Wormhole hack

Timotej Ponek

timotej.ponek@gmail.com

Keywords: Wormhole hack, Solana, Ethereum

WHAT IS WORMHOLE

Wormhole is a crosschain bridge that allows users to transfer their crypto assets from blockchain running under some protocol to another blockchain running under a different protocol. The mechanism of how it works can be described in these steps[4]:

- To move N tokens from blockchain A to blockchain B, the token owner first needs to lock N tokens into the bridge smart contract on blockchain A.
- After verification that N tokens have been locked, the bridge mints N equivalent tokens (or wrapped tokens) on blockchain B.
- To get back the original N tokens, the owner needs to burn N wrapped tokens on the bridge smart contract on blockchain B.
- After verification that N wrapped tokens have been burnt, the bridge will release N tokens on blockchain A and return them to the token owner.

Crosschain bridge could be likened to the old US gold certificates system. After depositing gold barse, the US bank issues gold certificates equivalent in value. The certificate is transferable and anyone can redeem it back to gold. The bank has to manage the security of gold storage and protect itself against counterfeit gold certificates.

Similar to the gold certificates system, crosschain bridges have two main security concerns:

- Make sure tokens locked inside smart contract (gold bars) are safe
- Make sure minted wrapped tokens (gold certificates) are valid

WHAT HAPPENED

On 2nd February 2022, an attacker was able to exploit Wormhole protocol and gain wrapped Ethereum tokens (wETH) on Solana blockchain without depositing the equivalent amount of Ethereum to the bridge smart contract.

MORE DETAILS

The root cause of the exploit was the usage of deprecated functions [1, 2, 3, 4, 5], that allowed attacker to use a fake account for verification. The faulty code can be viewed here.

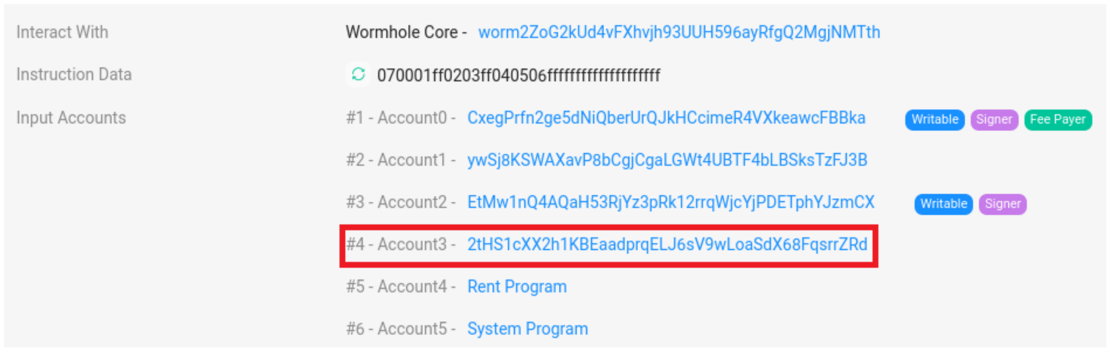
```
68  pub fn verify_signatures(
69    ctx: &ExecutionContext,
70    accs: &mut VerifySignatures,
71    data: VerifySignaturesData,
72  ) -> Result<()> {
73    accs.guardian_set
74      .verify_derivation(ctx.program_id, &(&*accs).into())?;
75    //...

91    //...
92    let current_instruction = solana_program::sysvar::instructions::load_current_index(
93      &accs.instruction_acc.try_borrow_mut_data()?,
94    );
95    if current_instruction == 0 {
96      return Err(InstructionAtWrongIndex.into());
97    }
98
99    // The previous ix must be a secp verification instruction
100   let secp_ix_index = (current_instruction - 1) as u8;
101   let secp_ix = solana_program::sysvar::instructions::load_instruction_at(
102     secp_ix_index as usize,
103     &accs.instruction_acc.try_borrow_mut_data()?,
104   )
105   .map_err(|_| ProgramError::InvalidAccountData)?;
106   //...
```

Listing 1. Usage of deprecated Solana functions in Wormhole code

In code snippet above, we can see that function *verify_signatures()* uses deprecated functions *load_current_index()* (line 92) and *load_instruction_at()* (line 101). Prior to the hack, the attacker must have found about the existance of these functions and their weaknesses, and figure out how they could be used to create the exploit.

A few hour before the hack happend, the attacker created an account on the Ethereum network holding just 0.1 ETH, that was later used in the exploit. To this account, he received 0.94 ETH from Tornado Cash, a privacy-focused Ethereum mixer, to facilitate the payment of transaction fees associated with their subsequent actions.

The exploit was condutected in following steps: At first, the attacker created a malicious account with fake instruction data. Then he invoked the *verify_signatures()* function with this forged account.



