Session Validation Module

Diving into theSession Validation Module , we explore its significance and interaction with theSession Key Manager Module via SDK.

note Understanding theSession Validation Modules is crucial for leveragingsession keys effectively in blockchain applications.

The Purpose of Session Validation Modules

At the core, a Session Validation Module is a smart contract designed to authenticate whether a user's operation complies with the permissions set within a session key. It functions to validate user operations based on pre-defined session key permissions.

info Key Functionality : We'll dissect a deployed contract that validates permissions any type of smart contract logic, enabling dApps to execute transactions without user signatures every time. Check the contracthere .

Breaking Down the Contract

The smart contract we focus on is structured to validate user operations (userOps) for any smart contract using session key signatures. It's not tailored for a specific type of smart contract, we can declare rules and permissions for any contract method.

warning Technical Deep Dive : The following contract breakdown is technical in nature, aimed at developers with a solid understanding of smart contract functionalities. // SPDX-License-Identifier: MIT pragma solidity ^ 0.8 .23 ; import

{ ECDSA }

from

"@openzeppelin/contracts/utils/cryptography/ECDSA.sol" ; import

"./ISessionValidationModule.sol" ;

/ *@title ABI Session Validation Module for Biconomy Smart Accounts. * @dev Validates userOps for any contract / method / params. * The _sessionKeyData layout: * Offset (in bytes) | Length (in bytes) | Contents * 0x0 | 0x14 | Session key (address) * 0x14 | 0x14 | Permitted destination contract (address) * 0x28 | 0x4 | Permitted selector (bytes4) * 0x2c | 0x10 | Permitted value limit (uint128) * 0x3c | 0x2 | Rules list length (uint16) * 0x3e + 0x23N | 0x23 | Rule #N * * Rule layout: * Offset (in bytes) | Length (in bytes) | Contents * 0x0 | 0x2 | Offset (uint16) * 0x2 | 0x1 | Condition (uint8) * 0x3 | 0x20 | Value (bytes32) * * Condition is a uint8, and can be one of the following: * 0: EQUAL * 1: LESS_THAN_OR_EQUAL * 2: LESS_THAN * 3: GREATER_THAN_OR_EQUAL * 4: GREATER_THAN * 5: NOT_EQUAL * * Inspired by https://github.com/zerodevapp/kernel/blob/main/src/validator/SessionKeyValidator.sol / contract ABISessionValidationModule is ISessionValidationModule { uint256 private constant RULE_LENGTH

=

35 ; uint256 private constant SELECTOR_LENGTH

=

4 ;

/** * @dev validates if the _op (UserOperation) matches the SessionKey permissions * and that _op has been signed by this SessionKey * Please mind the decimals of your exact token when setting maxAmount * @param

_op User Operation to be validated. * @param

_userOpHash Hash of the User Operation to be validated. * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * @param

_sessionKeySignature Signature over the the _userOpHash. * @return true if the _op is valid, false otherwise. */ function

validateSessionUserOp (UserOperation calldata _op , bytes32 _userOpHash , bytes calldata _sessionKeyData , bytes calldata _sessionKeySignature) external pure override returns

(bool)

{ bytes calldata callData = _op . callData ;

require (bytes4 (callData [0 : 4])

==

EXECUTE_OPTIMIZED_SELECTOR

|| bytes4 (callData [0 : 4])

==

EXECUTE_SELECTOR , "ABISV Not Execute Selector") ;

uint160 destContract ; uint256 callValue ; bytes calldata data ; assembly { //offset of the first 32-byte arg is 0x4 destContract : =

calldataload (add (callData . offset ,

SELECTOR_LENGTH)) //offset of the second 32-byte arg is 0x24 = 0x4 (SELECTOR_LENGTH) + 0x20 (first 32-byte arg) callValue : =

calldataload (add (callData . offset ,

0x24))

//we get the data offset from the calldata itself, so no assumptions are made about the data layout let dataOffset : =

add (add (callData . offset ,

0x04) , //offset of the bytes arg is stored after selector and two first 32-byte args // 0x4+0x20+0x20=0x44 calldataload (add (callData . offset ,

