13 JULY 2019

The 0x vulnerability, explained

On Friday July 12th, 0x shut down their v2 Exchange because a flaw in the signature verification routine meant that a signature of `0x04` was treated as a valid signature for all non-smart-contract accounts. This blog post explains how this is possible.

Background

0x is, when grossly oversimplified, a platform which allows users to trade with other users. If Alice wants to buy 1000 ZRX for 1 ETH, Alice can submit an order through the 0x protocol. If Bob then decides that he wants to take Alice's order, he can use 0x's Exchange contract to securely perform the exchange.

In order for the Exchange to make sure that Alice really did make the offer that Bob claims she's making, Bob needs to submit Alice's *signature* with the order data. This signature is the only thing preventing Bob from claiming that Alice is offering 1000000 ZRX for 1 wei, so it's imperative that signatures can't be forged.

0x supports several types of signatures. *EIP712* and *EthSign* signatures are based on ECDSA signatures and are cryptographically secure, while *Wallet* and *Validator* signatures query an address for the validity of the provided signature. The

Wallet signature in particular will query the sender address, and is intended to allow multi-signature wallets to make trades.

The Code

Let's take a look at how 0x verifies a *Wallet* signature.

```
function isValidWalletSignature(
    bytes32 hash,
    address walletAddress,
    bytes signature
)
    internal
    view
    returns (bool isValid)
{
    bytes memory calldata = abi.encodeWithSelector(
        IWallet(walletAddress).isValidSignature.selector,
        hash,
        signature
    );
    assembly {
        let cdStart := add(calldata, 32)
        let success := staticcall(
            gas,                // forward all gas
            walletAddress,      // address of Wallet contract
            cdStart,            // pointer to start of input
            mload(calldata),    // length of input
            cdStart,            // write output over input
            32                  // output size is 32 bytes
        )

        switch success
        case 0 {
            // Revert with `Error("WALLET_ERROR")`
            /* snip */
            revert(0, 100)
        }
        case 1 {
            // Signature is valid if call did not revert and r
            isValid := mload(cdStart)
        }
    }
```

```
        return isValid;
    }
```

Source

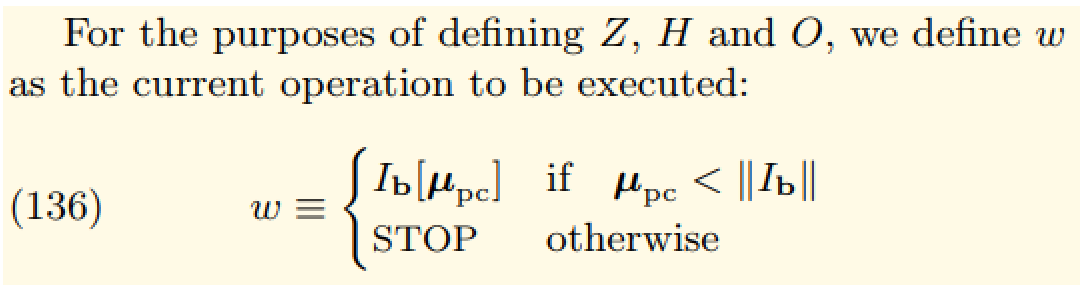
If we ignore the fact that this was written in inline assembly, it's fairly straightforward. First, lines 10-14 construct the ABI-encoded data which will be sent to the wallet. Lines 17-24 perform the call to the wallet. Finally, lines 26-34 check whether the call succeeded, and load the returned boolean.

The Problem

While the code to validate a *Wallet* signature was simple enough, it was written without knowledge of at least one of these two subtleties of the EVM:

1. Executing instructions outside the code is equivalent to executing STOP instructions

In most modern computers, executing undefined instructions means that your computer will execute garbage until the program crashes. However, the EVM is special because if execution happens to go outside the code of the smart contract, it's implicitly treated as a `STOP` instruction.



