Solidity CTF — Part 4: Read the Fine Print

Alex Towle
Sep 18, 2018 · 11 min read

EVM instructions dictate everything within Smart Contracts

This article is part 4 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.

New Challenge — Part 4: FinePrint

Part 4 has been deployed to Ropsten and explores how contract bytecode can be subtly manipulated. It ties in quite a lot of the material from the previous CTF challenges and also introduces the Merkle Tree data structure.

The goal of Part 4 is to take the contract's ether.

There is a Reddit thread for this challenge as well. Feel free to participate and ask questions!

Good luck!

Previous CTF Explanation — Part 3: "HoneyPot"

Part 3 was released in the previous article and broken by address `0xef045a554cbb0016275E90e3002f4D21c6f263e1`. The challenge was to steal all of ether from the contract, which was deployed to Ropsten here.

We are going to review the challenge's code step-by-step (Compiler version 0.4.24, no optimizer):

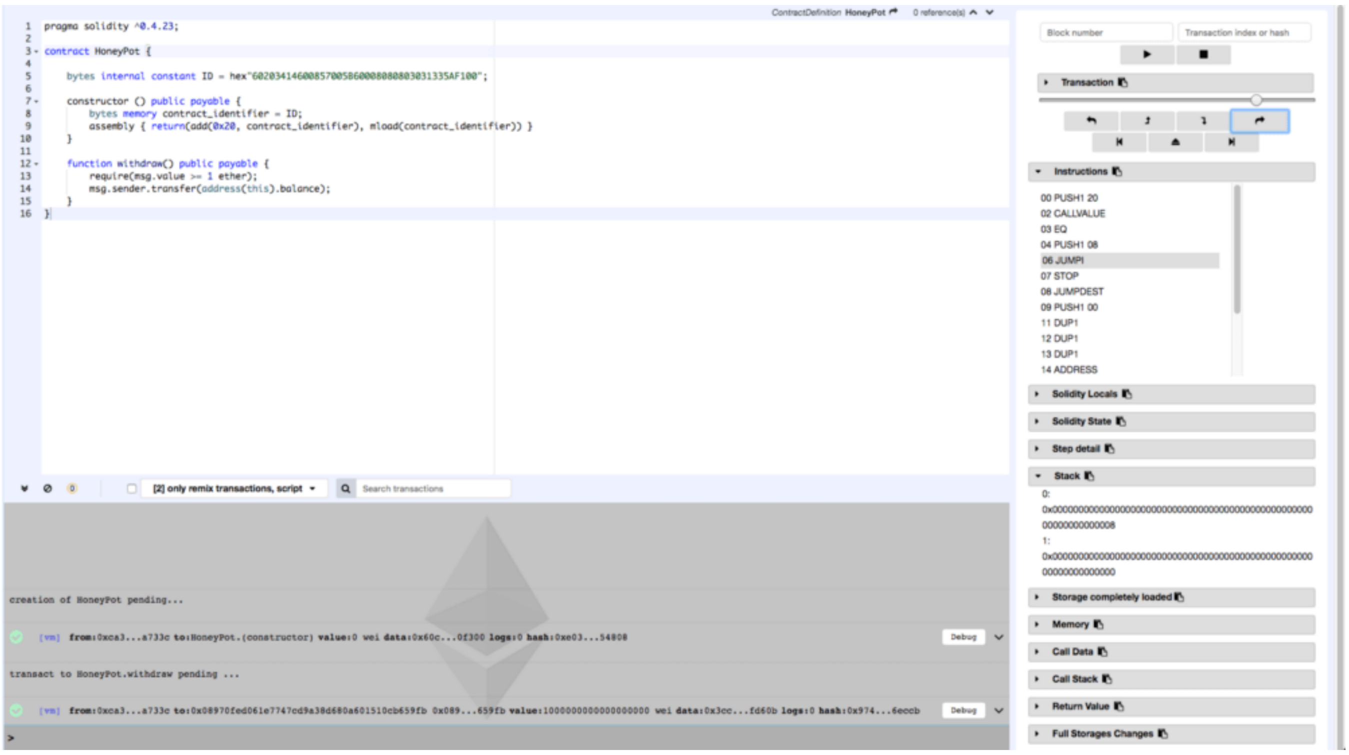
```
pragma solidity ^0.4.23;

contract HoneyPot {

    bytes internal constant ID = hex"60203414600857005B60008080803031335AF100";

    constructor () public payable {
        bytes memory contract_identifier = ID;
        assembly { return(add(0x20, contract_identifier), mload(contract_identifier)) }
    }

    function withdraw() public payable {
        require(msg.value >= 1 ether);
        msg.sender.transfer(address(this).balance);
    }
}
```

