Lesson 14

There's a bit of a trend toward writing "clever" Solidity code and it's not a good one. Boring code is better code.

Quote

The end of Stack too deep errors ?
Version 0.8.13

EIP712 and EIP2612

EIP712 a standard for signing transactions and ensuring that they are securely used

DOMAIN SEPARATOR (FOR AN ERC-20)

```
bytes32 eip712DomainHash = keccak256(
    abi.encode(
        keccak256(
            "EIP712Domain(string name,string version,
            uint256 chainId,address verifyingContract)"
        ),
        keccak256(bytes(name())), // ERC-20 Name
        keccak256(bytes("1")),    // Version
        chainid(),
        address(this)
    )
);
```

Permit Function

"Arguably one of the main reasons for the success of ERC-20 tokens lies in the interplay between `approve` and `transferFrom`, which allows for tokens to not only be transferred between externally owned accounts (EOA), but to be used in other contracts under application specific conditions by abstracting away `msg.sender` as the defining mechanism for token access control.

However, a limiting factor in this design stems from the fact that the ERC-20 `approve` function itself is defined in terms of `msg.sender`. This means that user's *initial action* involving ERC-20 tokens must be performed by an EOA .
If the user needs to interact with a smart contract, then they need to make 2 transactions

(`approve` and the smart contract call which will internally call `transferFrom`). Even in the simple use case of paying another person, they need to hold ETH to pay for transaction gas costs."

The permit function is used for any operation involving ERC-20 tokens to be paid for using the token itself, rather than using ETH.
EIP2612

EIP2612 adds the following to ERC20

```
function permit(address owner, address spender,
    uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external
function nonces(address owner) external view returns (uint)
function DOMAIN_SEPARATOR() external view returns (bytes32)
```

A call to `permit(owner, spender, value, deadline, v, r, s)` will set `approval[owner][spender]` to `value`,
increment `nonces[owner]` by 1, and
emit a corresponding `Approval` event,
if and only if the following conditions are met:

- The current blocktime is less than or equal to `deadline`.
- `owner` is not the zero address.
- `nonces[owner]` (before the state update) is equal to `nonce`.
- `r`, `s` and `v` is a valid `secp256k1` signature from `owner` of the message:

If any of these conditions are not met, the `permit` call must revert.

TRADITIONAL PROCESS = APPROVE + TRANSFERFROM

The user sends a transaction which will approve tokens to be used via the UI.
The user pays the gas fee for this transaction
The user submits a second transaction and pays gas again.

PERMIT PROCESS

User signs the signature — via Permit message which will sign the approve function.
User submits signature. This signature does not require any gas — transaction fee.
User submits transaction for which the user pays gas, this transaction sends tokens.

Original introduced by Maker Dao

IN MORE DETAIL

Taken from permit article

1. Our contract needs a domain hash as above and to keep track of nonces for addresses

2. We need a permit struct

```
bytes32 hashStruct = keccak256(
    abi.encode(
        keccak256("Permit(address owner,address spender,
        uint256 value,uint256 nonce,uint256 deadline)"),
        owner,
        spender,
        amount,
        nonces[owner],
        deadline
    )
);
```

This struct will ensure that the signature can only used for

the permit function
to approve from owner
to approve for spender
to approve the given value
only valid before the given deadline
only valid for the given nonce
The nonce ensures someone can not replay a signature, i.e., use it multiple times on the same contract.

