ECE 443/518 – Computer Cyber Security
Lecture 15 OpenSSL

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

October 6, 2025

Outline

Midterm Exam

- Lecture 1 $\sim$ Lecture 14, see Homework 1 and 2 for sample.
  - Points may be deducted if key steps are missing.
- Students registered for main campus section: Wed. 10/8, 11:25 AM – 12:40 PM, in class.
  - A physical calculator is allowed. Laptop or any other electronic device or calculator apps running on them are not allowed.
  - Closed book/notes. A letter-size page of cheat sheet is allowed.
- Online students may take the exam as above, or contact Charles Scott (scott@iit.edu) to make arrangement and confirm with me.
  - No make-up exam will be offered if you fail to do so.
- ADA Accommodations: contact Center for Disability Resource (disabilities@iit.edu)
- Emergency/extraordinary reasons for make-up midterm exams are accepted only with documented proof like docter's notes.

Reading Assignment

- This lecture: OpenSSL
  - Please install OpenSSL using a package manager come with your OS.
  - For Windows, use Win64 OpenSSL v3.5.2 Light from `https://slproweb.com/products/Win32OpenSSL.html`
- Next lecture (10/8): Midterm Exam
- No lecture on 10/13
- Lecture on 10/15: Secure Collaborations

Outline

OpenSSL

- A open-source software library to support TLS (Transport Layer Security) protocols.
  - Created in 1998, based on a previous open-source SSL (Secure Sockets Layer) implementation.
  - While TLS eventually replaced SSL, OpenSSL keeps its name as many other libraries and APIs.
- A toolkit to access cryptography in practice.
  - `libcrypto`: the core library of OpenSSL implementing standard cryptographic algorithms.
  - `libssl`: the OpenSSL library implementing TLS protocols based on libcrypto
  - `libcrypto` and `libssl` are widely used in applications, server software, and programming languages to provide support for cryptography and TLS.
  - `openssl`: the OpenSSL program to access its functionality from command-line.

Symmetric Cryptography with OpenSSL

- openssl-rand: CSPRNG

  `openssl rand -hex 16`

  - The `-hex` option allows to output the 16 random bytes in hex encoding like `00112233445566778899aabbccddeeff`.
- openssl-dgst: hash functions and digital signatures

  `openssl dgst -sha256 plain.txt`
- openssl-enc: symmetric ciphers
  - No, you should not use this in production since no AEAD algorithms are supported in command-line for security reasons.
  - You can access AEAD algorithms from a programming language or within a protocol like TLS.

Public-Key Cryptography with OpenSSL

- openssl-genpkey: key pair generation

```
openssl genpkey -algorithm RSA -out priv.pem
    -pkeyopt rsa_keygen_bits:2048
```

  - The `.pem` file allows to encode binary data in PEM, a textual format, so it can be distributed via email.
  - But you are not supposed to email the RSA private key!

- openssl-pkey: key pair processing

```
openssl pkey -in priv.pem -pubout -out pub.pem
```

  - Extract the public key and save it in PEM format that can be emailed.

- Inspect the RSA keys to show (p, q, d) and (n, e)

```
openssl pkey -in priv.pem -text -noout
openssl pkey -in pub.pem  -pubin -text -noout
```

Public-Key Cryptography with OpenSSL (Cont.)

- Use openssl-dgst for digital signatures.

```
openssl dgst -sha256 -sign priv.pem -out sig.bin msg.txt
openssl dgst -sha256 -verify pub.pem -signature sig.bin msg.
```

  - Need a hash algorithm to sign messages of arbitrary length.
  - Signatures are stored in bytes instead of PEM.
  - Signatures are usually part of a higher level protocol, like a TLS certificate, where an encoding to a textual format, like PEM, is defined.
- Other popular public-key algorithms.
  - `EC`: ECDSA (Elliptic Curve Digital Signature Algorithm) with a few choices of standard curves.
  - `Ed25519`: EdDSA (Edwards-curve Digital Signature Algorithm) with a specific curve.
- Unlike `RSA` where encryption/decryption is supported, most other public-key algorithms are for digital signatures only.
  - As discussed in our lectures, public-key algorithms are used for key exchange, and for authentication via digital signatures.

Outline

HTTPS Authentication Overview

- HTTPS use TLS protocols to authenticate servers and optionally clients.
  - Based on PKI
  - Since multiple parties are involved for multiple steps, it is important for us to understand where the inputs come from and where the outputs are used for each party and step.
- Certificate Authority (CA)
  - Make use of public-key cryptography: $k_{pub,CA}$ and $k_{pr,CA}$.
  - Use $k_{pr,CA}$ to sign certificates for servers and clients.
  - Distribute $k_{pub,CA}$ to servers and clients so they can verify certificates signed by CA.
  - Maintain CRL to revoke certificates

HTTPS Authentication Overview (Cont.)

