SharkTeam

SharkTeam: Move Language Security Analysis and Contract Audit Essentials —— The Replay Attack

Feb 24, 2023

SharkTeam, a leading blockchain security service team, offers smart contract audit services for developers. To satisfy the demands of different clients, thesmart contract audit services provide both manual auditing and automated auditing.

We implement almost 200 auditing contents that cover four aspects: high-level language layer, virtual machine layer, blockchain layer, and business logiclayer, ensuring that smart contracts are completely guaranteed and Safe.

In the previous series of “Top 10 Smart Contract Security Threats”, SharkTeam summarized and analyzed the top 10 most harmful vulnerabilities in the field of smart contracts based on historical smart contract security incidents.

These vulnerabilities usually appeared in Solidity smart contracts before, so will the same harm exist for the emerging Move smart contracts?

SharkTeam [A Vulnerability Perspective Analysis of Move Language Security] series of courses will discuss and deepen with you. The contents of this chapter are 【Replay Attack】.

Replay Attack is a malicious or fraudulent means of repeating or delaying valid data in traditional networks. It can be performed by the originator of a replay attack or by an adversary who intercepts data and retransmits it in order to deceive the system. This is a lower-level version of the “man-in-the-middle attack” and is used primarily in the authentication process to undermine the correctness of the authentication.

In blockchain, there are also replay attacks. In blockchain, the authentication process is the process of digital signature verification, and each authentication requires a new digital signature. An attacker may use a digital signature that has already been used for authentication and successfully pass it, and we call this attack method a replay attack in blockchain. In the following section, we only discuss replay attacks in blockchain and their impact on the Move ecosystem.

1. Classification of Replay Attacks

According to the replayed signature data and the different attack levels, we classify the replay attack into transaction replay and signature replay.

Transaction replay refers to the reuse of transactions and their signatures to put the transactions on the original chain into the target chain unchanged, and then the transactions can be executed normally on the target chain and complete the transaction verification after replay. Transaction replay is an attack at the blockchain level, which is a cross-chain attack method of the same ecology (such as Ethereum, BNB Chain, HECO Chain, etc. of Solidity ecology).

Signature replay means replaying the private key signature in the transaction data repeatedly, and the replaying process does not need to replay the whole transaction like transaction replay, but replay the corresponding data and its signature. Signature replay is a smart contract level attack, which can be a replay within the same chain, a cross-chain attack of the same ecology, or a cross-chain attack of signature-compatible public chains of different ecologies (e.g. Ethereum of Solidity ecology, Solana of Rust ecology, Aptos of Move ecology, etc.). Signature replay is more of a cross-chain attack of the same ecology; replay within the same chain is relatively rare; and cross-chain replay attacks of different ecologies require the most demanding conditions and are the least likely, but there is no guarantee that they will never happen. For hackers, whether they are project designers, developers, or auditors, they should not relax their vigilance heart.

2. Security Incidents

2.1 Optimism replay attack Incident

On June 9, 2022, hackers successfully stole 20 million OP tokens granted to Wintermute by the Optimism Foundation through transaction replay attacks. The transaction replay attack process is as follows:

(1) On May 27, the Optimism Foundation transferred 20 million OP tokens to Wintermute's multi-signature contract address on Optimism/L2. The multi-signature contract address is the multi-signature contract address of Wintermute on Ethereum/L1, and the multi-signature contract has been deployed. However, Wintermute does not deploy a corresponding multi-signature contract on Optimism/L2.

(2) On June 1, the attacker deployed the attack contract.

