Lecture 13: Certificates, Digital Signatures, and the Diffie-Hellman Key Exchange Algorithm

Lecture Notes on "Computer and Network Security"

by Avi Kak (kak@purdue.edu)

February 19, 2019
12:45 Noon

Goals:

- Authenticating users and their public keys with certificates signed by Certificate Authorities (CA)
- Exchanging session keys with public-key cryptography
- X.509 certificates
- **Perl and Python code for harvesting RSA moduli from X.509 certificates**
- The Diffie-Hellman algorithm for exchanging session keys
- The ElGamal digital signature algorithm
- **Can the certificates issued by CAs be forged?**

CONTENTS

13.1: USING PUBLIC KEYS TO EXCHANGE SECRET SESSION KEYS

- From the presentation on RSA cryptography in Lecture 12, you saw that public key cryptography, at least when using the RSA algorithm, is not suitable for the encryption of the actual message content.

- However, public key cryptography fulfills an extremely important role in the overall design and operation of secure computer networks because it leads to superior protocols for managing and distributing secret session keys that can subsequently be used for the encryption of actual message content using symmetric-key algorithms such as AES, 3DES, RC4, etc. [although, not RC4 as much any longer].

- How exactly public key cryptography should be used for exchanging a secret session key depends on the application context for secure communications and the risk factors associated with a loss of security.

- If a party A simply wants to receive all communications confidentially (meaning that A does not want anyone to snoop on the incoming message traffic) and that A is not worried about the authenticity of the messages received, all that A has to do is to publish his/her public key in some publicly accessible place (such as on a web page). Subsequently, anyone wanting to send a confidential message to A would encrypt that message with A's public key. Only A would be able to decrypt such messages. [You'd think that as long as A does not lose his/her private key, there is no danger of anyone else masquerading as A. You might even say that even if someone were to eavesdrop on the communications received by A, they would not be able decipher those messages. But think about the following: How would a party B wanting to send an encrypted message to A be sure that the webpage that claims to present A's public key is really authentic? What if it is a fake webpage that was put up for a short while just so that you could be tricked into parting with some sensitive information that you think you are sending to A?]

- If two parties A and B are sure about each other's identity, can be certain that a third party will not masquerade as either A or B vis-a-vis the other, they can use a simple and direct key exchange protocol for exchanging a secret session key. In general, such protocols will not require support from any coordinating or certificating agencies. A direct key exchange protocol is presented in Section 13.2. [Unfortunately, as you will see, the direct key exchange protocol is vulnerable to the man-in-the-middle attack.]

- The key exchange protocols are more complex for security that provides a higher level of either one-sided or mutual authentication between two communicating parties. **These protocols**

usually involve Certificate Authorities, as discussed in Section 13.3.

13.2: A DIRECT KEY EXCHANGE PROTOCOL

- If each of the two parties A and B has full confidence that a message received from the other party is indeed authentic (in the sense that the sending party is who he/she/it claims to be), the exchange of the secret session key for a symmetric-key based secure communication link can be carried out with a simple protocol such as the one described below:

  - Wishing to communicate with B, A generates a public/private key pair $\{PU_A,\ PR_A\}$ and transmits **an unencrypted message** to B consisting of PU_A and A's identifier, ID_A (which can be A's IP address). Note that PU_A is party A's public key and PR_A the private key.

  - Upon receiving the message from A, B generates and stores a secret session key K_S. Next, B responds to A with the secret session key K_S. This response to A is **encrypted** with A's public key PU_A. We can express this message from B to A as $E(PU_A,\ K_S)$. Obviously, since only A has access to the private key PR_A, only A can decrypt the message containing the session key.

- A decrypts the message received from B with the help of the private key PR_A and retrieves the session key K_S.

- A **discards both** the public and private keys, PU_A and PR_A, and B **discards** PU_A.

- Now A and B can communicate confidentially with the help of the session key K_S.

- However, this protocol is vulnerable to the **man-in-the-middle** attack by an adversary E who is able to **intercept** messages between A and B. This is how this attack takes place:

  - When A sends the very first unencrypted message consisting of PU_A and ID_A, E intercepts the message. (Therefore, B never sees this initial message.)

  - The adversary E generates its own public/private key pair $\{PU_E,\ PR_E\}$ and transmits $\{PU_E,\ ID_A\}$ to B.

  - Assuming that the message received came from A, B generates the secret key K_S, encodes it with PU_E, and sends it back to A.

– This transmission from B is again **intercepted** by E, who for obvious reasons is able to decode the message.

– E now encodes the secret key K_S with A's public key PU_A and sends the encoded message back to A.

– A retrieves the secret key and, not suspecting any foul play, starts communicating with B using the secret key.

– E can now successfully eavesdrop on all communications between A and B.

13.3: CERTIFICATE AUTHORITIES FOR AUTHENTICATING YOUR PUBLIC KEY

- A **certificate** issued by a **certificate authority** (CA) authenticates your public key. Said simply, a certificate is your public key signed by the CA's private key.

- The CAs operate through a strict hierarchical organization in which the trust can only flow downwards. The CAs at the top of the hierarchy are known as **Root CAs**. The CAs below the root are generally referred to as **Intermediate-Level CAs**. Obviously, each root CA sits at the top of a tree-like structure of intermediate-level CAs. **Your computer comes pre-loaded with the public keys for the root CAs. In a Linux machine, these certificates typically reside in the directory "`/etc/ssl/certs/`". You can view any of these certificates by executing the command "`openssl x509 -text < cert_file_name`".**

- CA based authentication of a user is based on the assumption that when a new user applies to a CA for a certificate, the CA can authenticate the identity of the applicant through other means.

- There are three kinds of certificates, depending on the level of "identity assurance and authentication" that was carried out with regard to the applicant organization. At the highest level, you have the Extended Validation (EV) certificates that are issued only after a rigorous identity verification process for establishing the legitimacy of the applicant organization. This process may include verifying that the applicant organization has a legal and physical existence and the information provided by the applicant matches what can be gleaned from other government and other records. This process also includes a check on whether the applicant has exclusive rights to the domain specified in the application. **When you visit a website that offers such a certificate to your browser, some part of the URL window will turn green.** It may take several days for a CA to issue such a certificate. These are the most expensive certificates.

- At the next lower level of "identity assurance and authentication", we have Organization Validation (OV) certificates. Identity checks are less intense compared to those carried out for EV certificates. Usually, the existence of the organization is verified, the name of the domain is verified, which may be followed by a phone call from the CA.

- At the lowest level of identity and domain validation are the Domain Validation (DV) certificates. The only check that is made before such a certificate is issued is that the applicant has the right to use a specific domain name. This is done solely on the basis of the information you provide when applying for a certificate, by comparing the domain name for which you want a certificate against the database of the currently existing domain names, and by checking various internet directories as a check on the information you have provided. Such certificates are the least expensive and are normally issued in just a few minutes.

- As mentioned previously, a website offering an EV certificate will change a part of your URL window to green. **In the green portion, you are likely to see a padlock, a logo and the name of the organization to which the certificate was issued.** The other two types of certificates, OV and DV, will only show a padlock in the URL window. [For example, the website `https://engineering.purdue.edu` only shows a padlock in the URL window.]

- At the beginning of this section I mentioned that the CAs operate in a strict hierarchy, with the Root CAs at the top of the hierarchy, and with the Intermediate-Level CAs forming a tree structure under the Root CAs. **There is a very practical reason for why Intermediate-Level CAs are needed:** As I said earlier, the public keys for the Root CAs come pre-loaded with your computer (and also with the browsers). Now consider the situation that would arise should the private key of a Root CA become compromised for some reason. The only fix for that problem would be for you to update your software in order to replace the now defunct public key for the Root CA. But with billions of computers and digital devices around the world, there must exist hundreds of millions of devices for which the software is rarely updated if ever at all. You don't run into this problem when the private key of an Intermediate-Level CA is compromised. The affected certificates can now simply be added to a "Certificate Revocation List" maintained by a higher-level CA. The affected CA can then proceed to issue fresh certificates to the affected parties.

- With regard to how a tree of CAs is used for validating a certificate, consider a certificate issued by a CA that is not just below the root in the tree of CAs, but somewhere further down in the tree. Before your browser trusts such a certificate, it will verify the public key of the next higher level CA that validated the certificate your browser has received. This process is recursive until the root certificate that is pre-loaded in your computer is invoked. **In order to save your browser from having to make repeated requests for the certificates as it goes up the tree of CAs, the webserver that sent you the certificate you are specifically interested in may send the whole bundle of higher level certificates also.**

- At its minimum, a certificate assigned to a user consists of the user's public key, the identifier of the key owner, a time stamp (in the form of a period of validity), etc., **the whole block encrypted with the CA's private key**. Encrypting of the block with the CA's private key is referred to as **the CA having signed the certificate**. We may therefore express a certificate issued to party A by

$$C_A \quad = \quad E\left(PR_{CA},\ [T,\ ID_A,\ PU_A]\right)$$

where PR_{CA} is the private key of the Certificate Authority, T the expiration date/time for the A's public key PU_A that is being validated by the CA, and ID_A the party A's identifier.

- Subsequently, when party A presents his/her certificate to party B, the latter can verify the legitimacy of the certificate by decrypting it with the CA's public key. Successful decryption authenticates both the certificate supplied by A and A's public key. [**CRITICAL TO WHY CA BASED AUTHENTICATION WORKS:** If the CA happens to be a root CA, its public key is already stored in your computer. That is, parties A and B in our example are likely to have immediate access to the public keys for the root CAs without having to download them from anywhere. You'll also soon see why having the public keys for the root CAs already stored in your computer makes the whole thing work with a reasonable level of reliability.] At least theoretically speaking, this also provides B with authentication for A's identity since only the real A could have provided a legitimate certificate with A's identifier in it — since, as mentioned in the previous bullet, the CA will not issue a certificate containing A's ID to A unless the CA is certain about A's identity. [An important question here is that if a third party C manages to steal A's certificate, can C pose as A vis-a-vis B? Not really, unless C also manages to steal A's private key.]

- Having established the certificate's legitimacy, having authenticated A, and having acquired A's public key, B responds back to

A with its own certificate. A processes B's certificate in the same manner as B processed A's certificate. [B **responding back with its own certificate makes for a two-way authentication. Most of the business transactions in e-commerce utilize only one-way authentication. To illustrate, before you upload your credit-card info to Amazon.com, your laptop must make certain that the website at the other end is truly Amazon.com. There is no need for Amazon.com to authenticate you or your laptop directly. Obviously, Amazon.com wants to get paid for the items ordered by you — that's something it does not need to worry about after your credit card info is accepted by the issuer of the card.**]

- This exchange results in A and B acquiring **authenticated public keys** for each other. The important thing to note here is that each of the two parties A and B acquires the other party's public key not directly but through the other party's certificate.

- The upper half of Figure 1 shows this approach to user and public key authentication. Next, we will explain the protocol that A and B use to exchange a secret session key. This is done with the help of the four messages shown in the bottom half of the figure.

- Another acronym closely related to CA is RA, which stands for Registration Authority. RAs act as resellers of certificates for CAs. That means, instead of directly approaching a particular CA for signing your certificate, you may approach an RA that works for the CA. RAs are not to be confused with intermediate level CAs. An intermediate level CA is a CA that is not the root CA (see Section 13.4 for what that means) and that issues a certificate under its own signature. On the other hand, an RA for a given CA is simply a conduit for obtaining a certificate signed by that CA. [See Section 13.8 for how an attacker recently compromised the security of an RA working for Comodo, a well-known root CA, and obtained forged certificates for some prominent domains.]

- As mentioned earlier in this section, in most practical situations involving e-commerce, what actually transpires between a client, such as your laptop, and an e-commerce website like Amazon.com is less elaborate than what is shown in the figure on the next page. That is for two reasons: (1) It is highly likely that a client will not possess a certificate; and (2) while it is important for your laptop to authenticate Amazon.com, the company does not really care as to who you are as long as your credit-card information proves to be valid. Therefore, a typical connection with an e-commerce website will involve only one-way authentication. Your laptop will request Amazon.com's certificate, verify its validity, use the Amazon.com's verified public key to encrypt a session key, and, finally, transmit the encrypted session key to the Amazon.com's website.

