Look here for the point.

ECRecover() and You:

Using Signed Messages for Flexibility and Security

Destry Saul (destrys)

Unchained Capital

TruffleCon 2018

UNCHAINED CAPITAL

TRUFFLECON

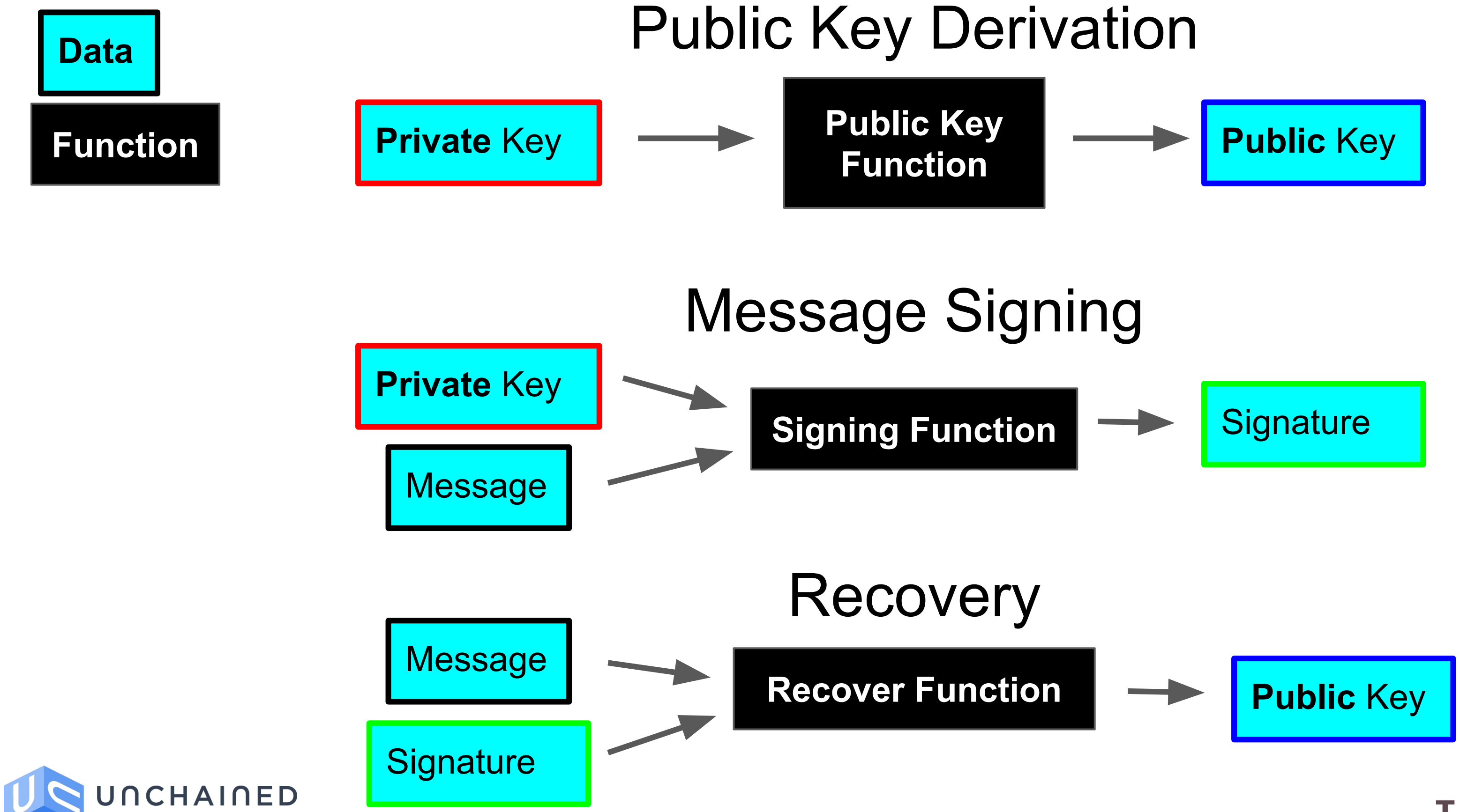
Agenda

- Promises from Asymmetric Crypto
- Basic Example - use in a contract, and web3 signature generation
- Signing Mechanics
  - Trezor, Ledger, and Metamask examples
- Example Usage
  - Kill-Switch, ColdOwner, Multisig, TimeClock, Registration, NaiveOracle
- Ecrecover in production
  - multisig custody @ unchained capital

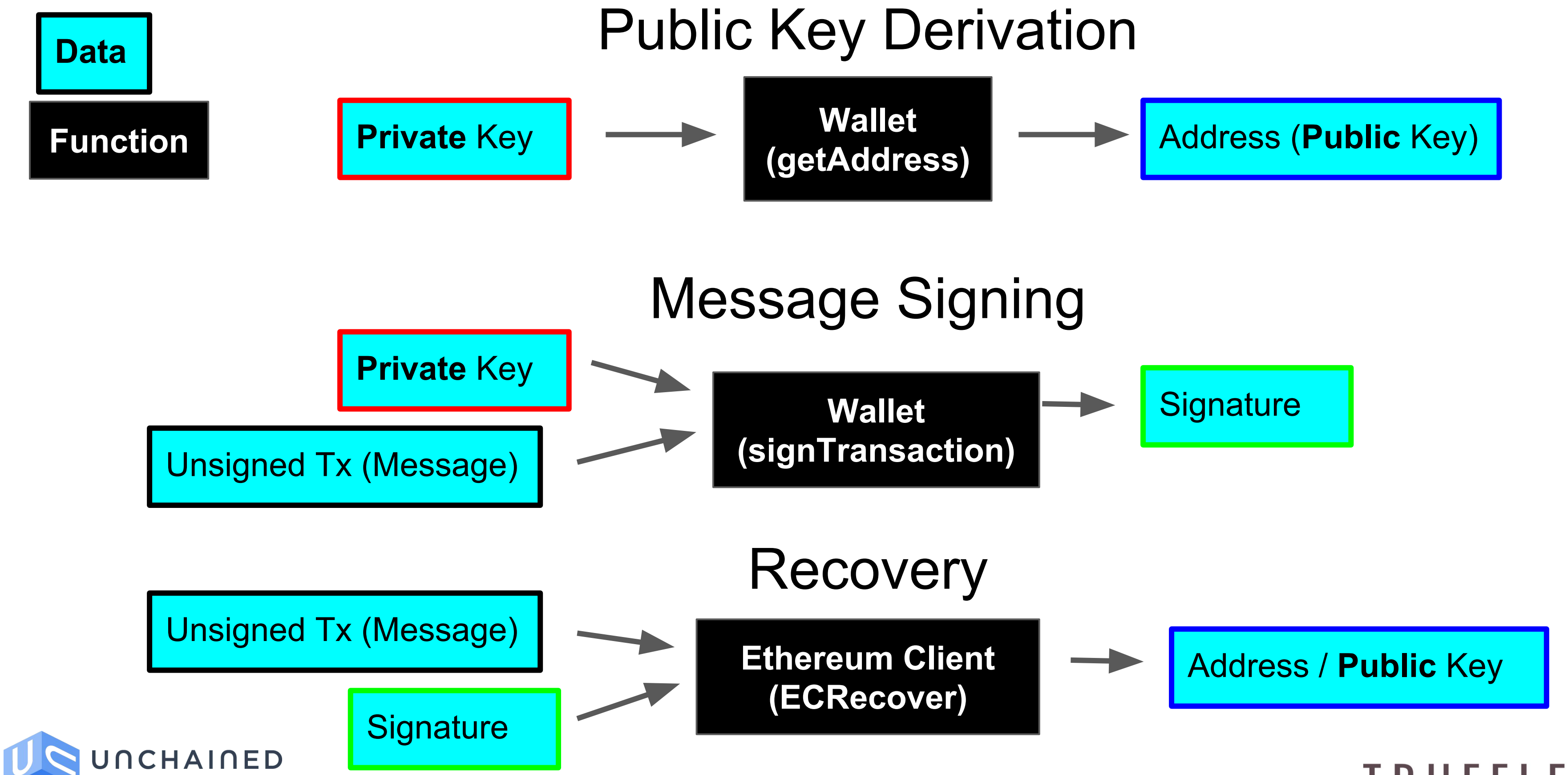
Goals: Learn the Basics, See the Dangers, Make you Think

Example Repo (in progress): github.com/destrys/ecrecover-demo

Three One-Way Functions Make Signing Work



Crypto Functions as used by Ethereum



Data

Function

Public Key Derivation

Private Key → **Wallet (getAddress)** → Address (**Public** Key)

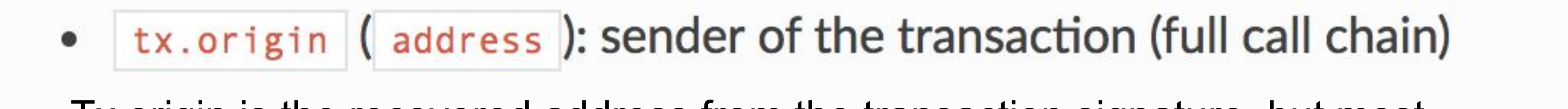
Message Signing

Private Key, Unsigned Tx (Message) → **Wallet (signTransaction)** → Signature

Recovery

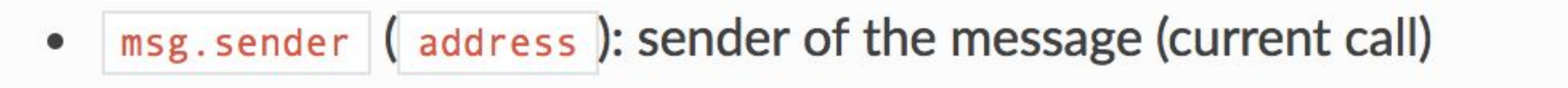
Unsigned Tx (Message), Signature → **Ethereum Client (ECRecover)** → Address / **Public** Key

Solidity: tx.origin is transaction signer (from ecrecover), msg.sender is function caller (from contract logic)



- `tx.origin` (`address`): sender of the transaction (full call chain)

Tx.origin is the recovered address from the transaction signature, but most contracts use msg.sender:



- `msg.sender` (`address`): sender of the message (current call)

Msg.sender is equal to tx.origin for the first call, but if a contract makes an external call, msg.sender is set to the calling contract’s address. Tx.origin is still available.