0x44)))

let length : =

calldataload (dataOffset) //data itself starts after the length which is another 32bytes word, so we add 0x20 data . offset : =

add (dataOffset ,

0x20) data . length : = length }

return _validateSessionParams (address (destContract) , callValue , data , _sessionKeyData)

== ECDSA . recover (ECDSA . toEthSignedMessageHash (_userOpHash) , _sessionKeySignature) ; }

/** * @dev validates that the call (destinationContract, callValue, funcCallData) * complies with the Session Key permissions represented by sessionKeyData * @param

destinationContract address of the contract to be called * @param

callValue value to be sent with the call * @param

_funcCallData the data for the call. is parsed inside the SVM * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * param _callSpecificData additional data, specific to the call, not used here * @return sessionKey address of the sessionKey that signed the userOp * for example to store a list of allowed tokens or receivers */ function

validateSessionParams (address destinationContract , uint256 callValue , bytes calldata _funcCallData , bytes calldata _sessionKeyData , bytes memory /*_callSpecificData*/) external pure virtual override returns

(address)

{ return _validateSessionParams (destinationContract , callValue , _funcCallData , _sessionKeyData) ; }

/** * @dev validates that the call (destinationContract, callValue, funcCallData) * complies with the Session Key permissions represented by sessionKeyData * @param

destinationContract address of the contract to be called * @param

callValue value to be sent with the call * @param

_funcCallData the data for the call. is parsed inside the SVM * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * @return sessionKey address of the sessionKey that signed the userOp * for example to store a list of allowed tokens or receivers */ function

_validateSessionParams (address destinationContract , uint256 callValue , bytes calldata _funcCallData , bytes calldata _sessionKeyData) internal pure virtual returns

(address)

{ // every address is 20bytes address sessionKey =

address (bytes20 (_sessionKeyData [0 : 20])) ; address permittedDestinationContract =

address (bytes20 (_sessionKeyData [20 : 40])) ; // every selector is 4bytes bytes4 permittedSelector =

bytes4 (_sessionKeyData [40 : 44]) ; // value limit is encoded as uint128 which is 16 bytes length uint256 permittedValueLimit =

uint256 (uint128 (bytes16 (_sessionKeyData [44 : 60]))) ; // rules list length is encoded as uint16 which is 2 bytes length uint256 rulesListLength =

uint256 (uint16 (bytes2 (_sessionKeyData [60 : 62]))) ;

if

(destinationContract != permittedDestinationContract)

{ revert ("ABISV Destination Forbidden") ; }

if

(bytes4 (_funcCallData [0 : 4])

!= permittedSelector)

{ revert ("ABISV Selector Forbidden") ; }

if

(callValue

permittedValueLimit)

{ revert ("ABISV Permitted Value Exceeded") ; }

// avoided explicit check that (_sessionKeyData.length - 62) is the multiple of RULE_LENGTH // also avoided calculating the rules list length from the rules list itself // both to save on gas // there is a test case that demonstrates that if the incorrect rules list length is provided // the validation will fail if

(! _checkRulesForPermission (_funcCallData , rulesListLength , bytes (_sessionKeyData [62 :])

//the rest of the _sessionKeyData is the rules list))

{ revert ("ABISV Arg Rule Violated") ; }

return sessionKey ; }

/** * @dev checks if the calldata matches the permission * @param

data the data for the call. is parsed inside the SVM * @param

rulesListLength the length of the rules list * @param

rules the rules list * @return true if the calldata matches the permission, false otherwise */ function

_checkRulesForPermission (bytes calldata data , uint256 rulesListLength , bytes calldata rules) internal pure returns

(bool)

