CSE 127: Computer Security

Asymmetric Crypto, TLS, PKI and CT

Deian Stefan

Adopted slides from Kirill Levchenko and Dan Boneh
Asymmetric Cryptography

• Also called public key cryptography

• Two separate keys
  ➢ Public key: known to everyone
  ➢ Private key: used to decrypt and sign
Asymmetric Primitives

- Encryption and decryption
- Signing and verification
- Diffie Hellman key exchange
Asymmetric Keys

- Each user has a public and private key
- Keys related to each other in algorithm-dependent way
  - Need a key generation function
  - Keygen(r) = (pk, sk)
    - pk: public key
    - sk: secret key
    - r: random bits
Public-key encryption

- **Encryption**: \((\text{public key}, \text{plaintext}) \rightarrow \text{ciphertext}\)
  - \(E_{pk}(m) = c\)

- **Decryption**: \((\text{secret key}, \text{ciphertext}) \rightarrow \text{plaintext}\)
  - \(D_{sk}(c) = m\)
Encryption properties

• Encryption and decryption are inverse operations
  ➤ \( D_{sk}(E_{pk}(m)) = m \)

• Secrecy: ciphertext reveals nothing about plaintext
  ➤ Computationally hard to decrypt without secret key

• What’s the point?
  ➤ Anybody with your public key can send you a secret message!
Implementations

- ElGamal encryption (1985)
  - Based on Diffie-Helman key exchange (1976), itself invented by Diffie, Hellman, and Merkle
  - Computational basis: hardness of discrete logarithms

- RSA encryption (1978)
  - Invented by Rivest, Shamir, and Adleman
  - Computational basis: hardness of factoring
Digital signatures

- **Signing:** (secret key, message) → signature
  - $S_{sk}(m) = s$

- **Verification:** (public key, message, signature) → bool
  - $V_{pk}(m,s) = true \mid false$
Signature properties

• Verification of signed message succeeds
  ➤ $V_{pk}(m, S_{sk}(m)) = true$

• Unforgettability: can’t compute signature for a message $m$ without secret key $sk$

• What’s the point?
  ➤ Anybody with your public key can verify that you signed something!
Implementations

• Digital Signature Algorithm (1991)
  ➢ Closely related to ElGamal signature scheme (1984)
  ➢ Computational basis: hardness of discrete logarithms

• RSA signatures
  ➢ Invented by Rivest, Shamir, and Adleman
  ➢ Computational basis: hardness of factoring
Encrypted and signed messaging

Alice

Bob

(this is a bad way to roll your own crypto)
Encrypted and signed messaging

Alice
(pk_{Alice-E}, sk_{Alice-E})
(pk_{Alice-S}, sk_{Alice-S})

Bob
(pk_{Bob-E}, sk_{Bob-E})
(pk_{Bob-S}, sk_{Bob-S})

(this is a bad way to roll your own crypto)
Encrypted and signed messaging

\[ E_{pk_{Alice-E}}(m_1) = c_1 \]
\[ (c_1, S_{sk_{Bob-S}}(c_1)) \]

Alice
(pk_{Alice-E}, sk_{Alice-E})
(pk_{Alice-S}, sk_{Alice-S})

Bob
(pk_{Bob-E}, sk_{Bob-E})
(pk_{Bob-S}, sk_{Bob-S})

(this is a bad way to roll your own crypto)
Encrypted and signed messaging

\[ E_{pk_{Alice-E}}(m_1) = c_1 \]
\[ (c_1, S_{sk_{Bob-S}}(c_1)) \]

if \( V_{pk_{Bob-S}}(c_1) \)
\[ D_{sk_{Alice-E}}(c_1) \]

(this is a bad way to roll your own crypto)
Encrypted and signed messaging

\[ E_{pk_{Alice-E}}(m_1) = c_1 \]
\[ (c_1, S_{sk_{Bob-S}}(c_1)) \]

if \( V_{pk_{Bob-S}}(c_1) \)

\[ D_{sk_{Alice-E}}(c_1) \]

(this is a bad way to roll your own crypto)
Practical Considerations

• Asymmetric cryptography operations are much more expensive than symmetric operations
  ➤ Even implementations based on elliptic curves!
  ➤ Don’t want to encrypt/sign huge messages

• Moreover: asymmetric primitives operate on fixed-size messages
What do we do in practice?

- Usually combined with symmetric for performance
  - Use asymmetric to bootstrap ephemeral secret
Typical Encryption Usage

• Encryption:
  ➤ Generate a ephemeral (one time) symmetric secret key
  ➤ Encrypt message using ephemeral secret key
  ➤ Encrypt ephemeral key using asymmetric encryption

• Decrpytion:
  ➤ Decrypt ephemeral key, decrypt message
Typical Signature Usage

• Signing:
  ➤ Compute cryptographic hash of message
  ➤ Sign it using asymmetric signature scheme

• Verification:
  ➤ Compute cryptographic hash of message
  ➤ Verify it using asymmetric signature scheme
Asymmetric Primitives

- Encryption and decryption
- Signing and verification
- Diffie Hellman key exchange
Diffie Hellman key exchange

• Establish a shared secret over public channel
  ➤ Invented by Diffie, Hellman, and Merkle

Alice

Bob
Diffie Hellman key exchange

- Establish a shared secret over public channel
  - Invented by Diffie, Hellman, and Merkle

Alice

\[(pk_{Alice}, sk_{Alice}) \leftarrow \text{DHKeygen()}\]

Bob

\[(pk_{Bob}, sk_{Bob}) \leftarrow \text{DHKeygen()}\]
Diffie Hellman key exchange