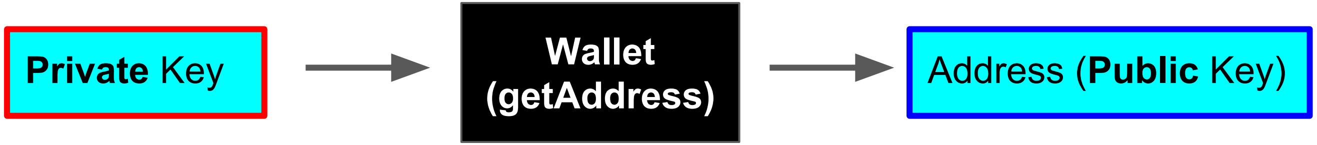
Solidity Docs:
https://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html

Messages and Transactions go through the same signing process

Data

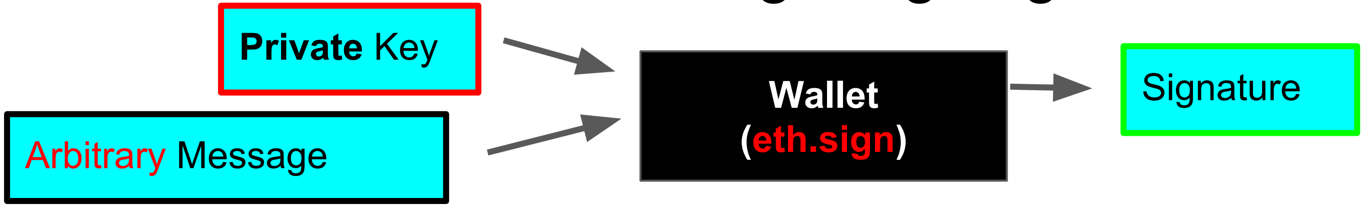
Function

Public Key Derivation



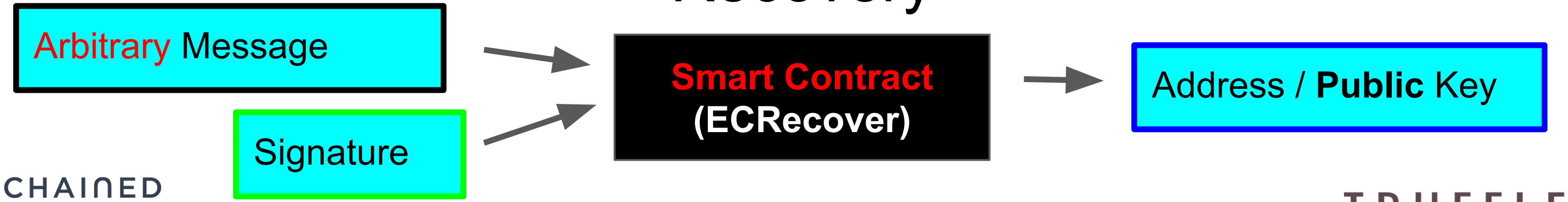
Private Key → **Wallet (getAddress)** → Address (**Public** Key)

Message Signing



Private Key, Arbitrary Message → **Wallet (eth.sign)** → Signature

Recovery

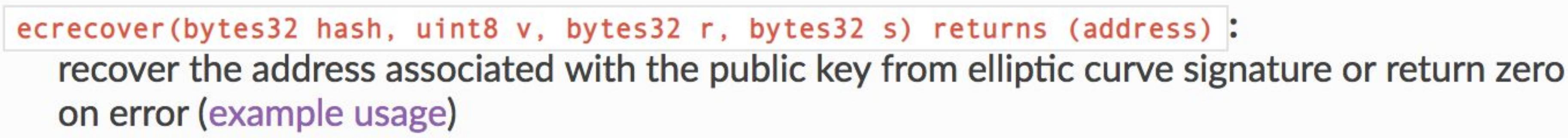


Arbitrary Message, Signature → **Smart Contract (ECRecover)** → Address / **Public** Key

UNCHAINED CAPITAL

TRUFFLECON

ecrecover docs aren’t great



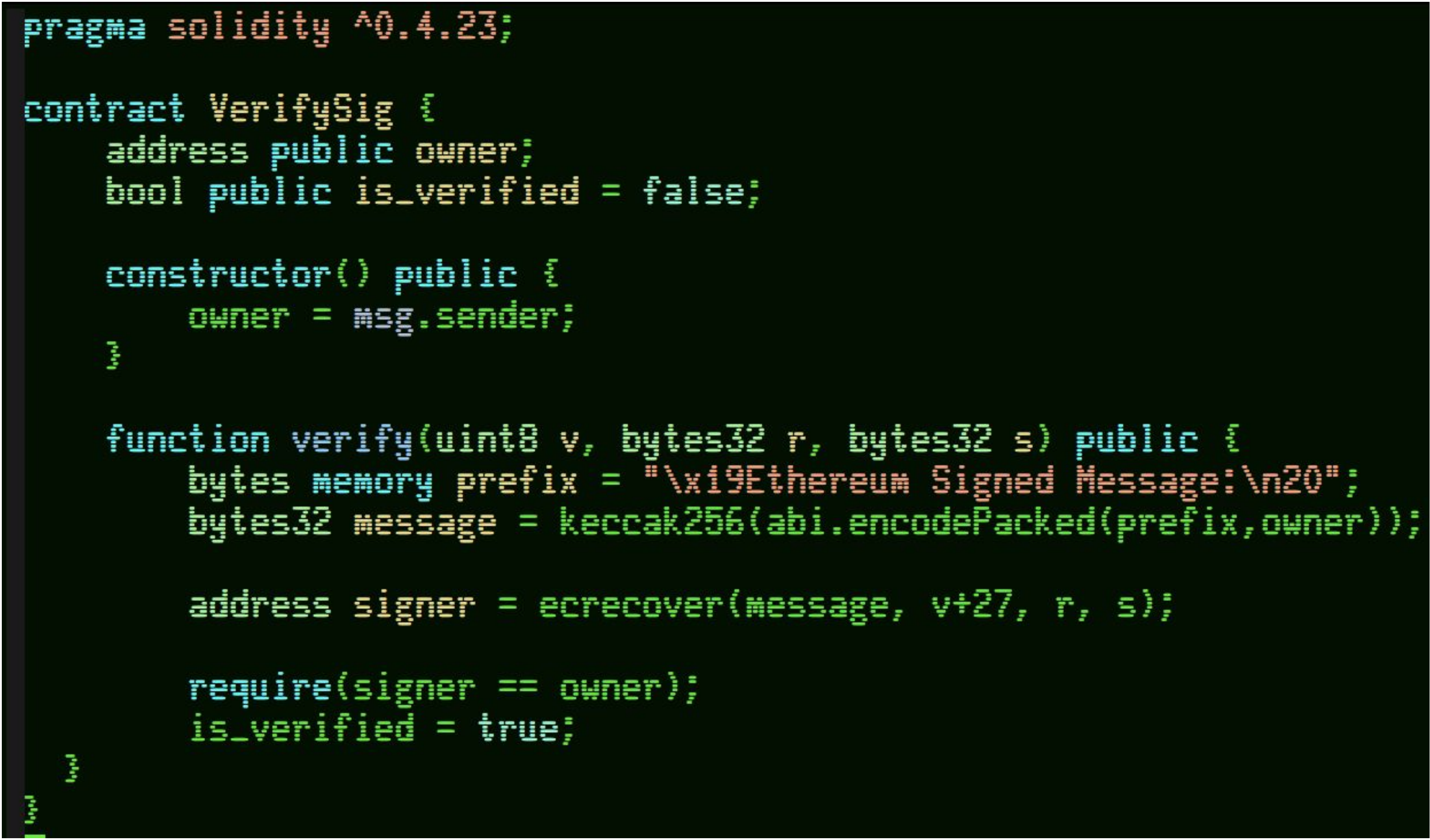
`ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)`:
recover the address associated with the public key from elliptic curve signature or return zero on error (example usage)

Great! So with V, R, and S, and a 32 byte hash, we can recover an address.

What are V, R, S? What is the hash?

That link goes to this stackoverflow page:
https://ethereum.stackexchange.com/questions/1777/workflow-on-signing-a-string-with-private-key-followed-by-signature-verificatio

Functioning ecrecover in a contract

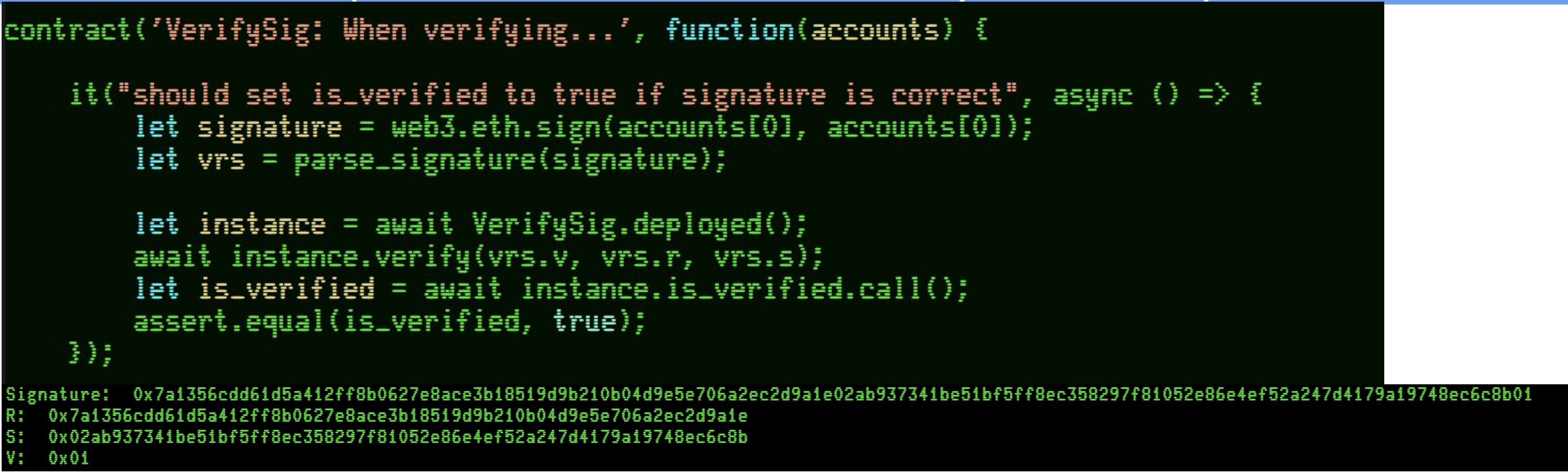


```
pragma solidity ^0.4.23;

contract VerifySig {
    address public owner;
    bool public is_verified = false;

    constructor() public {
        owner = msg.sender;
    }

    function verify(uint8 v, bytes32 r, bytes32 s) public {
        bytes memory prefix = "\x19Ethereum Signed Message:\n20";
        bytes32 message = keccak256(abi.encodePacked(prefix,owner));

        address signer = ecrecover(message, v+27, r, s);

        require(signer == owner);
        is_verified = true;
  }
}
```

- The Prefix is prepended by the WALLET, so we have to account for it.