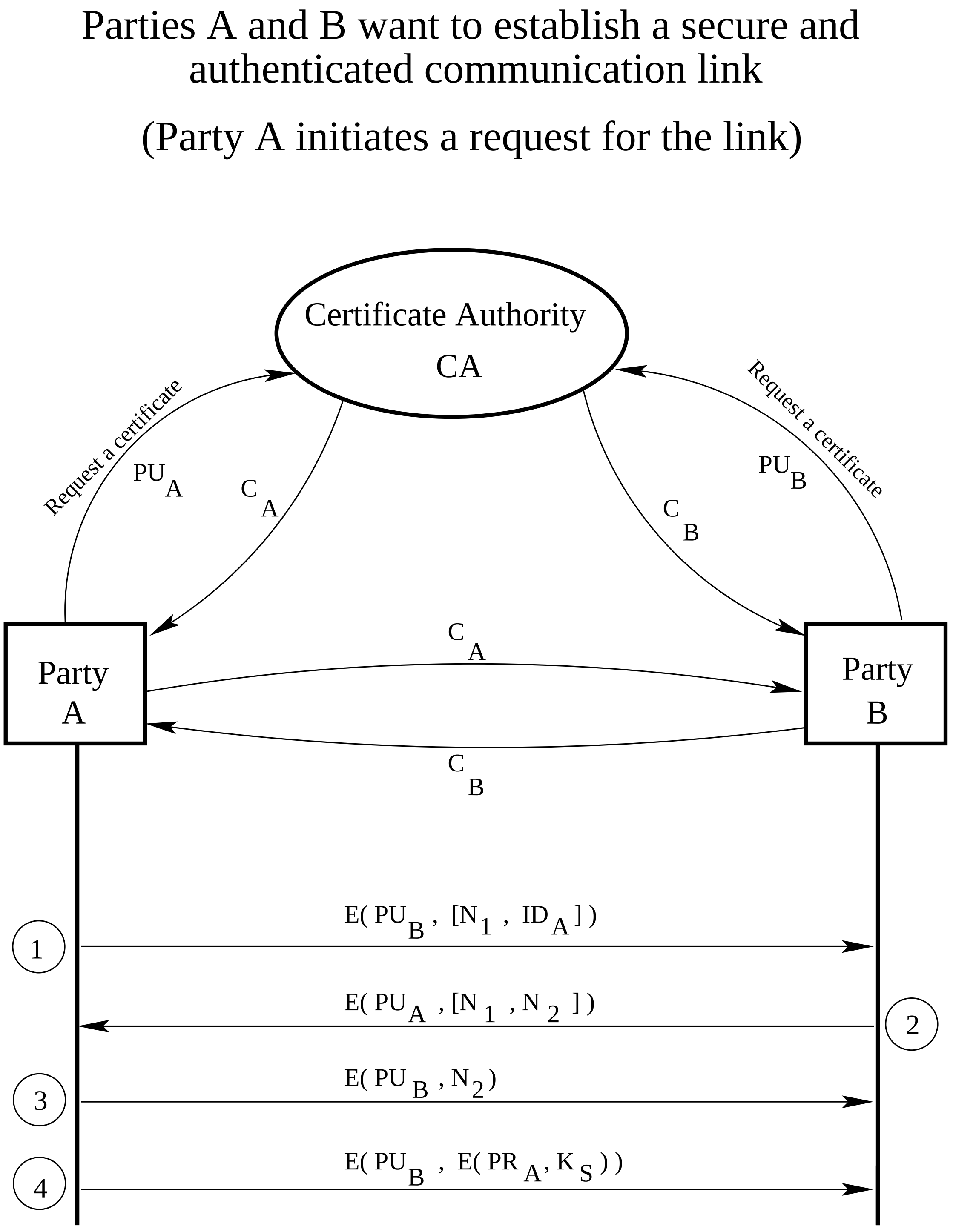
Parties A and B want to establish a secure and authenticated communication link
(Party A initiates a request for the link)
Certificate Authority CA
Request a certificate
PU_A
C_A
Request a certificate
PU_B
C_B
Party A
Party B
C_A
C_B
1
$E(PU_B , [N_1 , ID_A])$
2
$E(PU_A , [N_1 , N_2])$
3
$E(PU_B , N_2)$
4
$E(PU_B , E(PR_A, K_S))$

Figure 1: *Messages exchanged between two parties for acquiring each other's CA authenticated public keys.* (*This figure is from Lecture 13 of "Computer and Network Security" by Avi Kak.*)

13.3.1: Using Authenticated Public Keys to Exchange a Secret Session Key

- Having acquired the public keys (and having **cached** them for future use), the two parties A and B then proceed to exchange a secret session key.

- The bottom half of Figure 1 shows the messages exchanged for establishing the secret key.

- A uses B's public key PU_B to encrypt a message that contains A's identifier ID_A and a nonce N_1 as a transaction identifier. A sends this encrypted message to B. This message can be expressed as

$$E\,(PU_B, [N_1,\ ID_A])$$

- B responds back with a message encrypted using A's public key PU_A, the message containing A's nonce N_1 and new nonce N_2 from B to A. The structure of this message can be expressed as

$$E\,(PU_A, [N_1,\ N_2])$$

Since only B could have decrypted the first message from A to B, the presence of the nonce N_1 in this response from B further assures A that the responding party is actually B (since only B could have decrypted the original message containing the nonce N_1).

- A now selects a secret session key K_S and sends B the following message

$$M \quad = \quad E\left(PU_B,\ E\left(PR_A,\ K_S\right)\right)$$

Note that A encrypts the secret key K_S with his/her own private key PR_A before further encrypting it with B's public key PU_B. Encryption with A's **private key** makes it possible for B to authenticate the sender of the secret key. Of course, the further encryption with B's **public key** means that only B will be able to read it.

- B decrypts the message first with its own private key PR_B and then recovers the secret key by applying another round of decryption using A public key PU_A.

13.4: THE X.509 CERTIFICATE FORMAT STANDARD FOR PUBLIC KEY INFRASTRUCTURE (PKI)

- The set of standards related to the creation, distribution, use, **and revocation** of digital certificates is referred to as the **Public Key Infrastructure** (PKI). [In addition to PKI, another acronym that you will see frequently in the present context is PKCS, which, as previously mentioned in Section 12.6 of Lecture 12, stands for Public Key Cryptography Systems. If you search for information on the web, you will frequently see references to documents and protocols under the tag PKCS#N where N is usually a small integer. As stated in Lecture 12, these documents were produced by the RSA corporation that has been responsible for many of the PKI standards. Several of these documents eventually became IETF standards under the names that begin with RFC followed by a number. IETF stands for the Internet Engineering Task Force. A large number of standards that regulate the workings of the internet are IETF documents. Check them out at the `http://www.ietf.org` web page and find out about how the internet standardization process works.]

- **X.509** is one of the PKI standards. Besides other things, it is this standard that specifies the format of digital certificates. The X.509 standard is described in the IETF document RFC 5280 (also see its recent update in RFC 6818). [Just googling a string like "rfc5280" will take you directly to the source of such documents.]

- The X.509 standard is based on a strict hierarchical organization of the CAs in which the trust can only flow downwards. As mentioned previously at the beginning of Section 13.3, the CAs at the top of the hierarchy are known as **root CAs**. The CAs below the root are generally referred to as intermediate-level CAs.

- In order to verify the credentials of a particular CA as the issuer of a certificate, you approach the higher level CA for the needed verification. Obviously, this approach for establishing trust assumes that the root level CA must always be trusted implicitly.

- **IMPORTANT: The public keys of the root CAs, of which VeriSign, Comodo, and so on, are examples, are incorporated in your browser software and other applications that require networking so that the root-level verification is not subject to network-based man-in-the-middle attacks.** This also enables quick local authentication at the root level. In Linux machines, you'll find the root CA certificates in "`/etc/ssl/certs/`". [By the way, the status of the root CAs is verified annually by designated agencies. For example, Comodo's annual status as a root CA is verified annually by the global accounting firm KPMG. Again as a side note, Comodo owns 11 root keys. VeriSign is apparently the largest owner of root keys; it owns 13 root keys.]

- **For web-based applications, a certificate that cannot be authenticated by going up the chain of CAs all the way up to a root CA generates a warning popup from the browser.**

- The format of an X.509 certificate is shown in Figure 2. The different fields of this certificate are described below:

  - **Version Number**: This describes the version of the X.509 standard to which the certificate corresponds. We are now on the third version of this standard. Since the entry in this field is zero based, so you'd see 2 in this field for the certificates that correspond to the latest version of the standard.

  - **Serial Number**: This is the serial number assigned to a certificate by the CA.

  - **Signature Algorithm ID**: This is the name of the digital signature algorithm used to sign the certificate. The signature itself is placed in the last field of the certificate.

  - **Issuer Name**: This is the name of the Certificate Authority that issued this certificate.

  - **Validity Period**: This field states the time period during which the certificate is valid. The period is defined with two

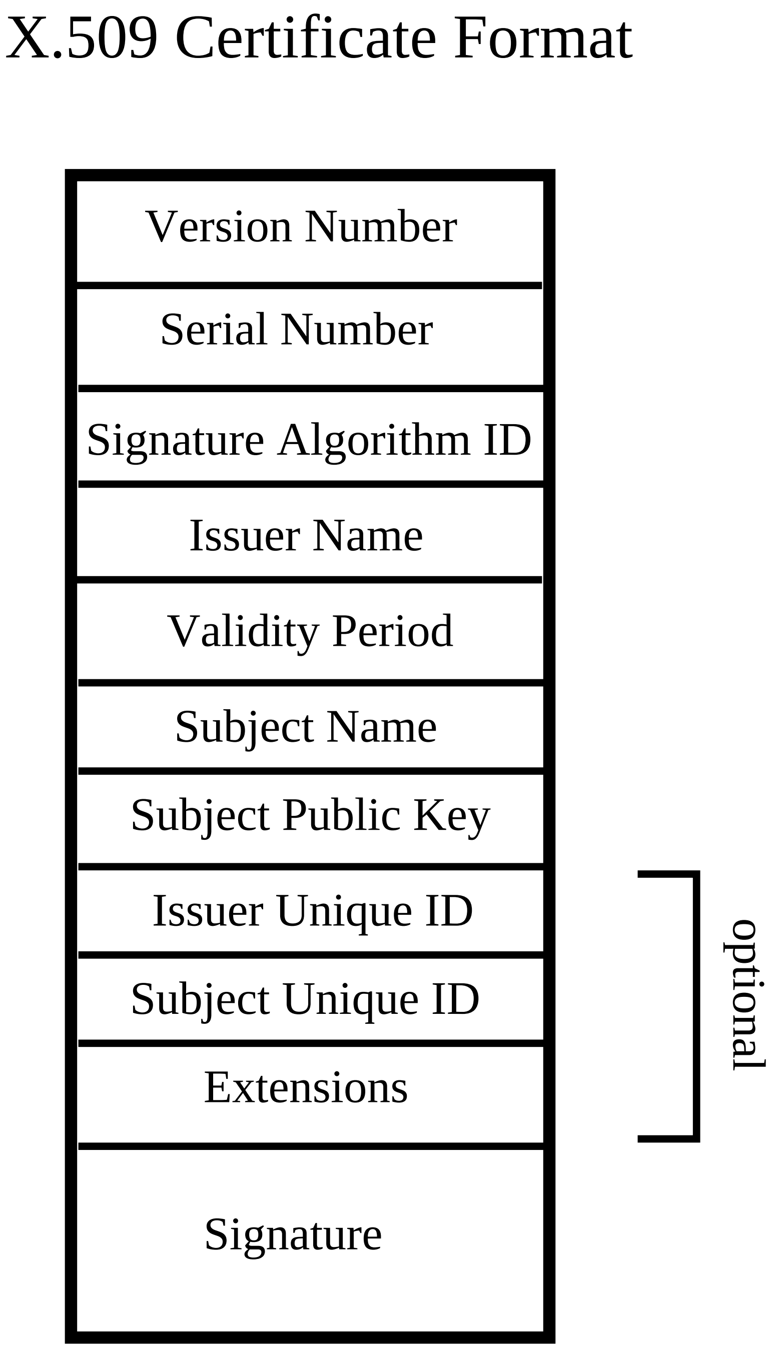

Figure 2: *The different fields of an X.509 certificate.* (*This figure is from Lecture 13 of "Computer and Network Security" by Avi Kak.*)

date-times, a **not before** date-time and a **not after** date-time.

– **Subject Name**: This field identifiers the individual/organization to which the certificate was issued. In other words, this field names the entity that wants to use this certificate to authenticate the public key that is in the next field.

– **Subject Public Key**: This field presents the public key that is meant to be authenticated by this certificate. This field also names the algorithm used for public-key generation.

– **Issuer Unique Identifier**: (optional) With the help of this identifier, two or more different CA's can operate as logically a single CA. The **Issuer Name** field will be distinct for each such CA but they will share the same value for the **Issuer Unique Identifier**.

– **Subject Unique Identifier**: (optional) With the help of this identifier, two or more different certificate holders can act as a single logical entity. Each holder will have a different value for the **Subject Name** field but they will share the same value for the **Subject Unique Identifier** field.

– **Extensions**: (optional) This field allows a CA to add additional private information to a certificate.

– **Signature:** This field contains the digital signature by the issuing CA for the certificate. **This signature is obtained by first computing a message digest of the rest of the fields with a hashing algorithm like SHA-1 (See Lecture 15) and then encrypting it with the CA's private key.** Authenticity of the contents of the certificate can be verified by using CA's public key to retrieve the message digest and then by comparing this digest with one computed from the rest of the fields.