HoneyPot.sol hosted with ❤ by GitHub view raw

HoneyPot code

As stated above the goal of this challenge was to "withdraw" all of the ether from this contract. This contract already has a function `withdraw()` that will transfer the contract's ether to the caller, so we'll begin the walk through by analyzing that function.

At first glance, it appears that taking the HoneyPot contract's ether is simple. All that need to be done is to call `withdraw()` with a message value of at least 1 ETH. Since this challenge is so easy, lets just try calling `withdraw()` with a message value of 1 ETH.



```
pragma solidity ^0.4.23;

contract HoneyPot {

    bytes internal constant ID = hex"6020341460085700586000808080303131335AF100";

    constructor () public payable {
        bytes memory contract_identifier = ID;
        assembly { return(add(0x20, contract_identifier), mload(contract_identifier)) }
    }

    function withdraw() public payable {
        require(msg.value >= 1 ether);
        msg.sender.transfer(address(this).balance);
    }
}
```

```
creation of HoneyPot pending...

[vm] from:0xca3...a733c to:HoneyPot.(constructor) value:0 wei data:0x60c...0f300 logs:0 hash:0xe03...54808

transact to HoneyPot.withdraw pending ...

[vm] from:0xca3...a733c to:0x08970fed061e7747cd9a38d680a601510cb659fb 0x089...659fb value:1000000000000000000 wei data:0x3cc...fd60b logs:0 hash:0x974...6eccb
```

Instructions

```
00 PUSH1 20
02 CALLVALUE
03 EQ
04 PUSH1 08
06 JUMPI
07 STOP
08 JUMPDEST
09 PUSH1 00
11 DUP1
12 DUP1
13 DUP1
14 ADDRESS
```

Stack

```
0:
0x00000000000000000000000000000000000000000000000000
00000000000008
1:
0x00000000000000000000000000000000000000000000000000
00000000000000
```

Instructions for the call to "withdraw()" with 1 ETH

This picture showcases the call to `withdraw` with one ETH. Even though the call succeeded, the HoneyPot's ether was not sent back. "Why didn't I get my ETH back?", you may ask. Well, to answer this question, we must analyze the execution of this transaction.

If we look at the instructions tab, we will see an interesting list of instructions. Generally, contracts have logic at the beginning of the bytecode to route function calls to the correct function. In this case, all of this logic is absent. Instead, the execution begins by pushing `0x20` onto the stack and then pushing `callvalue` onto the stack. Afterwards, an equality check on these values fails, leading execution to end successfully.

It turns out that this challenge is more difficult than it seems at first. This contract's deployed bytecode is not the bytecode of a Solidity contract, which means that to solve this problem we will need to dig further into the EVM.

Since the `withdraw` function was a dead end, the only other function left to analyze is the `constructor`. It is worth noting that constructors work completely differently then any other Solidity function. In the Ethereum Yellow Paper, https://ethereum.github.io/yellowpaper/paper.pdf, section 7 discusses the detailed process of contract creation. The code run at contract creation acts as a script that takes as input the contract's bytecode, as well as any additional parameters.

To take a closer look at constructors, I will compare the inputs provided to two different contracts, contract A and contract B. Consider the following code:

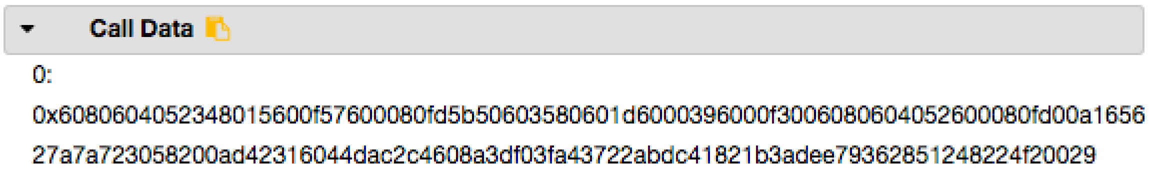
```
pragma solidity ^0.4.24;

contract A {
    constructor() { }
}

contract B {
    constructor(uint) public { }
}
```

AandB.sol hosted with ❤ by GitHub view raw

Code for contracts A and B

The only difference between the two above contracts is that contract B's constructor takes in a `uint` as a parameter, whereas A's constructor takes no parameters.