- The actual data that are signed is the hash of the prefix+message

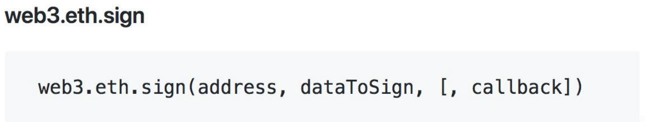
Web3 can generate a valid signature in a test (but careful which version you are on)



```
contract('VerifySig: When verifying...', function(accounts) {

    it("should set is_verified to true if signature is correct", async () => {
        let signature = web3.eth.sign(accounts[0], accounts[0]);
        let vrs = parse_signature(signature);

        let instance = await VerifySig.deployed();
        await instance.verify(vrs.v, vrs.r, vrs.s);
        let is_verified = await instance.is_verified.call();
        assert.equal(is_verified, true);
    });
```

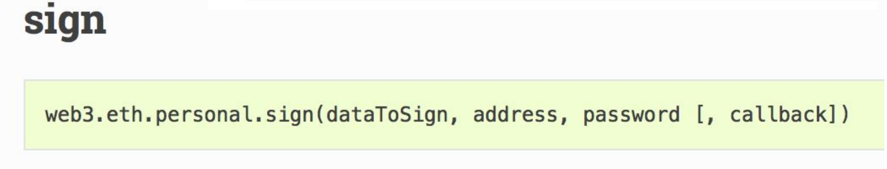
```
Signature:  0x7a1356cdd61d5a412ff8b0627e8ace3b18519d9b210b04d9e5e706a2ec2d9a1e02ab937341be51bf5ff8ec358297f81052e86e4ef52a247d4179a19748ec6c8b01
R:  0x7a1356cdd61d5a412ff8b0627e8ace3b18519d9b210b04d9e5e706a2ec2d9a1e
S:  0x02ab937341be51bf5ff8ec358297f81052e86e4ef52a247d4179a19748ec6c8b
V:  0x01
```

- Web3.eth.sign or web3.personal.sign -
- “Ethereum Signed Message” is prepended automatically
- Need to parse the signature



web3.eth.sign

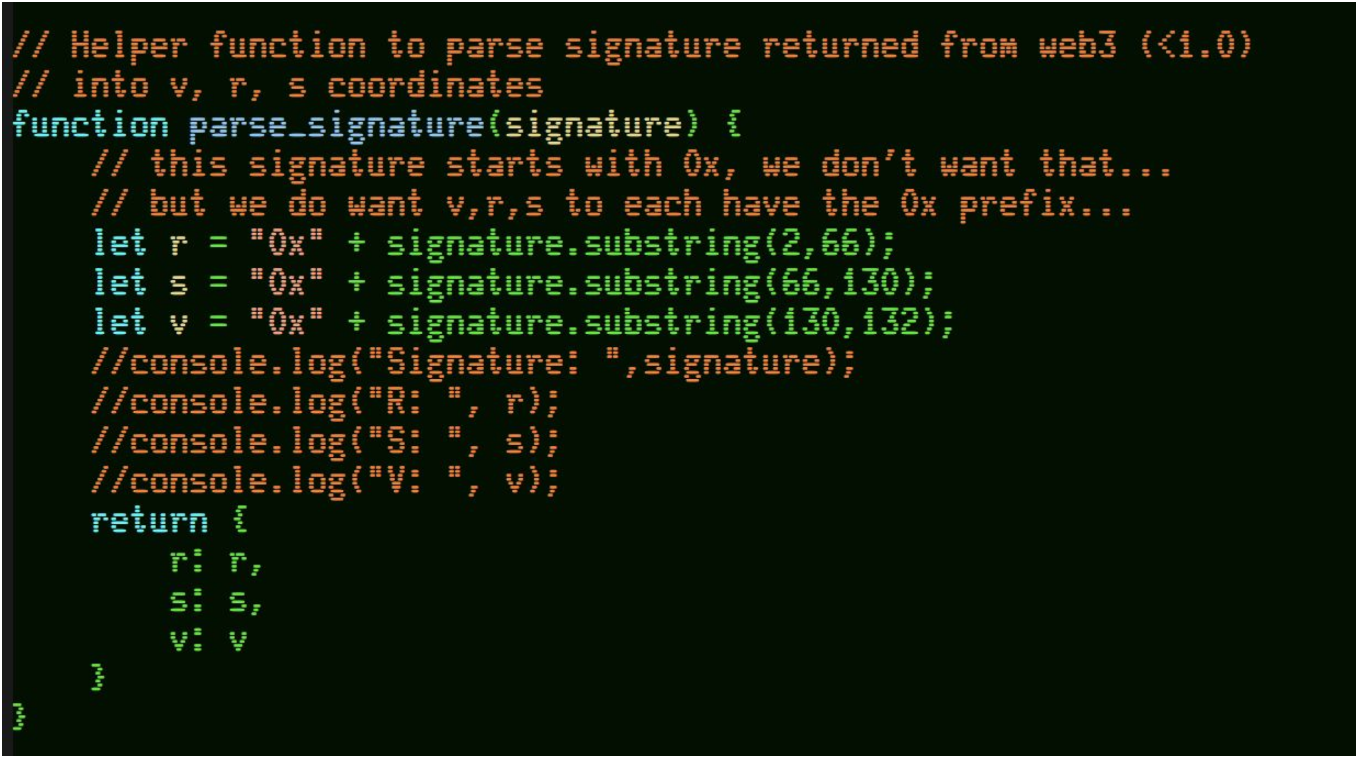
```
web3.eth.sign(address, dataToSign, [, callback])
```



sign

```
web3.eth.personal.sign(dataToSign, address, password [, callback])
```

Most (but not all) wallets require you to parse the sig into (v,r,s)



```
// Helper function to parse signature returned from web3 (<1.0)
// into v, r, s coordinates
function parse_signature(signature) {
    // this signature starts with 0x, we don't want that...
    // but we do want v,r,s to each have the 0x prefix...
    let r = "0x" + signature.substring(2,66);
    let s = "0x" + signature.substring(66,130);
    let v = "0x" + signature.substring(130,132);
    //console.log("Signature: ",signature);
    //console.log("R: ", r);
    //console.log("S: ", s);
    //console.log("V: ", v);
    return {
        r: r,
        s: s,
        v: v
    }
}
```

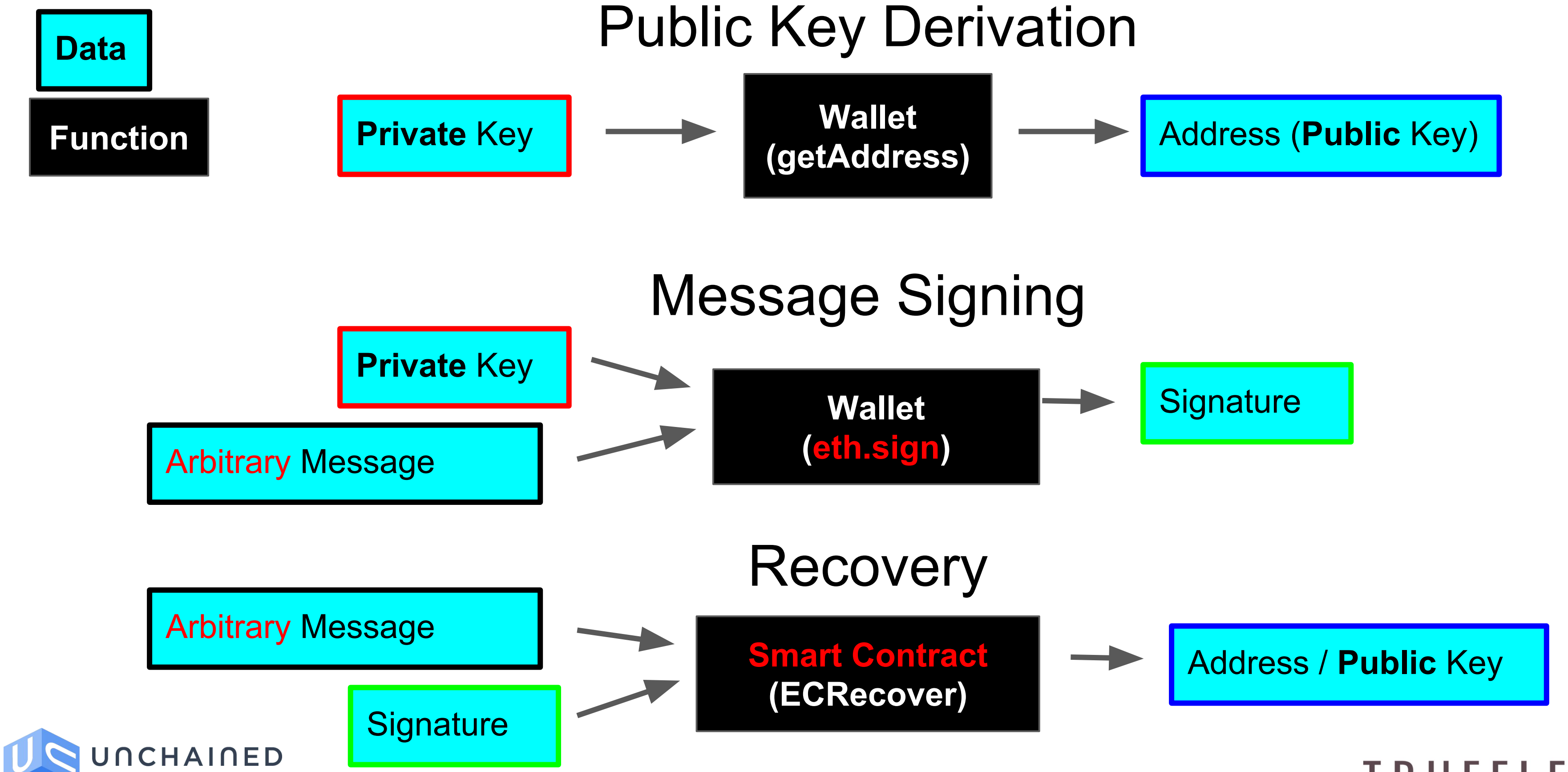
As you'll see in a minute, V can be (0|1), (27|28), or (1b|1c) depending on the wallet.

Let’s talk about signing for a minute

Remember that signing is just a function.
You just need the message, private key, and the right software to sign.

The tricky part is that unlike transaction signing, message signing isn’t standardized yet.