Interact With	Wormhole Core - worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth	
Instruction Data	070001ff0203ff040506ffffffffffffffffffff	
Input Accounts	#1 - Account0 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka	Writable Signer Fee Payer
	#2 - Account1 - ywSj8KSWAXavP8bCgjCgaLGWt4UBTF4bLBSksTzFJ3B	
	#3 - Account2 - EtMw1nQ4AQaH53RjYz3pRk12rrqWjcYjPDETphYJzmCX	Writable Signer
	#4 - Account3 - 2tHS1cXX2h1KBEaadprqELJ6sV9wLoaSdX68FqsrrZRd	
	#5 - Account4 - Rent Program	
	#6 - Account5 - System Program	

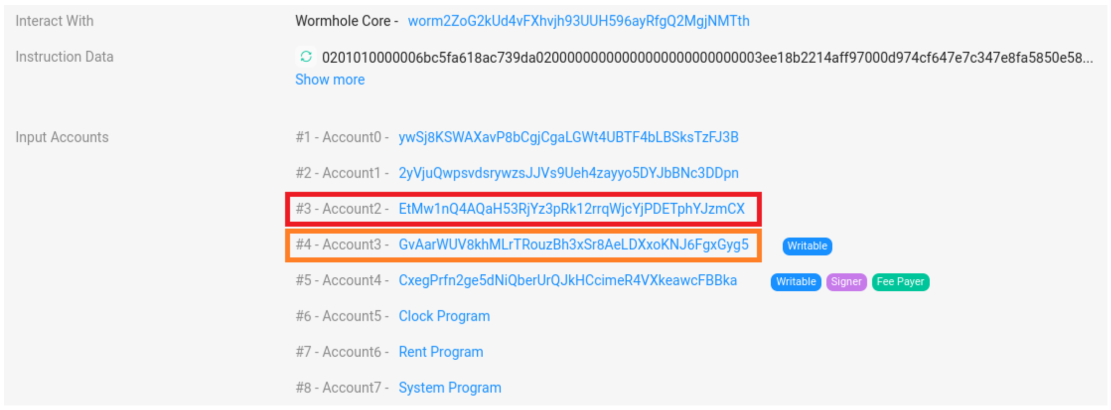
Figure 1. *verify_signatures* transaction with the malicious account (in red). If the account was valid, we would see string *Sysvar: Instructions* in place of account address.

The *verify_signatures()* function loads the current instruction utilizing *load_current_index()* (line 92) function. To attacker's advatage, this function does not validate whether the account at *accs.instruction_acc* (sysvar account) is actually a valid solana system sysvar account. So the attacker can insert any forged account with desired instruction because the account will never be checked for its validity.

We cannot say exactly what instruction the attacker inserted, but we can say for sure it was an instruction that allowed him to pass signature validation without ever invoking secp256k1 validation.

He was able to return from the *verify_signatures()* function with a valid (but not verified by secp256k1) signature set account, and proceed further in *post_vaa()* function.

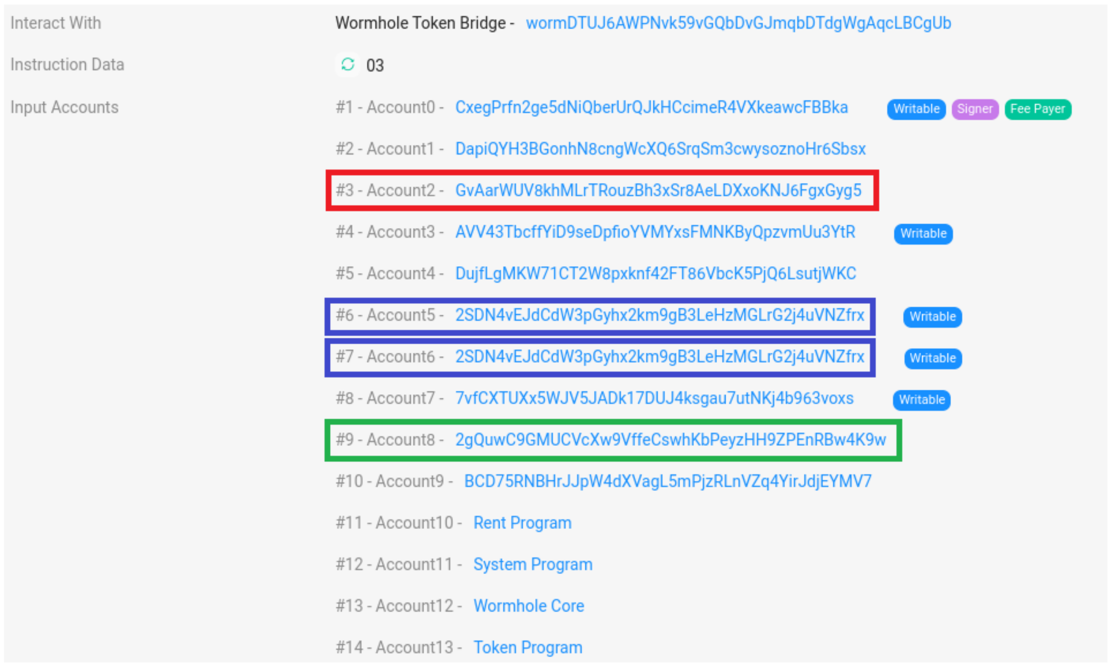
The *post_vaa()* function, not knowing it recieved unverified signatures, created the message account specifying the minting of 120,000 wETH (as the attacker desired) that was further passed to the *complete_wrapped()*.



```
Interact With       Wormhole Core - worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth
Instruction Data    0201010000006bc5fa618ac739da0200000000000000000000000000000003ee18b2214aff97000d974cf647e7c347e8fa5850e58...
                    Show more

Input Accounts      #1 - Account0 - ywSj8KSWAXavP8bCgjCgaLGWt4UBTF4bLBSksTzFJ3B
                    #2 - Account1 - 2yVjuQwpsvdsrywzsJJVs9Ueh4zayyo5DYJbBNc3DDpn
                    #3 - Account2 - EtMw1nQ4AQaH53RjYz3pRk12rrqWjcYjPDETphYJzmCX
                    #4 - Account3 - GvAarWUV8khMLrTRouzBh3xSr8AeLDXxoKNJ6FgxGyg5   Writable
                    #5 - Account4 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka   Writable  Signer  Fee Payer
                    #6 - Account5 - Clock Program
                    #7 - Account6 - Rent Program
                    #8 - Account7 - System Program
```