We can then put these together

```
bytes32 hash = keccak256(
    abi.encodePacked(uint16(0x1901), eip712DomainHash, hashStruct)
);
```

On receiving the signature we can verify with

```
address signer = ecrecover(hash, v, r, s);
require(signer == owner, "ERC20Permit: invalid signature");
require(signer != address(0), "ECDSA: invalid signature");
```

We can then increase the nonce and perform the approve

```
nonces[owner]++;
_approve(owner, spender, amount);
```

Example from Solmate

Draft Example from Open Zeppelin

```
abstract contract ERC20Permit is ERC20, IERC20Permit, EIP712 {
    using Counters for Counters.Counter;

    mapping(address => Counters.Counter) private _nonces;

    // solhint-disable-next-line var-name-mixedcase
    bytes32 private immutable _PERMIT_TYPEHASH =
        keccak256("Permit(address owner,address spender,uint256 value,
        uint256 nonce,uint256 deadline)");

    /**
     * @dev Initializes the {EIP712} domain separator using the `name`
parameter,
     * and setting `version` to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined
     * as the ERC20 token name.
     */
    constructor(string memory name) EIP712(name, "1") {}

    /**
     * @dev See {IERC20Permit-permit}.
     */
    function permit(
        address owner,
        address spender,
        uint256 value,
        uint256 deadline,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual override {
        require(block.timestamp <= deadline, "ERC20Permit: expired
deadline");

        bytes32 structHash = keccak256(abi.encode(_PERMIT_TYPEHASH, owner,
        spender, value, _useNonce(owner), deadline));

        bytes32 hash = _hashTypedDataV4(structHash);

        address signer = ECDSA.recover(hash, v, r, s);
        require(signer == owner, "ERC20Permit: invalid signature");

        _approve(owner, spender, value);
    }
```

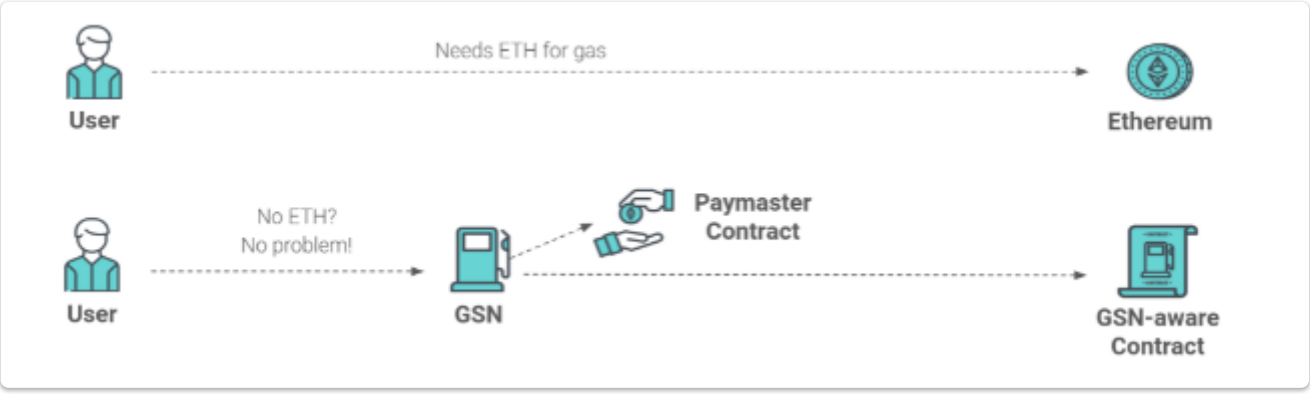
This is a step towards gassless transactions.