Messages and Transactions go through the same signing process



Data

Function

Public Key Derivation

Private Key

Wallet (getAddress)

Address (**Public** Key)

Message Signing

Private Key

Arbitrary Message

Wallet (eth.sign)

Signature

Recovery

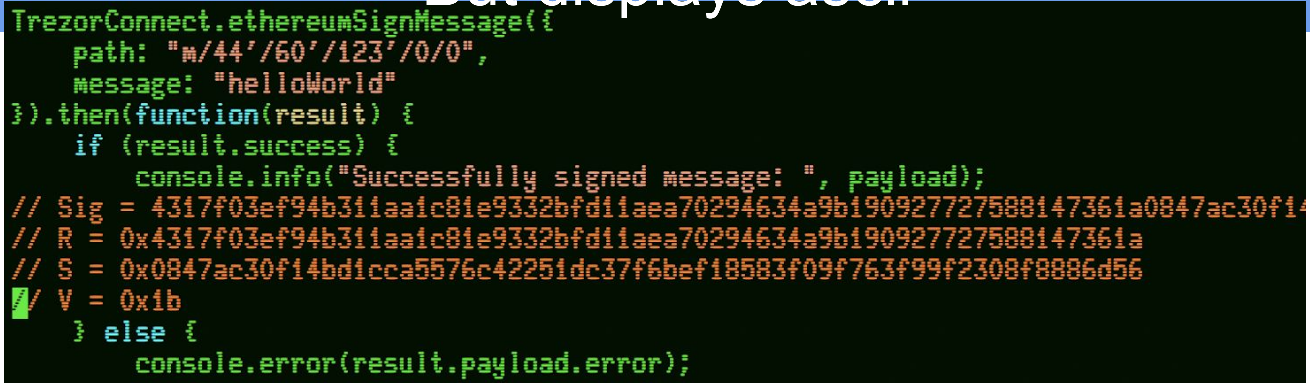
Arbitrary Message

Signature

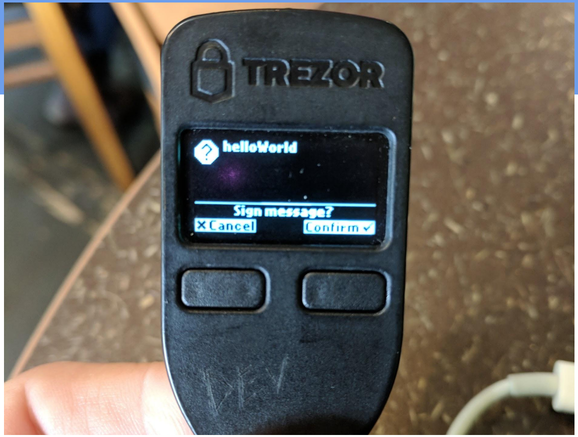
Smart Contract (ECRecover)

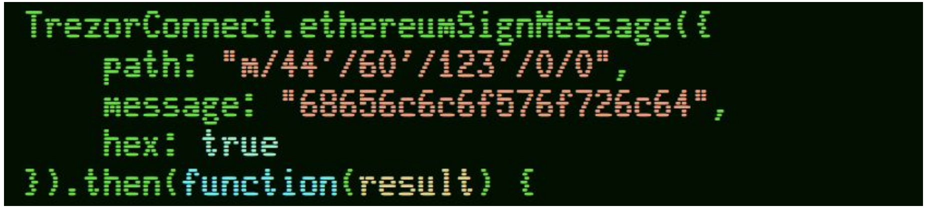
Address / **Public** Key

Trezor - accepts ascii (and now bytes!) But displays ascii

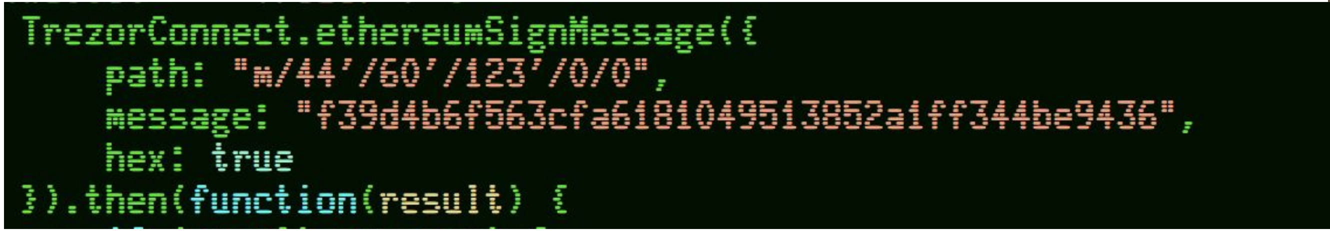


```
TrezorConnect.ethereumSignMessage({
    path: "m/44'/60'/123'/0/0",
    message: "helloWorld"
}).then(function(result) {
    if (result.success) {
        console.info("Successfully signed message: ", payload);
// Sig = 4317f03ef94b311aa1c81e9332bfd11aea70294634a9b1909277275881473 61a0847ac30f14
// R = 0x4317f03ef94b311aa1c81e9332bfd11aea70294634a9b190927727588147361a
// S = 0x0847ac30f14bd1cca5576c42251dc37f6bef18583f09f763f99f2308f8886d56
// V = 0x1b
    } else {
        console.error(result.payload.error);
```

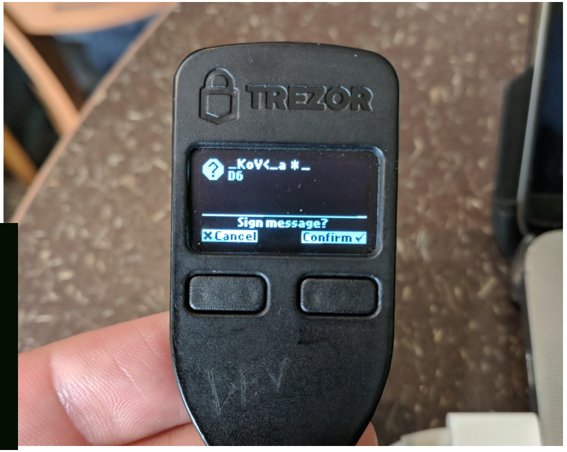




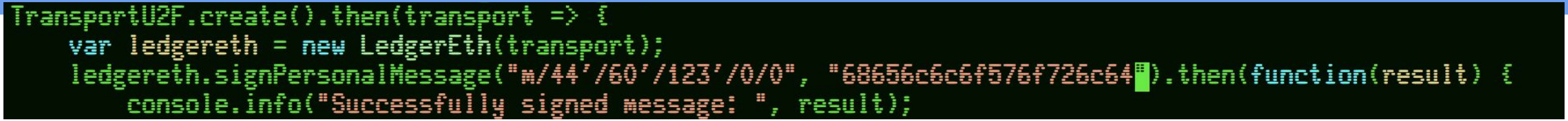
```
TrezorConnect.ethereumSignMessage({
    path: "m/44'/60'/123'/0/0",
    message: "68656c6c6f576f726c64",
    hex: true
}).then(function(result) {
```



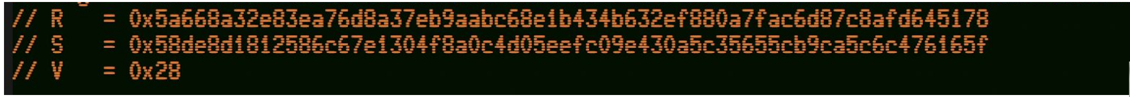
```
TrezorConnect.ethereumSignMessage({
    path: "m/44'/60'/123'/0/0",
    message: "f39d4b6f563cfa618104951385 2a1ff344be9436",
    hex: true
}).then(function(result) {
```



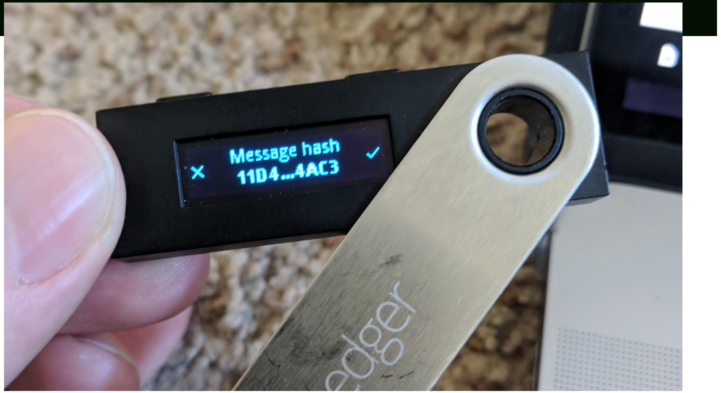
Ledger - accepts bytes - display practically useless



```
TransportU2F.create().then(transport => {
    var ledgereth = new LedgerEth(transport);
    ledgereth.signPersonalMessage("m/44'/60'/123'/0/0", "68656c6c6f576f726c64").then(function(result) {
        console.info("Successfully signed message: ", result);
```



```
// R   = 0x5a668a32e83ea76d8a37eb9aabc68e1b434b632ef880a7fac6d87c8afd645178
// S   = 0x58de8d1812586c67e1304f8a0c4d05eefc09e430a5c35655cb9ca5c6c476165f
// V   = 0x28
```

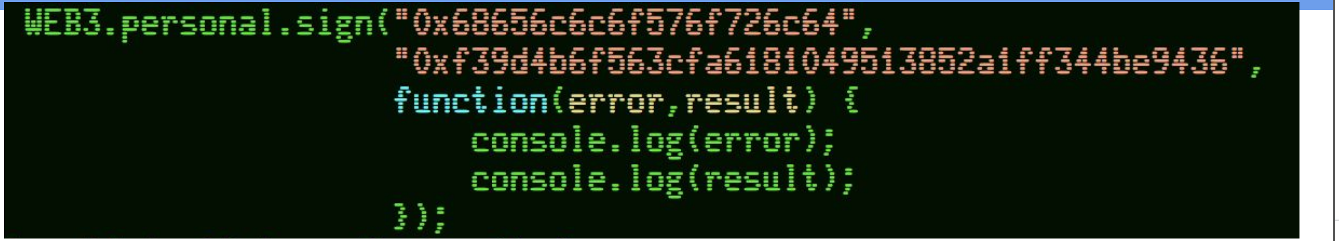


Ledger displays the (first4...last4) of the hash of the message.

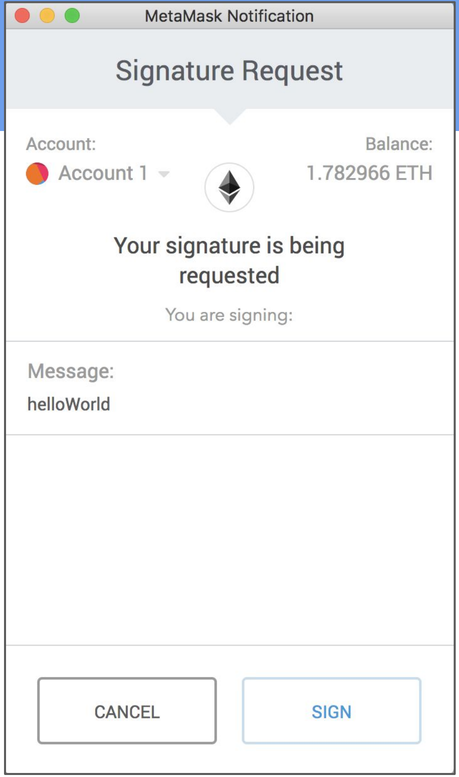
Dangers!

