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

Let's now define anencode() function used to generate the digest for each message before it is signed:

```
def encode ( message : str ) -> bytes :
    hasher = hashlib . sha256 ()
    hasher . update (message. encode ())
    return hasher . digest ()
```

This function is used to hash a string message using the SHA-256 algorithm and return the resulting digest as bytes.

We can now proceed and create our agents using theAgent class:

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

Let's now define asend_message() function foralice to send messages tobob :

```
@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)),
)
This function is decorated using the.on_interval()
decorator, which indicates that the function is called periodically every3.0
seconds to send messages tobob
's address. It takes in a single argumentctx
. The function first creates a message,msg
, and computes its digest using theencode
function. The message is then sent tobob
using thectx.send()
method, along with thedigest
and asignature
of the digest using thealice.sign_digest()
function.
Let's then define analice_rx_message()
function used to receive and process messages sent bybob
:
@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
}
"
)
This function is decorated using the.on_message()
, indicating that the function is triggered when a message is being received of typeMessage
. The function takes in three arguments:ctx
,sender
, andmsg
.
The function first verifies the authenticity of the message using theIdentity.verify_digest()
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.
We can now define abob_rx_message()
function used bybob
to receive and process messages sent byalice
:
@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)),
)
```

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 agent1qf5gfqm48k9acegez3sg82ney2aa6l5fvpwh3n3z0ajh0nam3ssgwnn5me7: hello there bob [alice]: Bob's message verified! [alice]: Received message from agent1qvjjle8dlf22ff7zsh6wr3gl8tdepzygftdxpc2vn8539ngt962a709c90s: hello there alice

Was this page helpful?

Almanac Contract Agentverse: Hosted Agents