Meta Transactions

See Open Zeppelin meta transactions

Gas Station Network

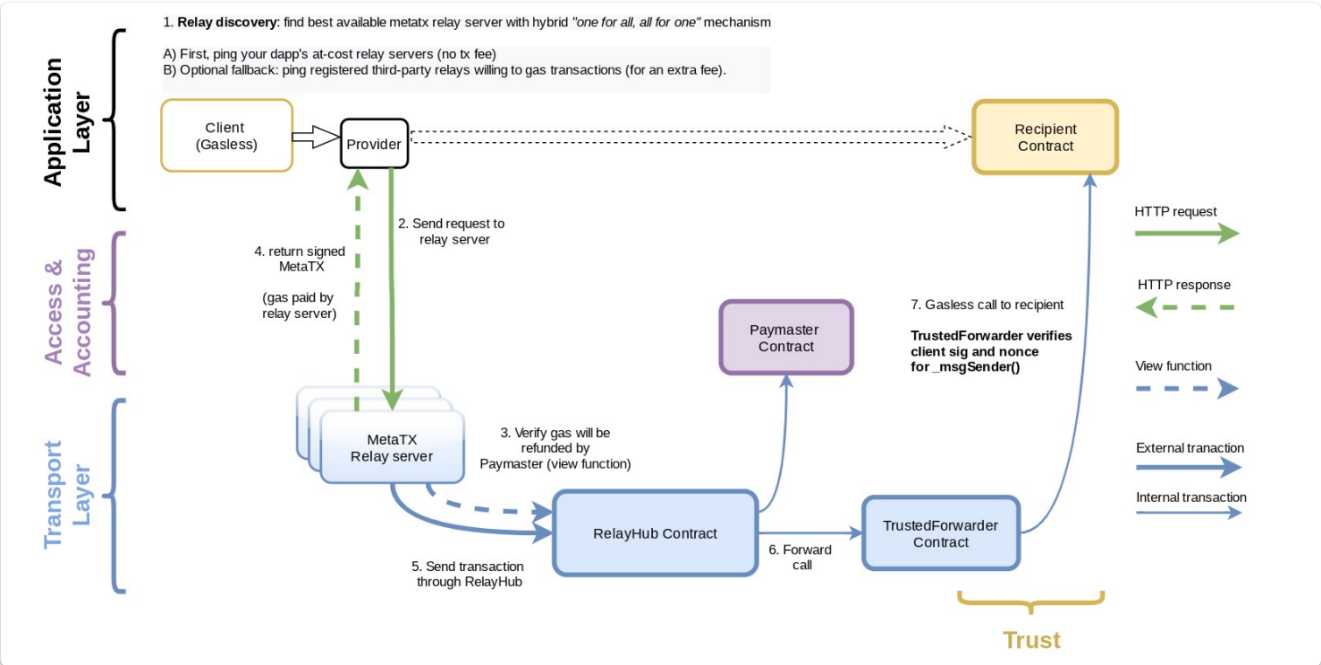
GSN1 is deprected in favour of Open GSN
We will follow the tutorial from them



USE CASES

- Pay gas in any token: Allow users to pay for gas in any token
- Pay gas in fiat: Allow users to pay for gas in fiat without having to go through KYC
- Privacy: Enabling ETH-less withdrawal of tokens sent to stealth addresses
- Onboarding: Allow dapps to subsidize the onboarding process for new users

ARCHITECTURE



- Client: signs & sends meta transaction to relay server

- Relay servers - provided by projects to give redundancy
- Paymaster: agrees to refund relay server for gas fees
  - The paymaster will have logic to decide whether to fund the transaction or not
  - If the paymaster decides to fund the transaction it pays for the transaction with its own ETH
- Trusted Forwarder: verifies sender signature and nonce
  - A simple security contract that verifies the accound details of the client
- Recipient contract: sees original sender (as _msgsender())
  - This contract will inherit from BaseRelayRecipient
- RelayHub: connecting participants trustlessly

Useful Libraries

Useful Open Source Collections

Open Zeppelin Token Contracts

Even though the concept of a token is simple, they have a variety of complexities in the implementation. Because everything in Ethereum is just a smart contract, and there are no rules about what smart contracts have to do, the community has developed a variety of standards (called EIPs or ERCs) for documenting how a contract can interoperate with other contracts.

- ERC20: the most widespread token standard for fungible assets, albeit somewhat limited by its simplicity.
- ERC721: the de-facto solution for non-fungible tokens, often used for collectibles and games.
- ERC777: a richer standard for fungible tokens, enabling new use cases and building on past learnings. Backwards compatible with ERC20.
- ERC1155: a novel standard for multi-tokens, allowing for a single contract to represent multiple fungible and non-fungible tokens, along with batched operations for increased gas efficiency.

