|  |  |  |  |
| --- | --- | --- | --- |
| **Course Name:** | **Information Security (116U01L602)** | **Semester:** | **VI** |
| **Date of Performance:** | **30/01/2025** | **DIV/ Batch No:** | **C - 3** |
| **Student Name:** | **Romil Lodaya** | **Roll No:** | **16010122096** |

|  |
| --- |
| **Title: Application of RSA Algorithm for various security services like**  **confidentiality, authentication, signature, non-repudiation and integrity** |

|  |
| --- |
| **Objectives:** |
| 1. Understand how the RSA algorithm is applied to achieve **confidentiality**, **authentication**, digital **signatures**, **non-repudiation**, and **data integrity**. 2. Explore the mathematical principles of RSA (e.g., modular arithmetic, prime factorization) and their role in cryptographic security. 3. Learn how public-private key pairs are used for different security services. 4. Analyze the interplay between RSA and hash functions (e.g., SHA-256) for integrity and non-repudiation. 5. Implement or simulate RSA-based mechanisms for each security service. |

|  |
| --- |
| **Expected Outcome of Experiment:** |
| **CO1 Explain various security goals, threats, vulnerabilities and controls**  **CO2 Apply various cryptographic algorithms for software security** |

|  |
| --- |
| **Books/ Journals/ Websites referred:** |
| 1. <https://sandilands.info/sgordon/simple-introduction-to-using-openssl-on-command-line> 2. <https://www.baeldung.com/openssl-self-signed-cert> |

|  |
| --- |
| **Pre Lab/ Prior Concepts:** |
| 1. Basic understanding of **asymmetric cryptography** (public vs. private keys). 2. Familiarity with **number theory** concepts: prime numbers, modular arithmetic, Euler’s theorem. 3. Knowledge of **hash functions** and their role in data integrity. 4. Awareness of digital signatures and their components (signing/verification). 5. General concepts of **security services**: confidentiality, authentication, etc. |

|  |
| --- |
| **New Concepts to be learned:** |
| 1. **RSA mechanisms** for specific security services:    * Confidentiality: Encryption/decryption using recipient’s public/private key.    * Authentication: Challenge-response protocols with RSA.    * Signature: Signing hashed messages with a private key.    * Non-repudiation: Binding signatures to sender identity.    * Integrity: Combining RSA with hash functions. 2. **Key generation process**: Selecting primes, computing modulus, Euler’s totient function, exponents. 3. **Padding schemes** (e.g., OAEP) to prevent vulnerabilities in raw RSA. 4. Practical limitations of RSA (e.g., key size, computational overhead). |

|  |
| --- |
| **Abstract:** |
| The RSA algorithm, a cornerstone of modern cryptography, enables multiple security services through its asymmetric key mechanism. By leveraging the mathematical complexity of factoring large primes, RSA ensures **confidentiality** via public-key encryption, **authentication** through challenge-response protocols, and **non-repudiation** via digitally signed messages. Combined with hash functions, it guarantees **integrity** by linking message content to cryptographic signatures. This lab explores RSA’s versatility in addressing core security requirements, emphasizing its role in secure communication, identity verification, and tamper-proof data exchange. |

|  |
| --- |
| **Related Theory:** |
| 1. **RSA Algorithm Overview**:    * **Key Generation**: Select primes p*p*, q*q*; compute n=p×q*n*=*p*×*q*, ϕ(n)=(p−1)(q−1)*ϕ*(*n*)=(*p*−1)(*q*−1), choose e*e* (public exponent) and d*d* (private exponent) such that e⋅d≡1mod  ϕ(n)*e*⋅*d*≡1mod*ϕ*(*n*).    * **Encryption**: C=Memod  n*C*=*Me*mod*n* (using recipient’s public key (e,n)(*e*,*n*)).    * **Decryption**: M=Cdmod  n*M*=*Cd*mod*n* (using recipient’s private key d*d*). 2. **Security Services with RSA**:    * **Confidentiality**: Encrypt data with the recipient’s public key; only the private key holder can decrypt.    * **Authentication**: Prove identity by decrypting a challenge encrypted with the claimant’s public key.    * **Signature**: Hash a message H(M)*H*(*M*), then encrypt the hash with the sender’s private key: S=H(M)dmod  n*S*=*H*(*M*)*d*mod*n*. Verification uses the sender’s public key: H(M)=Semod  n*H*(*M*)=*Se*mod*n*.    * **Non-repudiation**: Signatures are tied to the sender’s private key, preventing denial of authorship.    * **Integrity**: Hash functions ensure message consistency; any alteration invalidates the signature. 3. **Practical Considerations**:    * **Hash Functions**: SHA-256/512 for collision-resistant message digests.    * **Padding**: OAEP/PSS to prevent attacks like chosen-plaintext.    * **Key Size**: 2048+ bits for modern security.    * **Vulnerabilities**: Malleability in raw RSA, side-channel attacks, and key management risks. 4. **Real-World Applications**:    * SSL/TLS handshakes (authentication and key exchange).    * Digital certificates (X.509) for non-repudiation.    * Secure email (PGP/GPG) combining RSA with symmetric encryption. |