- The digital representation of an X.509 certificate, described in RFC 5280, is created by first using the following ASN.1 representation to generate a byte stream for the certificate and converting the bytestream into a printable form with Base64 encoding. [As mentioned in Section 12.8 of Lecture 12, ASN stands for Abstract Syntax Notation and the ASN.1 standard, along with its transfer encoding DER (for Distinguished Encoding Rules), accomplishes the same thing in binary format for complex data structures that the XML standard does in textual format.] Shown below is the ASN.1 representation of an X.509 certificate:

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
     issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                          -- If present, version MUST be v2 or v3
     extensions      [3]  EXPLICIT Extensions OPTIONAL
```

```
                         -- If present, version MUST be v3
     }

Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }

CertificateSerialNumber  ::=  INTEGER

Validity ::= SEQUENCE {
     notBefore      Time,
     notAfter       Time }

Time ::= CHOICE {
     utcTime        UTCTime,
     generalTime    GeneralizedTime }

UniqueIdentifier  ::=  BIT STRING

SubjectPublicKeyInfo  ::=  SEQUENCE  {
     algorithm            AlgorithmIdentifier,
     subjectPublicKey     BIT STRING  }

Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

Extension  ::=  SEQUENCE  {
     extnID      OBJECT IDENTIFIER,
     critical    BOOLEAN DEFAULT FALSE,
     extnValue   OCTET STRING
                 -- contains the DER encoding of an ASN.1 value
                 -- corresponding to the extension type identified
                 -- by extnID
```

- It is the hash of the bytestream that corresponds to what is stored for the field `TBSCertificate` that is encrypted by the CA's private key for the digital signature that then becomes the value of the `signatureValue` field. You may read `TBSCertificate` as the "To Be Signed" potion of what appears in the final certificate. As to what algorithms are used for hashing and for encryption with the CA's private key, that is identified by the value of the field `signatureAlgorithm`.

- Using the Base64 representation (see Lecture 2), an X.509 certificate is commonly stored in a printable form according to the RFC 1421 standard. In its printable form, a certificate will normally be bounded by the first string shown below at the beginning and the second at the end.

```
    -----BEGIN CERTIFICATE-----


    -----END CERTIFICATE-----
```

Shown below is an example of a certificate in Base64 representation and it resides in a file whose name carries the ".pem" suffix. The programming problem in Section 13.9 has more to say about the PEM format for representing keys and certificates.

```
-----BEGIN CERTIFICATE-----
MIIDJzCCApCgAwIBAgIBATANBgkqhkiG9w0BAQQFADCBzjELMAkGA1UEBhMCWkEx
FTATBgNVBAgTDFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJQ2FwZSBUb3duMR0wGwYD
VQQKExRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCYGA1UECxMfQ2VydGlmaWNhdGlv
biBTZXJ2aWNlcyBEaXZpc2lvbjEhMB8GA1UEAxMYVGhhd3RlIFByZW1pdW0gU2Vy
dmVyIENBMSgwJgYJKoZIhvcNAQkBFhlwcmVtaXVtLXNlcnZlckB0aGF3dGUuY29t
MB4XDTk2MDgwMTAwMDAwMFoXDTIwMTIzMTIzNTk1OVowgc4xCzAJBgNVBAYTAlpB
MRUwEwYDVQQIEwxXZXN0ZXJuIENhcGUxEjAQBgNVBAcTCUNhcGUgVG93bjEdMBsG
A1UEChMUVGhhd3RlIENvbnN1bHRpbmcgY2MxKDAmBgNVBAsTH0NlcnRpZmljYXRp
b24gU2VydmljZXMgRGl2aXNpb24xITAfBgNVBAMTGFRoYXd0ZSBQcmVtaXVtIFNl
cnZlciBDQTEoMCYGCSqGSIb3DQEJARYZcHJlbWl1bS1zZXJ2ZXJAdGhhd3RlLmNv
bTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwgYkCgYEA0jY2aovXwlue2oFBYo847kkE
VdbQ7xwblRZH7xhINTpS9CtqBo87L+pW46+GjZ4X9560ZXUCTe/LCaIhUdib0GfQ
ug2SBhRz1JPLlyoAnFxODLz6FVL88kRu2hFKbgifLy3j+ao6hnO2RlNYyIkFvYMR
uHM/qgeN9EJN50CdHDcCAwEAAaMTMBEwDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG
9w0BAQQFAAOBgQAmSCwWwlj66BZ0DKqqX1Q/8tfJeGBeXm43YyJ3Nn6yF8Q0ufUI
hfzJATj/Tb7yFkJD57taRvvBxhEf8UqwKEbJw8RCfbz6q1lu1bdRiBHjpIUZa4JM
pAwSremkrj/xw0llmozFyD4lt5SZu5IycQfwhl7tUCemDaYj+bvLpgcUQg==
-----END CERTIFICATE-----
```

- Ordinarily you would request a CA for a certificate for your public key. But that does not prevent you from generating your own certificates for testing purposes. If you have Ubuntu installed on your machine, try out the following command:

```
openssl req -new -newkey rsa:1024 -days 365 -nodes -x509 -keyout test.pem -out test.cert
```

where the first argument **`req`** to **`openssl`** is for generating an X509 certificate, the rest of the arguments being self-explanatory. This command will deposit a new private key for you in the file **`test.pem`** and the certificate in the file **`test.cert`**. [By the way, OpenSSL, the open-source library that supports the command **`openssl`** used above, is an amazingly useful library in C that implements the SSL/TLS protocol (that we will take up in greater depth in Lecture 20). It contains production-quality code for virtually anything you would ever want to do with cryptography — symmetric-key cryptography, public-key cryptography, hashing, certificate generation, etc. Check it out at **`www.openssl.org`**. If you are running Ubuntu and you have OpenSSL installed, do **`man openssl`** to see all the things that you can do with the command shown above as you give it different arguments.] When you invoke the above command, it will ask you for information related to you and your organization. It is not necessary to supply the information that you are prompted for, though.

- You can also use OpenSSL to make your own organization a CA. Visit `http://sandbox.rulemaker.net/ngps/m2/howto.ca.html` to find out how you can do it.

- Shown on the next page is the X.509 certificate that belongs to the InCommon root CA (`https://www.incommon.org/`). InCommon is used by several universities and research organizations in the US for data encryption for web servers. The certificate shown below can be downloaded from `https://www.incommon.org/cert/repository/InCommonServerCA.txt`.

- To see the role played by the InCommon's certificate shown on the next page, let's say the web browser in your computer requests a page from the `engineering.purdue.edu` web server that I use for hosting my computer and network security lecture notes. This server supplies all its content using the TLS/SSL protocol, meaning that all interactions with this server are encrypted. In order to create an encrypted session with the server, your browser first downloads `engineering.purdue.edu`'s certificate — which is signed by InCommon — and then authenticates it through InCommons's public key that is supplied by their own certificate shown on the next page. **IMPORTANT**: Note that InCommon is an intermediate level CA whose own certificate is signed by a root CA called AddTrust. Being a root CA, AddTrust's public key (in the form of a self-signed certificate) comes preloaded in your computer and resides in the directory "`/etc/ssl/certs/`". As mentioned earlier in this lecture, you can view any of these certificates by executing the command "`openssl x509 -text < cert_file_name`". Being preloaded in your computer, the acquisition of AddTrust's public key is NOT vulnerable to man-in-the-middle attack. The web browser running in your computer and the `engineering.purdue.edu`'s web server use the SSL/TLS protocol to create a session that cannot be eavesdropped on. For that, your browser first downloads the `engineering.purdue.edu`'s certificate as already mentioned. From the URL provided in this certificate to the InCommon web site, your browser next downloads the InCommon's certificate that is shown below. Next, it verifies InCommon's certificate using the pre-stored AddTrust certificate in the directory `/etc/ssl/certs/`. Subsequently, it uses the public key in the authenticated InCommon's certificate to authenticate the public key in `engineering.purdue.edu`'s certificate. Shown below is InCommon's certificate:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            7f:71:c1:d3:a2:26:b0:d2:b1:13:f3:e6:81:67:64:3e
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=SE, O=AddTrust AB, OU=AddTrust External TTP Network, CN=AddTrust External CA Root
        Validity
            Not Before: Dec  7 00:00:00 2010 GMT
            Not After : May 30 10:48:38 2020 GMT
        Subject: C=US, O=Internet2, OU=InCommon, CN=InCommon Server CA
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (2048 bit)
                Modulus (2048 bit):
                    00:97:7c:c7:c8:fe:b3:e9:20:6a:a3:a4:4f:8e:8e:
                    34:56:06:b3:7a:6c:aa:10:9b:48:61:2b:36:90:69:
                    e3:34:0a:47:a7:bb:7b:de:aa:6a:fb:eb:82:95:8f:
                    ca:1d:7f:af:75:a6:a8:4c:da:20:67:61:1a:0d:86:
                    c1:ca:c1:87:af:ac:4e:e4:de:62:1b:2f:9d:b1:98:
                    af:c6:01:fb:17:70:db:ac:14:59:ec:6f:3f:33:7f:
                    a6:98:0b:e4:e2:38:af:f5:7f:85:6d:0e:74:04:9d:
                    f6:27:86:c7:9b:8f:e7:71:2a:08:f4:03:02:40:63:
                    24:7d:40:57:8f:54:e0:54:7e:b6:13:48:61:f1:de:
                    ce:0e:bd:b6:fa:4d:98:b2:d9:0d:8d:79:a6:e0:aa:
                    cd:0c:91:9a:a5:df:ab:73:bb:ca:14:78:5c:47:29:
                    a1:ca:c5:ba:9f:c7:da:60:f7:ff:e7:7f:f2:d9:da:
                    a1:2d:0f:49:16:a7:d3:00:92:cf:8a:47:d9:4d:f8:
                    d5:95:66:d3:74:f9:80:63:00:4f:4c:84:16:1f:b3:
                    f5:24:1f:a1:4e:de:e8:95:d6:b2:0b:09:8b:2c:6b:
                    c7:5c:2f:8c:63:c9:99:cb:52:b1:62:7b:73:01:62:
                    7f:63:6c:d8:68:a0:ee:6a:a8:8d:1f:29:f3:d0:18:
                    ac:ad
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Authority Key Identifier:
                keyid:AD:BD:98:7A:34:B4:26:F7:FA:C4:26:54:EF:03:BD:E0:24:CB:54:1A

            X509v3 Subject Key Identifier:
                48:4F:5A:FA:2F:4A:9A:5E:E0:50:F3:6B:7B:55:A5:DE:F5:BE:34:5D
            X509v3 Key Usage: critical
                Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE, pathlen:0
            X509v3 Certificate Policies:
                Policy: X509v3 Any Policy

            X509v3 CRL Distribution Points:
                URI:http://crl.usertrust.com/AddTrustExternalCARoot.crl

            Authority Information Access:
                CA Issuers - URI:http://crt.usertrust.com/AddTrustExternalCARoot.p7c
                CA Issuers - URI:http://crt.usertrust.com/AddTrustUTNSGCCA.crt
                OCSP - URI:http://ocsp.usertrust.com

    Signature Algorithm: sha1WithRSAEncryption
        93:66:21:80:74:45:85:4b:c2:ab:ce:32:b0:29:fe:dd:df:d6:
        24:5b:bf:03:6a:6f:50:3e:0e:1b:b3:0d:88:a3:5b:ee:c4:a4:
        12:3b:56:ef:06:7f:cf:7f:21:95:56:3b:41:31:fe:e1:aa:93:
        d2:95:f3:95:0d:3c:47:ab:ca:5c:26:ad:3e:f1:f9:8c:34:6e:
```

```
        11:be:f4:67:e3:02:49:f9:a6:7c:7b:64:25:dd:17:46:f2:50:
        e3:e3:0a:21:3a:49:24:cd:c6:84:65:68:67:68:b0:45:2d:47:
        99:cd:9c:ab:86:29:11:72:dc:d6:9c:36:43:74:f3:d4:97:9e:
        56:a0:fe:5f:40:58:d2:d5:d7:7e:7c:c5:8e:1a:b2:04:5c:92:
        66:0e:85:ad:2e:06:ce:c8:a3:d8:eb:14:27:91:de:cf:17:30:
        81:53:b6:66:12:ad:37:e4:f5:ef:96:5c:20:0e:36:e9:ac:62:
        7d:19:81:8a:f5:90:61:a6:49:ab:ce:3c:df:e6:ca:64:ee:82:
        65:39:45:95:16:ba:41:06:00:98:ba:0c:56:61:e4:c6:c6:86:
        01:cf:66:a9:22:29:02:d6:3d:cf:c4:2a:8d:99:de:fb:09:14:
        9e:0e:d1:d5:c6:d7:81:dd:ad:24:ab:ac:07:05:e2:1d:68:c3:
        70:66:5f:d3
-----BEGIN CERTIFICATE-----
MIIEwzCCA6ugAwIBAgIQf3HB06ImsNKxE/PmgWdkPjANBgkqhkiG9w0BAQUFADBv
MQswCQYDVQQGEwJTRTEUMBIGA1UEChMLQWRkVHJ1c3QgQUIxJjAkBgNVBAsTHUFk
ZFRydXN0IEV4dGVybmFsIFRUUCBOZXR3b3JrMSIwIAYDVQQDExlBZGRUcnVzdCBF
eHRlcm5hbCBDQSBSb290MB4XDTEwMTIwNzAwMDAwMFoXDTIwMDUzMDEwNDgzOFow
UTELMAkGA1UEBhMCVVMxEjAQBgNVBAoTCUludGVybmV0MjERMA8GA1UECxMISW5D
b21tb24xGzAZBgNVBAMTEkluQ29tbW9uIFNlcnZlciBDQTCCASIwDQYJKoZIhvcN
AQEBBQADggEPADCCAQoCggEBAJd8x8j+s+kgaqOkT46ONFYGs3psqhCbSGErNpBp
4zQKR6e7e96qavvrgpWPyh1/r3WmqEzaIGdhGg2GwcrBh6+sTuTeYhsvnbGYr8YB
+xdw26wUWexvPzN/ppgL5OI4r/V/hWOOdASd9ieGx5uP53EqCPQDAkBjJH1AV49U
4FR+thNIYfHezg69tvpNmLLZDY15puCqzQyRmqXfq3O7yhR4XEcpocrFup/H2mD3
/+d/8tnaoS0PSRan0wCSz4pH2U341ZVm03T5gGMAT0yEFh+z9SQfoU7e6JXWsgsJ
iyxrx1wvjGPJmctSsWJ7cwFif2Ns2Gig7mqojR8p89AYrK0CAwEAAaOCAXcwggFz
MB8GA1UdIwQYMBaAFK29mHo0tCb3+sQmVO8DveAky1QaMB0GA1UdDgQWBBRIT1r6
L0qaXuBQ82t7VaXe9b40XTAOBgNVHQ8BAf8EBAMCAQYwEgYDVR0TAQH/BAgwBgEB
/wIBADARBgNVHSAECjAIMAYGBFUdIAAwRAYDVR0fBD0wOzA5oDegNYYzaHR0cDov
L2NybC51c2VydHJ1c3QuY29tL0FkZFRydXN0RXh0ZXJuYWxDQVJvb3QuY3JsMIGz
BggrBgEFBQcBAQSBpjCBozA/BggrBgEFBQcwAoYzaHR0cDovL2NydC51c2VydHJ1
c3QuY29tL0FkZFRydXN0RXh0ZXJuYWxDQVJvb3QucDdjMDkGCCsGAQUFBzAChi1o
dHRwOi8vY3J0LnVzZXJ0cnVzdC5jb20vQWRkVHJ1c3RVVE5TR0NDQS5jcnQwJQYI
KwYBBQUHMAGGGWh0dHA6Ly9vY3NwLnVzZXJ0cnVzdC5jb20wDQYJKoZIhvcNAQEF
BQADggEBAJNmIYB0RYVLwqvOMrAp/t3f1iRbvwNqb1A+DhuzDYijW+7EpBI7Vu8G
f89/IZVWO0Ex/uGqk9KV85UNPEerylwmrT7x+Yw0bhG+9GfjAkn5pnx7ZCXdF0by
UOPjCiE6SSTNxoRlaGdosEUtR5nNnKuGKRFy3NacNkN089SXnlag/l9AWNLV1358
xY4asgRckmYOha0uBs7Io9jrFCeR3s8XMIFTtmYSrTfk9e+WXCAONumsYn0ZgYr1
kGGmSavOPN/mymTugmU5RZUWukEGAJi6DFZh5MbGhgHPZqkiKQLWPc/EKo2Z3vsJ
FJ4OOdXG14HdrSSrrAcF4h1ow3BmX9M=
-----END CERTIFICATE-----
```