> **Safe Functions:**
>
> Safe functions (SafeTransferFrom etc.) were introduced to prevent tokens being sent to contracts that didn't know how to handle them, and thus becoming stuck in the contract.

Open Zeppelin Access Control / Security Contracts

- Ownable

- AccessControl
- TimeLockController
- Pausable
- Reentrancy Guard
- PullPayment

Open Zeppelin Governance Contracts

Implements on-chain voting protocols similar to Compound's Governor Alpha & Bravo

Open Zeppelin Cryptography Contracts

- ECDSA contract for checking signatures.
- MerkleProof for proving an item is in a Merkle tree.

Open Zeppelin Introspection Contracts

Contracts to allow runtime checks whether a target contract implements a particular interface

Open Zeppelin Maths Contracts

SafeMath - to prevent under / overflow etc. Some of this functionality is part of Solidity since version 0.8.0

Open Zeppelin Payment Contracts

- Payment splitter
- Escrow

Open Zeppelin Collections Contracts

- Enumerable Set
- Enumerable Map

Open Zeppelin Miscellaneous Contracts

- Address
- Multicall

Open Zeppelin Upgradability Contracts

- Proxy

MATHS

Solmate
Prb-Math

WadMath
ABDK
Example use : https://ortiz.sh/blockchain/2021/05/08/QMATH.html

Useful series about maths in Solidity

Recent Exploits

Function Selectors recap

The first four bytes of the call data for a function call specifies the function to be called.

The compiler creates something like

```
method_id = first 4 bytes of msg.data
if method_id == 0x25d8dcf2 jump to 0x11
if method_id == 0xaabbccdd jump to 0x22
if method_id == 0xffaaccee jump to 0x33
other code
0x11:
code for function with method id 0x25d8dcf2
0x22:
code for another function
0x33:
code for another function
```

Encoding the function signatures and parameters

Example

```
pragma solidity ^0.8.0;

contract MyContract {

  Foo otherContract;


  function callOtherContract() public view returns (bool){
      bool answer = otherContract.baz(69,true);
      return answer;
  }
}


contract Foo {
    function bar(bytes3[2] memory) public pure {}
```

```
    function baz(uint32 x, bool y) public pure returns (bool r) {
    r = x > 32 || y;
    }
    function sam(bytes memory, bool, uint[] memory) public pure {}
}
```

The way the call is actually made involves encoding the function selector and parameters

If we wanted to call ***baz*** with the parameters **69** and **true** , we would pass 68 bytes total, which can be broken down into:

1. the Method ID. This is derived as the first 4 bytes of
   the Keccak hash of the ASCII form of the signature baz(uint32,bool).

   ```
   ***0xcdcd77c0:***
   ```

2. the first parameter, a uint32 value 69 padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000045:

3. the second parameter - boolean true, padded to 32 bytes

   0x0000000000000000000000000000000000000000000000000000000000000001:

In total

0xcdcd77c000000000000000000000000000000000000000000000000000000000000045000000000000000000000000000000000000000000000000000000000000001

This is what you see in block explorers if you look at the inputs to functions

There are helper methods to put this together for you

```
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);
```

Alternatively you can then call functions in external contracts on a low level way via

```
bytes memory payload =
abi.encodeWithSignature("baz(uint32, boolean)", 69, true);

(bool success, bytes memory returnData) =
address(contractAddress).call(payload);

require(success);
```

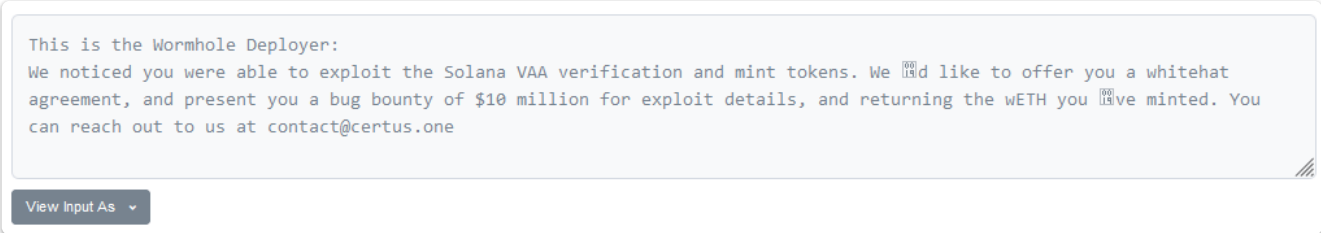
Solana's Wormhole Hack Post-Mortem Analysis