{ for

(uint256 i ; i < rulesListLength ;

```
++ i )
{ ( uint256 offset , uint256 condition , bytes32 value )
=
_parseRule ( rules , i ) ;
// get the 32bytes word to verify against reference value from the actual calldata of the userOp bytes32 param =
bytes32 ( data [ SELECTOR_LENGTH
+ offset : SELECTOR_LENGTH
+ offset +
32 ] ) ;
bool rulePassed ; assembly
( "memory-safe" )
{ switch condition case
0
{ // Condition.EQUAL rulePassed : =
eq ( param , value ) } case
1
{ // Condition.LESS_THAN_OR_EQUAL rulePassed : =
or ( lt ( param , value ) ,
eq ( param , value ) ) } case
2
{ // Condition.LESS_THAN rulePassed : =
lt ( param , value ) } case
3
{ // Condition.GREATER_THAN_OR_EQUAL rulePassed : =
or ( gt ( param , value ) ,
eq ( param , value ) ) } case
4
{ // Condition.GREATER_THAN rulePassed : =
gt ( param , value ) } case
5
{ // Condition.NOT_EQUAL rulePassed : =
iszero ( eq ( param , value ) ) } }
if
( ! rulePassed )
{ return
false ; } } return
true ; }
```

/** * @dev Parses a rule with a given index from the rules list * @param

rules the rules list as a bytes array * @param

index the index of the rule to be parsed * @return offset - the offset of the parameter in the calldata (multiplier of 32) * @return condition - the condition to be checked * @return value - the reference value to be checked against */ function

_parseRule (bytes calldata rules , uint256 index) internal pure returns

(uint256 offset , uint256 condition , bytes32 value)

{ // offset length is 2 bytes offset =

uint256 (uint16 (bytes2 (rules [index *

RULE_LENGTH : index *

RULE_LENGTH

+

2]))) ; // condition length is 1 byte condition =

uint256 (uint8 (bytes1 (rules [index *

RULE_LENGTH

+

2 : index *

RULE_LENGTH

+

3]))) ; // value length is 32 bytes value =

bytes32 (rules [index *

RULE_LENGTH

+

3 : index *

RULE_LENGTH

+

RULE_LENGTH]) ; } } The contract, extending theISessionValidationModule interface, contains essential functions likevalidateSessionUserOp andvalidateSessionParams , each serving distinct roles in operation validation.

Solidity Contract Breakdown

Here's the Solidity contract in question:

Function Analysis:validateSessionUserOp

note This function is essential forvalidating user operations againstsession key permissions and ensuring they are correctly signed. /** * @dev validates if the _op (UserOperation) matches the SessionKey permissions * and that _op has been signed by this SessionKey * Please mind the decimals of your exact token when setting maxAmount * @param

_op User Operation to be validated. * @param

_userOpHash Hash of the User Operation to be validated. * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * @param

_sessionKeySignature Signature over the the _userOpHash. * @return true if the _op is valid, false otherwise. */ function

validateSessionUserOp (UserOperation calldata _op , bytes32 _userOpHash , bytes calldata _sessionKeyData , bytes

calldata _sessionKeySignature) external pure override returns

(bool)

{ bytes calldata callData = _op . callData ;

require (bytes4 (callData [0 : 4])

==

EXECUTE_OPTIMIZED_SELECTOR

|| bytes4 (callData [0 : 4])

==

EXECUTE_SELECTOR , "ABISV Not Execute Selector") ;

uint160 destContract ; uint256 callValue ; bytes calldata data ; assembly { //offset of the first 32-byte arg is 0x4 destContract : =

calldataload (add (callData . offset ,

SELECTOR_LENGTH)) //offset of the second 32-byte arg is 0x24 = 0x4 (SELECTOR_LENGTH) + 0x20 (first 32-byte arg) callValue : =

calldataload (add (callData . offset ,

0x24))

//we get the data offset from the calldata itself, so no assumptions are made about the data layout let dataOffset : =

add (add (callData . offset ,

0x04) , //offset of the bytes arg is stored after selector and two first 32-byte args // 0x4+0x20+0x20=0x44 calldataload (add (callData . offset ,

0x44)))

let length : =

calldataload (dataOffset) //data itself starts after the length which is another 32bytes word, so we add 0x20 data . offset : =

add (dataOffset ,

0x20) data . length : = length }

return _validateSessionParams (address (destContract) , callValue , data , _sessionKeyData)

== ECDSA . recover (ECDSA . toEthSignedMessageHash (_userOpHash) , _sessionKeySignature) ; } Execution Steps:

1. Match Function Selectors:
2. This step ensures that the calldata of a given user operation calls the function with the allowed selectors only.
3. In this method, it checks whether the first four bytes of the call data match predefined selectors (EXECUTE_OPTIMIZED_SELECTOR or EXECUTE_SELECTOR).
4. Decode Session Key Data:
5. Here, the essential details from the session key data are extracted to understand the permissions and constraints associated with the user operation.
6. This includes retrieving information such as the destination contract, call value, and additional data from the call data provided in the user operation.
7. Verify Operation Details:
8. This step validates the details of the user operation against the permissions specified by the session key.
9. It checks whether the call data aligns with the permissions specified in the session key data.
10. Additionally, it ensures that any constraints or limits imposed by the session key, such as maximum transaction amounts or permitted recipients, are respected.
11. Signature Validation:
12. In this final step, the method confirms the authenticity of the operation by verifying its signature against the session key.
13. ECDSA (Elliptic Curve Digital Signature Algorithm) is used to validate that the provided signature matches the session key.
14. If the signature verification is successful, it indicates that the operation has been correctly signed by the session key.

Function Analysis:validateSessionParams

/** * @dev validates that the call (destinationContract, callValue, funcCallData) * complies with the Session Key permissions represented by sessionKeyData * @param

destinationContract address of the contract to be called * @param

callValue value to be sent with the call * @param

_funcCallData the data for the call. is parsed inside the SVM * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * param _callSpecificData additional data, specific to the call, not used here * @return sessionKey address of the sessionKey that signed the userOp * for example to store a list of allowed tokens or receivers */ function

validateSessionParams (address destinationContract , uint256 callValue , bytes calldata _funcCallData , bytes calldata _sessionKeyData , bytes memory /*_callSpecificData*/) external pure virtual override returns

(address)

{ return _validateSessionParams (destinationContract , callValue , _funcCallData , _sessionKeyData) ; }

/** * @dev validates that the call (destinationContract, callValue, funcCallData) * complies with the Session Key permissions represented by sessionKeyData * @param

destinationContract address of the contract to be called * @param

callValue value to be sent with the call * @param

_funcCallData the data for the call. is parsed inside the SVM * @param

_sessionKeyData SessionKey data, that describes sessionKey permissions * @return sessionKey address of the sessionKey that signed the userOp * for example to store a list of allowed tokens or receivers */ function

_validateSessionParams (address destinationContract , uint256 callValue , bytes calldata _funcCallData , bytes calldata _sessionKeyData) internal pure virtual returns

(address)

{ // every address is 20bytes address sessionKey =

address (bytes20 (_sessionKeyData [0 : 20])) ; address permittedDestinationContract =

address (bytes20 (_sessionKeyData [20 : 40])) ; // every selector is 4bytes bytes4 permittedSelector =

bytes4 (_sessionKeyData [40 : 44]) ; // value limit is encoded as uint128 which is 16 bytes length uint256 permittedValueLimit =

uint256 (uint128 (bytes16 (_sessionKeyData [44 : 60]))) ; // rules list length is encoded as uint16 which is 2 bytes length uint256 rulesListLength =

uint256 (uint16 (bytes2 (_sessionKeyData [60 : 62]))) ;

if

(destinationContract != permittedDestinationContract)

{ revert ("ABISV Destination Forbidden") ; }

if

(bytes4 (_funcCallData [0 : 4])

!= permittedSelector)

{ revert ("ABISV Selector Forbidden") ; }

if

(callValue

permittedValueLimit)

{ revert ("ABISV Permitted Value Exceeded") ; }

// avoided explicit check that (_sessionKeyData.length - 62) is the multiple of RULE_LENGTH // also avoided calculating the rules list length from the rules list itself // both to save on gas // there is a test case that demonstrates that if the incorrect rules list length is provided // the validation will fail if

(! _checkRulesForPermission (_funcCallData , rulesListLength , bytes (_sessionKeyData [62 :])

//the rest of the _sessionKeyData is the rules list))

{ revert ("ABISV Arg Rule Violated") ; }

return sessionKey ; } Operational Flow:

The method internally calls_validateSessionParams to perform the validation against the session permissions. It passes the necessary parameters to_validateSessionParams and returns the result.

_validateSessionParams validates the parameters of the call against the permissions defined for the session key.