|  |
| --- |
| **Implementation Details:** |
| 1] Encryption:    Here, I created the plaintext.txt and ciphertext.txt for encryption and decryption.        AES-256-CBC cipher is used to encrypt the data from the plaintext.txt and encrypted text is stored in ciphertext.txt    Also, AES-256-CBC is used to decrypt the data from ciphertext.txt and decrypted text is stored in decrypted.txt        Again, AES-256-CBC is used for encryption and decryption for different content.      2] Hashing:    Plaintext.txt is created with the content itself at CLI.      This command computes the **MD5 hash** of the file plaintext.txt using OpenSSL's digest utility. MD5 generates a 128-bit (16-byte) hash value    This command generates a **SHA-256 hash** of the file plaintext.txt using OpenSSL. SHA-256 produces a 256-bit (32-byte) hash    1st command Generates a **256-bit (32-byte) random cryptographic key** in hexadecimal format. This key (3bddaddldclaa65f...) is used for HMAC (Hash-Based Message Authentication Code).  2nd command Computes an **HMAC-MD5** for plaintext.txt using the generated key. HMAC combines MD5 hashing with a secret key to verify both **integrity** and **authenticity** of the file.    This command computes the **SHA-256 hash** of the file plaintext.txt. SHA-256 generates a 256-bit (64-character) hexadecimal hash to verify the file's **integrity** and ensure it has not been altered.  Changed content for plaintext.txt        Performed the same steps as above with different content to check the hashes of both the files.  So, hash for both of the files is different.  For the old content, it again generates the same hash as earlier so hash for same content remains same regardless of the time it was generated:    3] Certificates:  https://www.baeldung.com/openssl-self-signed-cert    Creating a password-protected, 2048-bit RSA private key (*domain.key*)    **If we want our certificate signed, we need a certificate signing request (CSR)**. The CSR includes the public key and some additional information (such as organization and country).  Creating a CSR (*domain.csr*) from our existing private key:      A self-signed certificate is **a certificate that’s signed with its own private key**. It can be used to encrypt data just as well as CA-signed certificates, but our users will be shown a warning that says the certificate isn’t trusted.  We can even create a private key and a self-signed certificate with just a single command:    We can be our own certificate authority (CA) by creating a self-signed root CA certificate, and then installing it as a trusted certificate in the local browser.  **Creating a Self-Signed Root CA**  Let’s create a private key (*rootCA.key*) and a self-signed root CA certificate (*rootCA.crt*) from the command line:      This is the CA certificate, rootCA.crt    **Sign Our CSR With Root CA**  We can sign our CSR (*domain.csr*) with the root CA certificate and its private key:  As a result, the CA-signed certificate will be in the *domain.crt* file. This would result in a working certificate    We can use the *openssl* command to view the contents of our certificate in plain text:      This is the directory in which all the files were created and operations were performed.    This is the users certificate, domain.crt    4] Digital Signatures:  **1. Generate RSA Private and Public Keys**  First, generate a private and public key pair if you don't have them already:  openssl genpkey -algorithm RSA -out private\_key.pem -pkeyopt rsa\_keygen\_bits:2048  openssl rsa -pubout -in private\_key.pem -out public\_key.pem  This creates:   * private\_key.pem: Your private key. * public\_key.pem: Your public key.         **2. Create a File to Sign**  For demonstration, let's assume you have a file named file.txt that you want to sign.      **3. Generate a Digital Signature**  To sign the file using your private key:  openssl dgst -sha256 -sign private\_key.pem -out file.sig file.txt    This command:   * Signs file.txt with the SHA-256 hash algorithm. * The signature is saved in file.sig.     **4. Verify the Digital Signature**  To verify the signature, you'll need to use the public key and the signature file. The file being signed (file.txt) and the signature file (file.sig) are required:  openssl dgst -sha256 -verify public\_key.pem -signature file.sig file.txt  If the verification is successful, OpenSSL will output:  Verified OK      **5] non - repudiation:**  Non-repudiation refers to the assurance that someone cannot deny the validity of their actions. In the context of cybersecurity, it is important for ensuring accountability, especially in digital communications or transactions. Non-repudiation involves techniques and mechanisms that provide evidence of the origin, receipt, or validity of data.  **Digital Signatures**: Digital signatures ensure that a message or transaction has been sent by the purported sender and has not been tampered with in transit. A digital signature is unique to both the message and the sender, providing non-repudiation by allowing the sender to not deny their involvement.  **Steps:**   1. **Generate a Private and Public Key Pair**. 2. **Sign a Message Using the Private Key**. 3. **Verify the Signature Using the Public Key**.   **1. Generate a Private and Public Key Pair**  First, generate a private key and public key using OpenSSL:  openssl genpkey -algorithm RSA -out private\_key.pem -aes256  openssl rsa -pubout -in private\_key.pem -out public\_key.pem     * This will create a private key (private\_key.pem) encrypted with aes256. * The second command generates the corresponding public key (public\_key.pem).         **2. Sign a Message Using the Private Key**  Now, create a file (message.txt) that contains the message you want to sign:  echo "This is a confidential message." > message.txt      Next, sign the message with your private key:  openssl dgst -sha256 -sign private\_key.pem -out message.sig message.txt     * This command creates a SHA-256 hash of message.txt and then signs it using the private key (private\_key.pem). * The signature will be saved in the file message.sig.       **3. Verify the Signature Using the Public Key**  To verify the signature, you will need the original message, the signature, and the public key. Use the following command:  openssl dgst -sha256 -verify public\_key.pem -signature message.sig message.txt     * This will verify the signature (message.sig) against the message (message.txt) using the public key (public\_key.pem). * If the signature is valid, it will return Verified OK. Otherwise, it will return Verification Failure. |

