Lesson 13 - DeFi Continued / Security

Today's topics

- Token 2022 Program
- Interoperability / L2 solutions
- Confidential Tokens
- Security

Token-2022 Program

The Token-2022 program is a program that includes the functionality of the Token Program, but adding new functionality with new instructions.

Data layout in accounts

The start of the data layout is the same for the Token-2022 and the Token Program, new fields are required for Token-2022, so these are introduced as extensions to the layout.

Extensions

See docs

Mint Account

- Transfer fees
  With Token-2022, it's possible to configure a transfer fee on a mint so that fees are assessed at the protocol level. On every transfer, some amount is withheld on the recipient account, untouchable by the recipient. These tokens can be withheld by a separate authority on the mint.
- Closing mint
  In Token-2022, it is possible to close mints

by initializing
the `MintCloseAuthority` extension before initializing the mint.

- Interest-bearing tokens
  With the Token-2022 extension model, however, we have the possibility to change how the UI amount of tokens are represented. Using
  the `InterestBearingMint` extension and
  the `amount_to_ui_amount` instruction, you can set an interest rate on your token and fetch its amount with interest at any time.
- Non-transferable tokens
  To accompany immutably owned token accounts, the `NonTransferable` mint extension allows for "soul-bound" tokens that cannot be moved to any other entity. For example, this extension is perfect for achievements that can only belong to one person or account.

Token Account

- Memo required on incoming transfers
  By enabling required memo transfers on your token account, the program enforces

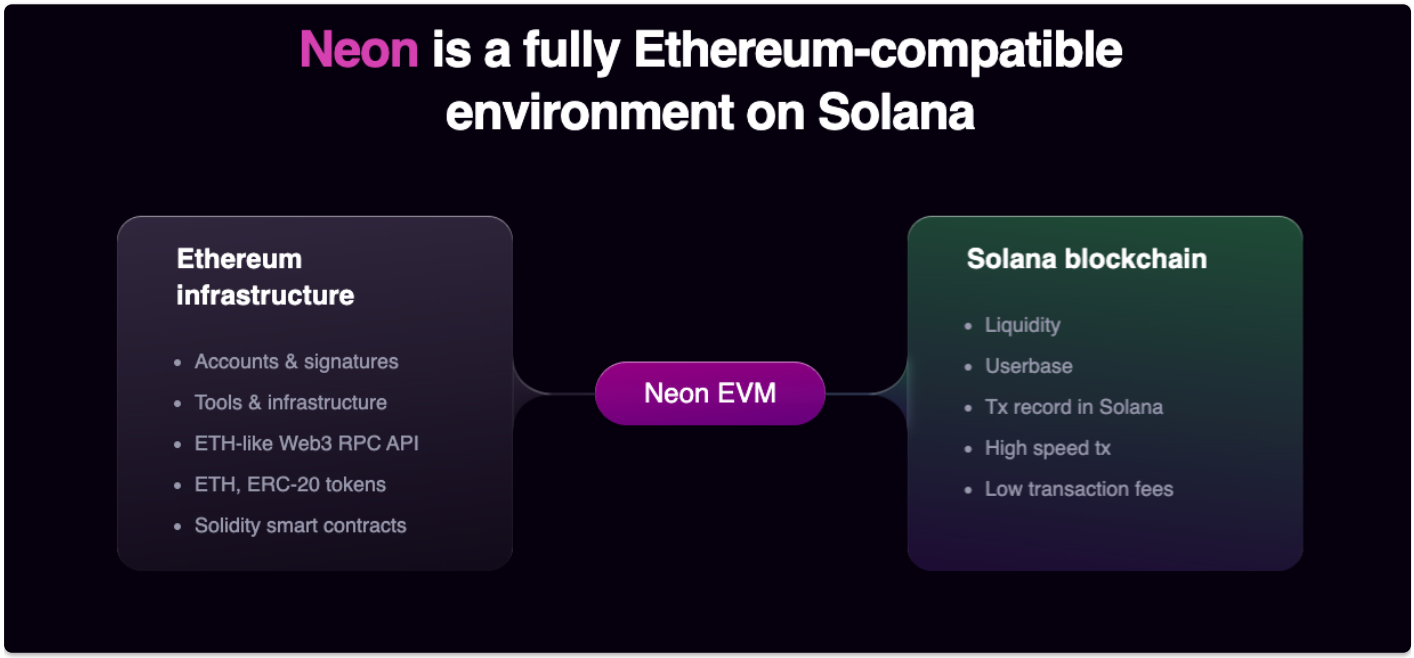
that all incoming transfers must have an accompanying memo instruction right before the transfer instruction.

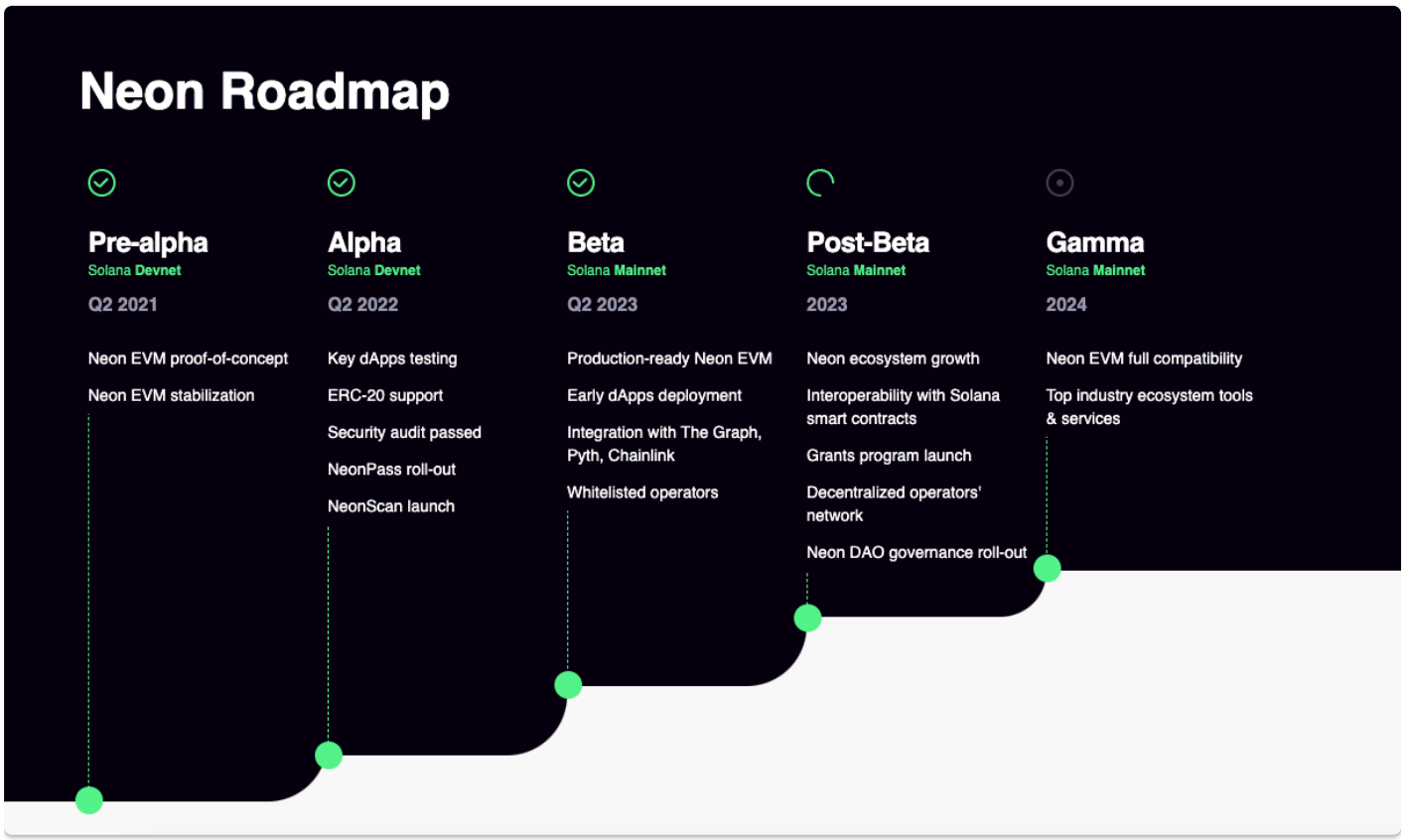
- Immutable ownership
  Token-2022 includes the `ImmutableOwner` extension, which makes it impossible to reassign ownership of an account. The Associated Token Account program always uses this extension when creating accounts.
- Default account state
  For example a mint creator may use the `DefaultAccountState` extension, which can force all new token accounts to be frozen. This way, users must eventually interact with some service to unfreeze their account and use tokens.
- Permanent Delegate
  With Token-2022, it's possible to specify a permanent account delegate for a mint. This authority has unlimited delegate privileges over any account for that mint, meaning that it can burn or transfer any amount of tokens.

