**Verifying a transaction in Metamask**

**Verifying MetaMask Transactions**

Possessing this better understanding of encoding empowers us to do something very cool, and that's verify the transactions in our Metamask wallet before signing them.

If we write to a contract on Etherscan, a transaction will pop up in our Metamask wallet, by navigating to the HEX tab, we can see the data being sent in this transaction.

We should recognize this calldata as similar to the data we sent in our previous lessons.

0xfb37e883000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000076578616d706c6500000000000000000000000000000000000000000000000000

Foundry includes a cast command which can conveniently decode bytecode like this for us.

--calldata-decode: Decode ABI-encoded input data [aliases: cdd]

❗ **PROTIP** You can run cast --help for an exhaustive list of available cast commands!

Now, if we just run cast calldata-decode it's going to tell us we need a function signature (SIG) and our calldata (CALLDATA). We know how we can verify the function signature of our contract easily enough. In the image above, it looks like we're intending to call "MintNFT(string)". What happens when we run:

cast sig "mintNFT(string)"

0xfb37e883

We can see that this matches the first 4 bytes of the calldata in our Metamask transaction, 0xfb37e883! Great, now we can verify the calldata being sent with the transaction.

cast --calldata-decode "mintNFT(string)" 0xfb37e883000000000000000000000000000000000000000000000000000000000000002000000000000000000000000000000000000000000000000000000000000000076578616d706c6500000000000000000000000000000000000000000000000000

Worked like a charm!

**Signature Collision**

There's something important to keep in mind with respect to function signatures. Sometimes, as a quirk of the encoding, two completely different functions will encode into the same function signature.

To see this yourselves, navigate to [**openchain.xyz/signatures**](https://openchain.xyz/signatures).

In the search field, enter 0x23b872dd. You'll see that this function signature is attributed to multiple, completely different functions!

Importantly, the Solidity compiler **will not** allow a contract to contain two or more functions which share a selector. You'll receive a compiler error:

I encourage you to try this out yourself in Remix! See if you can find any other conflicting function selectors! This is why it may be important to verify through the contract's code directly, which function is actually being called.

**Wrap Up**

With these new skills we can now verify any transaction proposed to our wallet! This is incredibly valuable, especially when interacting with frontends. You should always be sure the functions you're calling are behaving exactly as you expect them to.

In order to verify our transactions we need to:

1. Check the address
   * Verify the contract we're interacting with is what's expected
2. Check the function selector
   * Verify the provided function selector vs the function on the contract we expect to be calling
3. Decode the calldata
   * Verify the calldata to assure the parameters being sent to the function are what we expect to be sending.

It's too easy to unknowingly send a malicious transaction, but by following these steps you can be sure that your wallet is doing exactly what you intend it to.