Following a call to the constructor of A, the `calldata` during execution is:

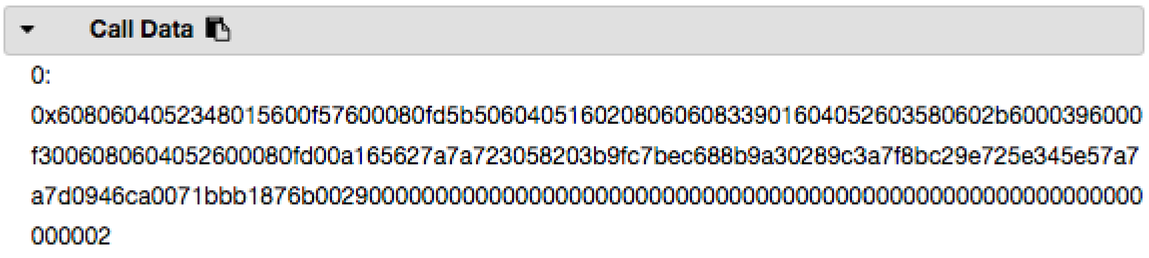


```
Call Data
0:
0x6080604052348015600f57600080fd5b5060358060 1d6000396000f3006080604052600080fd00a1656
27a7a723058200ad42316044dac2c4608a3df03fa43722abdc41821b3adee793628512482 24f20029
```

Calldata for the call to contract A's constructor

Taking a look at the above image, we can see that the input provided to A's constructor is definitely not what we would expect. In fact, the constructor of A takes *the bytecode of A as its calldata*.

Now that we have that knowledge, what is the `calldata` when we call contract B's constructor? Since this constructor takes a `uint` argument, I will provide the constructor with 2.



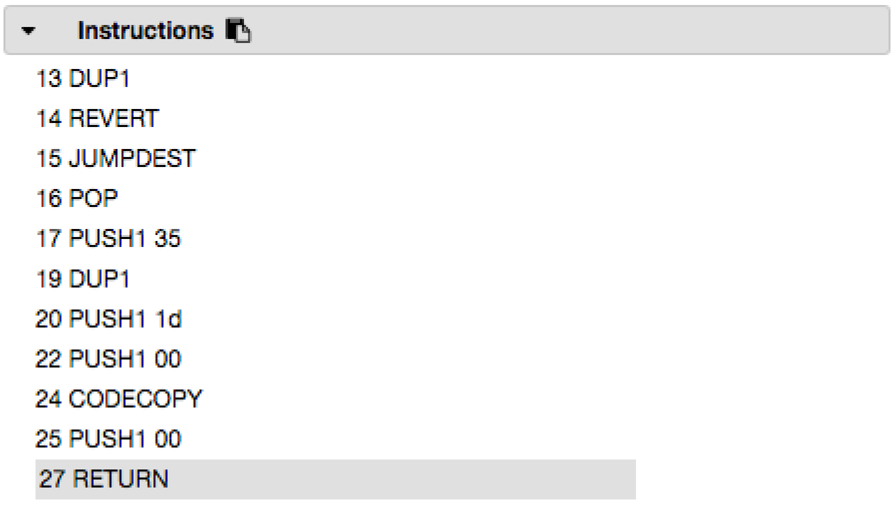
```
Call Data
0:
0x6080604052348015600f57600080fd5b50604051602080606083390160405260358060 2b6000396000
f3006080604052600080fd00a165627a7a723058203b9fc7bec688b9a30289c3a7f8bc29e725e345e57a7
a7d0946ca0071bbb1876b00290000000000000000000000000000000000000000000000000000000000
000002
```

Calldata for the call to contract B's constructor

In this case, the calldata provided to the constructor of contract B was the bytecode of the contract — exactly like before — but we can also see that our `uint` parameter, 2, was appended to the end of the `calldata`. From this comparison, we can see that *all* Solidity constructors take their contract's bytecode as input and some take in extra input. If at this point you think that constructors are weird, I don't blame you. They are, which makes it especially important to learn how they work.