L2 / Interoperability projects on Solana

Neon EVM

See Docs





EVM Features

- Familiar languages: Solidity, Vyper

- Well-known Ethereum tools: MetaMask, Hardhat, Truffle, Remix, etc.
- Ethereum RPC API compatibility
- Ethereum Accounts, Signatures, Token standards (ERC-20, ERC-721)
- No code changes required

Solana Features

- parallel execution of transactions
- low gas fee - number from site
- high transaction speed - number from site
- access to the growing Solana Ecosystem
- uses Solana consensus mechanism and state, no additional validators

Nitro SVM

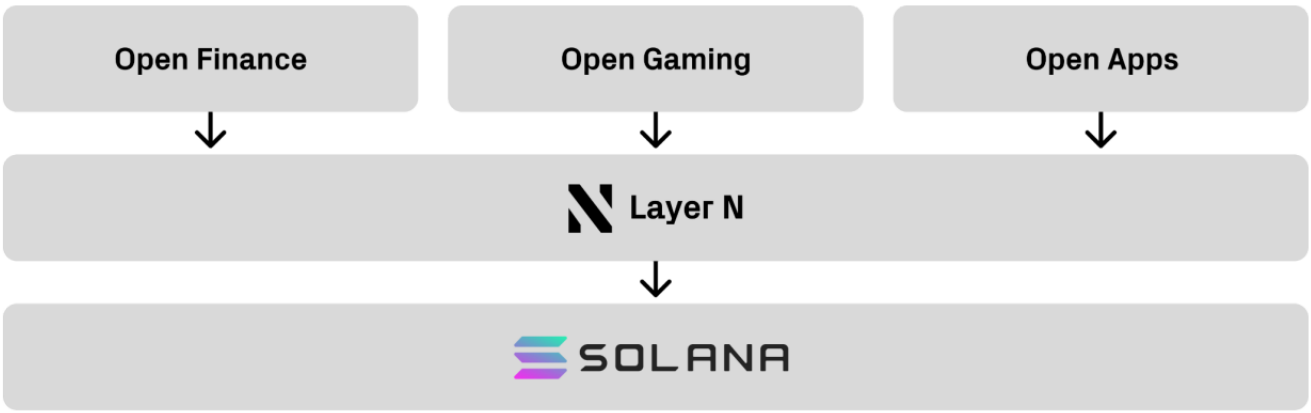
See Docs

Nitro is the first Solana VM chain, built on Cosmos as an optimistic rollup that features Sealevel Virtual Machine (SVM) compatibility. This enables developers to easily launch their Solana dApps on Cosmos and access IBC assets. Apps built on Nitro can leverage the powerful Solana VM while gaining the interoperability that is native to IBC.

Connection details see User Manual

LayerN

See Docs



Layer N: Innovations

Lightning-fast Transaction Confirmations

Optimistic confirmation scheme significantly decreases time to finality.

Integrated State Buffer

Intermediate state buffer enables parallel transaction execution, which significantly decreases latency.

Horizontal Scaling

Leveraging additional blockchains and custom data availability layers enable massive increases in throughput.

Segmented Complex Execution

Layer N enables the execution of highly intensive computation like BSM by auto-segmenting and partially executing it.

Enabling Universal Margin Interoperability

Layer N's design enables protocols to build wallet-level margining, which can maximize capital efficiency across protocols.

Multichain Wallet Support

Chain agnostic signing and multichain deposits/ withdrawals enable users to seamlessly use Layer N with any of their favorite wallets.

Cryptographic Fault Proofs

Fault proofs allow any validator to pinpoint state errors and to prevent mistakes from being made on the ledger.

Censorship Resistance

Cryptographic independence between sequencer and validator ensures MEV protection, and censorship resistance.

Decentralized Workers

Validator nodes will be able to run cron workers for decentralized applications.

Confidential Token Extension

See Docs

Tokens with Encryption and Proofs

The main state data structures that are used in the Token program are `Mint` and `Account`. The `Mint` data structure is used to store the global information for a class of tokens.

The `Account` data structure is used to store the token balance of a user.

```
/// Transfer instruction data
///
/// Accounts expected:
///   0. `[writable]` The source account.
///   1. `[writable]` The destination
account.
///   2. `[signer]` The source account's
owner.
struct Transfer {
  amount: u64,
}
```

First Approach to confidentiality is to use encryption

```
trait PKE<Message> {
  type SecretKey;
  type PublicKey;
  type Ciphertext;

  keygen() -> (SecretKey, PublicKey);
  encrypt(PublicKey, Message) ->
Ciphertext;
  decrypt(SecretKey, Ciphertext) ->
Message;
}
```

To hide the balance, we can encrypt the balance under the account owner's public key before storing it on chain.

We can similarly use encryption to hide transfer amounts in a transaction. Consider the following example of a transfer instruction. To hide the transaction amount, we can encrypt it under the

sender's public key before submitting it to the chain.

```
Transfer {
  amount: PKE::encrypt(pubkey_owner, 10),
}
```

By simply encrypting account balances and transfer amounts, we can add confidentiality to the Token program

One problem with this simple approach is that the Token program cannot deduct or add transaction amounts to accounts as they are all in encrypted form. One way to resolve this issue is to use a class of encryption schemes that are *linearly homomorphic* such as the ElGamal encryption scheme. An encryption scheme is linearly homomorphic if for any two numbers `x_0`, `x_1` and their encryptions `ct_0`, `ct_1` under the same public key, there exist ciphertext-specific add and subtract operations such that

```
let (sk, pk) = PKE::keygen();

let ct_0 = PKE::encrypt(pk, x_0);
let ct_1 = PKE::encrypt(pk, x_1);

assert_eq!(x_0 + x_1, PKE::decrypt(sk,
ct_0 + ct_1));
```

By using a linearly homomorphic encryption scheme to encrypt balances and transfer amounts, we can allow the Token program to process balances and transfer amounts in encrypted form. As linear homomorphism holds only when ciphertexts are encrypted under the same public key, we require that a transfer amount be encrypted under both the sender and receiver public keys.

Then, upon receiving a transfer instruction of this form, the token program can subtract and add ciphertexts to the source and destination accounts accordingly.

```
Account {
    mint:
Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwN
YB,
    owner:
5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8H
E6, // pubkey_sender
    amount: PKE::encrypt(pubkey_sender,
50) - PKE::encrypt(pubkey_sender, 10),
    ...
}
```

```
Account {
    mint:
Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwN
YB,
    owner:
0x89205A3A3b2A69De6Dbf7f01ED13B2108B2c43e7
, // pubkey_receiver
```

```
    amount: PKE::encrypt(pubkey_receiver,
50) + PKE::encrypt(pubkey_receiver, 10),
    ...
}
```

Another problem with encrypting account balances and transfer amounts is that the token program cannot check the validity of a transfer amount. For example, a user with an account balance of 50 tokens should not be able to transfer 70 tokens to another account. For regular SPL tokens, the token program can easily detect that there are not enough funds in a user's account. However, if account balances and transfer amounts are encrypted, then these values are hidden to the token program itself, preventing it from verifying the validity of a transaction.

To fix this, we require that transfer instructions include zero-knowledge proofs that validate their correctness.

zero-knowledge proofs consist of two pair of algorithms `prove` and `verify` that work over public and private data.
The `prove` algorithm generates a "proof" that certifies that some property of the public and private data is true.

The `verify` algorithm checks that the proof is valid.

```
trait ZKP<PublicData, PrivateData> {
  type Proof;

  prove(PublicData, PrivateData) -> Proof;
  verify(PublicData, Proof) -> bool;
}
```

In a transfer instruction, we require the following special classes of zero-knowledge proofs.

- *Range proof*: Range proofs are special types of zero-knowledge proof systems that allow users to generate a proof `proof` that a ciphertext `ct` encrypts a value `x` that falls in a specified range `lower_bound`, `upper_bound`:
  - For any `x` such that `lower_bound <= x < upper_bound`:

```
let ct = PKE::encrypt(pk, x);
let public_data = (pk, ct);
let private_data = (sk, x);

let proof = RangeProof::prove(public_data, private_data);
assert_eq!(RangeProof::verify(public_data, proof), true);
```

- Let `x` be any value that falls out of the bounds. Then for any `proof: Proof`:

```
let ct = PKE::encrypt(pk, x);
let public_data = (pk, ct);

assert_eq!(RangeProof::verify(public_data,
proof), false);
```

- The zero-knowledge property guarantees that the generated proof does not reveal the actual value of the input `x`, but only the fact that `lower_bound <= x < upper_bound`.

  In the confidential extension, we require that a transfer instruction includes a range proof that certify the following:

  - The proof should certify that there are enough funds in the source account. Specifically, let `ct_source` be the encrypted balance of a source account and `ct_transfer` be the encrypted transfer amount. Then we require that `ct_source - ct_transfer` encrypts a value `x` such that `0 <= x < u64::MAX`.
  - The proof should certify that the transfer amount itself is a positive 64

bit number. Let `ct_transfer` be the encrypted amount of a transfer. Then the proof should certify that `ct_transfer` encrypts a value `x` such that `0 <= x < u64::MAX`.

- *Equality proof*: Recall that a transfer instruction contains two ciphertexts of the transfer value `x`: a ciphertext under the sender public key `ct_sender = PKE::encrypt(pk_sender, x)` and one under the receiver public key `ct_receiver = PKE::encrypt(pk_receiver, x)`. A malicious user can encrypt two different values for `ct_sender` and `ct_receiver`.