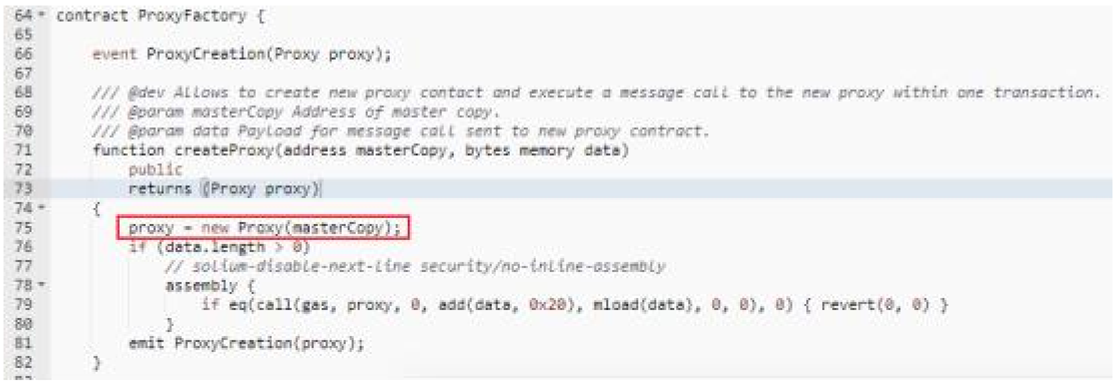
(3) On June 5, the attacker created the Gnosis Safe: Proxy Factory 1.1.1 contract by replaying the transactions on Ethereum/L1, and its address was

the same as that on Ethereum/L1; then the attacker deployed multi-signatures by attacking the contract Contract 0x4f3a, which is also the same address as Wintermute's multi-signature contract on Ethereum/L1, but the contract ownership belongs to the attacker. At this time, 20 million OP tokens have been transferred into the multi-sign contract.

(4) The attacker transfers 1 million OP to the attacker's address through the multi-signature contract 0x4f3a deployed by him, and converts 1 million OP into 720.7 Ether.

The key to the success of the entire attack process is that the attacker used the contract creation vulnerability in Solidity to create a multi-money contract with the same address as the Wintermute multi-signature contract on Ethereum/L1 through a replay attack. The ownership of the multi-signature contract Owned by the attacker.

In the Gnosis Safe: Proxy Factory 1.1.1 contract, the code for creating the proxy contract function createProxy is as follows:



```
contract ProxyFactory {

    event ProxyCreation(Proxy proxy);

    /// @dev Allows to create new proxy contact and execute a message call to the new proxy within one transaction.
    /// @param masterCopy Address of master copy.
    /// @param data Payload for message call sent to new proxy contract.
    function createProxy(address masterCopy, bytes memory data)
        public
        returns (Proxy proxy)
    {
        proxy = new Proxy(masterCopy);
        if (data.length > 0)
            // solium-disable-next-line security/no-inline-assembly
            assembly {
                if eq(call(gas, proxy, 0, add(data, 0x20), mload(data), 0, 0), 0) { revert(0, 0) }
            }
        emit ProxyCreation(proxy);
    }
```

The Solidity version used by the Gnosis Safe: Proxy Factory 1.1.1 contract is 0.5.3, and the contract is created by new. Here the opcode used by new to create the contract is CREATE instead of CREATE2.

Use the CREATE opcode to create a contract, and the contract address is calculated by the creator (contract creator address, that is, the current contract address using the CREATE opcode) and nonce. On Ethereum/L1, the creator who created the multi-signature contract 0x4f3a is the address of Gnosis Safe: Proxy Factory 1.1.1. The main purpose of the attacker to create the contract in

Gnosis Safe: Proxy Factory 1.1.1 is to replay the transaction on Optimism/L2 It is to ensure that the creator of contract 0x4f3a on Optimism/L2 is consistent with Ethereum/L1. Then keep the nonce consistent with Ethereum/L1 through transaction replay. Therefore, the attacker can call the createProxy function through the smart contract (contract 0xe714) to create a multi-signature contract with the same address (0x4f3a) as Ethereum/L1.

(5) On June 5, after receiving 20 million OPs, the multi-signed contract 0x4f3a transferred 1 million OPs to the hacker address 0x60b2, and then exchanged 1 million OPs for 720.7 Ether.

(6) On June 9, the contract 0x4f3a transferred 1 million OPs to the account address 0xd8da, and the other 18 million OPs are still in the contract 0x4f3a.

In summary, this security incident was caused by a combination of factors such as transaction replay, differences between the old and new versions of Solidity, and transaction signature verification on the main and side chains.

