Project 1 “Code Signing” Report

Group members: Niall Chiweshe, Bishesh Dulal, Pradip Sapkota

Abstract:

This project focuses on the study and implementation of code signing, a security mechanism that ensures the authenticity and integrity of software using digital signatures. Code signing prevents unauthorized modifications by allowing users to verify whether software has been altered or tampered with before execution. The project will involve researching digital signatures, implementing a software signing process, and creating a validator to check the legitimacy of signed software.

As part of the project, we will develop a product program, a simple script that prints "I am a software made by 11850393" (one of group member’s student ID) to the screen. This software will be digitally signed using the Elliptic Curve Digital Signature Algorithm (ECDSA). The signing process will involve generating a public-private key pair, where the private key is used by the software vendor to sign the program, and the public key is made available for users to verify its authenticity.

Additionally, we will create a validator program that checks the digital signature of the product before allowing execution. If the program’s signature is valid, execution is permitted; otherwise, it is blocked to prevent running potentially malicious or altered code. This project simulates real-world code signing practices used by major software vendors to prevent malware injection, unauthorized software modifications, and security threats. Through this implementation, we will gain practical experience in public-key cryptography, digital signatures, and software security, reinforcing our understanding of how cryptographic techniques enhance cybersecurity.

Introduction:

Problem Statement:

In today's digital landscape, software security is a critical concern, as cyber threats such as malware injection, unauthorized modifications, and software tampering pose significant risks to users and organizations. Attackers often exploit vulnerabilities in unsigned or poorly protected software to introduce malicious code, leading to data breaches, system compromises, and financial losses. Without a reliable mechanism to verify software authenticity and integrity, users risk running software that may have been altered by malicious actors.

Importance of the Problem:

Ensuring that software remains unaltered from its original state is essential for cybersecurity. Code signing is a fundamental technique that leverages cryptographic methods to verify that a piece of software originates from a trusted source and has not been modified. By digitally signing software, developers can assure users that their product is legitimate and secure, while users can verify authenticity before execution. Code signing helps prevent unauthorized alterations, safeguards against malware, and establishes trust in software distribution.

Approach Overview:

This project focuses on implementing a code signing mechanism using Elliptic Curve Digital Signature Algorithm (ECDSA) to secure software applications. Our approach involves:

1. Generating a key pair – A private key is used to sign the software, and a corresponding public key is distributed for validation. The first step is to generate an ECDSA key pair using the P-256 curve. The private key is generated and stored in the privateKey.pem file, while the corresponding public key is stored in the publicKey.pem file.
2. Signing the software – In this step, the product code (the product.py file) is signed using the private key generated earlier. A SHA-256 hash of the software is created, and the signature is generated using the private key. The signature is then saved to a product.sig file.

3. Validating the software – To ensure the integrity of the software, a validation mechanism is implemented. The validator reads the public key and signature, verifies the hash of the product.py file, and confirms that the signature matches the hash using the public key. If the signature is valid, the software is executed; otherwise, an error is reported, indicating the invalidity of the certificate.

Accomplishments:

Through this project, we successfully implemented a cryptographic framework for code signing and validation. The system ensures that only software with a valid signature can be executed, effectively blocking unauthorized modifications. By integrating Python-based cryptographic libraries, we built a functional prototype demonstrating how code signing enhances software security. This implementation provides a foundational understanding of public-key cryptography, digital signatures, and real-world security applications, contributing to the broader goal of securing software distribution and execution.

This project serves as a practical demonstration of how organizations can implement code signing to protect their software assets, ensuring trust, integrity, and security in software deployment and execution.