  Equality proofs are special types of zero-knowledge proof systems that allow users to prove that two ciphertexts `ct_0`, `ct_1` encrypt a same value `x`. In the confidential extension program, we require that a transfer instruction contains an equality proof that certifies that the two ciphertexts encrypt the same value.

  The zero-knowledge property guarantees that `proof_eq` does not reveal the actual values of `x_0`, `x_1` but only the fact that `x_0 == x_1`.

Global Auditor

As separate decryption keys are associated with each user accounts, users can provide read access to balances of *specific* accounts to potential auditors. The confidential extension also allows a *global* auditor feature that can be optionally enabled for mints.
Specifically, in the confidential extension, the mint data structure maintains an additional global auditor encryption key.
This auditor encryption key can be specified when the mint is first initialized and updated via the `ConfidentialTransferInstruction::ConfigureMint` instruction.
If the transfer auditor encryption key in the mint is not `None`, then any transfer instruction must additionally contain an encryption of the transfer amount under the auditor's encryption key.

```
Transfer {
  amount_sender:
PKE::encrypt(pke_pubkey_sender, 10),
  amount_receiver:
```

```
PKE::encrypt(pke_pubkey_receiver, 10),
  amount_auditor:
PKE::encrypt(pke_pubkey_auditor, 10),
  range_proof: RangeProof,
  equality_proof: EqualityProof,
  ...
}
```

This allows any entity with a corresponding auditor secret key to be able to decrypt any transfer amounts for a particular mint.

Similarly to how a dishonest sender can encrypt inconsistent transfer amounts under the source and destination keys, it can encrypt inconsistent transfer amount under the auditor encryption key. If the auditor encryption key is not `None` in the mint, then the token program requires that a transfer amount in a transfer instruction contain additional zero-knowledge proof that certifies that the encryption is done consistently.

Pending and available balance

One way an attacker can disrupt the use of a confidential extension account is by using *front-running*. Zero-knowledge proofs are verified with respect to the encrypted balance of an account. Suppose that a user Alice generates a proof with respect to her current encrypted account balance.
If another user Bob transfers some tokens to Alice, and Bob's transaction is processed first, then Alice's transaction will be rejected by the Token program as the proof will not verify with respect to the newly updated account state.

Under normal conditions, upon a rejection by the program, Alice can simply look up the newly updated ciphertext and submit a new transaction. However, if a malicious attacker continuously floods the network with a transfer to Alice's account, then the account may theoretically become unusable. To prevent this type of attack, we modify the account data structure such that the encrypted balance of an account is divided into two separate

components: the *pending* balance and *available* balance.

```
let ct_pending = PKE::encrypt(pke_pubkey, 10);
let ct_available = PKE::encryption(pke_pubkey, 50);

Account {
    mint: Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwNYB,
    owner: 5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8HE6,
    encryption_key: mpbpvs1LksLmdMhCEzyu5UEWEb3dsRPbB5,
    pending_balance: ct_pending,
    account_balance: ct_available,
    ...
}
```

Any outgoing funds from an account is subtracted from its available balance.

Any incoming funds to an account is added to its pending balance.

As an example, consider a transfer instruction that moves 10 tokens from a sender's account to a receiver's account.

```
let ct_transfer_sender =
PKE::encrypt(pke_pubkey_sender, 10);
let ct_transfer_receiver =
PKE::encrypt(pke_pubkey_receiver, 10);
let ct_transfer_auditor =
PKE::encrypt(pke_pubkey_auditor, 10);

Transfer {
  amount_sender: ct_transfer_sender,
  amount_receiver: ct_transfer_receiver,
  amount_auditor: ct_transfer_auditor,
  range_proof: RangeProof,
  equality_proof: EqualityProof,
  ...
}
```

Upon receiving this transaction and after verifying, the Token program subtracts the

encrypted amount from the sender's available balance and adds the encrypted amount to the receiver's pending balance.

```
Account {
    mint:
Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwN
YB,
    owner:
5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8H
E6,
    encryption_key:
mpbpvs1LksLmdMhCEzyu5UEWEb3dsRPbB5,
    pending_balance: ct_sender_pending,
    available_balance: ct_sender_available
- ct_transfer_sender,
    ...
}
```

This modification removes the sender's ability to change the receiver's available balance of a source account. As range proofs are generated and verified with respect to the available balance, this prevents a user's transaction from

being invalidated due to a transaction that is generated by another user.

An account's pending balance can be merged into its available balance via the `ApplyPendingBalance` instruction, which only the owner of the account can authorize.

Upon receiving this instruction and after verifying that the owner of the account signed the transaction, the token program adds the pending balance into the available balance.

```
Account {
    mint:
Es9vMFrzaCERmJfrF4H2FYD4KCoNkY11McCe8BenwN
YB,
    owner:
5vBrLAPeMjJr9UfssGbjUaBmWtrXTg2vZuMN6L4c8H
E6,
    encryption_key:
mpbpvs1LksLmdMhCEzyu5UEWEb3dsRPbB5,
    pending_balance: ct_pending_receiver -
ct_transfer_receiver,
    available_balance:
ct_available_receiver,
    ...
}
```

Zero Knowledge Proofs

The zero-knowledge proofs that are used in the confidential extension can be divided into two categories: *sigma protocols* and *bulletproofs*. Sigma protocols are simple systems that are tailor designed for the confidential extension use-cases.

We could have our encrypted transfer instruction like this

A transfer instruction has three associated ElGamal public keys: sender, receiver, and auditor. A transfer instruction data must include these three encryption public keys.

```
struct TransferPubkeys {
  source_pubkey: ElGamalPubkey,
  destination_pubkey: ElGamal Pubkey,
  auditor_pubkey: ElGamalPubkey,
}

struct TransferData {
  transfer_pubkeys: TransferPubkeys,
}
```

To cope with ElGamal decryption the transfer amount is restricted to 48-bit numbers and is encrypted as two separate numbers: `amount_lo` that represents the low 16-bits and `amount_hi` that represents the high 32-bits.

```
/// Ciphertext structure of the transfer
amount encrypted under three ElGamal
/// public keys
struct TransferAmountEncryption {
  commitment: PedersenCommitment,
  source_handle: DecryptHandle,
  destination_handle: DecryptionHandle,
  auditor_handle: DecryptHandle,
}

struct TransferData {
  ciphertext_lo: TransferAmountEncryption,
  ciphertext_hi: TransferAmountEncryption,
  transfer_pubkeys: TransferPubkeys,
}
```

In addition to these ciphertexts, transfer data must include proofs that these ciphertexts are generated properly. There are two ways that a user can potentially cheat the program.

1. A user may provide ciphertexts that are malformed. For example, even if a user may encrypt the transfer amount under a wrong public key, there is no way for the program to check the validity of a ciphertext. Therefore, we require that transfer data require a *ciphertext validity* proof that certifies that the ciphertexts are properly generated.
2. In addition to a ciphertext validity proof, a transfer instruction must include a *range proof* that certifies that the encrypted amounts `amount_lo` and `amount_hi` are positive 16 and 32-bit values respectively.

Finally, in addition to proving that the transfer amount is properly encrypted, a user must include a proof that the source account has enough balance to make the transfer. The canonical way to do this is for the user to generate a range proof that certifies that the ciphertext `source_available_balance - (ciphertext_lo + 2^16 * ciphertext_hi)`, which holds the available balance of the source account subtracted by the transfer amount, encrypts a positive 64-bit value. Since Bulletproofs supports proof aggregation, this additional range proof can be aggregated into the original range proof on the transfer amount.