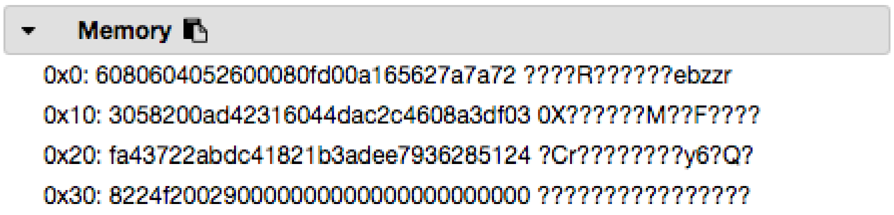
It is worthwhile to know what constructors take as input, but in this challenge, we care the most about what the constructor *returns*.

Using contract A from above, we will now analyze what happens at the end of contract construction. Below are pictures of the instructions, memory, and stack at the end of the construction call:



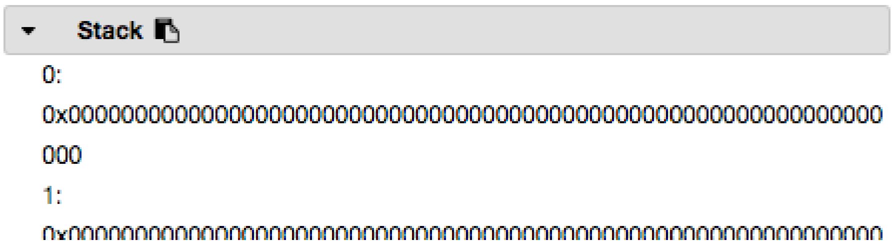
```
13 DUP1
14 REVERT
15 JUMPDEST
16 POP
17 PUSH1 35
19 DUP1
20 PUSH1 1d
22 PUSH1 00
24 CODECOPY
25 PUSH1 00
27 RETURN
```

The instructions at the end of the constructor call

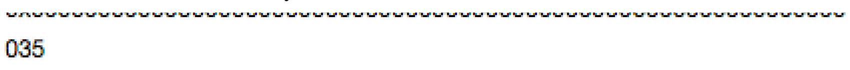


```
0x0: 6080604052600080fd00a165627a7a72 ????R??????ebzzr
0x10: 3058200ad42316044dac2c4608a3df03 0X??????M??F????
0x20: fa43722abdc41821b3adee7936285124 ?Cr????????y6?Q?
0x30: 8224f200290000000000000000000000 ????????????????
```

The memory at the end of the constructor call



```
0:
0x0000000000000000000000000000000000000000000000000000000000000
000
1:
0x0000000000000000000000000000000000000000000000000000000000000
```



```
035
```

The stack at the end of the constructor call

If we take a look at the instructions tab of the debugging window, we can see that there is a return statement at the end of the construction call. The `RETURN` opcode takes two arguments: A position to an interval in memory and the length of the interval. As seen in the stack, these arguments are `0x00` and `0x35` . This means that the data being returned will be `mem[0x00, 0x35)` , which can be seen in the memory tab above. This memory was placed in memory by the `CODECOPY` opcode at position `0x24` in the constructor's bytecode.

First, we can see that the `returndata` of this call is derived from *the contract creation code*. In fact, the data returned by this function call is exactly the code of contract A that is not part of the constructor's code. Looking through the data-to-be-returned in the memory tab, we can see that the `returndata` of this call is a substring of the `calldata` . In fact, the data returned by the constructor will be exactly the bytecode of the resulting deployed contract.