- Since all valid certificates are cached by your browser, if you previously visited the `engineering.purdue.edu` domain, the InCommon certificate I showed above is probably already in your computer. You can check whether or not that's the case through your browser's certificate viewer tool. For FireFox, you can get to the certificate viewer by clicking on the "edit" button in the menu bar of the browser and by further clicking as shown below:

```
Preferences -->
    Advanced -->
        Certificates -->
```

```
"View Certificates" button -->
    "Authorities" to view the CA certificates -->
        Scroll down to "AddTrust AB" -->
            Further scroll down to "InCommon Server CA"
```

where the last item will show up only if you previously visited the `engineering.purdue.edu` domain. Assuming it is there, when you double-click on the last item, you will see a popup with two buttons. The left button leads you to general information regarding the root CA and the right button shows the details regarding the root certificate through a tree structure. When you click on "Subject's public key", you will see the modulus and the public exponent used by this root. In the general information provided by the left button, you will notice that the serial number of the root certificate matches that of the root certificate that I downloaded directly from InCommon's web site and that is reproduced above.

- If you want to view the root CA certificates that have been deposited in your browser by different internet service provides (after they were verified by your browser), in the fifth action item in the indented list of actions shown above, click on "Servers".

13.4.1: Harvesting RSA Moduli From X.509 Certificates — Perl and Python Code

- As you now know from Section 12.8 of Lecture 12, if an attacker can somehow obtain two different moduli used for RSA cryptography from anywhere in the internet, and if it should happen that these moduli share a common factor, then the attacker can quickly determine the second factor in both the moduli and thus compromise the security of both hosts. What that means is that harvesting RSA moduli from the internet is a useful activity for network security research.

- Shown in this section is a script that you can use to harvest the moduli and the public exponents used in the X.509 SSL/TLS certificates around the world.

- The script uses **`gnutls-cli`** as a command-line SSL/TSL client to make a connection with the remote host on its port 443. On Linux/Ubuntu platforms, this utility is a part of the **`gnutls-bin`** package that you can download with the Synaptic Package Manager. [Port 443 is to the HTTPS protocol what port 80 is to the HTTP protocol. Secure web servers, such as those used by websites that require you to upload your credit-card information, must use the HTTPS protocol so that they can be authenticated by your computer before you upload your credit card information. HTTPS stands for "HTTP Secure," as you'd guess. The HTTPS protocol depends on X.509 certificates for the authentication of at least the server by the client and, sometimes, the authentication for both endpoints of a communication link.]

- With regard to the code in the script, the main point to note that since the IP addresses are selected purely randomly, a destination IP address is highly unlikely to be hosting an HTTPS server. So it is important to check that the port 443 is open at the destination and your computer can make a TCP connection with that port.

- Subsequent to making a successful connection, the script calls on the **`gnutls-cli`** client to download all the certificates offered by the remote host. It is common for large web sites to offer multiple certificates. The script then uses **`openssl`** commands to process each certificate for the extraction of the modulus and public key as stored in the certificate.

- For geographically distant hosts, the results you get will depend much on the value given to the **`Timeout`** option in the call to the socket constructor. You may want to experiment with larger values if the modulus yield is poor.

- The script as presented has the **`$NHOSTS`** set to 200, meaning that it will randomly select 200 hosts from the space of all IP addresses. You can change this value to whatever you want.

- Note that the moduli harvested are dumped cumulatively in a file named **`Dumpfile.txt`**. If you are just playing with this code, you may want to empty that file every once in a while.

```
#!/usr/bin/env perl

### ModulusHarvestor.pl

### Author:    Avi Kak (kak@purdue.edu)
### Date:      April 22, 2014
### Modified:  February 23, 2016

##  The script can be used in following two different modes:
##
##    --- With no command-line args.  In this case, the script scans the internet
##        with randomly synthesized IP addresses and, when it finds a site with its
##        port 443 open, it grabs the certificate(s) offered by that site and
##        extracts the various certificate parameters (modulus, public exponent,
##        etc.) from the certificate(s).
##
##    --- With just one command-line arg, which must be an IPv4 address.  In this
##        case, the script will try to connect with that address on its port 443 and
##        download the certificate offered by it.  So as not to waste your time, it
##        is best if you use an IP address that does offer an HTTPS service.  You can
##        check that with a simple port scanner like 'port_scan.pl' we will cover in
##        Lecture 16.

##  The basic purpose of this script is to harvest RSA moduli used for public keys in
##  SSL/TLS certificates.  Recent research has demonstrated that if two different
##  moduli share a common factor, they can both be factored easily, thus compromising
##  the security of both.

##  For harvesting moduli, the script first randomly selects $NHOSTS number of hosts
##  from the space of all possible IP addresses and tries to download their X.509
##  certificates using a GnuTLS client.  It subsequently extracts the modulus and
##  public key used in the certificates using openssl commands.  These are finally
##  dumped into a file called Dumpfile.txt.

use IO::Socket;                                                                  #(A1)
use Math::BigInt;                                                                #(A2)
use strict;
use warnings;

our $debug = 1;                                                                  #(A3)

our $mark1 = "-----BEGIN CERTIFICATE-----";                                      #(A4)
our $mark2 = "-----END CERTIFICATE-----";                                        #(A5)
our $dumpfile = "Dumpfile.txt";                                                  #(A6)
open DUMP, ">> $dumpfile";                                                       #(A7)
our @ip_addresses_to_scan;                                                       #(A8)
```

```
unless (@ARGV) {                                                              #(B1)
    our $NHOSTS = 200;                                                        #(B2)
    @ip_addresses_to_scan = @{get_fresh_ipaddresses($NHOSTS)};               #(B3)
} elsif (@ARGV == 1) {                                                        #(B4)
    @ip_addresses_to_scan = ($ARGV[0]);                                       #(B5)
} else {                                                                      #(B6)
    die "You cannot call $0 with more than one command-line argument\n";      #(B7)
}

foreach my $ip_address (@ip_addresses_to_scan) {                              #(C1)
    print "\nTrying IP address: $ip_address\n\n\n";                          #(C2)
    my $sock = IO::Socket::INET->new(PeerAddr => $ip_address,                 #(C3)
                                     PeerPort => 443,                         #(C4)
                                     Timeout  => "0.1",                       #(C5)
                                     Proto => 'tcp');                         #(C6)
    if ($sock) {                                                              #(C7)
        print DUMP "$ip_address\n\n";                                         #(C8)
        # The --print-cert option outputs the certificate in PEM format.
        # The --insecure option says not to insist on validating the certificate
        my $output = `gnutls-cli --insecure --print-cert $ip_address < /dev/null`;
                                                                              #(C9)
        my @certificates = $output =~ /$mark1(.+?)$mark2/gs;                  #(C10)
        my $howmany_certs = @certificates;                                    #(C11)
        print "Found $howmany_certs certificates\n\n" if $debug;              #(C12)
        foreach my $i (1..@certificates) {                                    #(C13)
            print "Certificate $i:\n\n" if $debug;                            #(C14)
            print "$certificates[$i-1]\n\n" if $debug;                        #(C15)
            open FILE, ">__temp.cert";                                        #(C16)
            print FILE "$mark1$certificates[$i-1]$mark2\n";                   #(C17)
            my $cert_text = `openssl x509 -text < __temp.cert`;               #(C18)
            print "$cert_text\n\n\n" if $debug;                               #(C19)
            my @all_lines = split /\s+/, $cert_text;                          #(C20)
            $cert_text = join '', grep $_, @all_lines;                        #(C21)
            my @params = $cert_text =~ /Modulus:(.+?)Exponent:(\d+)/gs;       #(C22)
            my $modulus ="0x" . join '', split /:/, $params[0];               #(C23)
            if ($debug) {                                                     #(C24)
                print "Modulus: \n";                                          #(C25)
                print Math::BigInt->new($modulus)->as_int();                  #(C26)
                print "\n\n";                                                 #(C27)
                print "Public exponent: $params[1]\n";                        #(C28)
                print "\n\n\n";
            }
            print DUMP "Modulus:\n";                                          #(C29)
            print DUMP Math::BigInt->new($modulus)->as_int();                 #(C30)
            print DUMP "\n\nPublic Exponent: $params[1]\n\n\n";               #(C31)
            unlink "__temp.cert";                                             #(C32)
        }
        print DUMP "\n\n\n";                                                  #(C33)
    }
}

## This subroutine was borrowed from the AbraWorm.pl code in Lecture 22.
sub get_fresh_ipaddresses {                                                   #(D1)
    my $howmany = shift || 0;                                                 #(D2)
    return 0 unless $howmany;                                                 #(D3)
    my @ipaddresses;                                                             #(D4)
    foreach my $i (0..$howmany-1) {                                              #(D5)
        my ($first,$second,$third,$fourth) =
                                  map {1 + int(rand($_))} (223,223,223,223);     #(D6)
        push @ipaddresses, "$first\.$second\.$third\.$fourth";                   #(D7)
    }
    return \@ipaddresses;                                                        #(D8)
}
```

- As mentioned in the comment block of the script, you can call this script with a single command-line argument if you want to see what exactly the script outputs without becoming overwhelmed by the output produced for a large number of certificates from many different websites. For example, if you make the call

  ```
  ModulusHarvestor.pl  170.149.159.130
  ```

  you will see the various parameters for all three certificates offered by `nyt.com`, which is the main website for The New York Times.

- However, when you call the script without any command-line args, you are likely to see an output like

  ```
  Trying IP address: 165.157.50.192

  Trying IP address: 157.156.164.166

  Trying IP address: 134.52.117.53

  Trying IP address: 27.99.72.169

  Trying IP address: 82.162.146.185

  Trying IP address: 92.127.112.199

  ...
  ...
  ```

Whenever the script finds an IP address that offers HTTPS service, it will download its certificate, extract the certificate parameters, and dump the information in the file `Dumpfile.txt`.