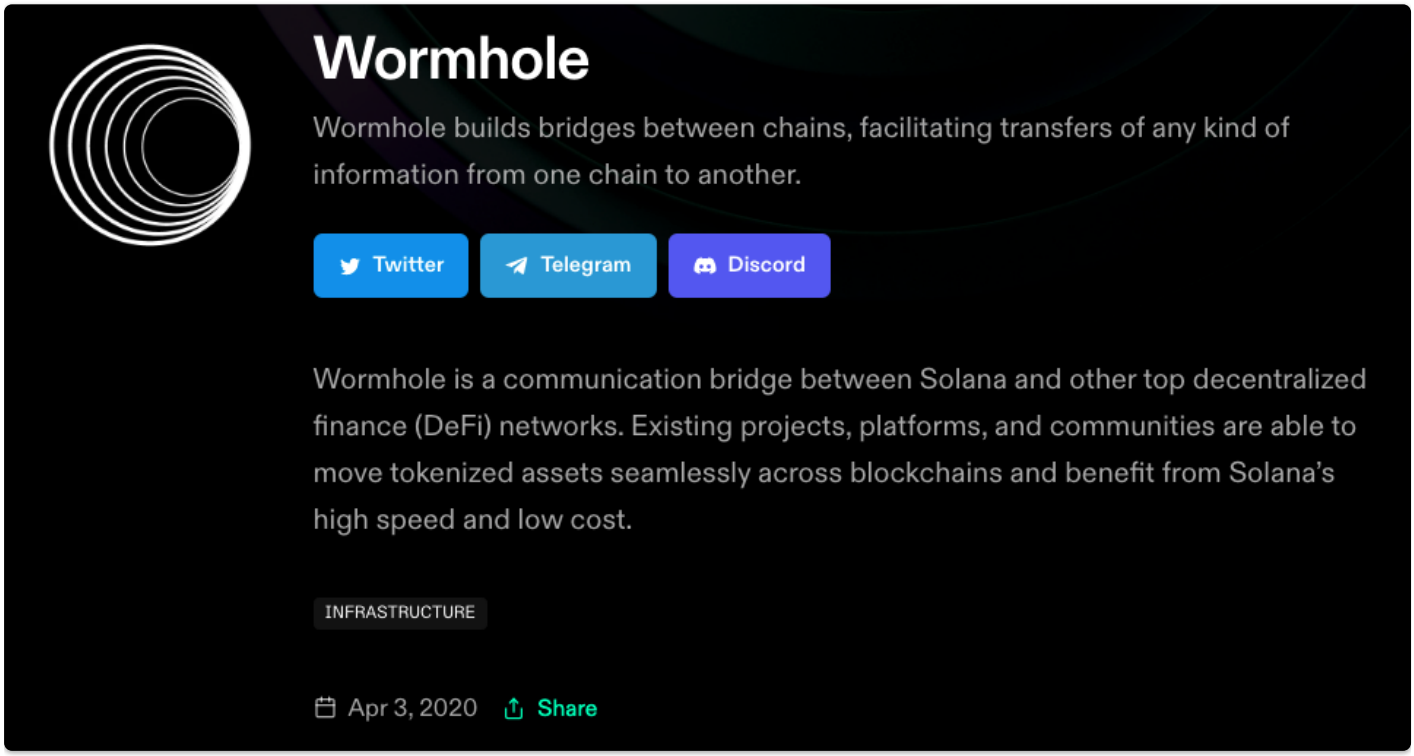
```
struct TransferProof {
  new_source_commitment:
PedersenCommitment,
  equality_proof: CtxtCommEqualityProof,
  validity_proof: ValidityProof,
  range_proof: RangeProof,
}

struct TransferData {
```

```
  ciphertext_lo: TransferAmountEncryption,
  ciphertext_hi: TransferAmountEncryption,
  transfer_pubkeys: TransferPubkeys,
  proof: TransferProof,
}
```

Solana Security

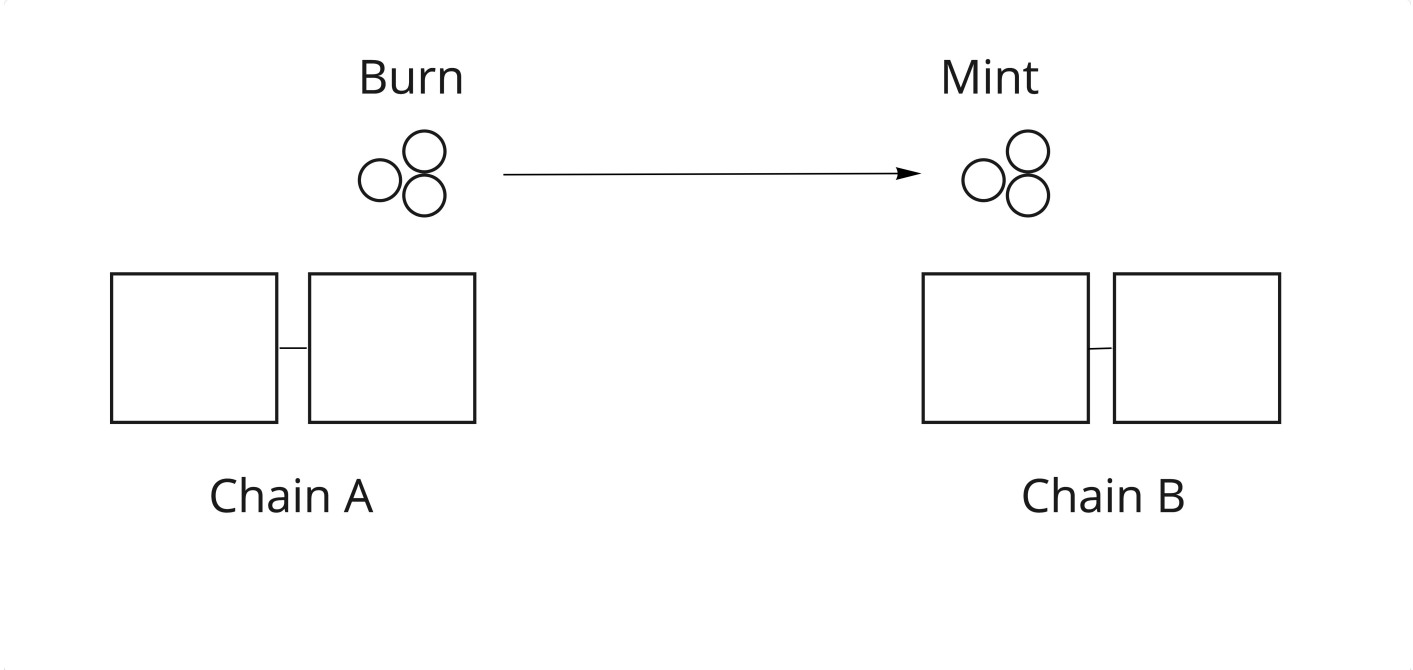
Wormhole attack

See our post mortem article



Wormhole is a “bridge”, basically a way to move crypto assets between different blockchains.

Bridges between blockchains



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.

This process needs to be authorised, Wormhole has a set of “guardians” that sign off on transfers between chains.

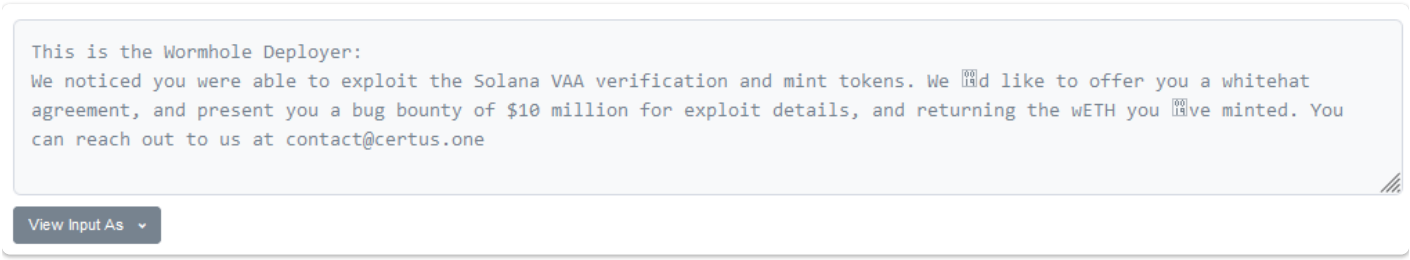
The affected bridge was between ETH on the Ethereum blockchain and wormhole ETH (whETH) on Solana
It acted as an escrow service holding a 1:1 ration of each token.

Attack - February 2nd 2022

The attacker managed to make it look as if 120k ETH had been deposited on Ethereum, allowing the attacker to mint the equivalent in wrapped whETH (Wormhole ETH) on Solana.

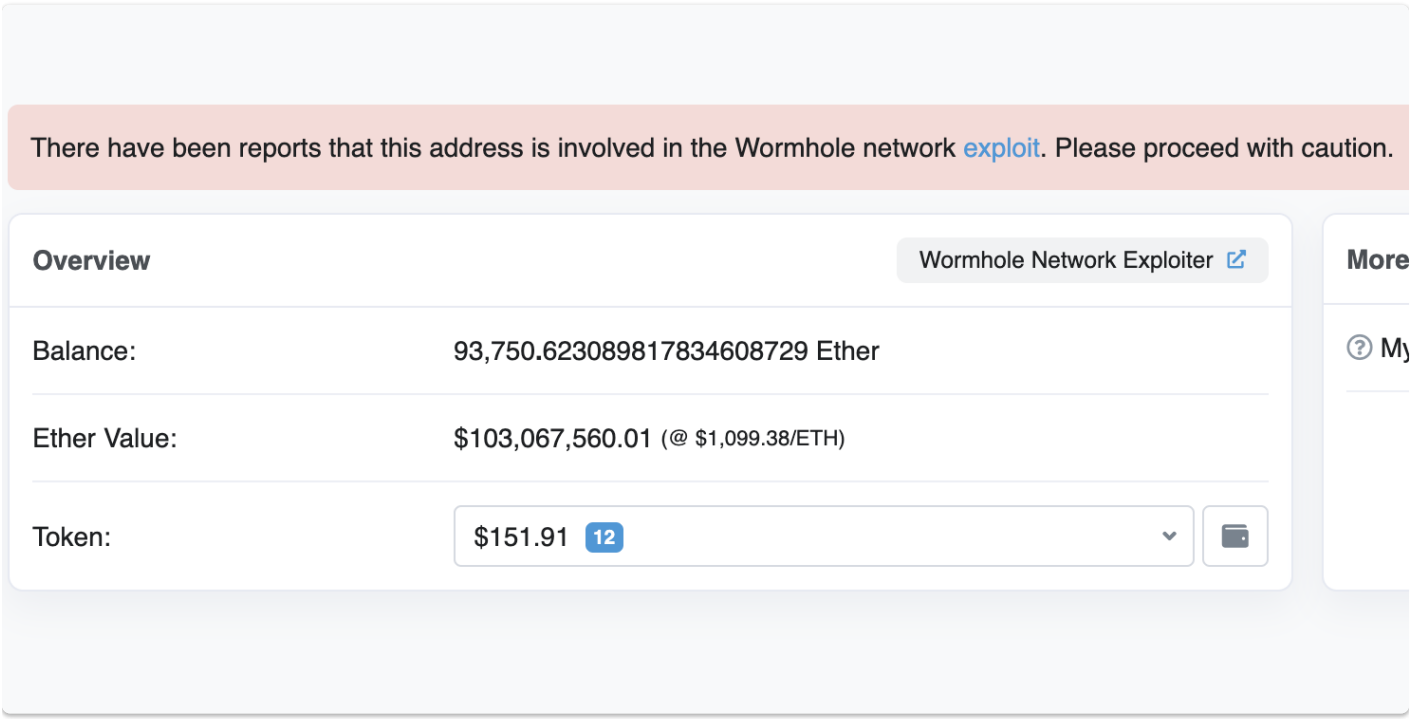
Shortly after the hack, an on-chain message was sent to the hacker from