Related Works:

Several studies and articles have explored the importance of code signing, digital signatures, and software security. Below are some key related works that provide insights into the field:

1. **Boneh & Shoup (2020) – "A Graduate Course in Applied Cryptography"**
    - This book provides a deep dive into cryptographic principles, including digital signatures and their role in security. The authors discuss the importance of ensuring software authenticity through mathematical proofs and cryptographic implementations. This work serves as a theoretical foundation for the use of ECDSA in our project.
2. **D. Cooper et al. (2018) – "Security Considerations for Code Signing" (NIST Report)**
    - This report from the National Institute of Standards and Technology (NIST) outlines best practices for code signing and the risks associated with improper implementation. The study highlights the threats posed by private key mismanagement, untrusted certificate authorities, and improper validation procedures, reinforcing the importance of a secure validation mechanism like the one implemented in our project.
3. **Code Signing Whitepaper – CA Security Council (2016)**
    - This whitepaper presents an industry perspective on how code signing certificates are used to establish trust between software vendors and end users. The paper discusses certificate revocation, timestamping, and trust chains, which are critical aspects for ensuring software remains valid even after updates. Our project aligns with these principles by implementing a public key infrastructure (PKI)-based validation process.
4. **Kim et al. (2021) – "Mitigating Software Supply Chain Attacks with Code Signing"**
    - This paper explores real-world software supply chain attacks and how code signing can mitigate risks. The authors analyze incidents like the SolarWinds attack, where attackers introduced malicious code into signed software updates. The study underscores the necessity of strong cryptographic verification

mechanisms, similar to our project's validator, to detect unauthorized modifications.

5. **Singh & Gupta (2019) – "A Comparative Analysis of Digital Signature Algorithms for Secure Code Distribution"**
    - This research compares various digital signature algorithms (DSAs), including RSA, DSA, and ECDSA, in terms of performance, security, and scalability. The study concludes that ECDSA offers better security with smaller key sizes, making it an ideal choice for modern code signing applications. Our project incorporates ECDSA based on these findings, ensuring efficient and secure software authentication.

Approach:

In this project, the goal is to implement a code signing mechanism using Elliptic Curve Digital Signature Algorithm (ECDSA) to ensure the integrity and authenticity of software applications. The approach follows a sequence of steps, which include key generation, code signing, and validation. Each step is essential in verifying the legitimacy of the software before execution, especially in the case where an attacker might attempt to modify the software. Below, I will explain each of the components of the project in detail.

Step 1: Key Pair Generation

The first step in the project is to generate a public/private key pair, which is crucial for the signing and validation processes. The private key will be used to sign the software, and the corresponding public key will be used for validation.

1. **Private Key Generation**:
    - We used the Crypto.PublicKey.ECC module from the PyCryptodome library to generate an elliptic curve private key using the P-256 curve. This curve is widely used and provides a good balance of security and performance.
    - The private key is generated by the ECC.generate() method, which ensures that the key adheres to the specifications of the curve.
2. **Public Key Extraction**:
    - After generating the private key, the corresponding public key is derived using the .public_key() method. The public key is necessary for signature verification and can be safely distributed to users who wish to validate the software.
3. **Key Storage**:
    - The generated private key is saved to a file named privateKey.pem in PEM (Privacy-Enhanced Mail) format. This key is kept confidential and is only accessible to the software vendor for signing purposes.
    - The public key is saved to a separate file, publicKey.pem, which is intended for distribution. This key is used by users to verify the software's authenticity and integrity.

4. **Key Generation implementation:**

```python
from Crypto.PublicKey import ECC

#ECDSA Key Pair
private_key = ECC.generate(curve='P-256')
public_key = private_key.public_key()

with open("privateKey.pem", "wt") as f:
    f.write(private_key.export_key(format='PEM'))

with open ("publicKey.pem", "wt") as f:
    f.write(public_key.export_key(format='PEM'))
```

Step 2: Code Signing

Once the key pair is generated, the next step is to sign the software. In this project, the software is represented by a simple Python script (product.py) that prints a message with one of our group member's student's ID.

1. **Signing Process**:
    - To sign the product.py file, the script is first read in binary mode (rb). The file is then hashed using SHA-256 to create a message digest. This digest represents the contents of the product in a fixed-length form, which is crucial for ensuring the integrity of the software.
    - The signing is performed using the private key with the DSS.new() method from the Crypto.Signature module. This method creates a Digital Signature Scheme (DSS) object that is used to generate the signature based on the private key and the hashed product.
2. **Signature Generation**:
    - After signing the hash, the signature is stored in the product.sig file. This signature is essentially cryptographic proof that the product has not been tampered with and that it was signed by the vendor with the private key.