- Certain length messages are signed wrong https://github.com/LedgerHQ/ledger-app-eth/issues/14
- Only use bip32 paths under m/44'/ https://github.com/LedgerHQ/ledger-app-eth/issues/16

Metamask - accepts bytes, displays ascii



```
WEB3.personal.sign("0x68656c6c6f576f726c64",
                   "0xf39d4b6f563cfa6181049513852a1ff344be9436",
                   function(error,result) {
                       console.log(error);
                       console.log(result);
                   });
```



Signature Request

Account: Account 1

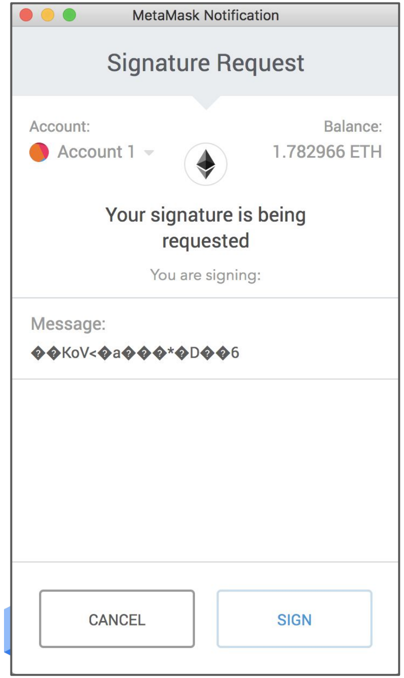
Balance: 1.782966 ETH

Your signature is being requested

You are signing:

Message:

helloWorld



Signature Request

Account: Account 1

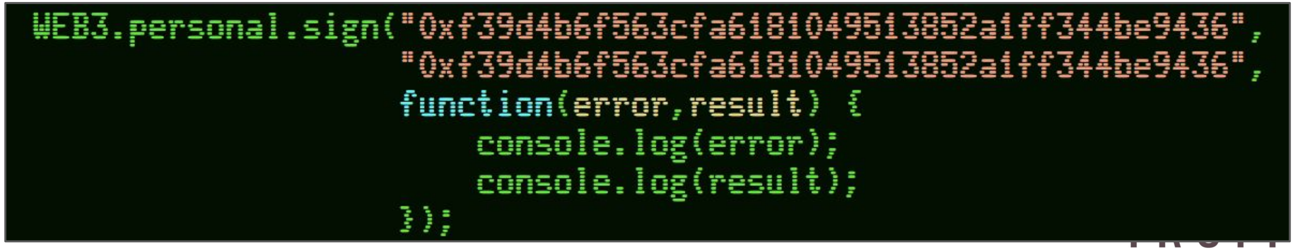
Balance: 1.782966 ETH

Your signature is being requested

You are signing:

Message:

��KoV<�a���*�D��6



```
WEB3.personal.sign("0xf39d4b6f563cfa6181049513852a1ff344be9436",
                   "0xf39d4b6f563cfa6181049513852a1ff344be9436",
                   function(error,result) {
                       console.log(error);
                       console.log(result);
                   });
```

Signing Review - **Test your signing code!**

Trezor
- Only very recently could you sign bytes
- Displays the ascii of the message you ask to sign

Ledger
- Displays the (first4...last4) of the hash of the un-prefixed message
- Restricted bip32 range

Metamask
- Displays ascii

Test Test Test Test Test Test Test

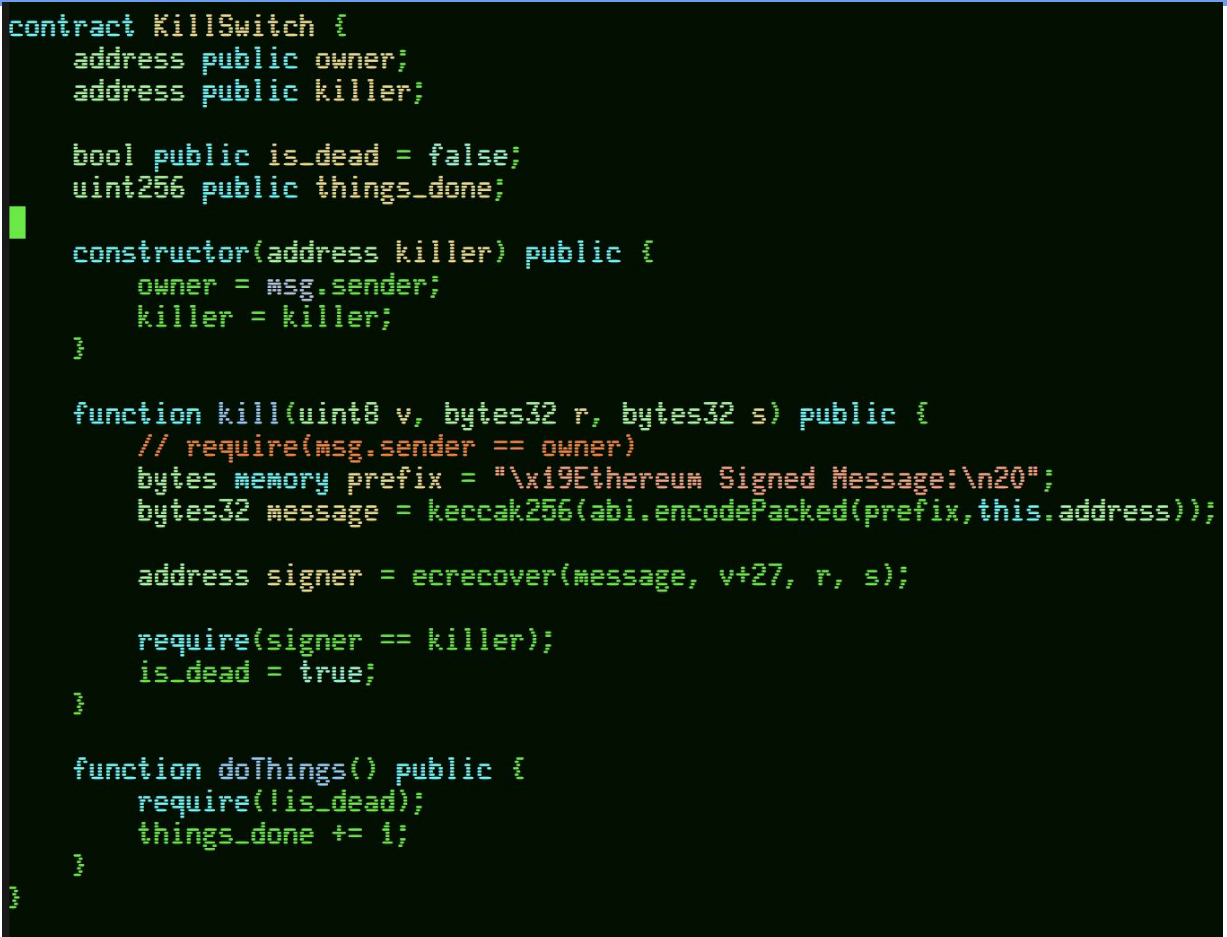
- You can use ganache to generate signatures
- You can initialized hardware wallets with the same mnemonic
- You can run hardware wallet emulators
- Verify you can generate the correct signatures with whatever wallet your signers will be using.

Great, we’ve covered how to call ecrecover in a contract, and how to generate a signature.

Now why would you want to use ecrecover in a contract???

- Example Usage
    - Kill Switch / ColdOwner
    - MultiSig
        - Nonce Considerations
    - TimeClock (central hot wallet)
    - Registrations (distribute single-use signatures)
    - Naive Oracle

Signature triggers one-time state change



```
contract KillSwitch {
    address public owner;
    address public killer;

    bool public is_dead = false;
    uint256 public things_done;

    constructor(address killer) public {
        owner = msg.sender;
        killer = killer;
    }

    function kill(uint8 v, bytes32 r, bytes32 s) public {
        // require(msg.sender == owner)
        bytes memory prefix = "\x19Ethereum Signed Message:\n20";
        bytes32 message = keccak256(abi.encodePacked(prefix,this.address));

        address signer = ecrecover(message, v+27, r, s);

        require(signer == killer);
        is_dead = true;
    }

    function doThings() public {
        require(!is_dead);
        things_done += 1;
    }
}
```

- Use the **`require()`** for extra security (unless you're trying to protect against loss of the owner account…)
- notice **`this.address`** in the message. Makes this signature only valid for this contract.
- as-is, any address can execute the **`kill()`** function if they have the correct signature.
- you could put a **`selfdestruct()`** or **`transfer()`** in the kill to transfer any eth in the contract

When you see ecrecover, ask yourself: “What happens when I (or someone else) use this sig again?”

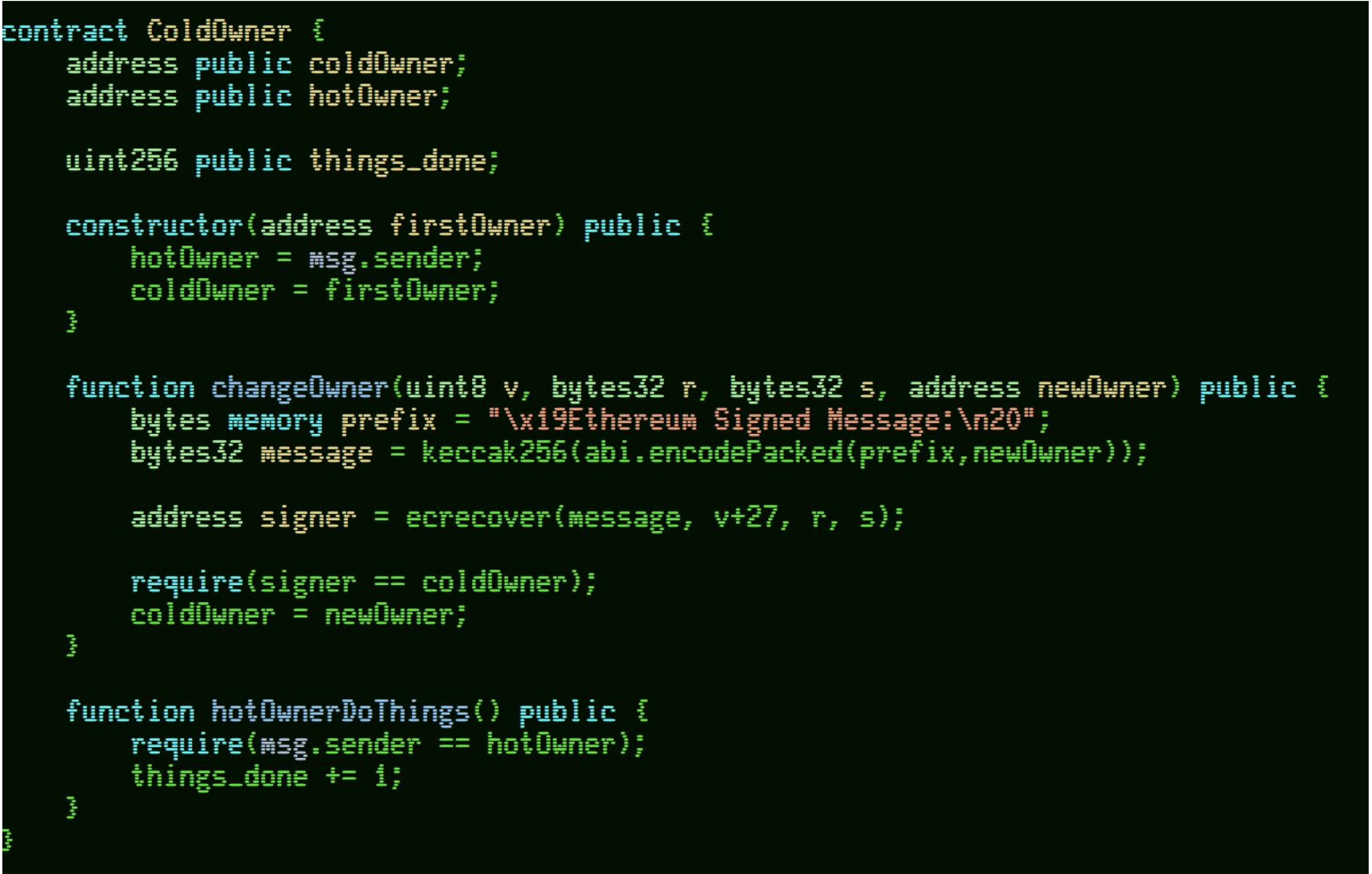
The biggest danger/nuance with ecrecover is that the contract and signatures need to be constructed to account for replay.