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.

Ethereum Wallet



The remaining ~36k whETH were liquidated on Solana into USDC and SOL.

This exploit broke the 1:1 peg by stealing 120k ETH from the collateral reserve.

At the time ETH wrapped by Wormhole comprises roughly 19% of the total ETH on the blockchain; if the 1:1 backing of whETH hadn't been replenished swiftly, it could have triggered a situation where DeFi positions may become undercollateralized and potentially fuel a bank run.

Jump Crypto, the cryptocurrency fund which has raised more than $700 million in capital, tweeted 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.

Exploit details

The exploit used the complete_wrapped function, which gets triggered whenever someone mints whETH 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` comes from a Solana program, 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*

The `verify_signatures` function is called to get the signed signature_set for the function post_vaa.

Basically, the wormhole program obtains the set of signatures via the instruction *sysvar program* (whose address is input by the user).

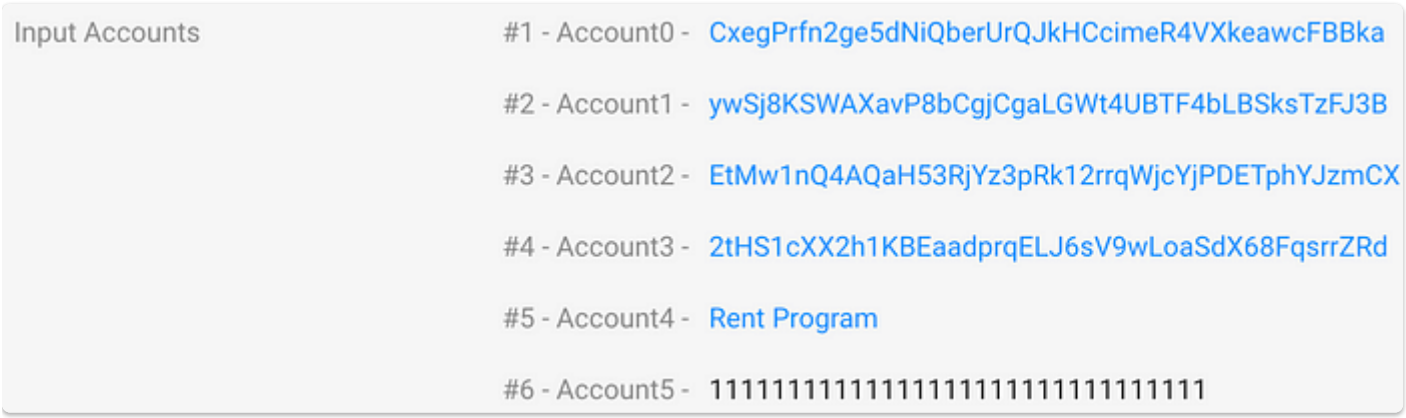
However, the `verify_signatures` function used the `load_instruction_at` function which outputs an instruction that is derived from the input data (which is the data of the instruction sysvar account).

This function does not check if the input sysvar program account is the real sysvar account. Basically, the instruction sysvar program was never checked.

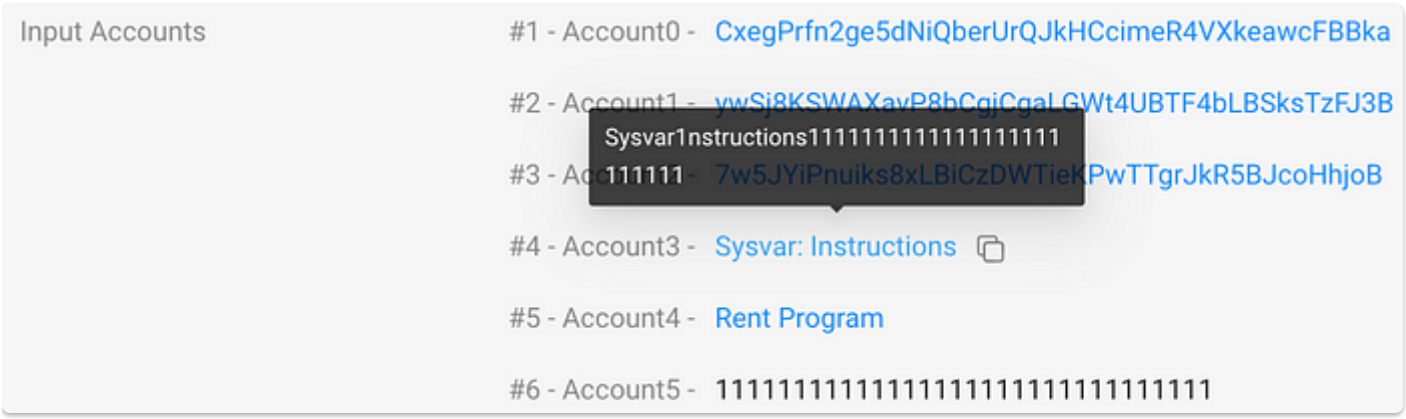
Thus, the attacker created a fake instruction sysvar account with fake data; therefore the signatures were spoofed with previously valid transferred tokens).

Thus, all signatures in the `signature_set` are marked as true falsely indicating that all the supposed guardian signatures were valid.

Here are the transactions parameters as supplied by the attacker



This is what they should have been



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

It is important to verify the validity of unmodified, reference-only accounts .
This is because a malicious user could create accounts with arbitrary data and then pass

these accounts to the program in place of valid accounts in Solana .

It is interesting that 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.

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.

Timeline

- 2021.10.20 06:01: Solana commit to deprecate `load_instruction_at`.
- 2022.01.13 14:29: Wormhole commit to update to Solana to 1.9.4.
- 2022.02.02 17:31: Pull request of the Solana update commit.
- 2022.02.02 18:24: Transaction to mint 120000 wormhole ETH on Solana.
- 2022.02.02 18:28: Transaction to pull out 80000 wETH from wormhole smart contract.

This is one of a number of attacks against blockchain bridges.

Comments about the security of bridges

Vitalik Buterin

> *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.*

Mango Markets attack - Oct 2022

Taken from the write up in rekt.news

The attacker's address was funded with over $5M (2M and 3.5M USDC) from FTX, which were deposited in Mango Markets and used to take out a large MNGO-PERP position.

By countertrading against the position from another account, the attacker succeeded in spiking the spot price of MNGO massively from $0.03 to $0.91. While the MNGO price remained high, the attacker was able to drain the lending pools using the unrealised profit from the long position as collateral.

The attacker borrowed heavily ~ $115M

	Deposits	Borrows	In Orders	Unsettled	Net Balance	Value
USDC	0	54,441,450.70	0	0	-54,441,450.70	-$54,441,450.70
MSOL	0	768,635.982	0	0	-768,635.982	-$25,432,243.06
SOL	0	761,782.908	0	0	-761,782.908	-$23,590,474.13
BTC	0	281.180369	0	0	-281.180	-$5,361,999.98
USDT	0	3,267,402.57	0	0	-3,267,402.57	-$3,267,026.79
SRM	0	2,355,667.42	0	0	-2,355,667.42	-$1,741,116.64
MNGO	0	32,420,404.56	0	0	-32,420,404.56	-$770,517.42

The extreme price manipulation was made possible by the MNGO token's low liquidity and volume. After some mixed messaging, Mango Markets later clarified that the incident was not an oracle failure, but rather genuine price manipulation.

In the process of pumping the price, over 4000 short liquidations were caused.

CQVKS...mKFfX

Do not send tokens directly to your account address.