The Solidity opcodes CREATE and CREATE2 are introduced as follows:

(1) The CREATE operation code creates a contract, and the new contract address is calculated as follows:

Hash(creator, nonce)

l creator: the address of the creator of the new contract, that is, the address of the contract using CREATE

l nonce: the nonce value of the transaction that created the contract

Create a new contract using CREATE via new:

Contract x = new Contract{value: _value}(params)

The value is optional and refers to the sent ether.

(2) The CREATE2 operation code creates a contract, and the new contract address is calculated as follows:

Hash(“0xff”, creator, salt, bytecode)

l “0xff”: a constant to avoid conflicts with CREATE

I creator: the address of the creator of the new contract

I salt: a salt value given by the creator

I bytecode: the bytecode of the contract to be deployed

Create a new contract using CREATE2 via new:

Contract x = new Contract{salt: _salt, value: _value}(params)

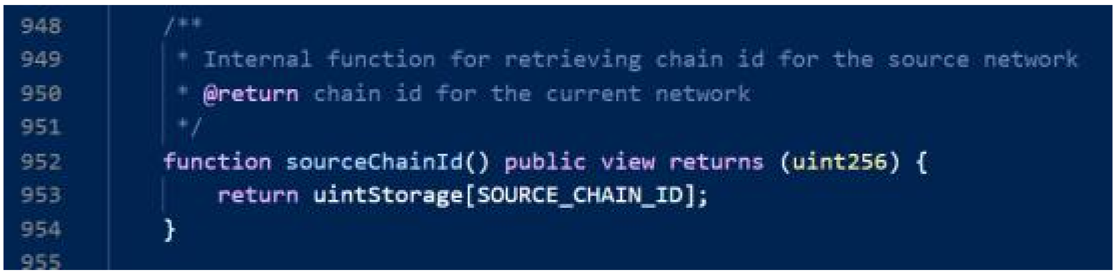
The value is optional and refers to the sent ether.

2.2 OmniBridge replay attack Incident

On September 18, 2022, the attacker transferred 200 WETH through the omni bridge of the Gnosis chain, and then replayed the same message on the PoW chain, obtaining an additional 200 ETHW.

In the same chain, we sort transactions by nonce to prevent transactions from being replayed. On different chains, we will identify the type of chain according to the chainid. For example, the chainid of the Ethereum mainnet is 1, and the chainid of the ETHW mainnet is 10001. Embedding chainid in the transaction can avoid cross-chain replay of the transaction. As for the signature, the signature generally used for cross-chain verification will also include the chainid to avoid cross-chain replay of the signature.

Ethereum enforced EIP-155 before the hard fork, which means that transactions on the ETH PoS chain cannot be replayed on the PoW chain. Therefore, this transaction replay is not a vulnerability of the chain itself. Analyzing the source code of Omni Bridge, it is found that in Omni Bridge's chainid verification logic, the chainid comes from the value stored in unitStorage, rather than the chainid on the chain directly read through the opcode CHAINID (0x46).



```
    /**
     * Internal function for retrieving chain id for the source network
     * @return chain id for the current network
     */
    function sourceChainId() public view returns (uint256) {
        return uintStorage[SOURCE_CHAIN_ID];
    }
```

After the hard fork of Ethereum, the chainid stored in the state variable is not updated to the new chainid, so the signature before and after the hard fork can be replayed. The attacker used this vulnerability to perform signature replay and obtained an additional 2 million ETHW.

3. Risk Prevention

By reviewing the attacks that have occurred, the root cause of both transaction replay and signature replay occurs due to the lack of uniqueness verification in the signature's verification mechanism. The replay attacks can be prevented by simply adding unique identifiers to the signature and verification mechanisms.

According to the different levels of replay attacks, the uniqueness identifiers include 3 types, namely

(1) The uniqueness identifier chainid (public chain ID) at the public chain level, and the verification of chainid can prevent replay attacks across chains (refer to EIP-155).

(2) The unique identifier nonce (transaction sequence number) at the transaction level, and the verification of nonce can prevent replay attacks on the same chain.

(3) Unique identifiers at the contract level, i.e., custom business identifiers, such as custom sequence numbers, and verification of custom business identifiers can prevent signature replay attacks on the business side of the contract.