- Server authentication
  - A server has its own key pair: $k_{pub,server}$ and $k_{pr,server}$
  - Request CA to sign its certificate with its identity ID_{server} and public key $k_{pub,server}$.
  - Send the certificate when clients connect via TLS.
- Client authentication (optional)
  - A client has its own key pair: $k_{pub,client}$ and $k_{pr,client}$
  - Request CA to sign its certificate with its identity ID_{client} and public key $k_{pub,client}$.
  - When the client connects via TLS, the server provides a list of CAs it will accept, then the client sends the certificate signed by one of the CAs.

CA Setup

- Generate private key $k_{pr,CA}$ with pass phrase protection.

```
openssl genpkey -algorithm RSA -out CA.key -aes256
    -pkeyopt rsa_keygen_bits:4096
```

- But servers and clients will not accept an arbitrary public key.
  - They only accept certificates signed by a known CA.
  - But this is our root CA and no other CA is supposed to sign a certificate for it.
- Create a self-signed root certificate to be distributed.

```
openssl req -x509 -new -key CA.key -days 3650
    -subj "/CN=MyRootCA" -out CA.pem
```

  - `openssl req -x509` specifically create a certificate for a public key and then sign it by its own private key.
  - X.509 (`-x509`) is a standard format for public-key certificate.
  - Subject (`-subj`) is the identity of a certificate.
- Inspect the root certificate.

```
openssl x509 -in CA.pem -text -noout
```

Server Certificate: Server Actions

- Generate private key $k_{pr,server}$

```
openssl genpkey -algorithm RSA -out server.key
    -pkeyopt rsa_keygen_bits:2048
```

  - Could use less protection for server – less RSA bits.
  - A server program usually need to load its private key at start without human input so no pass phrase here.
- Create a Certificate Signing Request (CSR)

```
openssl req -new -key server.key
    -subj "/CN=localhost" -out server.csr
```

  - `openssl req` (without `-x509`) creates a CSR.
  - Include $k_{pub,server}$ extracted from $k_{pr,server}$ (`server.key`)
  - Provide server's DNS name (`/CN`) for its identity (`-subj`) – use `localhost` for a testing server running on the same machine as clients.
  - Include a digital signature of the CSR itself signed by $k_{pr,server}$ – ensure the CSR is created by who owns the private key.

Server Certificate: CA Actions

- Once CA receives the CSR from a server, it signs a certificate based on the CSR, and send the certificate back to the server.

```
openssl x509 -req -in server.csr -CA CA.pem -CAkey CA.key
    -CAcreateserial -out server.pem -days 365 -sha256
    -extfile server.ext
```

  - Verify the digital signature in CSR.
  - CA's certificate (`-CA`) provides CA's information like its identity to be included into server's certificate.
  - CA's private key (`-CAkey`) is used to sign server's certificate.
  - `-CAcreateserial` provides a simple mechanism to maintain a unique serial number per certificate. This serial number is used later to identify a certificate, e.g. in the CRL.
  - `-extfile` provides additional information required by X.509v3 extensions for server authentications

- Inspect and verify the server certificate.

```
openssl x509 -in server.pem -text -noout
openssl verify -CAfile CA.pem server.pem
```

Client Certificate

- The general steps to create client certificates are the same as that for server certificates.
  - Generate private key $k_{pr,client}$ for the client
  - Create a CSR to be sent to CA.
  - CA signs the certificate based on the CSR.
- Differences
  - The private key $k_{pr,client}$ can be protected by a pass phrase.
  - Client identity (`-subj`) in CSR doesn't need to be a DNS name – could be username, email, phone number etc.
  - Use extensions (`-extfile`) for client authentications.

Practical Considerations

- Do you realize there is a critical question we haven't answer yet for production settings?
  - How does CA authenticate servers and clients? I.e. how does CA know the CSRs are from who they claim to be?
- Distributed server authentication with DNS
  - A server need to be reachable on Internet from the DNS name it claims to own.
  - Used by Let's Encrypt, where the Automatic Certificate Management Environment (ACME) protocol defines the communications between CA and server so a certificate can be created and delivered automatically.

PKCS #12

- What if DNS doesn't apply? E.g. for client authentications.
  - Or when organizations prefer a centralized approach.
- Asking servers and clients to generate their own private keys and then CSRs would require interactions with CA.
  - At least two messages, one for CSR and one for certificate.
  - Difficult to automate.
- Let CA generate keys, CSRs, and certificates for everyone
  - Less secure as CA now knows everyone's private key.
  - Simplify management. Easy to automate.
- PKCS #12: safe delivery of private key and certificate

```
openssl pkcs12 -export -inkey client.key -in client.pem
  -certfile CA.pem -out client.p12 -name "Alice"
```

  - Bundle a private key, its certificate, and possibly the CA chain into a single file to simplify communication.
  - Encrypt the file with a password so only who has the password can access the private key.

Summary

- Learn to use OpenSSL and other tools to work with cryptography standards and protocols.