Baroness Matte Kalepad

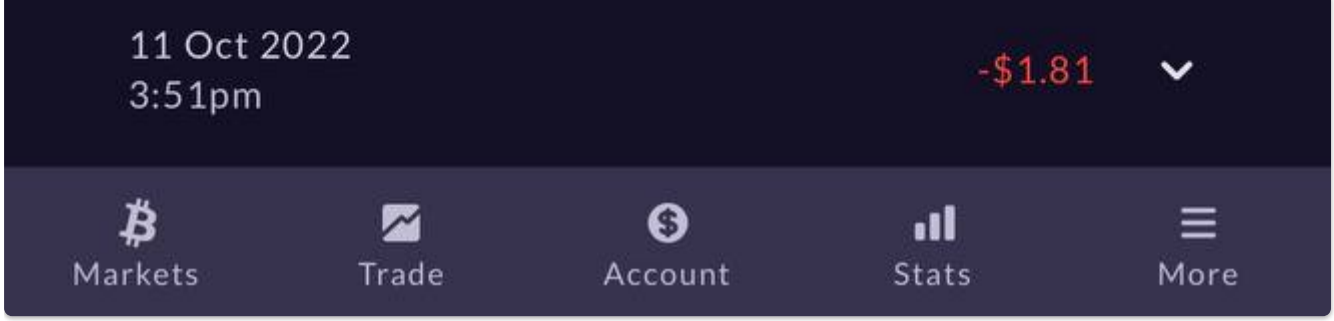
Orders | History | Performan

Trades | Deposits | Withdrawals | Interest | Funding | Liquidations | Volume

4304 Liquidations

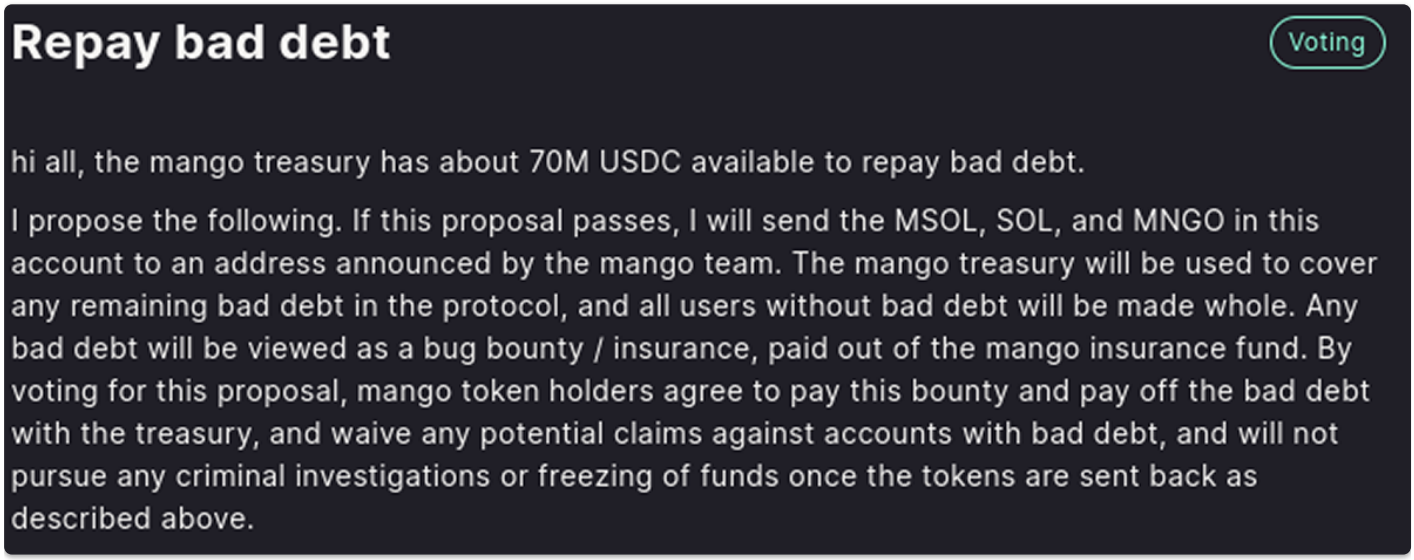
Export CSV

Date	Liquidation Fee
11 Oct 2022 3:51pm	-$16.32
11 Oct 2022 3:51pm	-$1,945.88
11 Oct 2022 3:51pm	-$142.51
11 Oct 2022 3:51pm	-$5.39
11 Oct 2022 3:51pm	-$16.06
11 Oct 2022 3:51pm	-$16.03

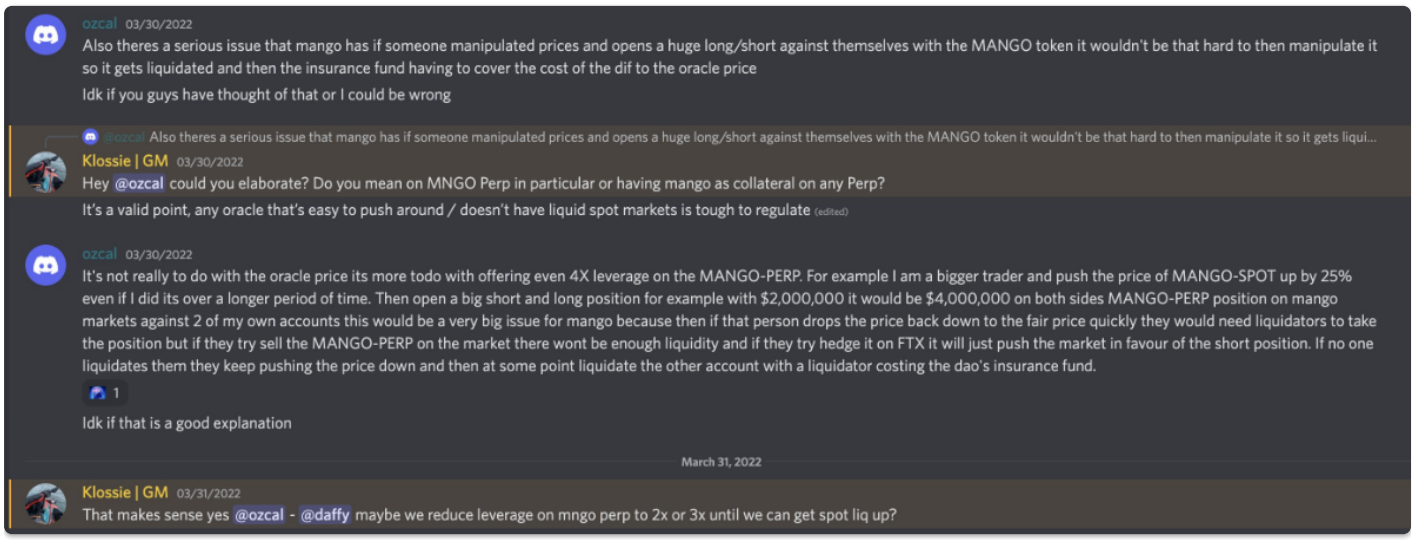


The attack drained all of Mango Markets' available borrow liquidity, with $70M remaining in the treasury. This leaves a shortfall of approximately $50M to cover the bad debt left by the incident, which the hacker is proposing to return.

The attacker proposed that Mango pay the hacker a bounty of ~$65M, and that they do not pursue any criminal investigation.



Concerns had been raised months earlier



Drift Labs uses Oracle guard rails to prevent this type of attack.

General Security

Open Source vs Close Source

Audits

- Risk identification
- Code improvements
- Gas optimisation
- Performance
- Credibility
- Compliance

Good Development Practices

- CI / CD
- Unit Tests
- TDD
- BDD

Post Deployment

- Monitoring
- Learning from exploits
- Bug Bounties

Other ways to get involved

- Whitehats
- Grants programs

Solana specific attacks

As Solana smart contract are stateless, logic data required for execution and subsequent state modification has to be provided from the client.

This data comes in two forms:

- instruction data specified from client
- addresses of the on-chain accounts containing state

To prevent many of the attacks which feature substitution of accounts, newly initiated accounts are zeroed out and only the program has the authority to modify them.

Advice from ArmaniOne

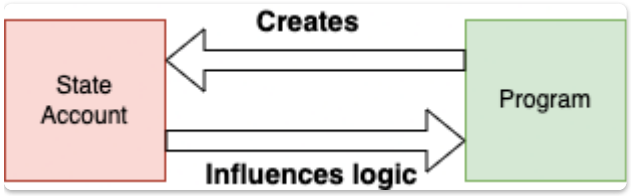
See thread

Feeding tailored account to influence logic

For a program logic to work as expected state is fed via on-chain accounts.

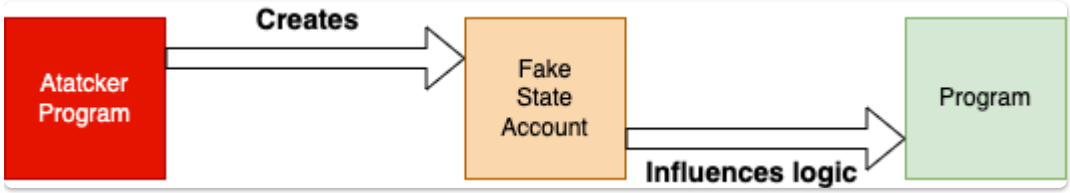
Account containing state can be owned by the program being invoked, but it doesn't necessarily have to be.

