WHAT TO DO WHEN YOU NEED CRYPTO

A look at cryptography in Python

NERDSINCHARGE

LVH

Principal Engineer
(Actually Does Things)

JARRET RAIM

Managed Security Lead
(Useless Manager Type)

ANDREW HARTNETT

Managed Security Eng Lead
(Less Useless Manager Type)

DOUG MENDIZABAL

Dev Extrordinaire
(Actually Does Things)

The content

THE TOOLS

- Python 2.7
- OpenSSL dev headers
- libffi dev headers
- Compiler (gcc/clang)
- (Windows might work, but no guarantees)
- virtualenv / pip

#oscon-crypto
Freenode

https://github.com/dmend/oscon-crypto-tutorial

http://bit.ly/oscon-crypto

The content

THE OVA

USERNAME: UBUNTU

PASSWORD: OSCON2015

(all lower case)

AN OUTLINE

The goal today is to take good programmers and give you all some intuition about how to use complex crypto systems to good effect. Sounds easy, no?

PASSWORDS & AUTHENTICATION

Let's be honest, passwords are terrible. However, until we all figure out something better, we need to deal with them. Here we figure out authentication.

KEY MANAGEMENT

All encryptions schemes are only as secure as their keys. In this session, we'll review the various types of key management for applications and review which type will be appropriate in different scenarios.

DATA AT REST

Data in applications comes in a huge variety of forms. We will review options for encrypting data and the pros and cons of each method.

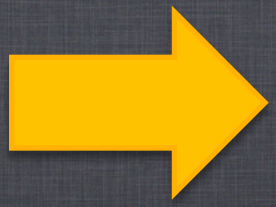
SIGNING & VERIFICATION

Many applications don't want to encrypt data for various reasons (performance, debuggability, etc) but do want to be able to verify that information hasn't been tampered with or that it comes from a known, valid user.

PASSWORDS

KEY DERIVATION

- Passwords are just a subtype of encryption key
- Encryption keys should be hard to guess
- Passwords are really easy to guess
- We need to make our passwords better keys



CUE OUR FRIENDS, THE
KEY DERIVATION FUNCTIONS

1. PBKDF2
2. BCRYPT
3. SCRYPT
4. HKDF

A DEFINITION

A key derivation function is a function that derives one or more secret values (the keys) from one secret value.

KEY STRENGTHENING

Passwords and other user chosen secrets generally do not have enough entropy to prevent them from being easily guessed.

MULTIPLE KEYS

When starting with a single secret, but the protocol needs multiple secrets, e.g. an encryption and MAC key.

```python
import os

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC


salt = os.urandom(16)
# derive
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
key = kdf.derive(b"my great password")
# verify
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
kdf.verify(b"my great password", key)
```

BCRYPT

- Based on the Blowfish symmetric cipher
- Used as default password storage for many systems including BSD
- 128-bit salt combined with 192-bit key

```
import bcrypt

key = bcrypt.hashpw(b"somepassword", bcrypt.gensalt())
if bcrypt.hashpw(b"somepassword", key) == key:
    print("verified")
else:
    print("invalid!")
```

SCRYPT

- Designed to be resistant to large-scale, customer hardware attacks (NSA, et. al)
- Raises resource demands of algorithm to make it more expensive to implement in hardware (e.g. more memory)

PBKDF2

- Part of Public-Key Cryptography Standards (PKCS) & RFC 2898
- Uses a pseudorandom function and a salt value and repeats a hash process for tunable a number of iterations

```
salt = os.urandom(16)
# derive
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(), length=32, salt=salt,
    iterations=100000, backend=default_backend()
)
```

HKDF

- HMAC based 'Extract and Expand'
- **Not for use with passwords!**

```
from cryptography.hazmat.primitives.kdf import HKDF

salt = os.urandom(16)
info = b"hkdf-example"
hkdf = HKDF(algorithm=hashes.SHA256(), length=32,
            salt=salt, info=info, backend=backend)
key = hkdf.derive(b"input key")
```