Once a signature is used in a function call, it’s on the blockchain, everyone has it.

Your standard ethereum addresses do this by having an increasing nonce associated with the address, it increments after each transaction.

You don’t need a nonce to use ecrecover in a smart contract, but you need to think carefully about how to use a single-use signature.

change the state so that the recovered address is invalid



```
contract ColdOwner {
    address public coldOwner;
    address public hotOwner;

    uint256 public things_done;

    constructor(address firstOwner) public {
        hotOwner = msg.sender;
        coldOwner = firstOwner;
    }

    function changeOwner(uint8 v, bytes32 r, bytes32 s, address newOwner) public {
        bytes memory prefix = "\x19Ethereum Signed Message:\n20";
        bytes32 message = keccak256(abi.encodePacked(prefix,newOwner));

        address signer = ecrecover(message, v+27, r, s);

        require(signer == coldOwner);
        coldOwner = newOwner;
    }

    function hotOwnerDoThings() public {
        require(msg.sender == hotOwner);
        things_done += 1;
    }
}
```

- change the signed message to be next cold owner.
- **Dangers**!: if an address was **`coldOwner`** on two contracts, the signature could be used on both (is this good or bad?)
- add contract address to message to make signature only usable by this contract
- **`coldOwner`** doesn’t need to know anything about the ethereum blockchain to sign.
  - Could end up in a loop

You can verify multiple signatures in one transaction!

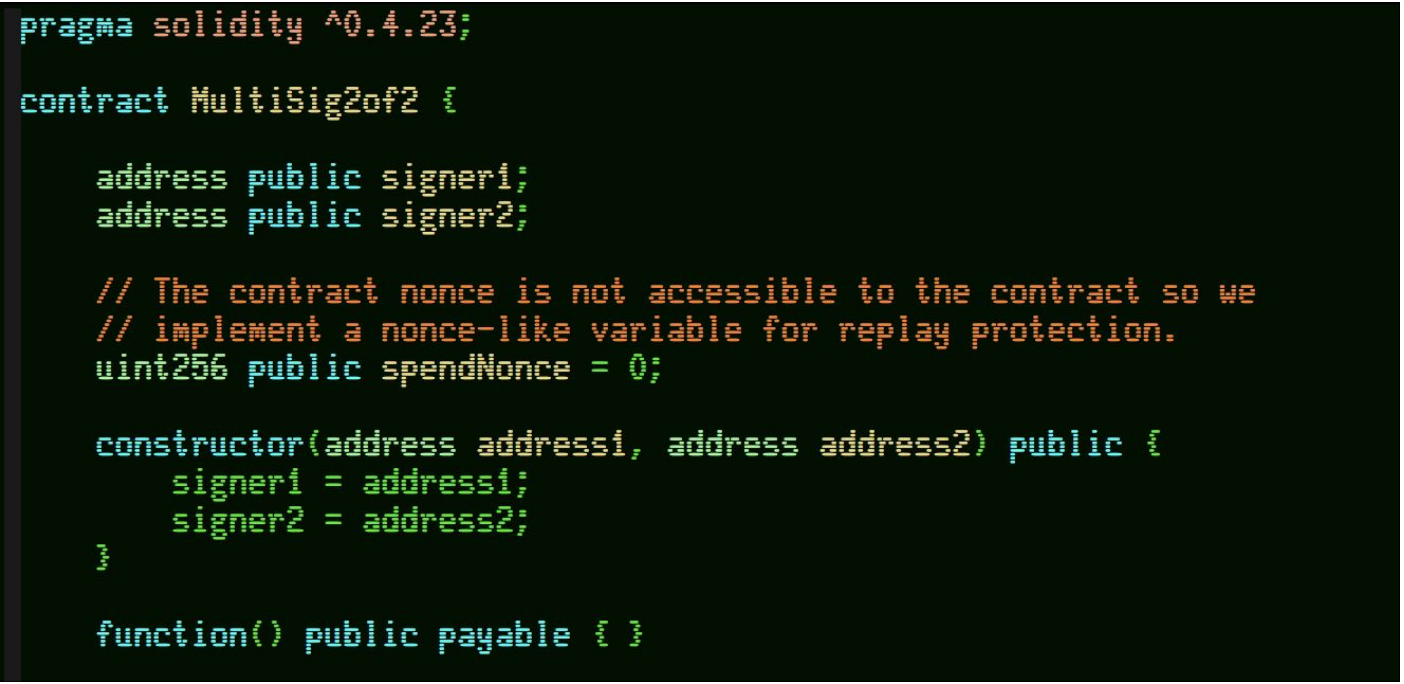
Since the signature can come from a different private key than the message sender, anytime you use ecrecover, you’re using multisig.

You can be more explicit about it and accept multiple signatures in the same function call.

Keep thinking about replay. - If you don’t want the function call to be repeatable, you may need to include a nonce.

For audited, in-production multisig contract, check out: https://github.com/unchained-capital/ethereum-multisig

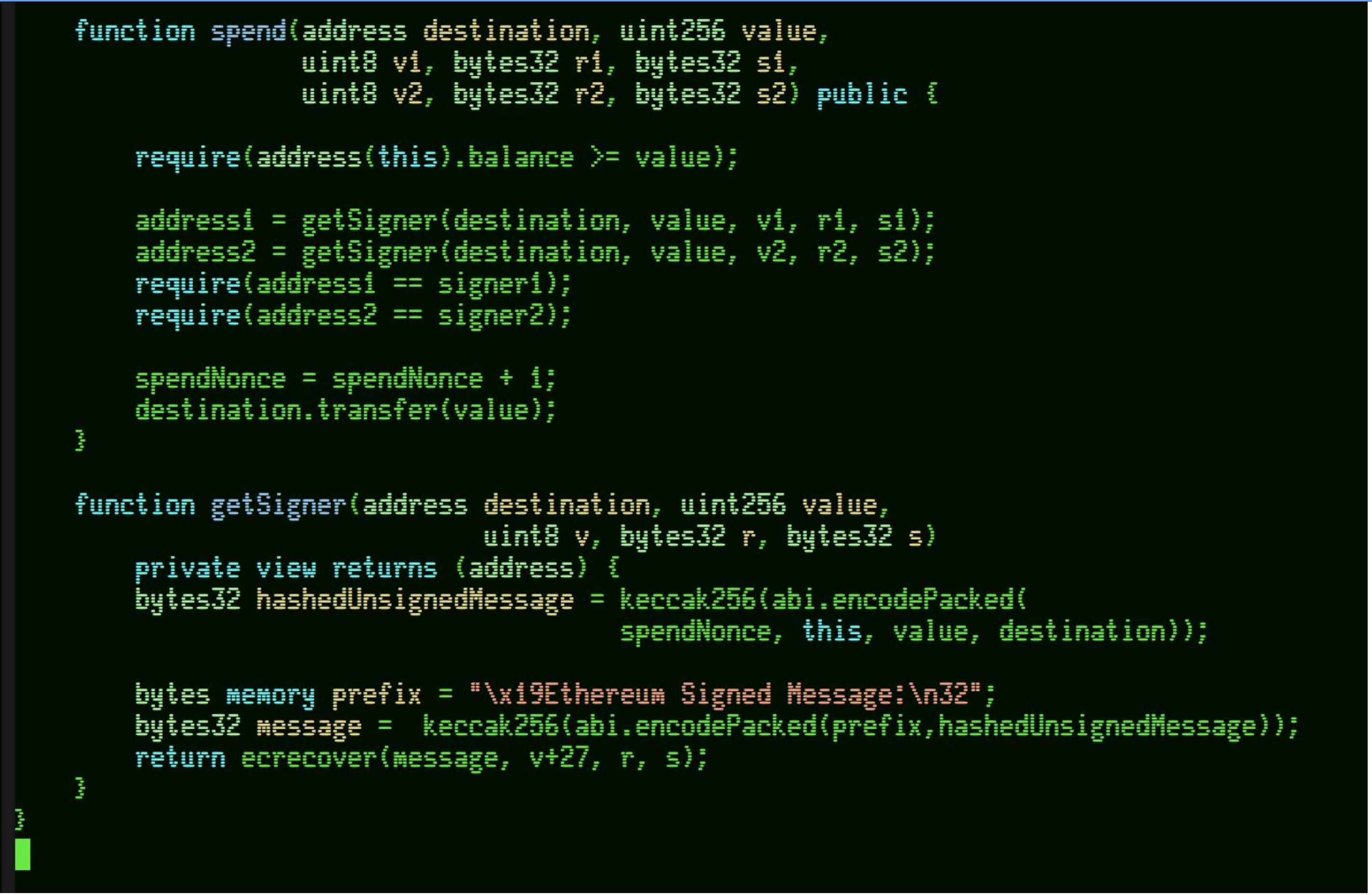
Multisig constructor



```
pragma solidity ^0.4.23;

contract MultiSig2of2 {

    address public signer1;
    address public signer2;

    // The contract nonce is not accessible to the contract so we
    // implement a nonce-like variable for replay protection.
    uint256 public spendNonce = 0;

    constructor(address address1, address address2) public {
        signer1 = address1;
        signer2 = address2;
    }

    function() public payable { }
```

- boring setup
- we have to use our own nonce because the built-in contract nonce is not available.
- use a payable fallback so the contract can accept ETH

Multisig code - spending



```
    function spend(address destination, uint256 value,
                   uint8 v1, bytes32 r1, bytes32 s1,
                   uint8 v2, bytes32 r2, bytes32 s2) public {

        require(address(this).balance >= value);

        address1 = getSigner(destination, value, v1, r1, s1);
        address2 = getSigner(destination, value, v2, r2, s2);
        require(address1 == signer1);
        require(address2 == signer2);

        spendNonce = spendNonce + 1;
        destination.transfer(value);
    }

    function getSigner(address destination, uint256 value,
                               uint8 v, bytes32 r, bytes32 s)
        private view returns (address) {
        bytes32 hashedUnsignedMessage = keccak256(abi.encodePacked(
                                        spendNonce, this, value, destination));

        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 message =  keccak256(abi.encodePacked(prefix,hashedUnsignedMessage));
        return ecrecover(message, v+27, r, s);
    }
}
```

- have to submit enough data to reconstruct the unsigned message
- notice the nonce increments and is **NOT** an argument
- instead of including a long message, the hash of the combined fields is signed. This keeps the message to a predictable 32 bytes.
-nonce, this, the value, and the destination are all included in the message
-the message would need to be constructed for signing (or have a function here)

Let’s talk about nonce.