3. **Signature Implementation:**

```
from Crypto.PublicKey import ECC
from Crypto.Signature import DSS
from Crypto.Hash import SHA256

with open ("privateKey.pem", "rt") as f:
    private_key = ECC.import_key(f.read())

with open ("product.py", "rb") as f:
    product = f.read()

product_hash = SHA256.new(product)
sign_obj = DSS.new(private_key, 'fips-186-3')
signature = sign_obj.sign(product_hash)

with open ("product.sig", "wb") as f:
    f.write(signature)
```

Step 3: Validation

The validation step ensures that only legitimate software is executed. Users rely on the validator to verify the authenticity of the software before running it.

1. **Reading the Public Key**:
    - The validator first reads the public key from the publicKey.pem file. This public key is used to verify the signature on the product.
2. **Signature Verification**:
    - The product file (product.py) is read again in binary mode. The hash of the product is generated using SHA-256, just as it was during the signing process.
    - The validator then reads the signature from the product.sig file. Using the public key, the validator attempts to verify the signature against the product hash using the DSS.new() method. If the signature is valid, the software is considered authentic, and execution is allowed.
3. **Execution Control**:
    - If the signature matches the hash of the product, the validator prints a message saying, "Code certificate valid: execution allowed," and the product is executed using subprocess.run().

- If the signature is invalid or if there is an attempt to modify the product (as will be shown in a later step), the validator prints a message saying, "Code certificate invalid: execution denied," and blocks the execution of the product.

4. **Validation Implementation:**

```
# Get the product to be verified
with open ("product.py", "rb") as f:
    product = f.read()

# Get the certificate
with open ("product.sig", "rb") as f:
    signature = f.read()

# Make the product hash,create the verifier with public key and
# check if signature matches the hash
product_hash = SHA256.new(product)
validator = DSS.new(public_key, 'fips-186-3')
validator.verify(product_hash, signature)

# Output for successful verification
print("\nCode certificate valid: execution allowed")
print("\nExecuting product:")
subprocess.run([sys.executable, "product.py"])  # Execute product
```

Step 4: Handling Malicious Code

To test the effectiveness of the code signing mechanism, we simulate an attack where an attacker modifies the product by adding malicious code.

1. **Modification**:
   - In this scenario, an attacker inserts a line of code into the product.py file that prints "Malicious code!" This simulates how an attacker might attempt to inject harmful code into the software.
2. **Verification Failure**:
   - When the modified product is validated using the public key, the signature will no longer match the hash of the modified product. As a result, the validator will detect the discrepancy and block the execution of the modified software, ensuring that the user is protected from the malicious code.

3. **Handling malicious code implementation:**

```
# Handle the invalid signature
except ValueError:
    print("\nCode certificate invalid: execution denied")
```

Results:

This section presents the outcomes of our implementation, verifying the correctness of our code signing approach using ECDSA. We evaluate key functionalities, including key generation, digital signing, signature verification, and detection of tampered software.

1. Key Pair Generation

- We successfully generated an ECDSA key pair using Python's Crypto.PublicKey.ECC module.
- The private key was securely stored in privateKey.pem, while the public key was stored in publicKey.pem, which is used for validation.