|  |
| --- |
| **Results/Output:** |
| VLAB: |

|  |
| --- |
| **Conclusion:** |
| **RSA enables confidentiality, authentication, signatures, non-repudiation, and integrity via asymmetric keys and hash functions, fulfilling core security goals** |

|  |
| --- |
| **Post-Lab Questions:** |
| 1. In the RSA algorithm, p= 7, q=11 and e= 13, then what will be the value of d?   **Calculation of d*d* in RSA** Given p=7*p*=7, q=11*q*=11, and e=13*e*=13:   * Compute *n*=*p*×*q*=77. * *ϕ*(*n*)=(*p*−1)(*q*−1)=60. * Solve *e*⋅*d*≡1mod*ϕ*(*n*): Using the Extended Euclidean Algorithm, *d*=37.   **Final Answer**: *d*=37.   1. Discuss various cryptanalysis attacks possible to be carried out on RSA  * **Factoring Attack**: Factor *n*=*p*×*q* to compute *ϕ*(*n*) and derive *d*. * **Brute Force**: Infeasible for large keys but possible with weak key sizes. * **Side-Channel Attacks**: Exploit timing/power consumption during decryption. * **Chosen Ciphertext Attack**: Decrypt chosen texts to deduce *d*. * **Wiener’s Attack**: Effective if *d* is small. * **Poor Padding Exploits**: e.g., Bleichenbacher’s attack on PKCS#1 v1.5. * **Broadcast Attack**: Encrypting the same message with multiple keys (Håstad).  1. Comment on drawbacks of RSA.  Discuss solution(s) over the same.   **Drawbacks**:   1. **Computational Overhead**: Slow for bulk data due to large keys. 2. **Quantum Vulnerable**: Shor’s algorithm breaks RSA on quantum computers. 3. **Key Management**: Large key sizes increase storage/transmission costs. 4. **Padding Risks**: Weak padding (e.g., PKCS#1 v1.5) enables attacks.   **Solutions**:   1. **Hybrid Encryption**: Use RSA for key exchange + AES for data. 2. **Post-Quantum Cryptography**: Adopt lattice-based algorithms (e.g., NTRU). 3. **Secure Padding**: Use OAEP instead of PKCS#1 v1.5. 4. **Quantum-Resistant Key Sizes**: Temporarily mitigate risk with 4096+ bit keys. |