

release-0.2.0 ▾

...

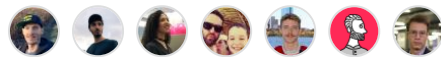
[cairo-contracts](#) / [src](#) / [openzeppelin](#) / [account](#) / [library.cairo](#) / <> Jump to ▾



martriay Bump SPDX Licence ids to 0.2.0 (#376) ... ✓

History

7 contributors



333 lines (282 sloc) | 9.57 KB

...

```

1  # SPDX-License-Identifier: MIT
2  # OpenZeppelin Contracts for Cairo v0.2.0 (account/library.cairo)
3
4  %lang starknet
5
6  from starkware.cairo.common.registers import get_fp_and_pc
7  from starkware.starknet.common.syscalls import get_contract_address
8  from starkware.cairo.common.signature import verify_ecdsa_signature
9  from starkware.cairo.common.cairo_builtins import HashBuiltin, SignatureBuiltin, BitwiseBuiltin
10 from starkware.cairo.common.alloc import alloc
11 from starkware.cairo.common.uint256 import Uint256
12 from starkware.cairo.common.memcpy import memcpy
13 from starkware.cairo.common.math import split_felt
14 from starkware.cairo.common.bool import TRUE
15 from starkware.starknet.common.syscalls import call_contract, get_caller_address, get_tx_info
16 from starkware.cairo.common.cairo_secp.signature import verify_eth_signature_uint256
17 from openzeppelin.introspection.ERC165 import ERC165
18
19 from openzeppelin.utils.constants import IACCOUNT_ID
20
21 #
22 # Storage
23 #
24
25 @storage_var
26 func Account_current_nonce() -> (res: felt):
27
28 end

```

```

29 @storage_var
30 func Account_public_key() -> (res: felt):
31 end
32
33 #
34 # Structs
35 #
36
37 struct Call:
38     member to: felt
39     member selector: felt
40     member calldata_len: felt
41     member calldata: felt*
42 end
43
44 # Tmp struct introduced while we wait for Cairo
45 # to support passing `[AccountCall]` to __execute__
46 struct AccountCallArray:
47     member to: felt
48     member selector: felt
49     member data_offset: felt
50     member data_len: felt
51 end
52
53 namespace Account:
54
55     #
56     # Initializer
57     #
58
59     func initializer{
60         syscall_ptr : felt*,
61         pedersen_ptr : HashBuiltin*,
62         range_check_ptr
63     }(_public_key: felt):
64         Account_public_key.write(_public_key)
65         ERC165.register_interface(IACCOUNT_ID)
66         return()
67     end
68
69     #
70     # Guards
71     #
72
73     func assert_only_self{syscall_ptr : felt*}():
74         let (self) = get_contract_address()
75         let (caller) = get_caller_address()
76         with_attr error_message("Account: caller is not this account"):
77             assert self = caller

```

```

78         end
79         return ()
80     end
81
82     #
83     # Getters
84     #
85
86     func get_public_key{
87         syscall_ptr : felt*,
88         pedersen_ptr : HashBuiltin*,
89         range_check_ptr
90     }() -> (res: felt):
91         let (res) = Account_public_key.read()
92         return (res=res)
93     end
94
95     func get_nonce{
96         syscall_ptr : felt*,
97         pedersen_ptr : HashBuiltin*,
98         range_check_ptr
99     }() -> (res: felt):
100         let (res) = Account_current_nonce.read()
101         return (res=res)
102     end
103
104     #
105     # Setters
106     #
107
108     func set_public_key{
109         syscall_ptr : felt*,
110         pedersen_ptr : HashBuiltin*,
111         range_check_ptr
112     }(new_public_key: felt):
113         assert_only_self()
114         Account_public_key.write(new_public_key)
115         return ()
116     end
117
118     #
119     # Business logic
120     #
121
122     func is_valid_signature{
123         syscall_ptr : felt*,
124         pedersen_ptr : HashBuiltin*,
125         range_check_ptr,
126         ecdsa_ptr: SignatureBuiltin*

```

```

127     }(
128         hash: felt,
129         signature_len: felt,
130         signature: felt*
131     ) -> (is_valid: felt):
132     let (_public_key) = Account_public_key.read()
133
134     # This interface expects a signature pointer and length to make
135     # no assumption about signature validation schemes.
136     # But this implementation does, and it expects a (sig_r, sig_s) pair.
137     let sig_r = signature[0]
138     let sig_s = signature[1]
139
140     verify_ecdsa_signature(
141         message=hash,
142         public_key=_public_key,
143         signature_r=sig_r,
144         signature_s=sig_s)
145
146     return (is_valid=TRUE)
147 end
148
149 func is_valid_eth_signature{
150     syscall_ptr : felt*,
151     pedersen_ptr : HashBuiltin*,
152     bitwise_ptr: BitwiseBuiltin*,
153     range_check_ptr
154 }(
155     hash: felt,
156     signature_len: felt,
157     signature: felt*
158 ) -> (is_valid: felt):
159     alloc_locals
160     let (_public_key) = get_public_key()
161     let (__fp__, _) = get_fp_and_pc()
162
163     # This interface expects a signature pointer and length to make
164     # no assumption about signature validation schemes.
165     # But this implementation does, and it expects a the sig_v, sig_r,
166     # sig_s, and hash elements.
167     let sig_v : felt = signature[0]
168     let sig_r : Uint256 = Uint256(low=signature[1], high=signature[2])
169     let sig_s : Uint256 = Uint256(low=signature[3], high=signature[4])
170     let (high, low) = split_felt(hash)
171     let msg_hash : Uint256 = Uint256(low=low, high=high)
172
173     let (local keccak_ptr : felt*) = alloc()
174
175     with keccak_ptr:

```

```

176         verify_eth_signature_uint256(
177             msg_hash=msg_hash,
178             r=sig_r,
179             s=sig_s,
180             v=sig_v,
181             eth_address=_public_key)
182     end
183
184     return (is_valid=TRUE)
185 end
186
187 func execute{
188     syscall_ptr : felt*,
189     pedersen_ptr : HashBuiltin*,
190     range_check_ptr,
191     bitwise_ptr: BitwiseBuiltin*
192 }(
193     call_array_len: felt,
194     call_array: AccountCallArray*,
195     calldata_len: felt,
196     calldata: felt*,
197     nonce: felt
198 ) -> (response_len: felt, response: felt*):
199     alloc_locals
200
201     let (__fp__, _) = get_fp_and_pc()
202     let (tx_info) = get_tx_info()
203     let (local ecdsa_ptr : SignatureBuiltin*) = alloc()
204     with ecdsa_ptr:
205         # validate transaction
206         with_attr error_message("Account: invalid signature"):
207             let (is_valid) = is_valid_signature(tx_info.transaction_hash, tx_info.signature_le
208             assert is_valid = TRUE
209         end
210     end
211
212     return _unsafe_execute(call_array_len, call_array, calldata_len, calldata, nonce)
213 end
214
215 func eth_execute{
216     syscall_ptr : felt*,
217     pedersen_ptr : HashBuiltin*,
218     range_check_ptr,
219     bitwise_ptr: BitwiseBuiltin*
220 }(
221     call_array_len: felt,
222     call_array: AccountCallArray*,
223     calldata_len: felt,
224     calldata: felt*,

```

```

225         nonce: felt
226     ) -> (response_len: felt, response: felt*):
227     alloc_locals
228
229     let (__fp__, _) = get_fp_and_pc()
230     let (tx_info) = get_tx_info()
231
232     # validate transaction
233     with_attr error_message("Account: invalid secp256k1 signature"):
234         let (is_valid) = is_valid_eth_signature(tx_info.transaction_hash, tx_info.signature_le
235         assert is_valid = TRUE
236     end
237
238     return _unsafe_execute(call_array_len, call_array, calldata_len, calldata, nonce)
239 end
240
241 func _unsafe_execute{
242     syscall_ptr : felt*,
243     pedersen_ptr : HashBuiltin*,
244     range_check_ptr,
245     bitwise_ptr: BitwiseBuiltin*
246 }(
247     call_array_len: felt,
248     call_array: AccountCallArray*,
249     calldata_len: felt,
250     calldata: felt*,
251     nonce: felt
252 ) -> (response_len: felt, response: felt*):
253     alloc_locals
254
255     let (caller) = get_caller_address()
256     with_attr error_message("Account: no reentrant call"):
257         assert caller = 0
258     end
259
260     # validate nonce
261
262     let (_current_nonce) = Account_current_nonce.read()
263
264     with_attr error_message("Account: nonce is invalid"):
265         assert _current_nonce = nonce
266     end
267
268     # bump nonce
269     Account_current_nonce.write(_current_nonce + 1)
270
271     # TMP: Convert `AccountCallArray` to 'Call'.
272     let (calls : Call*) = alloc()
273     _from_call_array_to_call(call_array_len, call_array, calldata, calls)

```

```

274     let calls_len = call_array_len
275
276     # execute call
277     let (response : felt*) = alloc()
278     let (response_len) = _execute_list(calls_len, calls, response)
279
280     return (response_len=response_len, response=response)
281 end
282
283 func _execute_list{syscall_ptr: felt*}(
284     calls_len: felt,
285     calls: Call*,
286     response: felt*
287 ) -> (response_len: felt):
288     alloc_locals
289
290     # if no more calls
291     if calls_len == 0:
292         return (0)
293     end
294
295     # do the current call
296     let this_call: Call = [calls]
297     let res = call_contract(
298         contract_address=this_call.to,
299         function_selector=this_call.selector,
300         calldata_size=this_call.calldata_len,
301         calldata=this_call.calldata
302     )
303     # copy the result in response
304     memcpy(response, res.retdat, res.retdat_size)
305     # do the next calls recursively
306     let (response_len) = _execute_list(calls_len - 1, calls + Call.SIZE, response + res.retdat)
307     return (response_len + res.retdat_size)
308 end
309
310 func _from_call_array_to_call{syscall_ptr: felt*}(
311     call_array_len: felt,
312     call_array: AccountCallArray*,
313     calldata: felt*,
314     calls: Call*
315 ):
316     # if no more calls
317     if call_array_len == 0:
318         return ()
319     end
320
321     # parse the current call
322     assert [calls] = Call(

```

```
323         to=[call_array].to,
324         selector=[call_array].selector,
325         calldata_len=[call_array].data_len,
326         calldata=calldata + [call_array].data_offset
327     )
328     # parse the remaining calls recursively
329     _from_call_array_to_call(call_array_len - 1, call_array + AccountCallArray.SIZE, calldata,
330     return ()
331 end
332
333 end
```

