

How to use agents to verify messages

Introduction

The emergence of decentralized technologies has introduced new possibilities for secure communication and data exchange. In this guide, we will delve into the process of setting up a scenario where two Agents communicate with each other, employing cryptographic methods to verify the messages exchanged between them. We will showcase how to create a secure messaging environment using Agents, where messages are not only exchanged but also signed and verified to prevent unauthorized access and tampering.

Walk-through

1. First of all, you need to navigate towards the directory you created for your project.
2. In here, let's create a Python script for this task and name it:touch message_verification.py
3. .
4. We now need to import the necessary classes fromuagents
5. (Agent
6. ,Bureau
7. ,Context
8. , andModel
9.),uagents.crypto
10. (Identity
11.) andhashlib
12. . Then we would need to define the messages format using theMessage
13. class as a subclass ofModel
14. :
15. import
16. hashlib
17. from
18. uagents
19. import
20. Agent
21. ,
22. Bureau
23. ,
24. Context
25. ,
26. Model
27. from
28. uagents
29. .
30. crypto
31. import
32. Identity
33. class
34. Message
35. (
36. Model
37.):
38. message
39. :
40. str
41. digest
42. :
43. str
44. signature
45. :
46. str
47. The message format has three attributes:
48.
 - message
49.
 - : a string representing the message text.
50.
 - digest
- 51.

- : a string representing the SHA-256 hash of the message.

52.

- signature

53.

- : a string representing the digital signature of the hash using the sender's private key.

54. Let's now define anencode()
55. function used to generate the digest for each message before it is signed:
56. def
57. encode
58. (
59. message
60. :
61. str
62.)
63. ->
64. bytes
65. :
66. hasher
67. =
68. hashlib
69. .
70. sha256
71. ()
72. hasher
73. .
74. update
75. (message.
76. encode
77. ())
78. return
79. hasher
80. .
81. digest
82.)
83. This function is used to hash a string message using the SHA-256 algorithm and return the resulting digest as bytes.
84. We can now proceed and create our agents using theAgent
85. class:
86. alice
87. =
88. Agent
89. (name
90. =
91. "alice"
92. , seed
93. =
94. "alice recovery password"
95.)
96. bob
97. =
98. Agent
99. (name
100. =
101. "bob"
102. , seed
103. =
104. "bob recovery password"
105.)
106. Let's now define asend_message()
107. function foralice
108. to send messages tobob
109. :
110. @alice
111. .
112. on_interval
113. (period
114. =
115. 3.0
116.)

```

117. async
118. def
119. send_message
120. (
121. ctx
122. :
123. Context):
124. msg
125. =
126. "hello there bob"
127. digest
128. =
129. encode
130. (msg)
131. await
132. ctx
133. .
134. send
135. (
136. bob.address,
137. Message
138. (message
139. =
140. msg, digest
141. =
142. digest.
143. hex
144. ()), signature
145. =
146. alice.
147. sign_digest
148. (digest)),
149. )
150. This function is decorated using the.on_interval()
151. decorator, which indicates that the function is called periodically every3.0
152. seconds to send messages tobob
153. 's address. It takes in a single argumentctx
154. . The function first creates a message,msg
155. , and computes its digest using theencode
156. function. The message is then sent tobob
157. using thectx.send()
158. method, along with thedigest
159. and asignature
160. of the digest using thealice.sign_digest()
161. function.
162. Let's then define analice_rx_message()
163. function used to receive and process messages sent bybob
164. :
165. @alice
166. .
167. on_message
168. (model
169. =
170. Message)
171. async
172. def
173. alice_rx_message
174. (
175. ctx
176. :
177. Context
178. ,
179. sender
180. :
181. str
182. ,
183. msg
184. :

```

```

185. Message):
186. assert
187. Identity
188. .
189. verify_digest
190. (
191. sender,
192. bytes
193. .
194. fromhex
195. (msg.digest), msg.signature
196. ),
197. "couldn't verify bob's message"
198. ctx
199. .
200. logger
201. .
202. info
203. (
204. "Bob's message verified!"
205. )
206. ctx
207. .
208. logger
209. .
210. info
211. (
212. f
213. "Received message from
214. {
215. sender
216. }
217. :
218. {
219. msg.message
220. }
221. "
222. )
223. This function is decorated using the.on_message()
224. , indicating that the function is triggered when a message is being received of typeMessage
225. . The function takes in three arguments:ctx
226. ,sender
227. , andmsg
228. .
229. The function first verifies the authenticity of the message using theIdentity.verify_digest()
230. function. If the message cannot be verified, the function raises an assertion error. Assuming the message is verified,
    the function logs a message indicating that the message was verified and another message indicating the contents of
    the message.
231. We can now define abob_rx_message()
232. function used bybob
233. to receive and process messages sent byalice
234. :
235. @bob
236. .
237. on_message
238. (model
239. =
240. Message)
241. async
242. def
243. bob_rx_message
244. (
245. ctx
246. :
247. Context
248. ,
249. sender
250. :