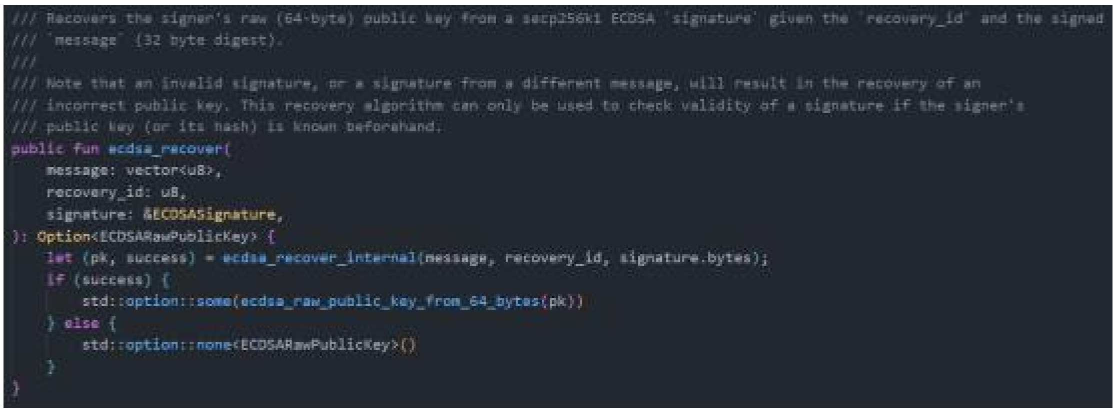
For the contract level, in addition to adding unique identifiers in the signature and verification mechanism, we can add signature verification records, such as using mapping to record the verification results of each unique signature.

4. Risk Analysis of Move Replay Attack

Replay attack is an attack method based on signature mechanism. Unlike Ethereum, which only supports ECDSA, a signature scheme, the Move ecosystem supports multiple signature mechanisms, especially at the contract level, and supports signature algorithms commonly used in blockchains, including the ECDSA signature scheme based on the Secp256k1 elliptic curve (Ethereum's unique signature scheme), single-signature scheme and k-of-N multi-signature scheme based on Ed25519 elliptic curve, BLS12–381 aggregate signature scheme, and even zero-knowledge proof schemes such as Groth16.

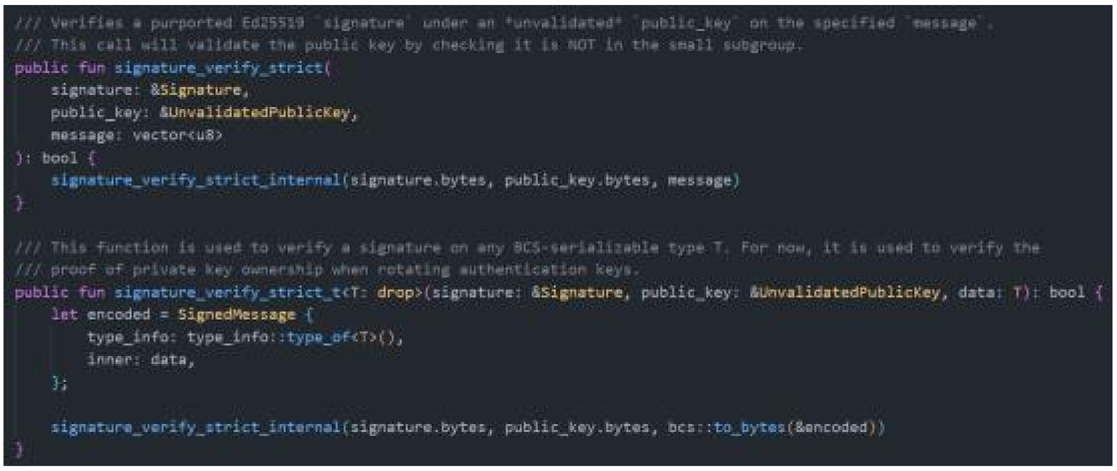
The Aptos public chain transaction signature supports the EdDSA single-signature scheme and the k-of-N multi-signature scheme based on the Ed25519 elliptic curve, and the single-signature scheme is selected by default. The contract level supports the verification of four signature mechanisms:

(1) ECDSA signature verification based on secp256k1 elliptic curve



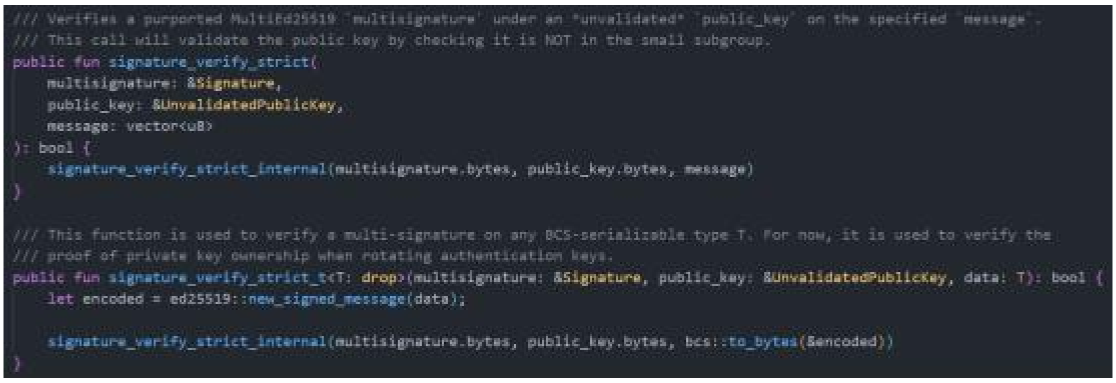
```
/// Recovers the signer's raw (64-byte) public key from a secp256k1 ECDSA `signature` given the `recovery_id` and the signed
/// `message` (32 byte digest).
///
/// Note that an invalid signature, or a signature from a different message, will result in the recovery of an
/// incorrect public key. This recovery algorithm can only be used to check validity of a signature if the signer's
/// public key (or its hash) is known beforehand.
public fun ecdsa_recover(
    message: vector<u8>,
    recovery_id: u8,
    signature: &ECDSASignature,
): Option<ECDSARawPublicKey> {
    let (pk, success) = ecdsa_recover_internal(message, recovery_id, signature.bytes);
    if (success) {
        std::option::some(ecdsa_raw_public_key_from_64_bytes(pk))
    } else {
        std::option::none<ECDSARawPublicKey>()
    }
}
```

(2) EdDSA signature verification based on Ed25519 elliptic curve



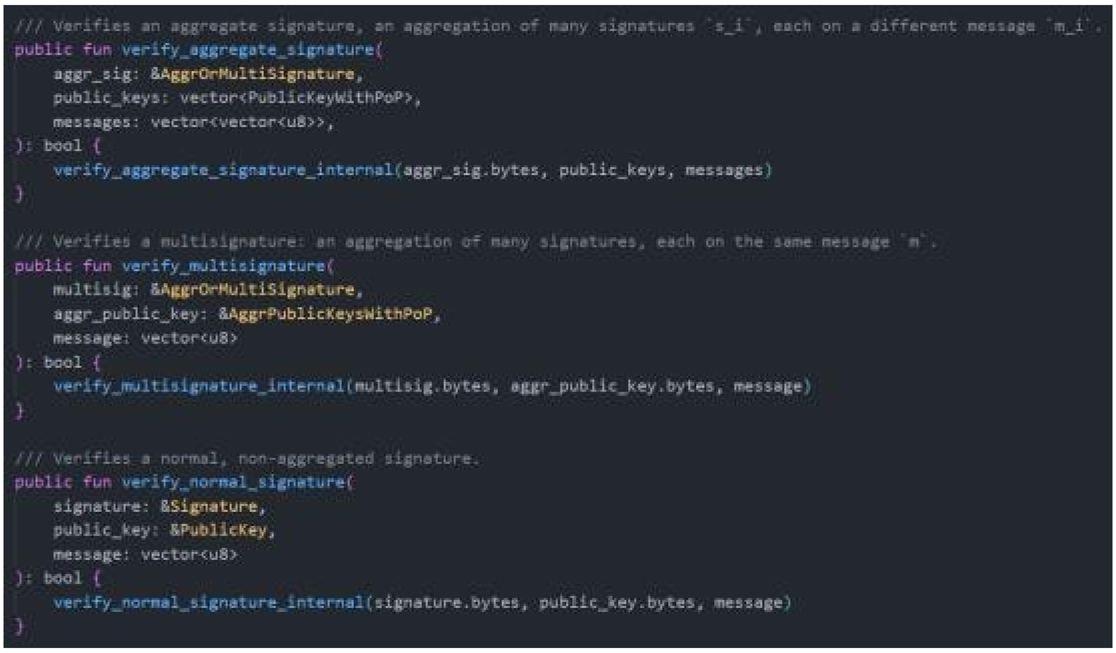
```
/// Verifies a purported Ed25519 `signature` under an *unvalidated* `public_key` on the specified `message`.
/// This call will validate the public key by checking it is NOT in the small subgroup.
public fun signature_verify_strict(
    signature: &Signature,
    public_key: &UnvalidatedPublicKey,
    message: vector<u8>
): bool {
    signature_verify_strict_internal(signature.bytes, public_key.bytes, message)
}

/// This function is used to verify a signature on any BCS-serializable type T. For now, it is used to verify the
/// proof of private key ownership when rotating authentication keys.
public fun signature_verify_strict_t<T: drop>(signature: &Signature, public_key: &UnvalidatedPublicKey, data: T): bool {
    let encoded = SignedMessage {
        type_info: type_info::type_of<T>(),
        inner: data,
    };

    signature_verify_strict_internal(signature.bytes, public_key.bytes, bcs::to_bytes(&encoded))
}
```