- **privateKey.pem:**

```
-----BEGIN PRIVATE KEY-----
MIGHAgEAMBMGByqGSM49AgEGCCqGSM49AwEHBG0wawIBAQQgMKe86nOD1VoivBaI
E+V/5Us+t0V7EIS3Y1XbQ2h152ihRANCAASFvSvYMgTQGx3Yv+ydomI5ylUTYrea
tXrj4y68LPsUvyv2gZnKnwS/0USOBk0M0r7v66VIRkJZgnRm8eEv1ALe
-----END PRIVATE KEY-----
```

- **publicKey.pem:**

```
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhb0r2DIE0Bsd2L/snaJiOcpVE2K3
mrV64+MuvCz7FL8r9oGZyp8Ev9FEjm5NDNK+7+ulSEZCWYJ0ZvHhL9QC3g==
-----END PUBLIC KEY-----
```

2. Code Signing

- The original product.py file was hashed using SHA-256.
- A digital signature was created using the private key and stored in product.sig.
- **Product.sig:**

```
5sí□ÈòØXgtp¿uøt+□□      □ \□ŸY v□□™•ÿŠv^Ê□)ÙW0WíÄÃÉÃ1ÇíR}RŸ¬<PP□□>ÌÑŠ
```

- This confirms that the product was successfully signed by the trusted vendor.

3. Verification of the Product (Valid Case)

- The validator component successfully verified the product using the public key.
- The execution of product.py was allowed, and the output displayed:
- **Output:**

```
vscode\extensions\ms-python.debugpy-2025.4.1-win32-x64\bundled\libs\debugpy\launcher' '51171' '--' 'c:\Users\Niall Chiw
rive\Documents\CodeSigning\validator.py'
Verified public key of the Vendor:
----------------------------------------
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhb0r2DIE0Bsd2L/snaJiOcpVE2K3
mrV64+MuvCz7FL8r9oGZyp8Ev9FEjm5NDNK+7+ulSEZCWYJ0ZvHhL9QC3g==
-----END PUBLIC KEY-----
----------------------------------------

Code certificate valid: execution allowed

Executing product:
I am a software made by 11850393
```

4. Detection of Tampered Software (Invalid Case)

- A modified version of product.py was created by inserting a malicious line:

```
print("Malicious code!")   #For the interception part
```

- The validator detected the modification since the new file's hash did not match the original signature.
- Execution was denied, displaying the output:

```
vscode\extensions\ms-python.debugpy-2025.4.1-win32-x64\bundled\libs\debugpy\launche
rive\Documents\CodeSigning\validator.py'
Verified public key of the Vendor:
----------------------------------------
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEhb0r2DIE0Bsd2L/snaJiOcpVE2K3
mrV64+MuvCz7FL8r9oGZyp8Ev9FEjm5NDNK+7+ulSEZCWYJ0ZvHhL9QC3g==
-----END PUBLIC KEY-----
----------------------------------------

Code certificate invalid: execution denied
```

5. Summary of Findings

- Our implementation correctly signed the software and prevented unauthorized modifications.

- The validator accurately detected changes, ensuring the integrity and authenticity of the software.
- This approach successfully demonstrates the importance of code signing in preventing software tampering and ensuring trust between software vendors and users.

References:

1. **Boneh, D., & Shoup, V.**, "A Graduate Course in Applied Cryptography," 2020.
2. **D. Cooper, S. Santesson, S. Farrell, H. H. Glasser, D. L. Housley, and W. Polk**, "Security Considerations for Code Signing," National Institute of Standards and Technology, NIST Special Publication 800-57 Part 1, 2018. [Online]. Available: https://csrc.nist.rip/external/nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.01262018.pdf. [Accessed: Mar. 20, 2025].
3. **CA Security Council**, "Code Signing Whitepaper," 2016. [Online]. Available: https://pkic.org/uploads/2016/12/CASC-Code-Signing.pdf. [Accessed: Mar. 20, 2025].
4. **Kim, K., J. Y. Lee, W. J. Yoo, and J. S. Kang**, "Mitigating Software Supply Chain Attacks with Code Signing," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 4, pp. 1302-1313, Jul. 2021.
5. **Singh, M., & Gupta, B.**, "A Comparative Analysis of Digital Signature Algorithms for Secure Code Distribution," *International Journal of Computer Science and Information Security*, vol. 17, no. 1, pp. 45-54, Jan. 2019.
6. **Pradip676**, "Code Signing Validator," GitHub repository, [Online]. Available: https://github.com/pradip676/Code-Signing-Validator. [Accessed: Mar. 20, 2025].