Now that we have the requisite knowledge of constructors, let's turn our attention to the HoneyPot contract's constructor:

```
bytes internal constant ID =
hex"6020341460085700 5B60008080803031335AF100";
constructor() public payable { bytes memory contract_identifier =
ID; assembly { return(add(0x20, contract_identifier),
mload(contract_identifier))
 }
}
```

It should now be fairly clear what is happening with our original HoneyPot contract. The data being returned in the assembly block — the hex-string called ID — is what will be deployed as the bytecode for the HoneyPot contract. Crazily, the HoneyPot contract presented at the beginning of the article is the **verified code** on that is shown on EtherScan.

Now that we know what code will actually be deployed, let's take a look at what it does. This is the bytecode that will be deployed by the HoneyPot's constructor:

```
6020341460085700 5B60008080803031335AF100
```

HoneyPot Bytecode hosted with ❤ by GitHub view raw

The bytecode deployed by the HoneyPot contract

And this is the opcode form of that bytecode (the two formats mean the same thing, but the opcode form is more readable for humans):

```
PUSH1 0x20
CALLVALUE
EQ
PUSH1 0x08
JUMPI
STOP
JUMPDEST
PUSH1 0x00
DUP1
DUP1
DUP1
ADDRESS
BALANCE
CALLER
GAS
CALL
STOP
```

HoneyPot Opcodes hosted with ❤ by GitHub view raw

The bytecode deployed by the HoneyPot contract

To solve this challenge, it is necessary to understand this opcode procedure. A list of all of the EVM opcodes and descriptions of their behavior is in the Yellow Paper's Appendix H.

The first instruction (it is actually at position 0 in the bytecode, disregard the line numbers) is `PUSH1 0x20`. The instruction `PUSH1` will push one byte of data that is provided in the bytecode onto the stack. In this case, the instruction will add the hexadecimal number `0x20` to the top of the EVM's stack. The `CALLVALUE` instruction will place the value sent with the message onto the stack. This value is exactly what `msg.value` returns. After this, the eq instruction takes the top two arguments off of the stack and checks for equality. In this case, `eq` will be comparing `0x20` and `msg.value`.

Following the equality check, the `PUSH1` instruction is again used to push `0x08` onto the stack. This value is used by the next instruction, `JUMPI`. The `JUMPI` instruction is a conditional jump — a jump that is only performed if a condition is met. `JUMPI` takes two arguments from the stack as input: the the target location in bytecode of the jump and a number representing whether or not a condition was met. In this case, the `JUMPDEST` of the `JUMPI` is `0x08` and the condition refers to the earlier equality check. To find out what instruction corresponds to position `0x08` of the bytecode, it is important to note that every opcode (ex. `JUMP` or `STOP`) has length of 1 byte. Secondly, every argument to a `PUSHX` instruction has length `X` bytes. In the case of a `PUSHX` operation, its argument is a *bytecode argument*; in other words, the argument of the `PUSHX` instruction takes up `X` bytes in bytecode after the `PUSHX` opcode. Using this information about bytecode positions, we can determine that the instruction at position `0x08` is the `JUMPDEST` on line 7. In the event that the `JUMPI` is not taken (when `0x20` is not equal to `msg.value`), the `STOP` instruction is met. This instruction stops the contract's execution and ends the transaction. If the `JUMPI` is taken, then the `JUMPDEST` instruction on line 7 is reached. If the transaction's `msg.value` is 32, we jump to position `0x08` in the bytecode. Otherwise, the transaction execution will end successfully.

A `JUMPDEST` instruction takes no stack arguments, and has no effect on the stack. It simply exists as a target for `JUMP` and `JUMPI` instructions; trying to jump to a position in code without a corresponding `JUMPDEST` will result in the EVM terminating execution. In this case, the `JUMPDEST` allows the `JUMPI` on line 5 to jump to position `0x08` in the contract bytecode. After reaching a `JUMPDEST` instruction, execution continues to the next instruction.

After the `JUMPDEST` instruction, there is a `PUSH1 0x00` instruction followed by three `DUP1` instructions. A `DUP1` instruction takes one value off of the stack — its argument — and then adds two values back onto the stack. These values are identical to the value taken off of the stack, so the result of a `DUP1` instruction is to duplicate the top stack item. This sequence of instructions will result in the top four arguments of the stack all being `0x00`. Next, the `ADDRESS` opcode will place the HoneyPot contract's address to the top of the stack. The `BALANCE` instruction will take this address off of the stack, and return the address's balance. To finish up the setup for a value transfer, the `CALLER` opcode is used to push the transaction sender's address onto the stack. Finally, the `GAS` opcode places all of the remaining gas in the transaction onto the stack.

