May 25, 2023 4 min read

Understanding smart contract metadata

Updated: Jun 3, 2023

When solidity generates the bytecode for the smart contract to be deployed, it appends metadata about the compilation at the end of the bytecode. We will examine the data contained in this bytecode.

A simple smart contract

Let's look at the compiler output of the simplest possible solidity smart contract

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Empty {
    constructor() payable {}
}
```

The contract literally does nothing. We can compile it to see the init code with solc --optimize-runs 1000 --bin C.sol. We get the following output

```
======= C.sol:Empty =======
Binary:
6080604052603e80600f5f395ff3fe60806040525f80fdfea2646970667358221220308
2dbb4f4db7e5d53b235f44d3e38f839dc82075e2cda9df05b88e6585bca8164736f6c63
430008140033
```

That seems awfully large for a contract that doesn't do anything, right? Let's understand what all this bytecode is.

When we compile the code with solc --optimize-runs 1000 --bin --no-cbor-metadata C.sol we get the following output:

```
======= C.sol:Empty =======
Binary:
60806040526600880600f5f395ff3fe60806040525f80fd
```

That is much smaller! So what is all that extra information?

Solidity Metadata

By default, the solidity compiler appends metadata at the end of the "actual" initcode, which gets stored to the blockchain when the constructor finishes executing. Here is the "extra" code below:

```
fea2646970667358221220 3082dbb4f4db7e5d53b235f44d3e38f839dc82075e2cda9df
05b88e6585bca8164736f6c63430008140033
```

The last two bytes 0033 mean "look backward 0x33 bytes, that is the metadata." This refers to all the code between the leading fe (which is the INVALID opcode) and the ending 0033. We can check this is indeed 0x33 bytes.

```
# fe and 0033 are not included
>>>hex(len('a2646970667358221220 3082dbb4f4db7e5d53b235f44d3e38f839dc820
75e2cda9df05b88e6585bca8164736f6c6343000814') // 2)
# '0x33'
```

So what is this 0x33 (51 decimal) string?

We can get a hint if we make one tiny, seemingly inconsequential change to the source code. The change is literally just an additional comment.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.20;

contract Empty {
    // nothing
    constructor() payable {}
}
```

The following screenshot is a before and after.

You can see the underlined sections have changed even though the code functionality has not changed. We will explain the code in the boxes in the next section.

Decoding the metadata

At first, it will seem like we are magically picking substrings out of thin air; bear with us.

Let's look at the hex in the blue box above

```
>>> bytes.fromhex("69706673").decode("ASCII")
'ipfs'
```

Next let's look at the code in the red box

```
>>> bytes.fromhex("736f6c63").decode("ASCII")
'solc'
```

That gives us a clue about what this data contains: an IPFS hash and the solidity compiler version.

The IPFS Hash

The section underlined in yellow, along with the turquoise box can be put into the following python script (note that we are using the version of the code with the // nothing comment)

```
import base58
hex_ipfs_hash =
"12206a68b6b8bcc01ba559ec3adf7a387b6c4210a5dc69a05d038e9d17cae3fa373b"
bytes_str = bytes.fromhex(hex_ipfs_hash)
print(base58.b58encode(bytes_str).decode("utf-8"))

# QmVW2XyafSxDtiSqirJRauuT5SaQtGnQYsxxyYHrFmRTEa
```

The Qm...RTEa is the IPFS hash of the metadata file produced by the compiler. This section of code (turquoise and yellow) is encoded differently from the boxes above.
Specifically, the IPFS hash (the turquoise and yellow) is a base58 encoded version of the hex data "1220...RTEa".

This is the IPFS hash you would get if you put the JSON file from the Solidity compiler onto IPFS. Here is the JSON file in question.

We can store the JSON file as an actual file and then validate the hash matches the one we produced in python above. You will need the ipfs commandline tool installed (how to install).

```
mkdir out
solc --optimize-runs 1000 --bin --metadata C.sol --output-dir out
# Compiler run successful. Artifact(s) can be found in directory "out".

ipfs add -qr --only-hash out/Empty_meta.json
# QmVW2XyafSxDtiSqirJRauuT5SaQtGnQYsxxyYHrFmRTEa
```

This matches the hash from earlier.

Won't this cause hash collisions?

If two contracts with identical source code and compiler configurations store their verified source code on IPFS, the IPFS hashes will collide, but this is desirable because it actually saves storage space. The smart contracts are uniquely identified by the combination of the chain id and their address, not the IPFS content.

Getting the solidity version

Finally, if we convert the section in the orange box, we see the solidity version.

```
>>> 0x00 # solidity is version 0
0
>>> 0x08 # major version
8
>>> 0x14 # minor version
20
# correct, we used solidity 0.8.20
```

Why does smart contract metadata exist?

This metadata adds an extra 53 bytes to the deployment cost, which translates to an extra 10,600 gas (200 per bytecode) + the calldata cost (16 gas per non-zero bytes, 4 gas per zero-byte). This translates to up to 848 additional gas in calldata cost.
So why include it?

This enables smart contract code to be rigorously verified. The metadata JSON the compiler output includes a hash of the source code. So if the source code changes a little bit, the the metadata JSON file will change and its IPFS hash will change.

One weird trick to lower gas via IPFS hash

One obvious way to reduce gas cost at deployment time is the use the --no-cbor-metadata option. But if you need this for contract verification, then you can still reduce the gas cost by mining for IPFS hashes that have a lot of zero bytes in them. When the contract is deployed, the zero bytes will reduce the cost of the calldata. Because the source code is hashed, comments included, this means one can mine for comments that lead to gas-efficient IPFS hashes that will be appended to the contract. Note this means we want the hex representation of the hash to have zeros, not the base58 encoding.

Further resources

You can see all the options for manipulating this metadata in the relevant solidity documentation. Sourcify provides a tool parse the metadata of existing smart contracts.

Learn More

See our Solidity Bootcamp to learn more advanced smart contract topics.