If it is owned by the program, the chances of it being malicious are much lower as the program is responsible for initialising and changing its state.



An account can be created by a different program specifically to recreate the structure of a similar account owned by the program, but with some crucial substitutions.

This account is then fed to the program to get it to achieve something that the designer did not intend to happen.



To prevent this attack any account assumed to be owned and created by the program needs to be checked for ownership by the program.

```
if state_account.owner != program_id {
        //exit
}
```

This should also extend to checks for ownership of other programs, if you are expecting an SPL-token account then the owner of it should be an SPL-Program.

Pretending to be the authority

As every non-wallet account is owned by a program, there is an additional permission field named usually either authority, admin or manger embedded within the account.
This is an example of field containing an address that has additional privileges:

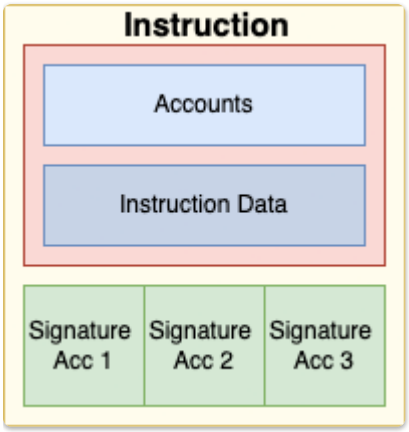
```
#[derive(BorshSerialize, BorshDeserialize, Debug, Clone, BorshSchema)]
pub struct Account {
        pub manager: Pubkey,
        ...
}
```

This account is provided during invocation and the caller can access otherwise restricted code area.

An attacker can pretend to be that authority by feeding a program owned account even though they are not the stated authority.

Each transaction includes not only accounts and instruction data, but additionally signatures of

parties that have seen the transaction and are happy with invocation.



To prevent this attack not only is it needed to check whether an account provided is owned by the program:

```
assert!(account.owner == program_id);
```

and whether the right privilege was attached:

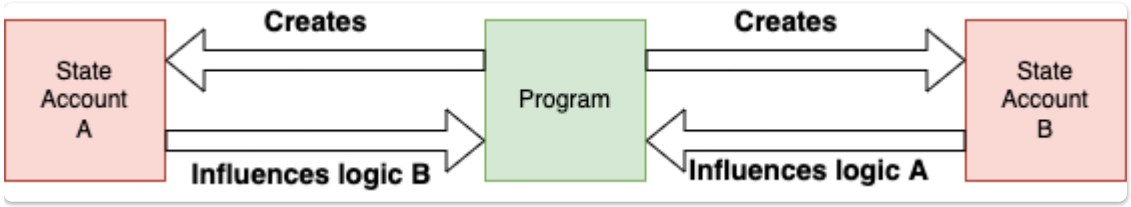
```
assert_eq!(manager.pubkey ==
account.manager);
```

but also whether intended account has signed the transaction:

```
assert!(manager.is_signer);
```

Program owned accounts substitution

Most program will feature multiple types of accounts managed in order to provide the full set of functionality.



An attacker could provide incorrect accounts for a given logic.
This vulnerability is known as type confusion and within the Solana world as account confusion.

For example a program may manage two accounts, `Funder` and `Admin`.

Layout of these could be as follows:

```
pub struct Funder {
        pub funder_key: Pubkey,
        pub project_funded: u8,
}


pub struct Admin {
```

```
        pub admin_key: Pubkey,
        pub admin_level: u8,
}
```

A function that requires additional privilege would require a caller to prove himself to be embedded within an account that states that privilege.

Providing a funder account masquerading as an admin account would allow the attacker to do so.

In this type of attack a simple check for ownership would not suffice, as both accounts are program owned. Likewise a signature check would not suffice.

To prevent this vulnerability it is important to tag each account accordingly and to ensure checks for this are made when dealing with the accounts.

```
#[derive(BorshSerialize, BorshDeserialize,
PartialEq)]
pub enum AccType {
        Funder,
        Admin,
}

#[derive(BorshSerialize, BorshDeserialize,
PartialEq)]
pub struct Funder {
        pub account_type: AccType,
        pub project_funded: u8,
}

#[derive(BorshSerialize, BorshDeserialize,
PartialEq)]
pub struct Admin {
        pub account_type: AccType,
        pub admin_key: Pubkey,
```

```
        pub admin_level: u8,
}
```

This gets done automatically with Anchor's `#[account]` macro.
It automatically adds an 8-byte discriminator to the start of the account

```
#[account]
pub struct Funder {
        pub project_funded: u8,
}

#[account]
pub struct Admin {
        pub admin_key: Pubkey,
        pub admin_level: u8,
}
```

In addition Anchor's discriminator prevents account from being initialised twice, but see example below.

Stack overflow

Logic requires mathematical operations, it could be changing balances, looping over sets or calculating profits. Despite being safe for invalid memory access, Rust does not enforce arithmetic checks.

The solution is to use functions that enforce safe mathematics. In essence they check how close to the "edge" of the variable a given arithmetic operation will take, and if within safe limits it will allow it.

Integer overflow/underflows are surprisingly common in smart contracts, because blockchain applications often compute math over financial data.

Rust is a popular language used in blockchains such as Solana and Polkadot. For many developers, it may be a misconception that Rust is memory-safe so it is free of arithmetic overflow/underflows.

There are ways of handling it included in the `spl_math` library.

Checked

On an operation that would result in an overflow it returns `None`.

```
pub fn checked_add(self, other: U256) ->
Option<U256>
```

Operation	Unsafe	Safe
addition	a + b	a.checked_add(b)
subtraction	a - b	a.checked_sub(b)
multiplication	a * b	a.checked_mul(b)
division	a / b	a.checked_div(b)

Saturation

On an operation that would result in an overflow it sets maximum value that does not overflow.

```
pub fn saturating_add(self, other: U256) -
> U256
```

Operation	Unsafe	Safe
addition	a + b	a.saturation_add(b)
subtraction	a - b	a.saturation_sub(b)
multiplication	a * b	a.saturation_mul(b)
division	a / b	a.saturation_div(b)

Rentrancy attack

The first version of this bug to be noticed involved functions that could be called repeatedly before the first invocation of the function was finished. This may cause the different invocations of the function to interact in destructive ways.

Common pitfalls from Neodyme

See Blog

- Missing ownership check
- Missing signer check
- Solana account confusions
- Arbitrary signed program invocation
- Integer overflow & underflow

Example lack of signer check

```
fn update_admin(accounts: &[AccountInfo],
admin: [u8; 32]) -> ProgramResult {
let acc_iter = &mut accounts.iter();
let admin_info =
next_account_info(acc_iter)?;
let staking_info =
next_account_info(acc_iter)?;

// There would be a problem without a
check here
// if !admin_info.is_signer {
// return
Err(ProgramError::MissingRequiredSignature
);
```

```
// }

let mut staking =
StakingInfo::try_from_slice(&staking_info.
data.borrow())?;
if staking.admin == [0; 32] {
staking.admin = admin;
} else if staking.admin ==
admin_info.key.to_bytes() {
staking.admin = admin;
} else {
return
Err(StakeError::AdminRequired.into());
}

let _ = staking.serialize(&mut &mut
staking_info.data.borrow_mut()[..]);
Ok(())
}
```