```

```
251. str
252. ,
253. msg
254. :
255. Message):
256. assert
257. Identity
258. .
259. verify_digest
260. (
261. sender,
262. bytes
263. .
264. fromhex
265. (msg.digest), msg.signature
266. ),
267. "couldn't verify alice's message"
268. ctx
269. .
270. logger
271. .
272. info
273. (
274. "Alice's message verified!"
275. )
276. ctx
277. .
278. logger
279. .
280. info
281. (
282. f
283. "Received message from
284. {
285. sender
286. }
287. :
288. {
289. msg.message
290. }
291. "
292. )
293. msg
294. =
295. "hello there alice"
296. digest
297. =
298. encode
299. (msg)
300. await
301. ctx
302. .
303. send
304. (
305. alice.address,
306. Message
307. (message
308. =
309. msg, digest
310. =
311. digest.
312. hex
313. ()), signature
314. =
315. bob.
316. sign_digest
317. (digest)),
318. )
```

```

319. This function is decorated using the.on_message()
320. decorator, indicating that the function is triggered when a message is being received of typeMessage
321. . It takes in three arguments:ctx
322. ,sender
323. , andmsg
324. .
325. The function first verifies the authenticity of the message using theidentity.verify_digest()
326. function. If the message cannot be verified, the function raises an assertion error. On the other hand, if the message is
    verified, the function logs a message indicating that the message was verified and another message indicating the
    contents of the message using thectx.logger.info()
327. method. It then creates a response message,msg
328. , and computes its digest using theencode()
329. function. The response message is then sent toalice
330. using thectx.send()
331. method.
332. We can now create abureau
333. object from theBureau
334. class and then add both agents to it so for them to be run together.
335. bureau
336. =
337. Bureau
338. ()
339. bureau
340. .
341. add
342. (alice)
343. bureau
344. .
345. add
346. (bob)
347. if
348. name
349. ==
350. "main"
351. :
352. bureau
353. .
354. run
355. ()
356. Save the script.

```

The overall script should look as follows:

```

message_verification.py import hashlib from uagents import Agent , Bureau , Context , Model from uagents . crypto import
Identity class

```

```

Message ( Model ): message :

```

```

str digest :

```

```

str signature :

```

```

str

```

```

def

```

```

encode ( message :

```

```

str ) ->

```

```

bytes : hasher = hashlib . sha256 () hasher . update (message. encode ()) return hasher . digest ()

```

alice

```

Agent (name = "alice" , seed = "alice recovery password" ) bob =

```

```

Agent (name = "bob" , seed = "bob recovery password" )

```

```

@alice . on_interval (period = 3.0 ) async

```

```

def
send_message ( ctx : Context): msg =
"hello there bob" digest =
encode (msg)
await ctx . send ( bob.address, Message (message = msg, digest = digest. hex (), signature = alice. sign_digest (digest)), )
@alice . on_message (model = Message) async

def
alice_rx_message ( ctx : Context ,
sender :
str ,
msg : Message): assert Identity . verify_digest ( sender, bytes . fromhex (msg.digest), msg.signature ),
"couldn't verify bob's message"
ctx . logger . info ( "Bob's message verified!" ) ctx . logger . info ( f "Received message from { sender } : { msg.message } " )
@bob . on_message (model = Message) async

def
bob_rx_message ( ctx : Context ,
sender :
str ,
msg : Message): assert Identity . verify_digest ( sender, bytes . fromhex (msg.digest), msg.signature ),
"couldn't verify alice's message"
ctx . logger . info ( "Alice's message verified!" ) ctx . logger . info ( f "Received message from { sender } : { msg.message } " )

```

msg

```

"hello there alice" digest =
encode (msg)
await ctx . send ( alice.address, Message (message = msg, digest = digest. hex (), signature = bob. sign_digest (digest)), )

```

bureau

```

Bureau () bureau . add (alice) bureau . add (bob)

if
name
==
"main" : bureau . run ()

```

Run your script

On your terminal, make sure to have activated your virtual environment.

Run the script: `python message_verification.py` .

The output should be as follows:

[bob]: Alice's message verified! [bob]: Received message from agent1qf5gfm48k9acegez3sg82ney2aa6l5fvpwh3n3z0ajh0nam3ssgwnn5me7: hello there bob [alice]: Bob's message verified! [alice]: Received message from agent1qvjjle8dlf22ff7zsh6wr3gl8tdepzygftdpc2vn8539ngt962a709c90s: hello there alice

Was this page helpful?

[Almanac Contract Agentverse: Hosted Agents](#)