- Shown below is a Python version of the script. Whereas in line (W) of the Perl script we used backticks to capture the certificates that were written by the `gnutls-cli()` call to the standard output, in the Python code shown below we do the same in line (Y) by first calling `subprocess.Popen()` to create a child process and then calling `communicate()` on the child process to capture whatever is written by the child process to its standard output.

- Another call to `subprocess.Popen()` in the script shown below is in line (l) for invoking `openssl x509 -text` command to create a text version of the certificate. In the Python version of the script shown earlier, the same thing was done with backticks in line (f) of that script.

```
#!/usr/bin/env python

### ModulusHarvestor.py

### Author:    Avi Kak (kak@purdue.edu)
### Date:      February 24, 2016

##  The script can be used in following two different modes:
##
##    --- With no command-line args.  In this case, the script scans the internet
##        with randomly synthesized IP addresses and, when it finds a site with its
##        port 443 open, it grabs the certificate(s) offered by that site and
##        extracts the various certificate parameters (modulus, public exponent,
##        etc.) from the certificate(s).
##
##    --- With just one command-line arg, which must be an IPv4 address.  In this
##       case, the script will try to connect with that address on its port 443 and
##       download the certificate offered by it.  So as not to waste your time, it
##       is best if you use an IP address that does offer an HTTPS service.  You can
##       check that with a simple port scanner like 'port_scan.pl' we will cover in
##       Lecture 16.

##  The basic purpose of this script is to harvest RSA moduli used for public keys in
##  SSL/TLS certificates.  Recent research has demonstrated that if two different
##  moduli share a common factor, they can both be factored easily, thus compromising
##  the security of both.

##  For harvesting moduli, the script first randomly selects $NHOSTS number of hosts
##  from the space of all possible IP addresses and tries to download their X.509
##  certificates using a GnuTLS client.  It subsequently extracts the modulus and
##  public key used in the certificates using openssl commands.  These are finally
##  dumped into a file called Dumpfile.txt.

import sys
import socket
import subprocess
import random
import re
import os

debug = 1

mark1 = "-----BEGIN CERTIFICATE-----"                                           #(A)
mark2 = "-----END CERTIFICATE-----"                                             #(B)
dumpfile = "Dumpfile.txt"                                                       #(C)
DUMP = open( dumpfile, 'w')                                                     #(D)
ip_addresses_to_scan = []                                                       #(E)

## This subroutine was borrowed from the AbraWorm.py code in Lecture 22.
def get_fresh_ipaddresses(howmany):                                             #(F)
    if howmany == 0: return 0                                                   #(G)
    ipaddresses = []                                                            #(H)
    for i in range(howmany):
        first,second,third,fourth = list(map(lambda x: random.randint(1, x),
                                                 [223] * 4))                    #(I)
        ipaddresses.append( "%s.%s.%s.%s" % (first,second,third,fourth) )       #(J)
    return ipaddresses                                                          #(K)


if __name__ == '__main__':

    if len(sys.argv) == 1:                                                      #(L)
        NHOSTS = 200                                                            #(M)
        ip_addresses_to_scan = get_fresh_ipaddresses(NHOSTS)                    #(N)
    elif len(sys.argv) == 2:
        ip_addresses_to_scan.append(sys.argv[1])                                #(O)
    else:
        sys.exit("You cannot call %s with more than one command-line argument"
                                                      % sys.argv[0])            #(P)
    for ip_address in ip_addresses_to_scan:                                     #(Q)
        print("\nTrying IP address: %s\n\n\n" % ip_address)                     #(R)
```

```
try:
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM )               #(S)
    sock.settimeout(0.1)                                                    #(T)
    sock.connect((ip_address, 443))                                         #(U)
except:                                                                     #(V)
    continue                                                                #(W)
DUMP.write("%s\n\n" % ip_address)                                           #(X)
proc = subprocess.Popen(['gnutls-cli --insecure --print-cert ' + \
     ip_address + ' < /dev/null'], stdout=subprocess.PIPE, shell=True)      #(Y)
(output,err) = proc.communicate()                                           #(Z)
regex = mark1 + r'(.+?)' + mark2                                            #(a)
certificates = re.findall( regex, output, re.DOTALL )                       #(b)
howmany_certs = len(certificates)                                           #(c)
if debug: print "Found %s certificates\n\n" % howmany_certs                 #(d)
for i in range(1, len(certificates)+1):                                     #(e)
    if debug:
        print "Certificate %s:\n\n" % i                                     #(f)
        print str(certificates[i-1]) + "\n\n"                               #(g)
    FILE = open("__temp.cert", 'w')                                         #(i)
    FILE.write(mark1 + str(certificates[i-1]) + mark2 + "\n")               #(j)
    FILE.close()                                                            #(k)
    proc2 = subprocess.Popen(['openssl x509 -text < __temp.cert'],
                         stdout=subprocess.PIPE, shell=True)                #(l)
    (cert_text, err) = proc2.communicate()                                  #(m)
    if debug: print cert_text + "\n\n\n"                                    #(n)
    all_lines = filter(None, re.split(r'\s+', cert_text)   )                #(o)
    cert_text = ''.join(all_lines)                                          #(p)
    params = re.findall(r'Modulus:(.+?)Exponent:(\d+)', cert_text,
                                                     re.DOTALL)             #(q)
    modulus = "0x" + ''.join( re.split(r':', params[0][0] ) )               #(r)
    if debug:
        print "Modulus:"                                                    #(s)
        print int(modulus, 16)                                              #(t)
        print "\n"
        print "Public exponent: %s\n" % params[0][1]                        #(u)
        print "\n\n\n";
    DUMP.write( "Modulus:\n" )                                              #(v)
    DUMP.write( modulus )                                                   #(w)
    DUMP.write("\n\nPublic Exponent: %s\n\n\n" % params[0][1])              #(x)
    os.unlink( "__temp.cert")                                               #(y)
    DUMP.write("\n\n\n")
```

- Don't forget to look at the contents of the file `Dumpfile.txt` in the directory in which you run the scripts shown in this section for the certificates and the RSA moduli extracted from randomly selected URLs around the world.

13.5: THE DIFFIE-HELLMAN ALGORITHM FOR GENERATING A SHARED SECRET SESSION KEY

- The previous approach for establishing a secret key (that could subsequently be used for communication using conventional encryption) assumed an RSA based approach for the exchange of the secret key. As was pointed out in Section 12.6 of Lecture 12, creating session keys in this manner makes them vulnerable to a man-in-the-middle attack in which an eavesdropper stores away the information exchanged between two parties with the hope that should he somehow acquire the private keys of the parties involved at a future date, he'll be able to figure out the secret session key at that time.

- When the authenticity of two parties can be established by other means (say, by the RSA algorithm), another approach for creating a shared secret key is based on the Diffie-Hellman Key Exchange algorithm. (See the note about the DHE-RSA algorithm at the end of Section 12.6 of Lecture 12.)

- Two parties A and B using this algorithm for creating a shared secret key first agree on a large **prime** number p and an element g of Z_p^* that generates a large-order **cyclic subgroup** of the multiplicative group Z_p^*. [**First note that the starting point for understanding the DH algorithm is NOT the finite field Z_p that you are so familiar with, but the multiplicative group Z_p^* that you know only cursorily from its definition in Section 11.8 of Lecture 11.** Before enlightening you further about Z_p^*, let me mention again that the **order of a group** is the **cardinality** of the group, meaning the number of elements in the group. We can also talk about the **order of an element** in a group: the order of an element $a \in G$ is the smallest value t such that $a^t \equiv a \circ a \circ \ldots (t\ times) \ldots \circ a \ = \ group\ identity\ element$ where $\circ$ is the group operator. **Now let's talk about the notation Z_p^*. The notion of Z_p^* is based on the observation that for prime p, the set $\{1, 2, 3, \cdots, p-1\}$ constitutes a group with the group operator being modulo p multiplication.** (Note that unlike what was the case with the field Z_p, we have no desire to map all the integers into the groupZ_p^*. That is, only the 16 integers 1 through 16 exist in Z_{17}^*. On the other hand, every integer exists in Z_{17}. The integers 17, for instance, is the same thing as 0 in Z_{17}. The same integer is simply outside the scope of Z_{17}^*. More technically speaking, the field Z_p is a set of equivalence classes: each element a of Z_p stands for **all** the integers whose modulo p value equals a. On the other hand, the group Z_p^* is merely a set of $p-1$ integers 1 through $p-1$.) Z_p^* is also frequently referred to as a multiplicative group of order $p-1$ with 1 being the group identity element. **As it turns out Z_p^* is a cyclic group for certain values of p.** Z_p^* is a cyclic group if all the elements of Z_p^* can be expressed as $g^i\ mod\ p$ for all $i = 0, 1, 2, \cdots$ and for some element $g \in Z_p^*$. For illustration, Z_{17}^* is a cyclic group with $g = 3$. That is, if you compute $3^i\ mod\ 17$ for $all\ i = 0, 1, 2, \cdots$, you will get the 16 numbers in the multiplicative group Z_{17}^*. Let's now focus on the cyclic subgroups of Z_p^*. A subset of Z_p^* forms a cyclic subgroup if the group operator continues to be modulo p multiplication and if all of the elements of the subgroup can be generated through the powers of one of the elements of the subgroup. In other words, for a subset of Z_p^* to constitute a cyclic subgroup, it must be possible to generate all of the elements of the subset by $g^i\ mod\ p$ for $all\ i = 0, 1, 2, \cdots$ for some g element in the subset. Again going back to the example of $p = 17$, if we use 2 as a generator element, we get the cyclic subgroup $\{1, 2, 4, 8, 16, 15, 13, 9\}$ whose order is 8. All of the elements in this subgroup are given by $2^i\ mod\ 17$ for all $i = 0, 1, 2, \cdots$. In general, if M is the order of a cyclic subgroup of Z_p^*, M will be a divisor of $p-1$. **(This is known**

as Lagrange's Theorem in Group Theory.) Also note that within each order-M cyclic subgroup of Z_p^*, we have $g^M = 1$ if g is the generator for that subgroup. In order words, using the terminology of Section 11.8 of Lecture 11, g is the *primitive element* of the cyclic subgroup generated by it. More commonly, though, g is called the *generator* of the multiplicative subgroup that is generated by raising g to all possible power. **We are specifically interested in those cyclic subgroups of Z_p^* whose order M is large. More specifically, we want to choose for the DH protocol a g so that the order M is a large prime factor of $p - 1$.**]

- The pair of numbers (p, g) is public. This pair of numbers may be used for several runs of the protocol. These two numbers may even stay the same for a large number of users for a long period of time. [**A typical value used for g is 2, as stated in RFC 2412 "OAKLEY Key Determination Protocol", but may be larger. Obviously, the choices for g and p must yield a large order cyclic subgroup of Z_p^*. This RFC defines the protocol that is used for exchanging the relevant information between two hosts for establishing a secret session key according to the Diffie-Hellman algorithm.**]

- Subsequently, A and B use the algorithm described below to calculate their public keys that are then made available by each party to the other:

  - We will denote A's and B's **private keys** by X_A and X_B. And their **public keys** by Y_A and Y_B. In other words, X stands for private and Y for public.

  - A selects a random number X_A from the set $\{2, \ldots, p-2\}$ to serve as his/her **private key**. A then calculates a **public-key** integer Y_A that is guaranteed to exist:

$$Y_A \quad = \quad g^{X_A} \bmod p$$

A makes the **public key** Y_A available to B. [Regarding the "guaranteed to exist" comment about Y_A regardless of the choice of X_A (as long as it is *any* integer between 1 and $p-2$), it follows from the very definition of a cyclic subgroup of Z_p^*. For greater clarity regarding this issue, see the example based on $p = 17$ that is presented in the next main bullet. In that example, we choose $X_A = 5$, which is a member of Z_{17}^*. While this X_A is NOT in the cyclic group generated by the root element $g = 2$, the number $2^5 \ mod \ 17$ is. **And if you are curious as to why $p-1$ is EXCLUDED as a candidate for the private key X_A, recall that for any g, we have $g^{p-1} \ mod \ p \ = \ 1$ by FLT.**]

– Similarly, B selects a random number X_B from from the set $\{2, \ldots, p-2\}$ to serve as his/her **private key**. B then calculates an integer Y_B that serves his/her **public key**:

$$Y_B \quad = \quad g^{X_B} \bmod p$$

B makes the public-key Y_B available to A.