INTRODUCTION

Solana's Wormhole is a communication enabling the transfer of tokenized assets seamlessly across blockchains and benefit from Solana's high speed and low cost.

The Wormhole bridge was hacked on February 2nd, the attack exploited unpached Rust contracts in Solana that
were manipulated into crediting 120k ETH as having been deposited on Ethereum, allowing for the hacker to mint the equivalent in wrapped whETH (Wormhole ETH) on Solana.

Shortly after the hack, an on-chain message was sent to the hacker from Certus One, the team behind the Wormhole bridge:



However the hacker didn't make any contact, instead 93,750 ETH was bridged back to Ethereum over the course of 3 transactions, where it still remains in the hacker's wallet. The remaining ~36k whETH were liquidated on Solana into USDC and SOL.

Jump Crypto, the cryptocurrency fund which has raised more than $700 million in capital, tweeted Thursday afternoon that it had "replaced" 120,000 stolen Ethereum-based tokens "to make community members whole" and support Wormhole but provided no further details about the bailout. Wormhole confirmed on Twitter that funds involved in the hack had indeed been "restored" and wrote "all funds are safe" on Telegram.

Prior to the exploit, the bridge held a 1:1 ratio of ethereum to wrapped ethereum on the solana blockchain, acting essentially as an escrow service. This exploit broke the 1:1 peg by stealing 120k ETH from the collateral reserve. Solscan (a blockchain explorer for Solana), shows that ETH wrapped by Wormhole comprises roughly 19% of the total ETH on the blockchain; if the 1:1 backing of wETH hadn't been replenished swiftly, it could have triggered a situation where DeFi positions may become undercollateralized and potentially fuel a bank run. Such a situation would add to Solana's well-documented difficulties after it has suffered from multiple downtimes due to DoS attacks on the network over the last few months.

HOW IT HAPPENED

Wormhole is a "bridge", basically a way to move crypto assets between different blockchains. Specifically, Wormhole has a set of "guardians" that sign off on transfers between chains.

Bridges like Wormhole work by having two smart contracts — one on each chain. In this case, there was one smart contract on Solana and one on Ethereum. A bridge like Wormhole takes an ethereum token, locks it into a contract on one chain, and then on the chain at the other side of the bridge, it issues a parallel token. The attacker submitted transaction showed that it contained valid signatures from the guardians.

