PyWARP Documentation

Andrey Kislyuk

Feb 02, 2023

Contents

PyWARP is an implementation of the W3C WebAuthn standard's Relying Party component in Python. The WebAuthn standard is used to provide advanced authentication security for two-factor, multifactor and passwordless authentication models through the use of dedicated hardware security keys and biometric devices such as Yubico YubiKey, Google Titan, TPM, and Touch ID. PyWARP's design goal is to provide an ergonomic and intuitive API to guide the implementer with good defaults and trusted dependencies.

Compared to earlier two-factor standards like HOTP (RFC 4226) and TOTP (RFC 6238), the FIDO U2F profile of WebAuthn uses asymmetric cryptography to avoid using a shared secret design, which strengthens your authentication solution against server-side attacks. Hardware U2F also sequesters the client secret in a dedicated single-purpose device, which strengthens your clients against client-side attacks. And by automating scoping of credentials to relying party IDs (application origin/domain names), WebAuthn/U2F adds protection against phishing attacks.

PyWARP implements the *Relying Party* component of WebAuthn. A Relying Party is a server-side application that instructs the browser (user agent) to use WebAuthn APIs to authenticate its users.

To see an example of PyWARP in action, check the `examples` directory. Two demos are included: an AWS Chalice app and a Flask app.

In addition to reading the WebAuthn standard, we recommend that implementers read the OWASP Authentication Cheat Sheet and NIST SP 800-63-3: Digital Authentication Guideline for a high level overview of authentication best practices.

CHAPTER 1

Installation

```
pip install pywarp
```

PyWARP requires Python 3.6+. Python 2.7 and <= 3.5 is not supported.

PyWARP depends on cryptography, which in turn requires OpenSSL and CFFI. See the cryptography installation docs for more details.

CHAPTER 2

Synopsis

```
from pywarp import RelyingPartyManager, Credential
# Using DynamoDB as an example. See "storage backends" below for other databases.
from pywarp.backends import DynamoBackend

rp_id = "myapp.example.com"  # This must match the origin domain of your app, as seen␣
↪by the browser.
rp = RelyingPartyManager("PyWARP demo", rp_id=rp_id, credential_storage_
↪backend=DynamoBackend())

# Get options for navigator.credentials.create() - pass these to your frontend when␣
↪registering a user
opts = rp.get_registration_options(email=str)

# Run the protocol in https://www.w3.org/TR/webauthn/#registering-a-new-credential,
# then call the credential storage backend to store the credential public key.
rp.register(attestation_object=bytes, client_data_json=bytes, email=bytes)

# Get options for navigator.credentials.get() - pass these to your frontend when␣
↪logging in a user
opts = rp.get_authentication_options(email=str)

# Run the protocol in https://www.w3.org/TR/webauthn/#verifying-assertion,
# calling the credential storage backend to retrieve the credential public key.
# If no exception is raised, proceed with user login.
rp.verify(authenticator_data=bytes, client_data_json=bytes, signature=bytes, user_
↪handle=bytes, raw_id=bytes,
          email=bytes)
```

See examples/chalice/app.py and examples/chalice/chalicelib/index.html (frontend) for a complete example.

CHAPTER 3

Storage backends

Your application is presumably using an application server like uWSGI, a database backend like MySQL, PostgreSQL or MongoDB, and maybe a framework like Flask or Django to tie them together. PyWARP makes no assumptions about your database, schema, or model. Instead, it provides an abstract class (`pywarp.backends.CredentialStorageBackend`) representing an interface for storing and retrieving WebAuthn credential data for your users.

To deploy PyWARP, declare a subclass of `CredentialStorageBackend`. In your subclass, implement bindings to your database, then pass an instance of your subclass to `pywarp.RelyingPartyManager(credential_storage_backend=...)`:

```
from pywarp import RelyingPartyManager, Credential
from pywarp.backends import CredentialStorageBackend

class MyDBBackend(CredentialStorageBackend):
    def __init__(self, ...):
        self.database_client = ...

    def get_credential_by_email(self, email):
        user_record = self.database_client.get(email)
        return Credential(credential_id=user_record["cred_id"],
                          credential_public_key=user_record["cred_pub_key"])

    def save_credential_for_user(self, email, credential):
        self.database_client.update(email, {"cred_id": credential.credential_id,
                                            "cred_pub_key": bytes(credential.public_
↪key)})

    def save_challenge_for_user(self, email, challenge, type):
        self.database_client.update(email, {type + "challenge": challenge})

    def get_challenge_for_user(self, email, type):
        user_record = self.database_client.get(email)
        return user_record[type + "challenge"]
```

(continues on next page)

(continued from previous page)

```
my_rp = RelyingPartyManager(credential_storage_backend=MyDBBackend(...), ...)
```

CHAPTER 4

Example: Chalice app

The Chalice app example (in the `examples/chalice` directory) can be deployed as an AWS Lambda application when used with conventional AWS account credentials (configured via `aws configure` in the AWS CLI). This example uses DynamoDB as a storage backend.

To deploy this example, run `make -C examples/chalice` after configuring your AWS CLI credentials.

See the API documentation for more.

CHAPTER 5

Authors

- Andrey Kislyuk

CHAPTER 6

Links

- Project home page (GitHub)
- Documentation (Read the Docs)
- Package distribution (PyPI)
- Change log

CHAPTER 7

Bugs

Please report bugs, issues, feature requests, etc. on GitHub.

CHAPTER 8

License

CHAPTER 9

API documentation

CHAPTER 10

Table of Contents