– A now calculates the secret key K from his/her private key X_A and B's public key Y_B:

$$K \quad = \quad (Y_B)^{X_A} \bmod p \qquad (1)$$

– B carries out a similar calculation for locally generating the shared secret key K from his/her private key X_B and A's public key Y_A:

$$K = (Y_A)^{X_B} \bmod p \tag{2}$$

– The following equalities demonstrate that the secret key K in both the equation Eq. (1) and Eq. (2) will be the same:

$$\begin{aligned}
K\ as\ calculated\ by\ A &= (Y_B)^{X_A} \bmod p \\
&= (g^{X_B} \bmod p)^{X_A} \bmod p \\
&= (g^{X_B})^{X_A} \bmod p \\
&= g^{X_B X_A} \bmod p \\
&= (g^{X_A})^{X_B} \bmod p \\
&= (g^{X_A} \bmod p)^{X_B} \bmod p \\
&= (Y_A)^{X_B} \bmod p \\
&= K\ as\ calculated\ by\ B
\end{aligned}$$

- To illustrate the Diffie-Hellman key exchange with a silly little example, consider the case when the prime p is 17 and the primitive root g is 2. So we start with the multiplicative group $Z_{17}^* = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$. Let's now choose $g = 2$ for the root element and see what cyclic subgroup of Z_{17}^* is generated by this root element. Just by calculating 2^i *mod* 17 for all $i = 0, 1, 2, 3, \cdots$, we can see that the cyclic subgroup is given by $\{1, 2, 4, 8, 16, 15, 13, 9\}$ where I have intentionally shown the elements in the order of the consecutive powers of 2. [However, as you well know, the order of appearance in a set is unimportant.] Let's say that party A chooses $X_A = 5$ as his/her private key. (Note that X_A is an element of Z_p^*. It does not specifically have to be an element of the cyclic subgroup generated by the chosen primitive root.) A's public key would be given by $Y_A = 2^{X_A}\ mod\ 17 = 2^5\ mod\ 17 = 15$. And let's assume that party B chooses $X_B = 13$ as his/her private key. Party B's public key would be given by $Y_B = 2^{X_B}\ mod\ 17 = 2^{13}\ mod\ 17 = 15$. As it happens, in this case, both parties have the number 15 as their public keys. The secret session key as calculated by A: $K_A = Y_B^{X_A}\ mod\ 17 = 15^5\ mod\ 17 = 2$. And the secret session key as calculated by B: $K_B = Y_A^{X_B}\ mod\ 17 = 15^{13}\ mod\ 17 = 2$. [In this example, you might wonder as to what purpose was served by displaying the cyclic subgroup generated by the root element 2. I did that to emphasize the fact that the private key itself does NOT have to belong to the cyclic subgroup — the private key can be any integer at all as long as it is in the set Z_p^*. Note also that, as you will see later, the security of the DH protocol depends critically on the size of this cyclic subgroup.]

- A seemingly **magical thing** about the DH protocol is that an eavesdropper having access to the public keys for both A and B would still not be able to figure out the secret key K.

- Another seemingly **magical thing** about this protocol is that it allows two parties A and B to create a shared secret K without either party having to send it directly to the other.

- The DH protocol is also referred to as the **ephemeral** secret key agreement protocol because, typically, the secret key K is used only once. [At least that is the mode in which the DH protocol is used in the Transport Layer Security (TLS) protocol that we will talk about in Lecture 20.]

- A well-known variant of the Diffie-Hellman protocol is known as the ElGamal protocol in which A's public key Y_A remains fixed (and publicly available) over a long period of time. Party B encrypts his/her message M by calculating $M \times K \ mod \ p$ where K is the same as defined earlier. [That is, party B can directly encrypt the message M without having to resort to a block cipher for content encryption. For reasons of computational efficiency, this works well only when M is small (as is likely to be the case if you are encrypting the hash value of a document).] The decryption by A consists of dividing the received ciphertext by K modulo p. This mechanism is useful in some implementations of anonymous client connections.

- The security of the Diffie-Hellman algorithm is based on the fact that whereas it is relatively easy to compute the powers of an integer in a finite field, it is extremely hard to compute the discrete logarithms. (See Section 11.8 of Lecture 11 for what is meant by a discrete logarithm).

- That is, whereas the following can be calculated readily

$$Y_A \quad = \quad g^{X_A} \bmod p$$

by A in order to determine his/her public key, for a adversary to figure out the private keys X_A or X_B from a knowledge of all of the publicly available information $\{p,\ g,\ Y_A,\ Y_B\}$, the adversary would have to carry out the following sort of a discrete logarithm calculation

$$X_A \quad = \quad d\log_{g,p} Y_A$$

for which there do not exist any efficient algorithms. The difficulty of determining the secret shared key K from the publicly available p, g, Y_A, and Y_B is sometimes referred to as the *Computational Diffie-Hellman Assumption*.

- Even if you accept the security of DH on the basis of the difficulty of solving the discrete logarithm problem, the DH protocol possesses a number of vulnerabilities. If interested, see the publication "Security Issues in the Diffie-Hellman Key Agreement Protocol" by Raymond and Stiglic for a list of these vulnerabilities.

- **One of the most serious vulnerabilities of DH is to the man-in-the-middle attack.** Let's say there is an adversary who can intercept — as opposed to merely eavesdrop on — the messages between A and B. The adversary intercepts the public key Y_A that is sent by A to B and replaces it with $Y_A^{'}$. The adversary does the same to the public key Y_B that is sent by B to A — it gets replaced by $Y_B^{'}$. The secret key generated by A will now be different from the key generated by B, but both these keys will be known to adversary. Unless A and B each authenticates the other party independently, neither will realize that they are using different session keys. (What makes this attack scenario worse is that the adversary has the freedom to change the content of the message received from A before it is encrypted again for B using the key that B knows.)

- Because of the vulnerability to the man-in-the-middle attack, use of the DH protocol should be preceded by sender authentication. When DH is used with sender authentication, the resulting overall protocol is sometimes referred to as *authenticated DH*.

- In *authenticated DH*, each party acquires a certificate for the other party. The DH public key that each party sends to the other party is digitally signed by the sender using the private key that corresponds to the public key on the sender's certificate. [A reader might ask that if the two parties are going to use certificates anyway, why not fall back on the "traditional" approach of having one of the parties encrypt a session key with the other party's public key, since, subsequently, only the other party would be able to retrieve the session key through decryption with their private key. While that point is valid, DH does give you additional security because it creates a shared secret without any transmission of the secret between the two parties.]

13.6: THE ElGamal ALGORITHM FOR DIGITAL SIGNATURES

- Typically, when you say you have digitally signed a document, it means that you first calculated a hash of the document (using one of the methods described in Lecture 15), you then encrypted the hash with your private key, and you made this encrypted block available (as your signature) along with the document. When a party wants to verify that the document is authentic, they use your public key to extract the hash of the document from the encrypted block, and compare this hash with the hash their computer calculates directly from the document. [Earlier you saw an example of this in Section 13.4 when we talked about how a CA signs a certificate.]

- Although the above description is what is generally meant by a digital signature, there does exist a somewhat more elaborate Digital Signature Algorithm that has been promulgated as a standard by NIST. The standard itself is referred to as the Digital Signature Standard (DSS). It is based on the famous ElGamal algorithm for constructing the digital signature of a document. In what follows, we present without proof the main steps of this algorithm.

- Let's say that you would like to sign the documents you make available to others on the internet. As with all public key cryptography systems, the first thing you'd need to do is to create a public key – private key pair. You will execute the following steps for this:

  – Select a large prime p and then randomly select two numbers, denoted g and X, less than p. You will make the numbers p and g publicly available and you will treat X as your private key.

  – Next you calculate your public key Y by

  $$Y \quad = \quad g^X \;\; mod\, p$$

  Obviously, you will also make publicly available your public key Y (along with g and p).

  – In addition, you will generate a one-time random number K such that $0 < K < p-1$ and $gcd(K, p-1) = 1$. [Note that K is coprime to $p-1$, which is an even integer since p is a prime.] You are going to need K for constructing a digital signature. By one-time we mean that you will discard K after each use. That is, each digital signature you create will be with a different K. Even though you use each K only once, you must not let anyone else get hold of this number, since otherwise they will be able to figure out your private key from the signature and from all the other information you must make public. **For the logic of how an adversary can figure out your private key if you use the same K on different documents, see Section 14.13 of Lecture 14.**

– Now you are ready to construct a digital signature of a document. Let M be the integer that represents whatever it is you want to sign. Typically, M will be the output of a hashing function applied to the document. See Lecture 15 on hashing functions.

– The digital signature you construct for M will consist of two parts that we will denote sig_1 and sig_2.

– You construct sig_1 by

$$sig_1 \quad = \quad g^K \;\; mod\, p$$

and you construct sig_2 by

$$sig_2 \quad = \quad K^{-1} \times (M \;\; - \;\; X \times sig_1) \qquad mod\; (p-1)$$

where K^{-1} is the multiplicative inverse of K modulo $p-1$ that can be obtained with the Extended Euclid's Algorithm (See Sections 5.6 and 5.7 of Lecture 5).

  – As mentioned, sig_1 and sig_2 taken together constitute your digital signature of M.

- Let's say you have sent the message M along with your signature (sig_1, sig_2) to some recipient and the recipient wishes to make sure that he/she is not receiving a modified message. The recipient can verify the authenticity of M by checking the following equality

$$Y^{sig_1} \times {sig_1}^{sig_2} \quad \equiv \quad g^M \quad (mod\ p)$$

- Since the random number K is specific to each signature, the ElGamal algorithm give you the ability to create one-time signatures. Let's say you use your laptop to sign a document today with this algorithm. If your laptop were to be stolen tomorrow, the thief would not be able to recreate that signature even if he/she gained access to your private key X.

- The Digital Signature Standard is described in the document FIPS 186-3 that can be downloaded from:

  `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`

- An aside: *Taher ElGamal (also written Taher El Gamal) played a central role in the development of the SSL (Secure Socket Layer) protocol in his capacity as the Chief Scientist* *of Netscape Communications in the late 1990's. SSL [and its later cousin TLS (for Transport Layer Security)] forms the security backbone for a large number of protocols, as you will see later in this course.*

13.7: ON SOLVING THE DISCRETE LOGARITHM PROBLEM

- Obviously, if an adversary can solve the following equation

$$g^s \quad = \quad k \bmod p \tag{3}$$

  for s for given values of g and k, the Diffie-Hellman encryption will be broken. As mentioned earlier, solving this equation for s is the famous **discrete logarithm problem**.

- One obvious way to solve the discrete logarithm problem is by brute force. This involves calculating g^i for $i = 0, 1, 2, ...$ until a solution is found. The computational complexity of this is proportional to p. If p requires an n bit representation, then the complexity, being proportional to 2^n, grows **exponentially** with the size of p in bits.

- A slightly more efficient way to solve the discrete logarithm problem is by the **baby-step giant-step** method:

– Compute, sort, and store the m elements g^0, g^1, g^2, ..., g^m in a table. Since the exponents increase by 1 as you go from one row to another in this table, this constitutes **baby steps**.

– Now compute $\frac{k}{g^m}$ and check to see if it is in the above table. If not, compute $\frac{k}{g^{2m}}$ and check to see if it is in the table. If not, repeat until you find a j so that $\frac{k}{g^{jm}}$ is in the table. Let's say that from the table we find

$$\frac{k}{g^{jm}} \quad = \quad g^i \tag{4}$$

for some j and i. Dividing k by successively larger powers of g^m constitute the **giant steps**.

– The above equation implies that the solution s we are looking for must satisfy

$$s \quad = \quad jm \; + \; i$$

– The time complexity of this algorithm is $O(p/m)$ and the memory requirement $O(m)$. The product of the two is $O(p) = O(2^n)$, which is still exponential in n, the size of p.

- A second approach to solving the discrete logarithm problem is known as the $Pollard - \rho$ method. [Source: van Tilborg, NAW, Sept. 2001]

– This method is based on the assumption that g can serve as the **generator** of a subgroup of prime order q within Z_p. That means that the set $\{g^0, g^1, \ldots\}$ would form a subgroup within the set Z_p.

– Another concept that the $Pollard-\rho$ method is based on can be explained as follows: Let f be a random mapping function from a **finite** set A to itself. Now starting from a randomly selected $a_0 \in A$, define a sequence $\{a_i\}_{i \geq 0}$ recursively by

$$a_{i+1} \quad = \quad f(a_i)$$

The sequence a_0, a_1, a_2, ..., will eventually cycle because A was assumed to be finite. It has been shown that the average length of the cycle and the length of the beginning segment until the cycle starts are both given by $\sqrt{\pi |A|/8}$.

– The $Pollard-\rho$ method uses the mapping $f : Z_q \times Z_q \times Z_q \rightarrow Z_q \times Z_q \times Z_q$ as given by

$$f(x, u, v) \quad = \quad \begin{cases} (x^2, 2u, 2v), & if \;\; x \equiv 0 \pmod 3, \\ (kx, u, v+1), & if \;\; x \equiv 1 \pmod 3, \\ (gx, u+1, v), & if \;\; x \equiv 2 \pmod 3 \end{cases}$$

The sequence $\{(x_i, u_i, v_i)\}_{i \geq 0}$ is defined recursively by

$$\begin{aligned} (x_0, u_0, v_0) \quad &= \quad (1, 0, 0), \\ (x_{i+1}, u_{i+1}, v_{i+1}) \quad &= \quad f(x_i, u_i, v_i) \end{aligned}$$

– The recursion shown above generates the sequence $x_i = g^{u_i} k^{v_i}$ for all $i \geq 0$. [This fact can be verified by induction. Assume for a moment that $x \equiv 0 \pmod 3$. Now $g^{u_{i+1}} k^{v_{i+1}} = g^{2u_i} k^{2v_i} = (g^{u_i} k^{v_i})^2 = (x_i)^2 = x_{i+1}$.]