Re initialisation attack with Candy Machine

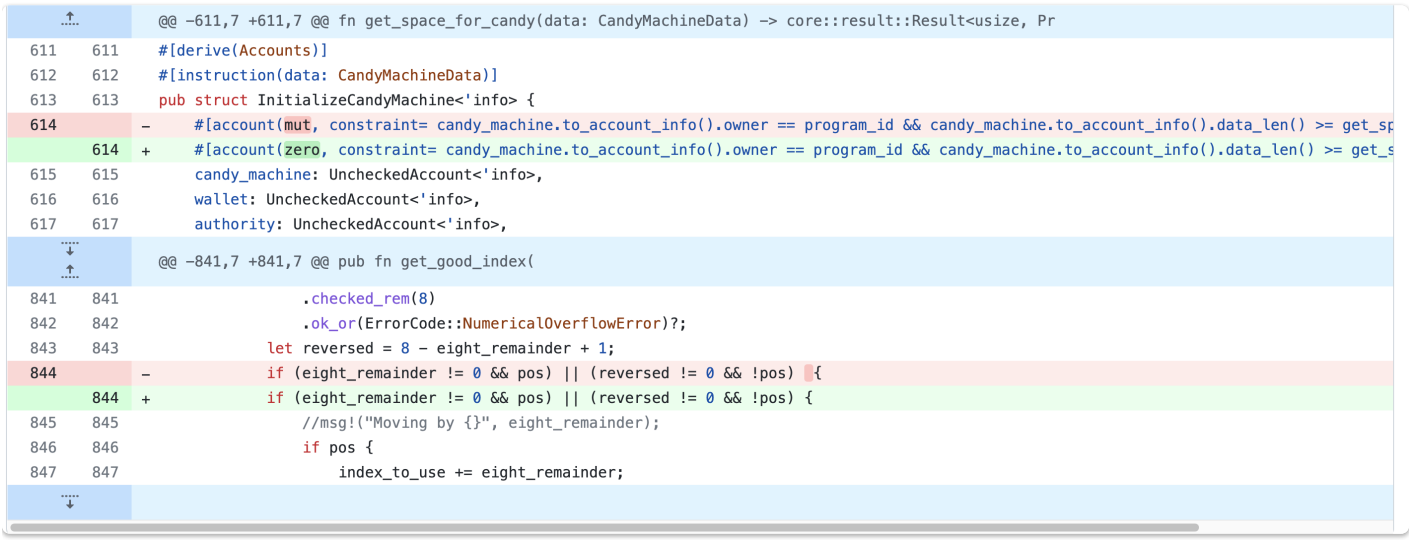
See article

As a simple example, given a candy machine account A that has already been created and initialized, an attacker could send the same account again to this instruction, where they change the wallet account to a wallet of their own.

```
#[derive(Accounts)]
#[instruction(data: CandyMachineData)]
pub struct InitializeCandyMachine<'info> {
    #[account(mut, constraint=
candy_machine.to_account_info().owner ==
program_id &&
candy_machine.to_account_info().data_len()
>= get_space_for_candy(data)?)]
    candy_machine:
UncheckedAccount<'info>,
    wallet: UncheckedAccount<'info>,
    authority: UncheckedAccount<'info>,
    payer: Signer<'info>,
    system_program: Program<'info,
System>,
```

```
    rent: Sysvar<'info, Rent>,
}
```

This is the fix that was applied



```
@@ -611,7 +611,7 @@ fn get_space_for_candy(data: CandyMachineData) -> core::result::Result<usize, Pr
611 611  #[derive(Accounts)]
612 612  #[instruction(data: CandyMachineData)]
613 613  pub struct InitializeCandyMachine<'info> {
614     -    #[account(mut, constraint= candy_machine.to_account_info().owner == program_id && candy_machine.to_account_info().data_len() >= get_sp
    614 +    #[account(zero, constraint= candy_machine.to_account_info().owner == program_id && candy_machine.to_account_info().data_len() >= get_s
615 615      candy_machine: UncheckedAccount<'info>,
616 616      wallet: UncheckedAccount<'info>,
617 617      authority: UncheckedAccount<'info>,
@@ -841,7 +841,7 @@ pub fn get_good_index(
841 841                  .checked_rem(8)
842 842                  .ok_or(ErrorCode::NumericalOverflowError)?;
843 843              let reversed = 8 - eight_remainder + 1;
844     -            if (eight_remainder != 0 && pos) || (reversed != 0 && !pos) {
    844 +            if (eight_remainder != 0 && pos) || (reversed != 0 && !pos) {
845 845                  //msg!("Moving by {}", eight_remainder);
846 846                  if pos {
847 847                      index_to_use += eight_remainder;
```

From anchor docs

`#[account(zero)]`

Checks the account discriminator is zero. Enforces rent exemption unless skipped with `rent_exempt = skip`.

Use this constraint if you want to create an account in a previous instruction and then initialize it in your instruction instead of using `init`. This is necessary for accounts that are larger than 10 Kibibyte because those accounts cannot be created via a CPI (which is what `init` would do).

Anchor adds internal data to the account when using `zero` just like it does with `init` which is why `zero` implies `mut`.

Example:

```
#[account(zero)]
pub my_account: Account<'info, MyData>
```

Other

This list is not exhaustive, many other errors exist that can't be fully summarised here.

Some others include

- Preventing completion due to running out of gas
- Numerical precision errors
- Casting truncation
- Oracle manipulation
- Vampire attacks
- DoS
- Timestamp dependence
- Cross-function Reentrancy
- Economic Attacks

But there are tools that can help spotting them.

Solana Auditing and Security Resources

See Repo

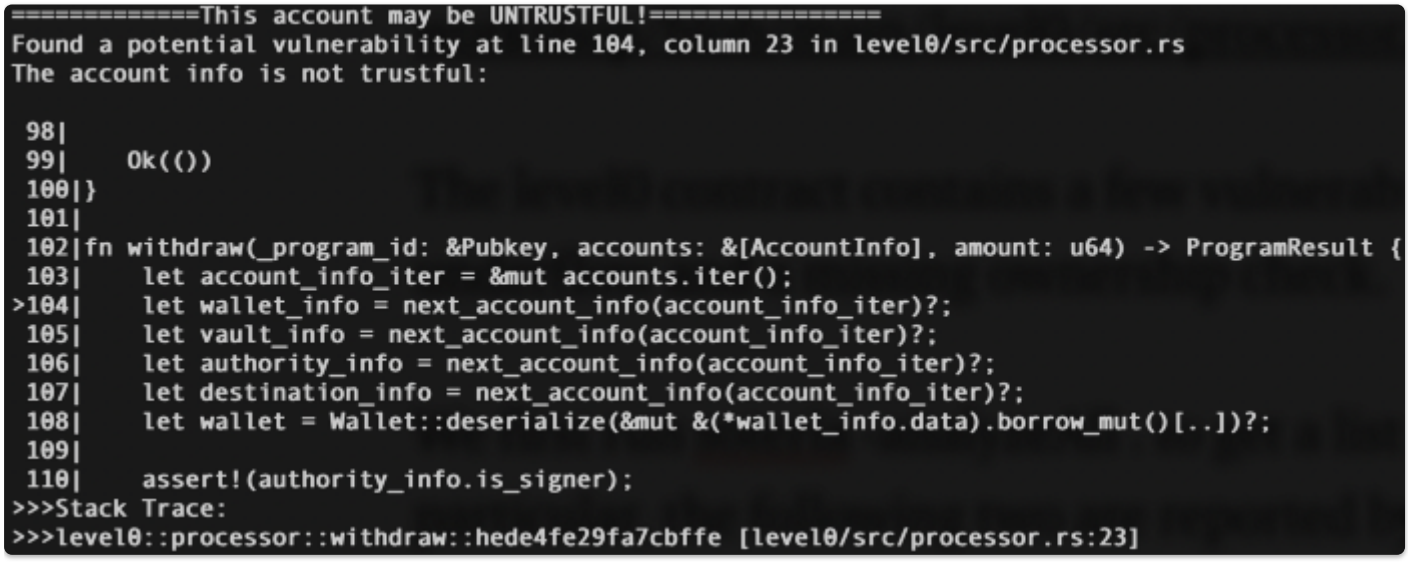
- Solana Auditing Workshop Slides
- Overview of Solana security from Kudelski Security

Security Tools

Soteria

See article

Static analysis tool that automatically detects vulnerabilities by checking all execution paths.



It can be installed with

```
sh -c "$(curl -k
https://supercompiler.xyz/install)"
```

```
export PATH=$PWD/soteria-linux-
develop/bin/:$PATH
```

or via docker

```
docker run -v $PWD/jet-v1/:/workspace -it
greencorelab/soteria:latest /bin/bash
```

Rust-Analyzer

See Docs

IDE extension and semantic analysis of Rust code.

Neodyme POC framework

See Docs

Tool for creating local testing environments within which bugs can be detected.

Solana Security Policy

See Docs

Security Bug Bounties from Solana

We offer bounties for critical security issues. Please see below for more details. Either a demonstration or a valid bug report is all that's necessary to submit a bug bounty. A patch to fix the issue isn't required.

Loss of Funds:

$2,000,000 USD in locked SOL tokens (locked for 12 months)

- Theft of funds without users signature from any account
- Theft of funds without users interaction in system, token, stake, vote programs
- Theft of funds that requires users signature - creating a vote program that drains the delegated stakes.

Consensus/Safety Violations:

$1,000,000 USD in locked SOL tokens (locked for 12 months)

- Consensus safety violation

- Tricking a validator to accept an optimistic confirmation or rooted slot without a double vote, etc.

Liveness / Loss of Availability:

$400,000 USD in locked SOL tokens (locked for 12 months)

- Whereby consensus halts and requires human intervention
- Eclipse attacks,
- Remote attacks that partition the network,

DoS Attacks:

$100,000 USD in locked SOL tokens (locked for 12 months)

- Remote resource exhaustion via Non-RPC protocols

Supply Chain Attacks:

$100,000 USD in locked SOL tokens (locked for 12 months)

- Non-social attacks against source code change management, automated testing, release build, release publication and

release hosting infrastructure of the monorepo.

RPC DoS/Crashes:

$5,000 USD in locked SOL tokens (locked for 12 months)

- RPC attacks

Mango Markets Bug Bounties

Bug Bounty

Program Overview

This bug bounty is specifically for Mango Markets' on-chain program code; UI only bugs are omitted.

Severity	Description	Bug Bounty
Critical	Bugs that freeze user funds or drain the contract's holdings or involve theft of funds without user signatures.	10% of the value of the hack up to $1,000,000.
High	Bugs that could *temporarily* freeze user funds or incorrectly assign value to user funds.	$10,000 to $50,000 per bug, assessed on a case by case basis
Medium/Low	Bugs that don't threaten user funds	$1,000 to $5,000 per bug, assessed on a case by case basis