In the last example, we had a nonce that increased once per use.

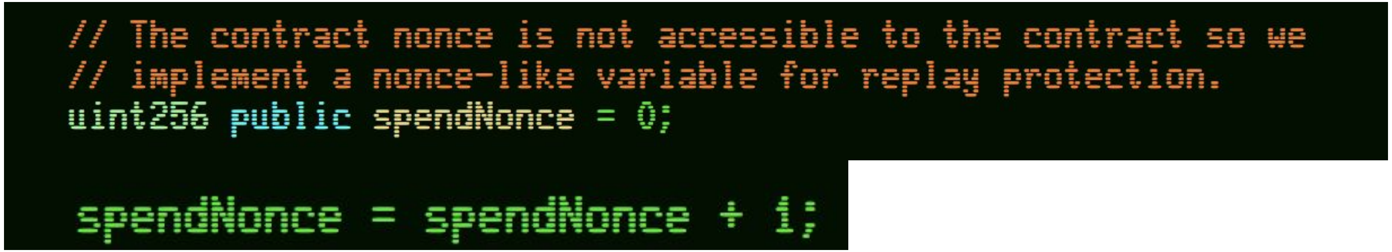
That means if you have multiple transactions you want to broadcast, they have to be submitted in order.

This is true for standard ethereum accounts if you’re signing with an air-gapped computer.

By using a mapping to store used nonces, we can get around ordered transactions! At the cost of storage.

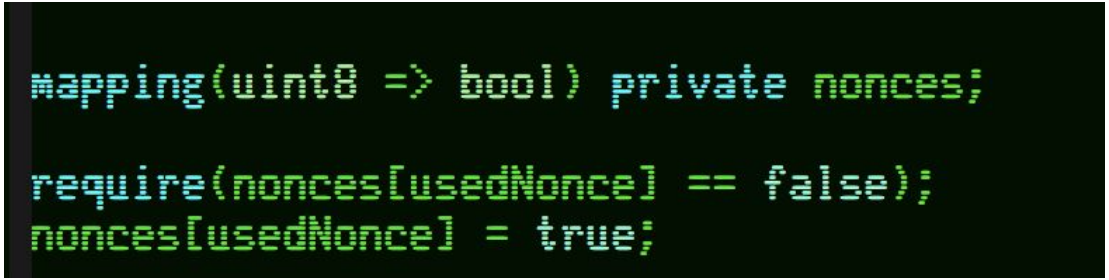
Nonces can be programmatic, or you can track them

Ordered nonce:



```
// The contract nonce is not accessible to the contract so we
// implement a nonce-like variable for replay protection.
uint256 public spendNonce = 0;

 spendNonce = spendNonce + 1;
```

Mapping nonce:



```
mapping(uint8 => bool) private nonces;

require(nonces[usedNonce] == false);
nonces[usedNonce] = true;
```

Example: submitting signatures to a central hot-wallet

What if you want to have many private keys (accounts, users, etc.) but don’t want to have to fund many accounts just to pay for gas for each transaction coming from each account?

And/or what if you want to allow users to write data/change state without them having to pay for the gas?

TimeClock example

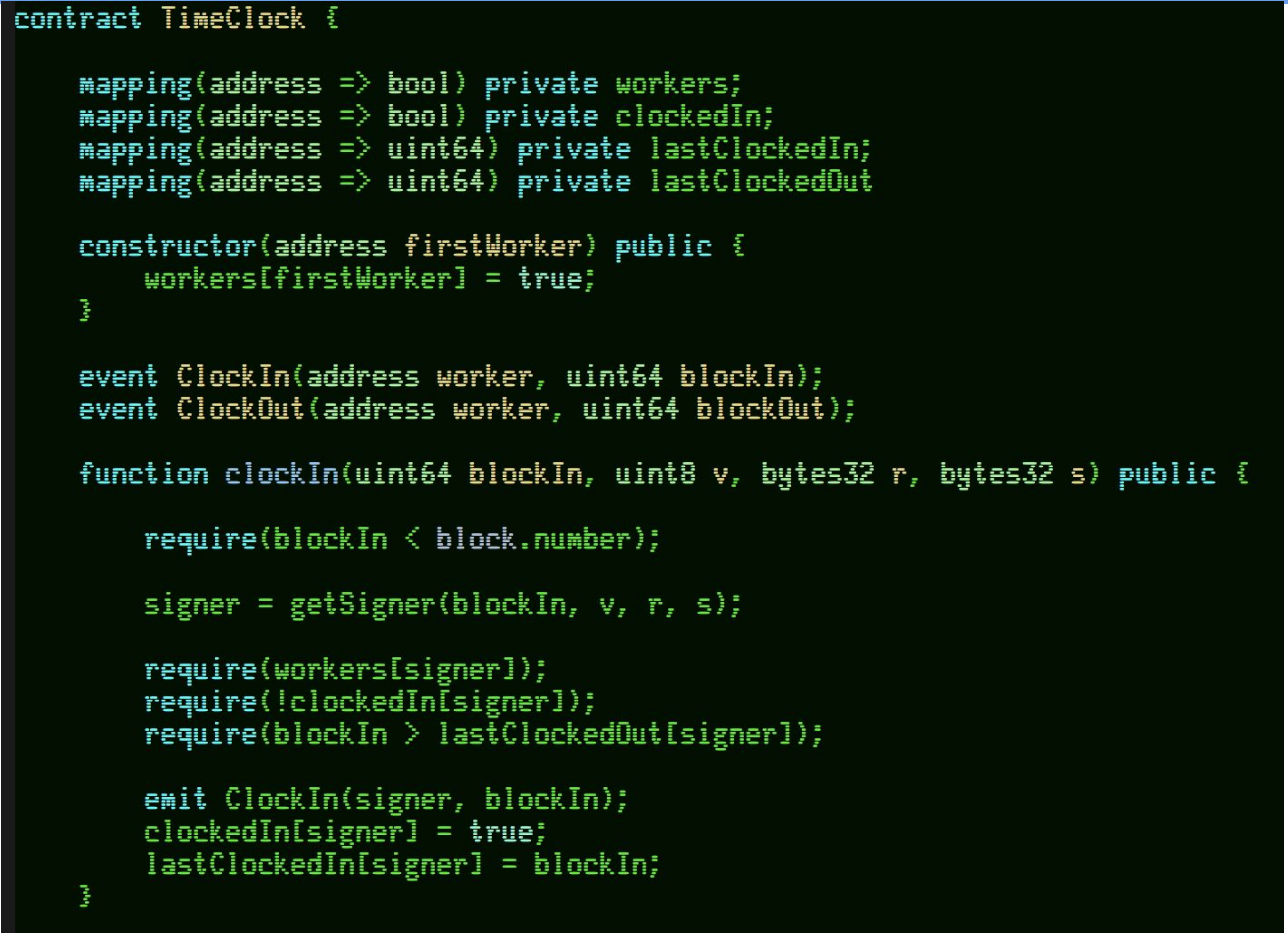
You have a bunch of employees and want to use a contract to track clock-in clock-out times.

Each employee could have their own wallet and send transactions to you contract, OR you could have them submit a valid signature to a hot wallet you keep funded to pay for fees.

To save yourself from wasting gas, you could confirm the validity of the signature before attempting to broadcast.

The hot wallet has no privileges, nothing centralized to hack.

TimeClock Code



```
contract TimeClock {

    mapping(address => bool) private workers;
    mapping(address => bool) private clockedIn;
    mapping(address => uint64) private lastClockedIn;
    mapping(address => uint64) private lastClockedOut

    constructor(address firstWorker) public {
        workers[firstWorker] = true;
    }

    event ClockIn(address worker, uint64 blockIn);
    event ClockOut(address worker, uint64 blockOut);

    function clockIn(uint64 blockIn, uint8 v, bytes32 r, bytes32 s) public {

        require(blockIn < block.number);

        signer = getSigner(blockIn, v, r, s);

        require(workers[signer]);
        require(!clockedIn[signer]);
        require(blockIn > lastClockedOut[signer]);

        emit ClockIn(signer, blockIn);
        clockedIn[signer] = true;
        lastClockedIn[signer] = blockIn;
    }
```

- the block number acts like a nonce and is restricted to a range of values
- events are used to record clock-in and clock-out (could use block.number instead)
- any address can call clockIn
- the worker’s address doesn’t need to be funded since it’s not paying for gas
-if you want to prevent pre-signing

Flip it - users broadcast provided signatures (registration?)

What if you want to control access to a contract, but you don't want to pay for the gas to change it's state?

This is the opposite of the last example, but again ecrecover can help us!

