CCDC Inject

INJECT NAME	Explain a Digital Signature Script
INJECT ID	CRYP13A

INJECT DESCRIPTION:

Consider the following Python script for placing a digital signature on a syslog file. Explain what each of the 4 steps are doing in the process of completing this.

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
import os

# Step 1: EXPLAIN STEP 1
def generate_keys():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()

    # Save the private key to a file
    with open("private_key.pem", "wb") as f:
        f.write(private_key.private_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PrivateFormat.TraditionalOpenSSL,
            encryption_algorithm=serialization.NoEncryption()
        ))

    # Save the public key to a file
    with open("public_key.pem", "wb") as f:
        f.write(public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        ))

# Step 2: EXPLAIN STEP 2
def hash_file(file_path):
    hasher = hashes.Hash(hashes.SHA256())
    with open(file_path, 'rb') as f:
        while chunk := f.read(4096):
            hasher.update(chunk)
    return hasher.finalize()

# Step 3: EXPLAIN STEP 3
```

```
def sign_hash(private_key_path, file_hash):
    with open(private_key_path, "rb") as key_file:
        private_key = serialization.load_pem_private_key(
            key_file.read(),
            password=None
        )
    signature = private_key.sign(
        file_hash,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

# Step 4: EXPLAIN STEP 4
def save_signature(signature, output_path):
    with open(output_path, 'wb') as f:
        f.write(signature)

# Step 5: Verify the signature (Optional, for testing purposes)
def verify_signature(public_key_path, file_hash, signature):
    with open(public_key_path, "rb") as key_file:
        public_key = serialization.load_pem_public_key(key_file.read())
    try:
        public_key.verify(
            signature,
            file_hash,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except Exception as e:
        return False

if __name__ == "__main__":
    # Paths
    syslog_file = "syslog.log"  # Replace with your syslog file path
    signature_file = "syslog_signature.sig"

    if not os.path.exists("private_key.pem") or not os.path.exists("public_key.pem"):
        print("Generating RSA keys...")
        generate_keys()

    print("Hashing the syslog file...")
    file_hash = hash_file(syslog_file)

    print("Signing the hash...")
    signature = sign_hash("private_key.pem", file_hash)
    print("Saving the signature...")
    save_signature(signature, signature_file)

    print("Signature created and saved to", signature_file)

    # Optional: Verify the signature
    print("Verifying the signature...")
    if verify_signature("public_key.pem", file_hash, signature):
        print("Signature verification successful.")
    else:
        print("Signature verification failed.")
```

INJECT DELIVERABLE

Respond with a business memo that describes what each of the 4 steps in the script are accomplishing from a crypto functionality perspective.