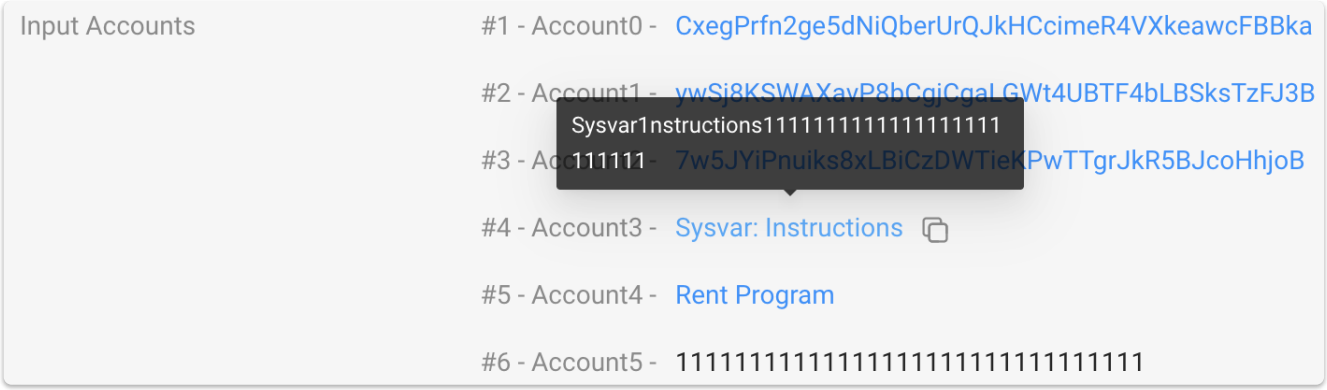
So how did the attacker mint 120,000 ETH out of thin air? The answer can be found on Wormhole's Github, specifically the complete_wrapped function, which gets triggered whenever someone minted Wormhole ETH on Solana. One of the parameters that this function takes is a `transfer_message`, basically a message signed by the guardians that says which token to mint and how much. This `transfer_message` is actually a contract on Solana, and is created by triggering a function called `post_vaa`, which checks if the message is valid by checking the signatures from the guardians.

> *all guardians perform the same computation upon observing an on-chain event, and sign a so-called Validator Action Approval (VAA). If a 2/3+ majority of all guardian nodes have observed and signed the same event using their individual keys, then it is automatically considered valid by all Wormhole contracts on all chains and triggers a mint/burn. — Leo from Certus One*

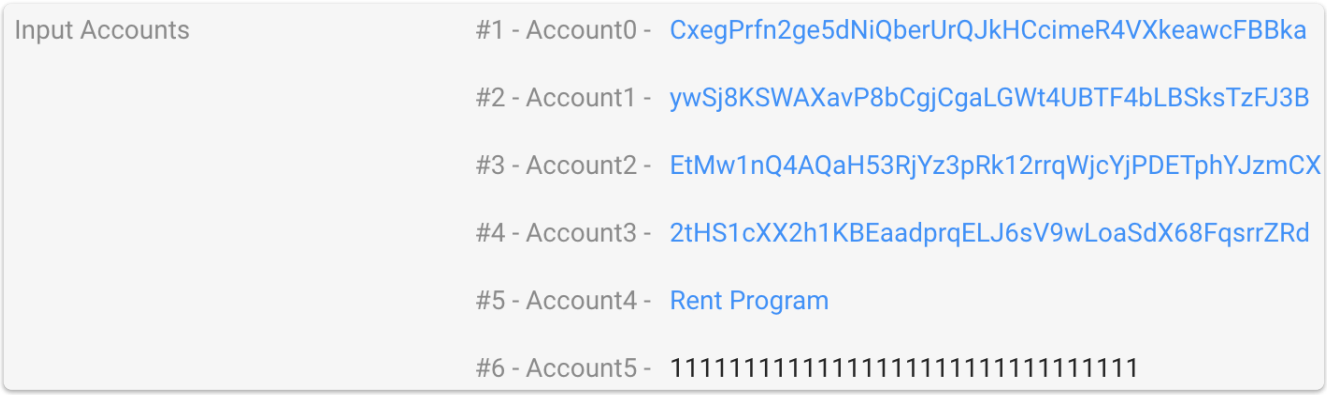
`post_vaa` doesn't actually check the signatures, however, instead, there's another smart contract which gets created by calling the verify_signatures function. One of the inputs to the `verify_signatures` function is a Solana built-in system program which contains various utilities the contract can use. So the signature verification was outsourced to this program, which was where the bug lied.

The Wormhole contracts used the function load_instruction_at to check that the Secp256k1 function was called first. By looking at Github's internal commits, the load_instruction_at function was deprecated on January 13th by the team as it did not check that the signature verification was baing performed by a whitelisted address a.k.a. "system address"!
You're supposed to provide the system address as the program you're executing here (it's the third-to-last program input), but here's the `verify_signatures` transaction for the fake deposit of 120k ETH; the system address was substituted for a program's address (the equivalent of an Ethereum smart contract) that didn't check signatures at all.