Execution flow in_validateSessionParams

- Extracts essential details from _sessionKeyData, such as session key address, permitted destination contract, permitted method selector, value limit, and rules list length.
- Checks if the destinationContract matches the permitted destination contract, if the function selector in _funcCallData matches the permitted selector, and if the callValue does not exceed the permitted value limit.
- Calls_checkRulesForPermission
- to verify if the call data complies with the permission rules.

Execution flow in_checkRulesForPermission

- Iterates through the list of rules.
- Parses each rule to determine the offset, condition, and value.
- Verifies if the call data arguments satisfy conditions defined by the rules.

Rules

Rules define permissions for the args of an allowed method. With Rules you can precisely define what should be the args of the transaction that is allowed for a given Session. Every Rule works with a single static arg or a 32 bytes chunk of the dynamic arg.

Since the ABI Encoding translates every static param into a 32bytes word, even the shorter ones (likeaddress oruint8), every Rule defines a desired relation (Condition) between n-th 32bytes word of thecalldata and a reference Value (that is obviously a 32bytes word as well).

So, when dApp is creating a_sessionKeyData to enable a session, it should convert every shorter static arg to a 32bytes word to match how it will be actually ABI encoded in theuserOp.callData .

For the dynamic args, likebytes , every 32bytes word of thecalldata such as offset of the bytes arg, length of the bytes arg, and n-th 32bytes word of the bytes arg can be controlled by a dedicated Rule.

Offset

The offset in the ABI SVM contract helps locate the relevant data within the function call data, it serves as a reference point from which to start reading or extracting specific information required for validation. When processing function call data, particularly in low-level languages like Solidity assembly, it's necessary to locate where specific parameters or arguments are stored. The offset is used to calculate the starting position within the calldata where the desired data resides. Suppose we have a function call with multiple arguments passed as calldata. Each argument occupies a certain number of bytes, and the offset helps determine where each argument begins within the calldata.

Using the offset to Extract Data: In the contract, the offset is used to calculate the position within the calldata where specific parameters or arguments are located. Since every arg is a 32-bytes word, offsets are always multiplier of 32 (or of 0x20 in hex).

Let's see how the offset is applied to extract the to and value arguments of a transfer(address to, uint256 value) method:

Extracting to Argument: The to argument is the first parameter of the transfer function, representing the recipient address. Every calldata starts with the 4-bytes method selector. However, the ABI SVM is adding the selector length itself, so for the first argument the offset will always be 0 (0x00);

Extracting value Argument: The value argument is the second parameter of the transfer function, representing the amount of tokens to be transferred. To extract this argument, the offset for the value parameter would be calculated based on its position in the function calldata. Despite to is a 20-bytes address, in the solidity abi encoding it is always appended with zeroes to a 32-bytes word. So the offset for the second 32-bytes argument (which is value in our case) will be 32 (or 0x20 in hex).

If you need to deal with dynamic-length arguments, such as bytes, please refer to this documenthttps://docs.soliditylang.org/en/v0.8.24/abi-spec.html#function-selector-and-argument-encoding to learn more about how dynamic arguments are represented in the calldata and which offsets should be used to access them.

Condition

The condition is used to determine how we are checking the actual reference value, the condition can be of many types:

- 0: EQUAL
- 1: LESS_THAN_OR_EQUAL
- 2: LESS_THAN
- 3: GREATER_THAN_OR_EQUAL
- 4: GREATER_THAN
- 5: NOT_EQUAL
- In our example the condition is 0, this means we check that the receiver of the NFT is EQUAL to what we set it to be.

Value

This is the reference value. The actual arg value is decoded from the userOp.callData using the Offset. Then it is compared to the reference value using the Condition.

BothvalidateSessionUserOp andvalidateSessionParams are integral to our dApp's security framework, ensuring strict adherence to permissions and enhancing transaction integrity.

Next Steps

With a foundational understanding of the Session Validation Module, we're set to move forward. Up next, we'll embark on initializing the frontend and integrating the Biconomy SDK, crucial steps in bringing our dApp to life.

tip Explore the Interface : Familiarize yourself with theISessionValidationModule interfacehere for a comprehensive understanding. Previous Introduction Next Initialize Frontend