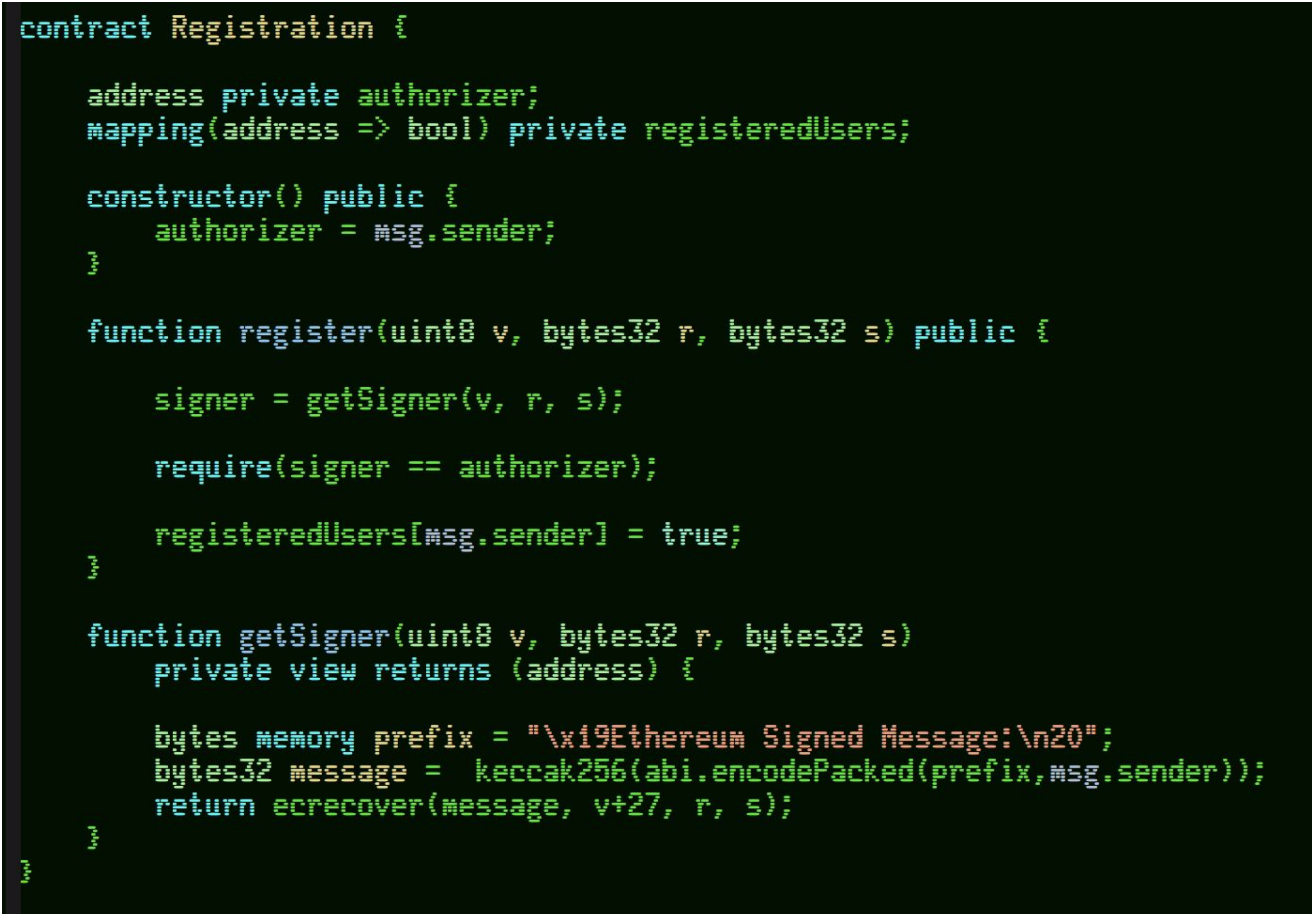
Registration example

You have a contract with a whitelist - only registered users can interact with the contract.

To register, a user has to log into your website, KYC/AML, and submit an address.

You can provide them a single-use signature, only usable by their registered address, that they can use to interact with the contract.

Registration Code



```
contract Registration {

    address private authorizer;
    mapping(address => bool) private registeredUsers;

    constructor() public {
        authorizer = msg.sender;
    }

    function register(uint8 v, bytes32 r, bytes32 s) public {

        signer = getSigner(v, r, s);

        require(signer == authorizer);

        registeredUsers[msg.sender] = true;
    }

    function getSigner(uint8 v, bytes32 r, bytes32 s)
        private view returns (address) {

        bytes memory prefix = "\x19Ethereum Signed Message:\n20";
        bytes32 message =  keccak256(abi.encodePacked(prefix,msg.sender));
        return ecrecover(message, v+27, r, s);
    }
}
```

- no nonce since registration only goes one-way (revoking would make this more complicated)
- the signature must be sent from the correct address to be valid
- the authorizer doesn’t spend any gas to register an address, the address being registered does

Naive Oracle - instead of single-use individual codes, Broadcast signed data for anyone to use

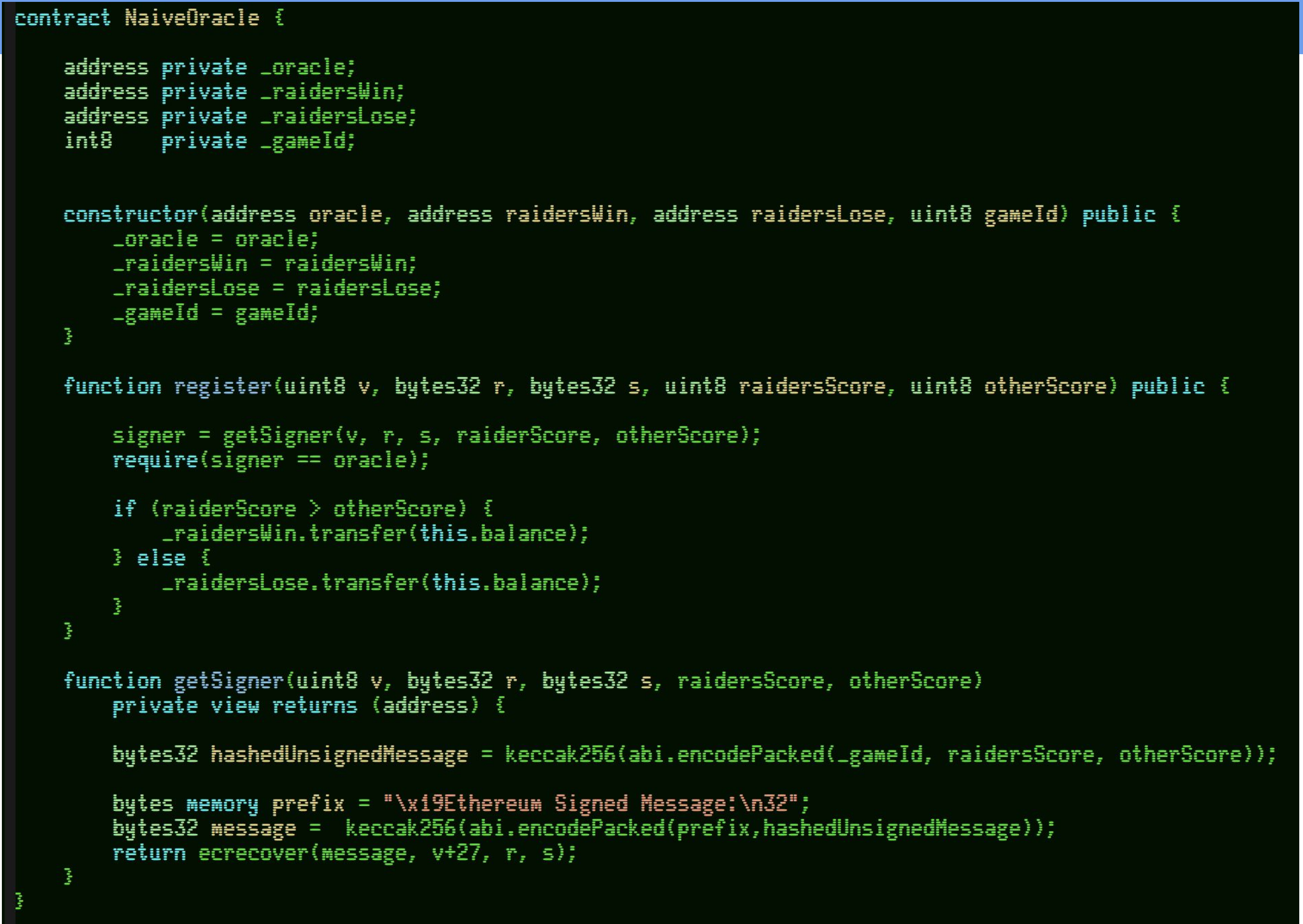
Very similar to the last example.

You can broadcast signatures of anything, example: sports scores.
You could publish the public key and other people could set up their own contracts that depend on the signature you'll provide.

You publish the signature off-chain, costing you nothing, and people can use that as an input to their contracts.

This is a high-trust oracle, but simple.

NaiveOracle code



```
contract NaiveOracle {

    address private _oracle;
    address private _raidersWin;
    address private _raidersLose;
    int8    private _gameId;


    constructor(address oracle, address raidersWin, address raidersLose, uint8 gameId) public {
        _oracle = oracle;
        _raidersWin = raidersWin;
        _raidersLose = raidersLose;
        _gameId = gameId;
    }

    function register(uint8 v, bytes32 r, bytes32 s, uint8 raidersScore, uint8 otherScore) public {

        signer = getSigner(v, r, s, raiderScore, otherScore);
        require(signer == oracle);

        if (raiderScore > otherScore) {
            _raidersWin.transfer(this.balance);
        } else {
            _raidersLose.transfer(this.balance);
        }
    }

    function getSigner(uint8 v, bytes32 r, bytes32 s, raidersScore, otherScore)
        private view returns (address) {

        bytes32 hashedUnsignedMessage = keccak256(abi.encodePacked(_gameId, raidersScore, otherScore));

        bytes memory prefix = "\x19Ethereum Signed Message:\n32";
        bytes32 message =  keccak256(abi.encodePacked(prefix,hashedUnsignedMessage));
        return ecrecover(message, v+27, r, s);
    }
}
```

- no nonce since sig is used once, but using **`gameId`** so that the sig isn’t re-usable for other games.

- The oracle address doesn’t pay any fees

- high-trust in the oracle

Ecrecover in production

At Unchained Capital, we use a hardware-wallet compatible multisig contract that uses ecrecover for signature verification.

Our Requirements:

1. Anyone can deposit
2. N-of-M signatures required for spending/withdrawal
3. The private keys for the M signatures are controlled by Hardware Wallets
4. Contract can be created without access to hardware wallet
5. Private keys are not reused.
6. ‘Hot’ wallets have no privileges
7. Asynchronous Signing
8. Simplest code possible

https://github.com/unchained-capital/ethereum-multisig

Let’s wrap up

Ecrecover allows you to:

- Verify data was approved by the signer
- Multiple signatures in a single transaction
- Asynchronous signing
- Have a different account submit the transaction and pay for the gas
- Separates privilege from broadcast

Pros:

- Lots of use-cases
- Single-tx multisig
- Unfunded Address signing
- Unordered signing

Cons:

- More Code
- Wallet-Specific Signing
- Nonce / Replay Protection
- Contracts can’t create signatures

Thank You! - Questions?

- Promises from Asymmetric Crypto
- Basic Example - use in a contract, and web3 signature generation
- Signing Mechanics
    - Trezor, Ledger, and Metamask examples
- Example Usage
    - Kill-Switch, ColdOwner, Multisig, TimeClock, Registration, NaiveOracle
- Ecrecover in production
    - multisig custody @ unchained capital