The below screenshot was taken from a regular transaction on Solana, but the following one is from the hacker's transaction; notice input parameter #4, in the latter image it contains the hacker's supplied program address rather than "Sysvar: Instructions", which is the alias for the program that's supposed to check signatures.



This is what a regular verify_signature transaction should look like



During the execution of the verify_signatures method, the hacker managed to substitute the "Sysvar: Instructions" address with his own supplied address

Using an account created hours earlier with a single serialised instruction corresponding to the Sep256k1 contract, the attacker was able to fake the 'SignatureSet', call 'complete_wrapped' and fraudulently mint 120k whETH on Solana using VAA verification that had been created in a previous transaction.

WHAT WENT WRONG?

A commit made on January 13th and pushed to Github on February 2nd replacing usage of `load_instruction_at` with `load_instruction_at_checked`, which actually confirms that the program being executed is the system program. What is interesting however is that the exploit came a few hours *after* this commit as team didn't have time to deploy the new version yet. It is possible that an attacker was keeping an eye on the repository and looking out for suspicious commits.

Roger Grimes, a data-driven defense evangelist for KnowBe4, said that the programming error Wormhole made was rather common, but serious nonetheless:

> *"The function inside of the multiple nested smart contracts which was supposed to verify the signature was not coded to ensure the integrity check actually happened. So, there was no integrity guaranteed in the integrity check."*

Analysis of these transactions has determined that the exploit resulted from Wormhole's failure to validate "guardian" accounts:

> *Wormhole didn't properly validate all input accounts, which allowed the attacker to spoof guardian signatures and mint 120,000 ETH on Solana*

At this point, the full extent of this attack "still remains to be seen," CertiK said. It could turn out to be a precursor to other attacks, the firm suggested, if, for example, Wormhole's bridge to a different cryptocurrency – the Terra blockchain – shares the same vulnerability as its Solana bridge.

CLOSING THOUGHTS

Given the seriousness of this incident, along with very recent Multichain exploits such as Meter on February 6th, Qubit on January 28th, the week before that $3 million was drained by multiple hackers attacking cross-chain router protocol, and of course last summer's mammoth attack on Poly Network.

Vitalik Buterin seems to have been proven right about his recent security concerns around cross-chain protocols, as he argued in a Reddit post that bridges will not be popular in the future, due to "fundamental limits to the security of bridges that hop across multiple 'zones of sovereignty'", i.e. risks to the backing of bridged assets. Vitalik also noted that "it's always safer to hold Ethereum-native assets on Ethereum or Solana-native assets on Solana than it is to hold Ethereum-native assets on Solana or Solana-native assets on Ethereum."

Ronghui Gu, co-founder of CertiK, said in an interview:

> *Bridges are an attractive target for hackers: they hold millions of dollars of tokens in what is essentially an escrow contract, and by operating across multiple chains they multiply their potential points of failure.*

From Rekt:

> *It remains to be seen whether the future of DeFi will be cross-chain or 'multi-chain', but either way, the journey there will be long and dangerous. That being said, Solana is not yet the battle-hardened network that Ethereum has grown to be. Every exploit, for all the damage it does, offers a lesson for how to secure an evolving ecosystem.*

Resources

- https://rekt.news/wormhole-rekt/
- https://www.forbes.com/sites/jonathanponciano/2022/02/03/vc-backer-replaces-325-million-in-stolen-crypto-one-day-after-solanas-biggest-hack-ever/
- https://old.reddit.com/r/ethereum/comments/rwojtk/ama_we_are_the_efs_research_team_pt_7_07_january/hrngyk8/
- https://twitter.com/samczsun/status/1489044939732406275
- https://twitter.com/kelvinfichter/status/1489041221947375616
- https://twitter.com/AlexSmirnov__/status/1489044639768268803

- https://medium.com/certus-one/introducing-the-wormhole-bridge-24911b7335f7