Figure 2. *post_vaa* transaction with unchecked but valid signatures (in red) and message account (in orange), stating 120, 000 wETH to be minted

In *complete_wrapped()* function, 120, 000 wETH tokens were minted and assigned to the attacker's account.



```
Interact With       Wormhole Token Bridge - wormDTUJ6AWPNvk59vGQbDvGJmqbDTdgWgAqcLBCgUb
Instruction Data    03
Input Accounts      #1 - Account0 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka   Writable  Signer  Fee Payer
                    #2 - Account1 - DapiQYH3BGonhN8cngWcXQ6SrqSm3cwysoznoHr6Sbsx
                    #3 - Account2 - GvAarWUV8khMLrTRouzBh3xSr8AeLDXxoKNJ6FgxGyg5
                    #4 - Account3 - AVV43TbcffYiD9seDpfioYVMYxsFMNKByQpzvmUu3YtR   Writable
                    #5 - Account4 - DujfLgMKW71CT2W8pxknf42FT86VbcK5PjQ6LsutjWKC
                    #6 - Account5 - 2SDN4vEJdCdW3pGyhx2km9gB3LeHzMGLrG2j4uVNZfrx   Writable
                    #7 - Account6 - 2SDN4vEJdCdW3pGyhx2km9gB3LeHzMGLrG2j4uVNZfrx   Writable
                    #8 - Account7 - 7vfCXTUXx5WJV5JADk17DUJ4ksgau7utNKj4b963voxs   Writable
                    #9 - Account8 - 2gQuwC9GMUCVcXw9VffeCswhKbPeyzHH9ZPEnRBw4K9w
                    #10 - Account9 - BCD75RNBHrJJpW4dXVagL5mPjzRLnVZq4YirJdjEYMV7
                    #11 - Account10 - Rent Program
                    #12 - Account11 - System Program
                    #13 - Account12 - Wormhole Core
                    #14 - Account13 - Token Program
```

Figure 3. *complete_wrapped* transaction with malevolent message account (in red). In blue is account that will recieve the newly minted wETH (which is owned by the signer of the transaction) and in green is the mint authority (that only mints wETH relying that the checks were performed before)

AFTERMATH

Following the minting of wETH, the attacker orchestrated a series of assets movements:

- $93,750$ ETH was bridged back to Ethereum within 3 transactions, and still remains in the hacker's wallet
- Remaining wETH was exchanged for $432,662.14$ SOL and 1444.16 \$

Within the day the attack happend, Wormhole team sent to the attacker's Ethereum wallet a transaction with a message offering him a whitehat agreement and 10 milion \$, if he returns the minted wETH and shares exploit details. Attacker did not respond.

On 3nd February 2022, the stolen assets were replaced by the Wormhole's parent company Jump Crypto, as the company believes in future of this project and wanted to maintain the trust in the project. The deprecated functions were also replaced with their checked versions, removing possibility of anyone using the same exploit again.

REFERENCES

[1] ackeeblockchain. 2022 solana hacks explained: Wormhole. `https://ackeeblockchain.com/blog/2022-solana-hacks-explained-wormhole/`, 1 2023.

[2] certik. Wormhole bridge exploit incident analysis. `https://www.certik.com/resources/blog/1kDYgyBcisoD2EqiBpHE5l-wormhole-bridge-exploit-incident-analysis`, 2 2022.

[3] immunebytes. Wormhole bridge exploit incident analysis. `https://www.immunebytes.com/blog/wormhole-bridge-hack-feb-2-2022-detailed-hack-analysis/`, 8 2023.

[4] G. Nguyen. \$320 million wormhole hack explained. `https://www.linkedin.com/pulse/320-million-wormhole-hack-explained-giap-nguyen/`, 2 2022.

[5] rekt. Wormhole - rekt. `https://rekt.news/wormhole-rekt/`, 2 2022.