– Assume that we can find an x_i such that $x_{2i} = x_i$. When that happens, $g^{u_{2i}} k^{v_{2i}} = g^{u_i} k^{v_i}$. Substituting in this our original equation $k = g^s$, we have $g^{u_{2i}} g^{s v_{2i}} = g^{u_i} g^{s v_i}$. From this, it is almost always the case that we can write the following solution for s [Source: van Tilborg, NAW, 2001]:

$$s \quad = \quad \frac{u_{2i} \; - \; u_i}{v_i \; - \; v_{2i}} \bmod (q-1)$$

To find an index i such that $x_{2i} = x_i$, it is not necessary to list all values of the sequence x_i. If for a given i, $x_i \neq x_{2i}$, we calculate $x_{i+1}, u_{i+1}, v_{i+1} = f(x_i, u_i, v_i)$ and $x_{2i+2}, u_{2i+2}, v_{2i+2} = f(f(x_{2i}, u_{2i}, v_{2i}))$ and compare their first coordinates again.

– The time complexity of the $Pollard - \rho$ method is $O(2^{n/2})$ if it takes n bits to represent the prime integer p.

• Two other methods for solving the discrete logarithm problem are the $Pollard - \lambda$ method and the Index-Calculus method.

13.8: How Diffie-Hellman May Fail in Practice

- The title of this section was inspired by the title of a wonderful 2015 publication entitled "*Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*" by David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thome, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Beguelin, and Paul Zimmerman. Googling the title of this publication will take you straight to a download site.

- To appreciate the issues raised by the authors, first realize that it can be computationally cumbersome to find the primes with the desirable properties for use with the Diffie-Hellman algorithm. At the least, the primes must yield multiplicative subgroups of large order. They must also possess several other properties that are reproduced below from the RFC 2412 document.

- For reasons mentioned above, most applications (SSH, VPN, Tor, etc.) use the DHE parameters mentioned in RFC 2412 "*OAKLEY Key Determination Protocol*" that governs the exchange of messages between two hosts for establishing a session key with the Diffie-Hellman algorithm. This RFC recommends "safe" primes of length 768 bits (Oakley Group 1), 1024 bits (Oakley Group 2), and 1536 bits (Oakley Group 5). [The word "group" in "Oakley Group n" for different n refers to a Z_p^* multiplicative group with a prime modulus p and a generator element g that is typically 2. Note that even if multiple hosts use the same multiplicative group, that does not automatically mean that their DH security is compromised. An adversary eavesdropping on a communication link will see the parameters (p, g) and the public key Y. If a good random number generator is used for choosing the private key X, the value of Y will be different for different links. To figure out X from Y would require solving the discrete log problem $X = dlog_{g,p}Y$. When the prime p is large, solving the problem for one Y would not automatically result in a solution for a different Y. However, when p is insufficiently large, all bets are off.]

- For example, for Oakley Group 1, RFC 2412 recommends the following decimal value for a 768-bit prime:

```
155251809230070893513091813125848175563133404943451431320235
119490296623994910210725866945387659164244291000768028886422
915080371891804634263272761303128298374438082089019628850917
0691316593175367469551763119843371637221007210577919
```

and the generator element $g = 2$ to go with this prime. And for Oakley Group 2, the RFC 2412 recommends the following decimal value for a 1024-bit prime:

```
179769313486231590770839156793787453197860296048756011706444
423684197180216158519368947833795864925541502180565485980503
646440548199239100050792877003355816639229553136239076508735
759914822574862575007425302077447712589550957937778424442426
617334727629299387668709205606050270810842907692932019128194
```

with the generator g again being 2. [The following properties of the primes shown here are reproduced from RFC 2412: The high order 64 bits for both the primes shown here are forced to be 1s. This helps the classical remainder algorithm, because the trial quotient digit can always be taken as the high order word of the dividend, possibly +1. The low order 64 bits are forced to 1. This helps the Montgomery-style remainder algorithms, because the multiplier digit can always be taken to be the low order word of the dividend. The middle bits are taken from the binary expansion of π. This guarantees that they are effectively random, while avoiding any suspicion that the primes have secretly been selected to be weak. Additionally, because both primes are based on pi, there is a large section of overlap in the hexadecimal representations of the two primes. The primes are chosen to be Sophie Germain primes (i.e., $(P-1)/2$ is also prime), to have the maximum strength against the square-root attack on the discrete logarithm problem. **The starting trial numbers were repeatedly incremented by 2^{64} until suitable primes were located.** Because these two primes are congruent to 7 (mod 8), 2 is a quadratic residue of each prime. All powers of 2 will also be quadratic residues. This prevents an opponent from learning the low order bit of the Diffie-Hellman exponent (AKA the subgroup confinement problem). Using 2 as a generator is efficient for some modular exponentiation algorithms. The RFC gives credit to Richard Schroeppel for work related to the establishing these primes as possessing the good properties mentioned here.]

- The basic weakness of DH lies in fact that a large number of servers use the same set of DH parameters as mentioned above. As the paper says, this "dramatically reduces the cost of large-scale attacks, bringing some within range of feasibility today." An adversary can carry out a large number of precomputations for these choices of the primes for solving the discrete log problem in order to figure out the private keys from the public keys.

- While the Oakley groups for the DH parameters are still considered safe — especially those that involve large sized primes — there is a basic flaw in the TLS protocol that allows some legacy servers to offer 512-bit primes. The authors were able to calculate the discrete logs in about a minute for two commonly used such primes.

- The authors state that solving the discrete-log problem for 768-bit primes is now within reach for academic researchers and for 1024-bit primes within reach for state-level attackers.

13.9: CAN THE CERTIFICATES ISSUED BY A CA BE FORGED?

- The short answer is yes.

- In mid-2008, it was shown by a group of security researchers (Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger) how the weak collision resistance property of the MD5 hashing function could be exploited to construct a forged certificate. [Lecture 15 talks about hashing functions and their collision resistance properties.] They acquired some real certificates from a root CA and then proceeded to attach the CA's signature to a different public-key embedded in a digital document whose MD5 signature was the same as that in one of the legal certificates. This exploit is described in detail at `http://www.win.tue.nl/hashclash/rogue-ca/`. [What made this exploit particularly potent was that the researchers created a rogue certificate for an intermediate level CA. Subsequently, the rogue CA thus brought into existence could have issued its own rogue certificates to any number of end users. Most of the world's browsers would not have found any problems with those end-user rogue certificates since the browsers would have been able to validate them against the rogue intermediate CA certificate that was forged by the researchers and, that certificate, in turn, would have been validated by the root CA in the usual manner. As mentioned earlier in this lecture, the public keys of the Root CAs, of which VeriSign, Comodo, etc., are examples, are incorporated in your browser software so that the root-level verification is not subject to network-based man-in-the-middle attacks.]

- Another way to obtain forged certificates came to light on March 11, 2011. An attacker breached the account of an Italian reseller of the Comodo-signed certificates. [As mentioned earlier, Comodo is a large root CA; it owns 11 root public keys. Some of the Comodo root keys should already be programmed into your web browser, in keeping with the explanation presented earlier in this lecture.] Apparently, the reseller used cleartext-based password authentication for folks filling out CSR (Certificate Signing Request) forms. The attacker used this weakness to break into the reseller's account and created for himself a new user account with authorization to issue Comodo certificates. The attacker then proceeded to create Comodo-signed forged certificates for the domains: `mail.goggle.com`, `www.google.com`, `login.yahoo.com`, `login.skype.com`, `addons.mozilla.org`, and `login.live.com`. **Technically speaking, these certificates were forged because the attacker held the private keys corresponding to the public keys signed by the Comodo's private key. This would have allowed the attacker to act like Google, Yahoo, Skype, etc.** This unauthorized issuance of certificates was discovered within hours and these certificates revoked immediately. [A CA can revoke a certificate by adding its serial number to its CRL (Certificate Revocation List). Before the browser software validates a certificate downloaded from web server, its serial number is checked with the CRL maintained by the signer of the certificate.] This exploit is described at `http://blogs.comodo.com/it-security/data-security/the-recent-ra-compromise/`

- Let's now address the question of what harm an attacker may bring to bear on the organizations whose certificate the attacker has forged.

- Let's say the attacker has created or obtained a forged certificate for the domain `www.citibank.com`. [As mentioned in the previous bullet, this means that the attacker has the private key for what is supposedly CitiBank's public key that is signed in the certificate.] The attacker then proceeds to create a Citibank lookalike web site and attaches the forged certificate with this rogue site. The problem now for the attacker is that unless the client traffic can be directed to this rogue website, no harm will come from the forged certificate.

- In order to direct client traffic to his rogue website, the attacker would need to poison the DNS cache likely to be used by the client applications. (See Lecture 17 on how that can be done.) As a result of the scare that was caused by Dan Kaminsky when he demonstrated how vulnerable DNS servers were to cache poisoning exploits, a majority of the world's DNS servers have been patched and are protected against such exploits. So the odds are against the attacker succeeding with cache poisoning — unless the attacker has ISP and/or state level cooperation.

- Another way the attacker could direct unsuspecting users to his rogue webserver would be through a phishing attack. As mentioned in Section 17.15 of Lecture 17, phishing is online fraud that attempts to steal sensitive information such as usernames, passwords, and credit card numbers. A common way to do this is to display familiar strings like `www.amazon.com` or `www.paypal.com` in the browser window while their actual URL links are to rogue web servers.

- **Note that it is easy to make yourself a "fake" root CA with the help of the opensource library called** `OpenSSL` For example, you can run the following command to create a self-signed "Root CA Certificate": [All root CA certificates are self-signed for obvious reasons.]:

```
openssl req -new -x509 -keyout private/CAkey.pem -out CAcert.pem -days 365
```

  that deposits the root CA certificate in a file named `CA.pem` and the corresponding private key in a file called `CAkey.pem` in the directory `private`. Subsequently, all you have to do is to somehow get innocent parties to add this certificate to the collection of root certificates already in their computers. (You might be able to do that with a social engineering attack as described in Lecture 30.) Next you can set up an e-commerce business that uses certificates signed by you in your capacity as a root CA. Now all you have to do is to lure customers to your e-commerce website with deals they cannot resist. Should there be folks who take the bait and upload their credit card information to your website, just imagine how quickly you could become rich — assuming that the law does not get any wind of your deeds.

13.10: HOMEWORK PROBLEMS

1. Let's say the browser in your laptop wants to download a page from the `engineering.purdue.edu` domain. Since the web server for this domain runs under the **`HTTPS`** protocol, your browser must engage in what is known as SSL handshaking with the server for the purpose of creating a secret session key that can be used for content encryption by both the web server and your browser. [We will cover SSL handshaking as used in the HTTPS protocol in Lecture 20. For now just assume that this handshaking requires authenticating a certificate-supplied public key with the public key of the applicable CA and then using the authenticated public key for encrypting a message that can only be deciphered with the private key that corresponds to the authenticated public key.] Let's say you have not provided your laptop with a public key, let alone a certificate. [This is indeed the case for most of the users of of web services.] Given this scenario, which end of the connection between your browser and the web server for `engineering.purdue.edu` do you believe will generate a session key and send it over to the other side?

2. Would your answer to the previous question change if I mentioned that the secret session key would be generated through the Diffie-Hellman algorithm after your laptop has authenticated `engineering.purdue.edu`'s public key?

3. What is man-in-the-middle attack?

4. What is the Diffie-Hellman algorithm for creating a secret session key?

5. Difficulty of breaking RSA cipher is because of the difficulty of factorizing large numbers. To what do we owe the difficulty of breaking the Diffie-Hellman cipher?