The next opcode in the instruction list is the `CALL` opcode. This opcode takes seven arguments off of the stack: (1) the gas to send with the call, (2) the address to call, (3) the value to send with the call, (4) a pointer in memory to the beginning of the input array, (5) the size of the input array, (6) a pointer in memory to the beginning of the output array, and (7) the size of the output array. The high-level version of this call is:

```
msg.sender.call.value(address(this).balance)();
```

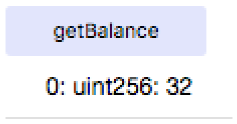
where `balance` is the HoneyPot's balance. This instruction will send a transaction to the sender address with no `calldata` and value equal to the balance of the HoneyPot contract. The result of this transaction is to send ether back to the sender's address.

Now that the explanation is over, it should be clear that to capture this contract's ether, we must call the `withdraw` function with a message value of `0x20` , 32 in decimal.

To see if our analysis was correct, I am going to be using the Take contract from below to call the HoneyPot's withdraw.

```
pragma solidity ^0.4.24;

contract HoneyPot {

    bytes internal constant ID = hex"60203414600857005B60008080803031335AF100";

    constructor () public payable {
        bytes memory contract_identifier = ID;
        assembly { return(add(0x20, contract_identifier), mload(contract_identifier)) }
    }

    function withdraw() public payable {
        require(msg.value >= 1 ether);
        msg.sender.transfer(address(this).balance);
    }
}

contract Take {
    function take(address _honeypot) external payable {
        HoneyPot(_honeypot).withdraw.value(msg.value)();
    }

    function getBalance() external view returns (uint) {
        return address(this).balance;
    }

    function () external payable { }
}
```

Fantastic! The call succeeds when I provide 32 wei as the message value to `take(address)`, as evidenced by the return value of `getBalance()`:



```
getBalance
0: uint256: 32
```

The ending balance of the Take contract

Contract Verification

Now that we looked at the HoneyPot contract under a microscope, let's try to understand why the full contract code is verified on Etherscan.

Etherscan has a feature that allows contracts to be verified after deploying them to the blockchain. To verify a contract on Etherscan, Solidity code must be provided to Etherscan that compiles to bytecode that matches the bytecode deployed to the blockchain. After reviewing the HoneyPot contract and the bytecode that it deploys to the blockchain after construction, it may seem counterintuitive that the HoneyPot contract is verified on Etherscan. It turns out that Etherscan is doing nothing incorrect by verifying the contract, but this verification does showcase the need to exercise caution when using verified contracts.

Etherscan's contract verification link is here: https://etherscan.io/verifyContract. This link outlines that Etherscan verifies contracts "[i]f the Bytecode generated matches the existing **Creation Address** Bytecode, the contract is then Verified." This means that if a contract compiles to bytecode that matches the contract creation code, then the contract is verified. Contract creation code is code that is deployed during the contract construction process. This code is used to deploy the code that will be permanently added to the blockchain after the initial transaction. The issue with this verification method is that `constructor()` functions can be used in weird ways, leading to completely different code being deployed than expected. The Solidity compiler prevents high-level return statements in constructors, but this can be circumvented by using the `return(uint a, uint b)` in inline-assembly. In general, it is not safe to use Smart Contracts that have assembly in the `constructor()` without a good understanding of what is going on.

What have we learned?

1. Solidity constructors are prevented from defining return values (save through assembly), because the data returned during contract creation is the resulting

contract's bytecode.

2. Etherscan only verifies **Contract Creation Code** rather than the code deployed after construction. Make sure that contracts that you want to use do not use assembly in the constructor (unless you know what you are doing).
3. All internal transfers use the CALL opcode, even Solidity functions including `address.send` and `address.transfer` .
4. We can make conditional jumps, like if-statements and for-loops, by using the `JUMPI` opcode and pushing numbers onto the stack!

Tune in next time when we take a deep dive into Solidity functions!

. . .

Originally published at authio.org.

Ethereum Security Smart Contracts Solidity Blockchain

About Help Legal