KDF DEMO

```python
import os

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC


salt = os.urandom(16)
# derive
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
key = kdf.derive(b"my great password")
# verify
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
kdf.verify(b"my great password", key)
```

PASSWORDS EXERCISE

1. Get the code
2. Start the app
3. Look at authentication code
4. Look at database
5. New auth scheme / code

- System should use a KDF for password strengthening
- System should support upgrading of passwords
- All tests still need to pass (fortunately there aren't any)

KEYMANAGEMENT

RANDOMNESS

GENERATING KEYS

Ensure that you are using a cryptographically secure random number generator that has been correctly seeded to generate random data.

VIRTUAL MACHINES

It is vanishingly unlikely that using a virtual machine will cause randomness exhaustion.

/DEV/URANDOM

Just use it, the man page is wrong. Seriously.

```
import os

randomness = os.urandom(16)
```

STORAGE EXERCISE

1. Get the code
2. Start the app
3. Look at how the encryption key is stored
4. Improve it

- Start with keys in source code
- Migrate to config file
- Migrate to environment

DATAATREST

ALTERNATE APPROACHES

There are three general ways to protect data at rest. Each has situations in which it is appropriate.

Hashing

Symmetric Encryption

Asymmetric Encryption

TOHASHORNOTTOHASH

A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size.

ONE WAY

Hashes are only useful for recognizing a known value (e.g. file integrity) as they cannot be reversed. Data that is hashed cannot be reconstituted from the hash itself.

USES

Most common use is authenticating sensitive data like passwords (not as just a single hash!!), social security numbers, etc.

```
1  import hashlib
2
3  password = "9384hfhjs98hf8("
4  hex_digest = hashlib.sha256(password)
5                .hexdigest()
6
7  print (hex_digest)
8
```

SYMMETRIC ENCRYPTION

A function that uses the same keying material for both encryption and decryption.

SINGLE KEY

A single key is used for all encryption operations. This key is an enforced secret, disclosure renders encrypted data viewable.

USES

Protecting sensitive data that must be retrieved, but isn't exposed to outside systems.

```
from cryptography.fernet import Fernet

key = Fernet.generate_key()
f = Fernet(key)
token = f.encrypt(b"my deep dark secret")
f.decrypt(token)
```

SYMMETRIC EXERCISE

1. Get the code
2. Start the app
3. Look at upload/download
4. Add encryption/decryption

- System should use an authenticated encryption scheme
- System should support algorithm upgrade over time.

ASYMMETRIC ENCRYPTION

Also known as public-key cryptograph, asymmetric cryptography requires two separate keys (public and private – natch), one of which is sharable.

MULTIPLE KEYS

A public key can be shared, allowing another party to encrypt a secret that can only be decrypted by the corresponding private key.

USES

Data and connections that must be shared outside a trust boundary.

```
message = b"encrypted data"
ciphertext = public_key.encrypt(
    message,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA1()),
        algorithm=hashes.SHA1(),
        label=None
    )
)
```

SIGNING AND VERIFICATION

ADESCRIPTION

Signing data allows to user to verify it's authenticity (e.g. do I know who it came from) and integrity (e.g. has this data been modified).

KEY GENERATION ALGORITHM

Signature algorithms can be built from other encryption algorithms, most usually a cryptography hash function.

SIGNATURE GENERATION ALGORITHM

Signatures use asymmetric cryptography (since they need to be shared with third parties). In this case, the private key is used to create the message so that it can be verified by anyone with the public key.

SIGNATURE VERIFICATION ALGORITHM

Any user with access to the public key can hash the signed data and verify that the hash was encrypted with the corresponding private key.

A STORAGE MECHANISM

It is usually useful to combine the signed data and signature into a single message. This can be small data (TLS) or for larger data where something like fernet is more useful.

SIGNING AND VERIFICATION

1. Get the code
2. Start the app
3. Sign some data
4. Verify it
5. What happens if you alter the data before verification?

- System should use an authenticated encryption scheme
- System should support algorithm upgrade over time.

GOODLUCK!