6. **Programming Assignment 1:**

   The main goal of this assignment is to extract the number parameters used in RSA keys that are stored in PEM formatted files that may either be key files or certificate files. [The name of the PEM format stands for "Privacy Enhanced Mail." It is the format used by OpenSSL to represent public keys, private keys, digital signatures, and certificates. **PEM is basically the same thing as the Base64 format that you are already familiar with, except for the addition of the header and the footer lines.**] You may be interested in extracting the number parameters n and e from a public key in order to check if the modulus n is factorizable or has previously been factored by someone else. Many folks still use 1024-bit RSA despite the fact that moduli of this size have been factorized successfully. We will proceed in the following manner for this homework:

   In Section 13.4, we talked about using the following command from the OpenSSL library to generate an X509 certificate for testing purposes. That command is reproduced below:

```
openssl req -new -newkey rsa:1024 -days 365 -nodes -x509 -keyout test.pem -out test.cert
```

As mentioned in Section 13.4, this outputs two files, `test.pem` and `test.cert`, the former containing a new private key for 1024-bit RSA and the latter an X509 certificate that contains the public key and a self-signed version of the same. (You should already know about the format of an X509 certificate from Section 13.4. You also know about the format of an RSA private key from Section 12.8 of Lecture 12.) To verify that the certificate file `test.cert` contains all the goodies, you can invoke

```
    openssl  verify  test.cert
```

This command will print out a message saying this is a self-signed certificate. If you want to see in text form in your terminal window the contents of the certificate, execute the following:

```
    openssl  x509  -in  test.cert  -text  -noout
```

As you will see, this will also display in the terminal window any information you supplied about yourself and your organization during the certificate creation process. But you will notice that your public key (that corresponds to the private key in the `test.pem` file) as well the signature are in Base64 encoded form. Let's say you are interested in extracting the number parameters that went into the public key stored in the certificate file `test.cert` and the private key that is in the file `test.pem`. **How does one do that? — Which brings us to the main point of this programming exercise as described below.**

In order to see the specific parameters used in the public key stored in the certificate `test.cert` and the private key file `test`

`.pem`, let's first generate for practice purposes a new pair of keys as follows:

```
openssl  genrsa  -out  my_private_key.pem  1024

openssl  rsa  -in  my_private_key.pem  -pubout  >  my_public_key.pem
```

where the first command generates a private key and the second puts out the corresponding public key.

If you want to extract the number parameters from a PEM file containing a private key, the following will do the job:

```
openssl  rsa  -text  <  my_private_key.pem
```

The command that does the same for a public key is

```
openssl rsa -text -pubin < my_public_key.pem
```

Both of the above commands will show the number parameters as colon-delimited hex strings. If you want the modulus to be displayed as a single continuous hex string, you can execute:

```
openssl rsa -text -pubin -modulus < my_public_key.pem
```

You have surely noticed that all of our invocations to print out the number parameters above used the **`rsa`** as the first option to the **`openssl`** command. That makes sense because all of those calls were on files containing RSA keys. If you want to look inside an X509 certificate at a level that prints out the number information in the form of hex strings (and without Base64 encoding), try

```
openssl  x509  -text  <  test.cert
```

After you have become comfortable with these `openssl` commands, output the modulus and the public exponent that you can extract from the certificate file `test.cert`. Feed this modulus into the web site `http://www.factordb.com` to see if they can supply you with the prime factors of the modulus.

As you can see, the OpenSSL library is extremely useful. In addition to visiting `http://www.openssl.org`, you may also want to visit `http://www.madboa.com/geek/openssl` for a nicely organized page that shows how you can use OpenSSL commands for accomplishing different things.

7. **Programming Assignment 2:**

The goal of this homework is to "play" with: (1) the public key used by a web server running under the HTTPS protocol, (2) the public key of the CA used by the web server for authenticating its own public key, and (3) the digital signature placed at the end of the certificate supplied by the web server. Another goal is to become familiar with viewing certificates through the Certificate Viewer in your browser.

Point your web browser to a page in the `engineering.purdue.edu` domain and click on the lock symbol that you will see at the left side of the one-line URL window at the top of the browser window. You should see a popup that (1) tells you that you are running an encrypted session with the server; (2) gives you the name of the CA that issued the certificate for the domain of the URL; and

(3) shows you a button that you can click on for further information regarding the certificate. When you click on the button, you should see another popup for "View Certificate". Clicking on this button takes to what's known as a Certificate Viewer. The Certificate Viewer should show two panels, one that gives you general information regarding the certificate and the other that gives you all of the fields in `engineering.purdue.edu`'s X.509 certificate. Click on "Subject's Public Key" to see the modulus and the public exponent in the public key used by this domain. Finally, click on the "Certificate Signature Value" to see the value produced by encrypting the SHA-1 hash of the relevant certificate fields with CA's private key. [We will cover hash functions in general and SHA-1 in particular in Lecture 15.] If we represent this signature by C, then $C^e \ mod \ n$ should give you the 20-byte SHA-1 hash of what is stored for the `TBSCertificate` field in the ASN.1 representation of an X.509 certificate that was shown earlier in Section 13.4.

Now compare the 20 byte SHA-1 hash you obtain with the value you will find on the "General" panel of the Certificate Viewer. The two values will turn out not to be same. Why?

The n and e values I mentioned above are the modulus and the public exponent as used by the CA. To get these values, you must invoke the Certificate Viewer directly in your browser. For FireFox, this you can do with the "Preferences $\rightarrow$ Advanced $\rightarrow$ View Certificates" options that you can access through the "Edit" menu button at the top of the browser window. Now go to the CA's own certificate and, through mouse actions similar to those already described, extract the n and e values you need. [This homework problem may also alert you to a security vulnerability in your browser. You may be able to view any passwords you previously stored in your browser in clear text. For example, the same popup that gives me the button "View Certificate" also has a button for "View Saved Passwords".]

8. This problem focuses on verifying certificates. Let's say that our goal is to explicitly verify the certificate made available by the `engineering.purdue.edu` that I use for hosting my lecture notes. Obviously, the very first thing you'd need to do is to get hold of the certificate document itself. To get the document, when you are on the "Details" panel of the Certificate Viewer mentioned in the previous problem, click on the "Export" button. This will deposit a Base-64 encoded certificate in a directory of your choice. The name of this certificate file will be

    ```
    engineering.purdue.edu.crt
    ```

    You can read this file with "`cat engineering.purdue.edu.crt`" to see the following in your terminal window:

```
-----BEGIN CERTIFICATE-----
MIIFoTCCBImgAwIBAgIQITA/w6Nfe9hoVhW/LVjRYjANBgkqhkiG9w0BAQUFADBR
MQswCQYDVQQGEwJVUzESMBAGA1UEChMJSW50ZXJuZXQyMREwDwYDVQQLEwhJbkNv
bW1vbjEbMBkGA1UEAxMSSW5Db21tb24gU2VydmVyIENBMB4XDTEzMDEwOTAwMDAw
MFoXDTE2MDEwOTIzNTk1OVowggEFMQswCQYDVQQGEwJVUzEOMAwGA1UEERMFNDc5
MDcxEDAOBgNVBAgTB0luZGlhbmExFzAVBgNVBAcTDldlc3QgTGFmYXlldHRlMSAw
HgYDVQQJExc0NjUgTm9ydGh3ZXN0ZXJuIEF2ZW51ZTE1MDMGA1UECRMsRWxlY3Ry
aWNhbCBhbmQgQ29tcHV0ZXIgRW5naW5lZXJpbmcgQnVpbGRpbmcxGjAYBgNVBAoT
EVB1cmR1ZSBVbml2ZXJzaXR5MSUwIwYDVQQLExxFbmdpbmVlcmluZyBDb21wdXRl
ciBOZXR3b3JrMR8wHQYDVQQDExZlbmdpbmVlcmluZy5wdXJkdWUuZWR1MIIBIjAN
BgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEApTxPMFDWtsNPaeYl4OG9472rOTkL
GQ9kBSlWKFeAd63FAZ/QGuaVbRX1gXgdqsdZljy4YM5mc1zLOUsbLkKvwAhmqMbG
Ep6OD/q9lq+LXNngnT8JSkRn92pmaggA7TJ2rURlUJbSXeUXEyHxeifFwXPdOJCb
jdXt7EaV7rBmfSOjiNLktbbj4ernZWEBLlFw0a1JPQxAVwrvekYOT5RDAVQP2sD3
k6HOkyGQEDlCnWUkqlURvRsBmW8Iv5lMHKyNHl16UPtblZYpJiuc7fewLl0rU9Wc
E5C8IFwtNCs4GGsZP7xmwzcGYS01cohbQCFDG6gJBklE8n5en3UEo13vxQIDAQAB
o4IBvTCCAbkwHwYDVR0jBBgwFoAUSE9a+i9Kml7gUPNre1Wl3vW+NF0wHQYDVR0O
BBYEFFtlqE6NBraWgs7VSceSlGEgPvO7MA4GA1UdDwEB/wQEAwIFoDAMBgNVHRMB
Af8EAjAAMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEFBQcDAjBnBgNVHSAEYDBe
MFIGDCsGAQQBriMBBAMBATBCMEAGCCsGAQUFBwIBFjRodHRwczovL3d3dy5pbmNv
```

```
bW1vbi5vcmcvY2VydC9yZXBvc2l0b3J5L2Nwc19zc2wucGRmMAgGBmeBDAECAjA9
BgNVHR8ENjA0MDKgMKAuhixodHRwOi8vY3JsLmluY29tbW9uLm9yZy9JbkNvbW1v
blNlcnZlckNBLmNybDBvBggrBgEFBQcBAQRjMGEwOQYIKwYBBQUHMAKGLWh0dHA6
Ly9jZXJ0LmluY29tbW9uLm9yZy9JbkNvbW1vblNlcnZlckNBLmNydDAkBggrBgEF
BQcwAYYYaHR0cDovL29jc3AuaW5jb21tb24ub3JnMCEGA1UdEQQaMBiCFmVuZ2lu
ZWVyaW5nLnB1cmR1ZS5lZHUwDQYJKoZIhvcNAQEFBQADggEBAJE7Um53QPZPnCS3
sS+LK3aS+ufhLfE/8Dkg2mhVVZCBujijXajglpDncyWEqCxtfuiclgJPgyyiqycW
q+ahr7dThzFotHqpTgQu7sdvzCxDIWP2qRV28LhCmNbRTWGcWGytGLwx66l2oTDg
dgUSmfyefzlx6c/Cx4cBxyRaPj6ulRiDGoX7bAiKMo6wZ2rBf5ogqyAHWHoJEVah
UrMESl2VoNx8D67rfvs4kMiSEA6A2xdtQv1jnsrIlIaeSKmQYcAvMX/Dr0JQKKGJ
FzTDkbDblWiRxm2SXk5FmLblzqtmS2jNDaVqu0F8NsVmovE30q7jmSAo96hj2As7
DrCP2vM=
-----END CERTIFICATE-----
```

Since the certificate shown above is Base-64 encoded, you are not able to see any of its fields. To actually see the contents of the certificate, execute the following command

```
    openssl x509 -in engineering.purdue.edu.crt -text -noout
```

This will output all of the certificate fields to your terminal window. If you wish, you can direct the output into a text file. As mentioned in Section 13.4, the digital signature you'll see at the bottom is the output of encrypting the hash of the data in all of the certificate fields with CA's private key. As the certificate itself mentions, the CA used the SHA-1 algorithm for hashing; this is something we will take up in Lecture 15. Now execute the following command to verify this certificate

```
    openssl verify -CAfile InCommonServerCA.crt engineering.purdue.edu.crt
```

where I have assumed that you downloaded the CA's public key into the file `InCommonServerCA.crt` in accordance with the discussion in Section 13.4. In general, if the non-root CA certificates in your computer are stored in a directory that you know about, you can also invoke the following command for certificate verification:

```
    openssl verify -CApath directory_to_ca_certs  engineering.purdue.edu.crt
```

I should also mention that all of the root SSL certificates that your machine knows about are stored in the directory

```
    /etc/ssl/certs/
```

The `openssl` tools should already know about this location. So if you are trying to verify a certificate that was signed by a root CA directly, you can use the following command line for verification:

```
    openssl verify InCommonServerCA.crt
```

where, as you already know, `InCommonServerCA.crt` is the certificate for the intermediate level CA InCommon. This certificate, as mentioned previously, is signed by the root CA AddTrust.

9. Digital certificates started out as a promising solution to the problem of identity fraud in web-based interactions. The idea at the beginning was that the CAs would verify that the requester of a certificate was a valid individual or entity. However, over the years, that idea has mostly fallen by the wayside. The CAs now issue certificates to anyone requesting them for a fee, the only identity verification carried out being the validity of the IP address from which you supply the required information. As a result, as matters stand today, all that a digital certificate in a protocol such as HTTPS ensures is that you are running an encrypted session with the web server, but you cannot be 100% certain about the true identity of the party at the other end.

   The fact that a digital certificate cannot ordinarily be banked on to establish trust in the identity of a web service provider is an important issue in e-commerce applications where you are asked to supply credit card, financial, and, sometimes, personal information. To meet the need for a greater degree of identity trust, a new type of an X.509 certificate was recently created that is known as Extended Validation Certificate (EV). An EV certificate also conforms to the X.509 standard. However, a CA will subject an entity to a higher proof of identity before issuing this type of a certificate. Your browser identifies an EV certificate through the *object identifier* (OID) number that is placed in the **`extensions`** field in the ASN.1 representation of a certificate that was shown in Section 13.4.

The goal of this homework is for you to verify that, in terms of structure and content layout, there is no difference between a regular X.509 certificate, as, for example, supplied by the domain `engineering.purdue.edu` and an EV certificate, as, for example, supplied by the domain `http://www.paypal.com`. Download the certificates from these two or other similar web sites, create their readable textual representations using the **`openssl`** commands that you are already familiar with, and then compare them. [When you point your browser to a web site that supplies an EV certificate, its presentation in the URL window will change. In FireFox, the color of the lock symbol along with the name of the web site will turn green.]