(3) k-of-N EdDSA multi-signature verification based on Ed25519 elliptic curve



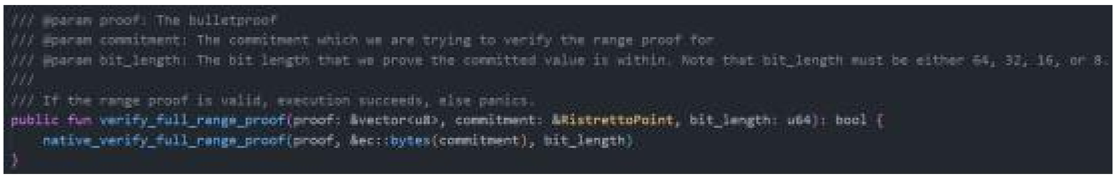
```
/// Verifies a purported MultiEd25519 `multisignature` under an *unvalidated* `public_key` on the specified `message`.
/// This call will validate the public key by checking it is NOT in the small subgroup.
public fun signature_verify_strict(
    multisignature: &Signature,
    public_key: &UnvalidatedPublicKey,
    message: vector<u8>
): bool {
    signature_verify_strict_internal(multisignature.bytes, public_key.bytes, message)
}

/// This function is used to verify a multi-signature on any BCS-serializable type T. For now, it is used to verify the
/// proof of private key ownership when rotating authentication keys.
public fun signature_verify_strict_t<T: drop>(multisignature: &Signature, public_key: &UnvalidatedPublicKey, data: T): bool {
    let encoded = ed25519::new_signed_message(data);

    signature_verify_strict_internal(multisignature.bytes, public_key.bytes, bcs::to_bytes(&encoded))
}
```

(4) Verification of bls12–381 aggregate signature, multi-signature signature and single-signature signature