For the purposes of defining Z, H and O, we define w as the current operation to be executed:

$$(136) \qquad w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

Section 9.4

A side effect of this is that accounts with *no code* can still be executed - they'll just immediately halt.

2. While the CALL family of instructions allow specifying where the output should be copied, the output area is not cleared beforehand

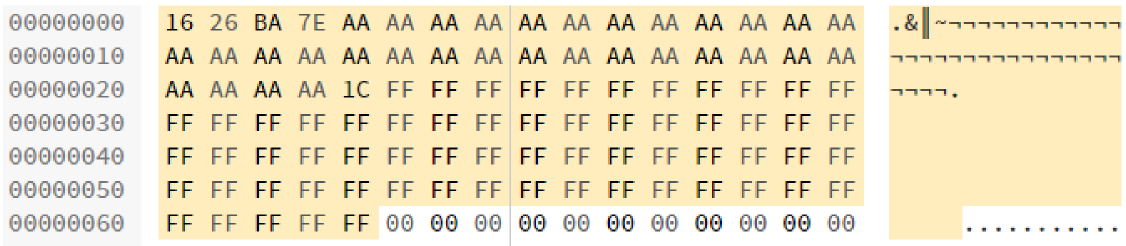
Take a look at this excerpt from the formal definition of the `CALL` instruction, where `μ[5]` is where in memory the return data should be copied, `μ[6]` is the length of the return data to be copied, and **o** is the data returned by the `CALL` instruction.

$$n \equiv \min(\{\boldsymbol{\mu}_{\mathbf{s}}[6], |\mathbf{o}|\})$$
$$\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[5] \dots (\boldsymbol{\mu}_{\mathbf{s}}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$$

Appendix H

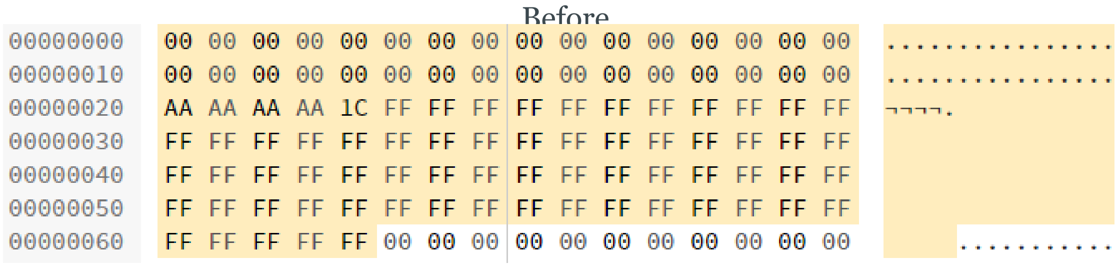
This states that only `n` bytes will be copied from the returned data to main memory, where `n` is the minimum of the number of bytes expected and the number of bytes returned. This implies that if less bytes are returned than expected, *only* the number of bytes returned will be copied to memory.

Going back to the validation routine, the authors instructed the EVM to overwrite the input data with the returned data, likely to save gas. Under normal operation, given a hash of `0xAA...AA` and a signature of `0x1CFF...FF`, the memory before and after a call might look something like this.



```
00000000  16 26 BA 7E AA AA AA AA AA AA AA AA AA AA AA AA  .&║~¬¬¬¬¬¬¬¬¬¬¬¬
00000010  AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA  ¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬
00000020  AA AA AA AA 1C FF FF FF FF FF FF FF FF FF FF FF  ¬¬¬¬.
00000030  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000040  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000050  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000060  FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00       ...........
```