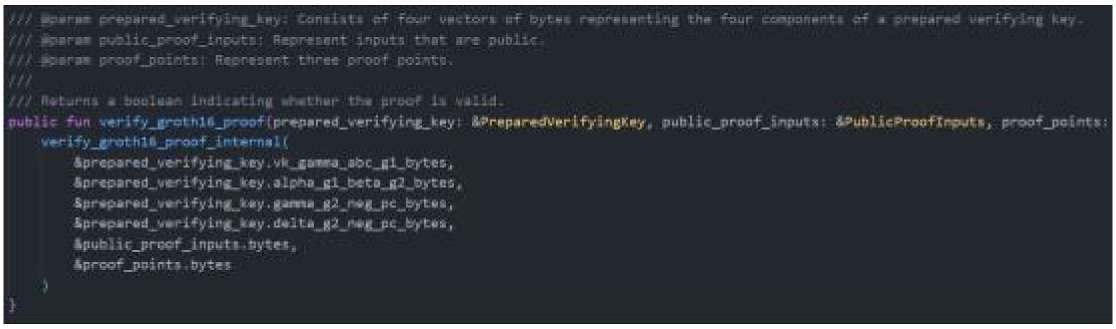


```
/// Verifies an aggregate signature, an aggregation of many signatures `s_i`, each on a different message `m_i`.
public fun verify_aggregate_signature(
    aggr_sig: &AggrOrMultiSignature,
    public_keys: vector<PublicKeyWithPoP>,
    messages: vector<vector<u8>>,
): bool {
    verify_aggregate_signature_internal(aggr_sig.bytes, public_keys, messages)
}

/// Verifies a multisignature: an aggregation of many signatures, each on the same message `m`.
public fun verify_multisignature(
    multisig: &AggrOrMultiSignature,
    aggr_public_key: &AggrPublicKeysWithPoP,
    message: vector<u8>
): bool {
    verify_multisignature_internal(multisig.bytes, aggr_public_key.bytes, message)
}

/// Verifies a normal, non-aggregated signature.
public fun verify_normal_signature(
    signature: &Signature,
    public_key: &PublicKey,
    message: vector<u8>
): bool {
    verify_normal_signature_internal(signature.bytes, public_key.bytes, message)
}
```

The Sui public chain Move contract also supports the above 4 types of signature verification, in addition to range proof verification and Groth16 zero-knowledge proof verification.



```
/// @param proof: The bulletproof
/// @param commitment: The commitment which we are trying to verify the range proof for
/// @param bit_length: The bit length that we prove the committed value is within. Note that bit_length must be either 64, 32, 16, or 8.
///
/// If the range proof is valid, execution succeeds, else panics.
public fun verify_full_range_proof(proof: &vector<u8>, commitment: &RistrettoPoint, bit_length: u64): bool {
    native_verify_full_range_proof(proof, &ec::bytes(commitment), bit_length)
}
```



```
/// @param prepared_verifying_key: Consists of four vectors of bytes representing the four components of a prepared verifying key.
/// @param public_proof_inputs: Represent inputs that are public.
/// @param proof_points: Represent three proof points.
///
/// Returns a boolean indicating whether the proof is valid.
public fun verify_groth16_proof(prepared_verifying_key: &PreparedVerifyingKey, public_proof_inputs: &PublicProofInputs, proof_points:
    verify_groth16_proof_internal(
        &prepared_verifying_key.vk_gamma_abc_g1_bytes,
        &prepared_verifying_key.alpha_g1_beta_g2_bytes,
        &prepared_verifying_key.gamma_g2_neg_pc_bytes,
        &prepared_verifying_key.delta_g2_neg_pc_bytes,
        &public_proof_inputs.bytes,
        &proof_points.bytes
    )
}
```

Move public chain supports multiple signature verification mechanisms to facilitate the cross-chain transfer of assets at the contract level, which is beneficial to the occurrence of Move ecology. When using signatures to achieve cross-chain, it is also necessary to strictly prevent signature replay attacks, especially the interaction between on-chain and off-chain data and the signature and verification at the business contract level, which should be treated with caution to avoid replay attacks on business data signatures at the contract level of the same chain or cross-chain.

About SharkTeam

Our vision is to improve security globally. We believe that by building this security barrier, we can significantly improve lives around the world.SharkTeam composes of members with many years of cyber security experiences and blockchain, team members are based in Suzhou, Beijing, Nanjing and Silicon Valley, proficient in the underlying theories of blockchain and smart contracts, and we provide comprehensive services including threat modeling, smart contract auditing, emergency response, etc. SharkTeam has established strategic and long-term cooperations with key players in many areas of the blockchain ecosystem, such as Huobi Global, OKX, polygon, Polkadot, imToken, ChainIDE, etc

Twitter：https://twitter.com/sharkteamorg

Discord：https://discord.gg/jGH9xXCjDZ

Telegram：https://t.me/sharkteamorg

web：https://www.sharkteam.org/

SharkTeam

In Math, We Trust!