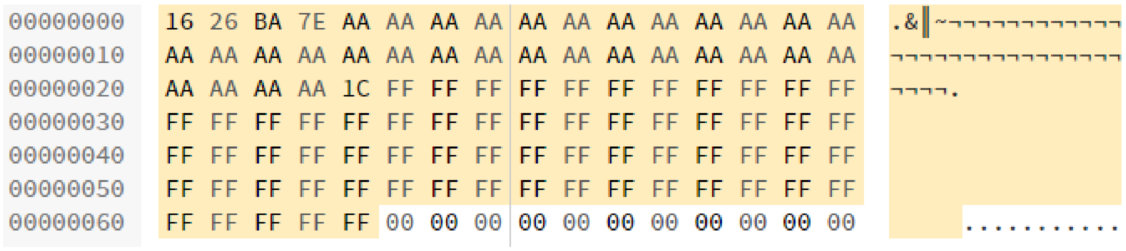
Before



```
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000020  AA AA AA AA 1C FF FF FF FF FF FF FF FF FF FF FF  ¬¬¬¬.
00000030  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000040  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000050  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000060  FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00       ...........
```

After

However, if there was no data returned by the call, then the memory after the call would look like this:



```
00000000  16 26 BA 7E AA AA AA AA AA AA AA AA AA AA AA AA  .&║~¬¬¬¬¬¬¬¬¬¬¬¬
00000010  AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA AA  ¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬¬
00000020  AA AA AA AA 1C FF FF FF FF FF FF FF FF FF FF FF  ¬¬¬¬.
00000030  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000040  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000050  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00000060  FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00       ...........
```

After

In other words, the memory is unchanged. Now, when the code on line 33 loads the first 32 bytes into a boolean variable, the nonzero value is coerced into `true`. Then, this `true` is returned from the function, indicating that "yes, the signature provided is fine" when it clearly isn't.

As for why the magic signature of `0x04` is always considered valid, it's because `0x04` is the ID number for a *Wallet* type signature, and the signature type is the last byte in the signature array.

```
// Allowed signature types.
enum SignatureType {
    Illegal,         // 0x00, default value
    Invalid,         // 0x01
    EIP712,          // 0x02
    EthSign,         // 0x03
    Wallet,          // 0x04
    Validator,       // 0x05
    PreSigned,       // 0x06
    NSignatureTypes  // 0x07, number of signature types. Alway
}
```

Source

```
function isValidSignature(
    bytes32 hash,
    address signerAddress,
    bytes memory signature
)
    public
    view
    returns (bool isValid)
{
    /* snip */

    // Pop last byte off of signature byte array.
    uint8 signatureTypeRaw = uint8(signature.popLastByte());

    /* snip */

    SignatureType signatureType = SignatureType(signatureTypeR

    /* snip */
    if (signatureType == SignatureType.Wallet) {
        isValid = isValidWalletSignature(
            hash,
            signerAddress,
            signature
        );
        return isValid;
    }
    /* snip */
}
```

Source

The Solution

To their credit, 0x triaged and fixed this vulnerability in a couple of hours. The relevant commit can be viewed here, but there's really only two sections of interest.

The first change requires that the wallet address contains some code. This behavior matches what the Solidity compiler inserts before performing a function call.

```
if iszero(extcodesize(walletAddress)) {
    // Revert with `Error("WALLET_ERROR")`
    /* snip */
    revert(0, 100)
}
```

The second change requires the return data to be exactly 32 bytes long. This is also what the Solidity compiler inserts after performing a function call.

```
if iszero(eq(returndatasize(), 32)) {
    // Revert with `Error("WALLET_ERROR")`
    /* snip */
    revert(0, 100)
}
```

Conclusion

This isn't the first time that an EVM subtlety has bitten a smart contract developer, and it won't be the last. However, it's usually through incidents like this that we all learn about a new dangerous pattern to watch out for. Hopefully through this incident, future developers and auditors will be able to catch similar problems before they make it to mainnet.

Further Reading

- ConsenSys Diligence's post-mortem on this issue
- 0x's post-mortem on this issue

samczsun
Read more posts by this author.

Read More

The Livepeer slashing vulnerability

What happens when good intentions go bad?

3 MIN READ

ConsenSys CTF - Rop EVM

A second CTF from ConsenSys Diligence. The solution is a blast from the past.

4 MIN READ

Latest Posts Twitter Ghost