- Establish a shared secret over public channel
  - Invented by Diffie, Hellman, and Merkle

\[ (pk_{Alice}, sk_{Alice}) \leftarrow DHKeygen() \]

\[ (pk_{Bob}, sk_{Bob}) \leftarrow DHKeygen() \]
Diffie Hellman key exchange

- Establish a shared secret over public channel
  - Invented by Diffie, Hellman, and Merkle

Alice

Bob

$(pk_{Alice}, sk_{Alice}) \leftarrow \text{DHKeygen()}$

$(pk_{Bob}, sk_{Bob}) \leftarrow \text{DHKeygen()}$
**Diffie Hellman key exchange**

- Establish a shared secret over public channel
  - Invented by Diffie, Hellman, and Merkle

Alice

\[
(pk_{Alice}, sk_{Alice}) \leftarrow \text{DHKeygen()}
\]

\[ k \leftarrow \text{DHSecret}(sk_{Alice}, pk_{Bob}) \]

Bob

\[
(pk_{Bob}, sk_{Bob}) \leftarrow \text{DHKeygen()}
\]

\[ k \leftarrow \text{DHSecret}(sk_{Bob}, pk_{Alice}) \]
Diffie Hellman key exchange

- Establish a shared secret over public channel
  - Invented by Diffie, Hellman, and Merkle

Alice

Bob

\[(pk_{Alice}, sk_{Alice}) \leftarrow \text{DHKeygen}()\]

\[k \leftarrow \text{DHSecret}(sk_{Alice}, pk_{Bob})\]

\[(pk_{Bob}, sk_{Bob}) \leftarrow \text{DHKeygen}()\]

\[k \leftarrow \text{DHSecret}(sk_{Bob}, pk_{Alice})\]
How do we get keys?
How do we get keys?

• Public keys are public: just ask for them!
How do we get keys?

- Public keys are public: just ask for them!
How do we get keys?

- Public keys are public: just ask for them!
How do we get keys?

- Public keys are public: just ask for them!
How do we get keys?

• Public keys are public: just ask for them!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!

Alice  pk_{Bob}  Bob
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

• Public keys are public: just ask for them!
  ➤ No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  ➤ No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

- Public keys are public: just ask for them!
  - No! Vulnerable to Man-in-the-Middle attacks!
How do we get keys?

• Public directory contains everyone’s public key
• To encrypt to a person, get their public key from directory
• No need for shared secrets!
How do we know Alice’s key is actually Alice’s?
Key Verification

• Alice and Bob need a way to know that each has the real public key of the other

• **Ideal solution:** Alice and Bob meet in person and exchange public keys

• **Equivalent:** Alice and Bob meet in person and exchange public key fingerprints
  
  ➢ Key fingerprint: cryptographic hash of public key
  ➢ Key itself can be sent in the open
Where have you seen this?

https://signal.org/blog/safety-number-updates/
Key Verification

• Problem with ideal: Alice and Bob need to meet
  ➤ Impractical to meet and verify key of everyone ...

• Practical solution: Use a trusted intermediary
  ➤ Alice and Bob have already exchanged keys w/ Claire
  ➤ Claire sends signed message with Alice’s key to Bob
  ➤ Claire sends signed message with Bob’s key to Alice
  ➤ Alice and Bob trust Claire to send the public keys
  ➤ Alice and Bob now have each other’s public key
Key Verification Improved

- Claire creates a certificate:
  “I, Charlie, verified that Alice’s key is ... ”

- Claire signs the message and gives it to Alice
  - Alice now has certificate attesting to her public key

- Alice sends Bob her public key and Claire’s cert

- Bob verifies signature on certificate

- Bob trusts Claire, accepts public key from Alice
Who is Claire?

- PGP world: Claire is any other person you trust
- Keybase world: Claire is a set of
- SSL world: Claire is a Certificate Authority
PGP (Web of Trust)

- PGP allows one user to attest to the accuracy of another user’s public key — **key signing**
  - PGP does not use the term “certificate”
  - Public key has set of signatures (equiv. certificates)
- A user can indicate how much they trust another user’s signature on a key
PGP (Web of Trust)

• Claire’s signature of Bob’s PGP key means Claire has verified that this is really Bob’s key
  ➤ Email address associated with key is Bob’s address
  ➤ Name associated with key is Bob

• Other people who trust Claire can use her signature on Bob’s key to be sure it is Bob’s key
Where is PGP used?
Where is PGP used?
Where is PGP used?

-----BEGIN PGP SIGNATURE-----

iQiCAABAcgAGBQJcYsgnAAoJE0t3RJHZ/wbir4kP/j6knpATcXLrzcQD1kELx63fZTKDzw8KoXRguZWXgMuaRX8yVup/7x1TZ55URADadAh8yta5hqvq0Ob1fSRT
M92Yu12m0x5SAAzlBFCMnG0jvz7Lbt6LM2SuvL4mw0ifH4e3OshQnNpz9e+C1UI
ShHNHlHo3N0A4ffYeNVSNjreLInj1JbsF5ZUFaznbFJlh5nvcDPSBCp13M5bElW
bj/n+0teFvfh BKvjRfLwEytcjr7UbDO/8cz0q7du7wR4u7ixgwYnEnh33kCaL
Ah45Qe3p9yY285XRTyYn3bjmXhg8p3DkJCz411TCYvZ0N0bJr84WxfIrVFJnbjJ
UtS+boY8t24uhd9imr6YdNbKrygQMSAT1LaMuw0t03b2H6aTE115VK3k5DVzqj
4oyWmYcX3geQsFg8CdGW5iraK2bW0AkHTCdVWJuHBByBzDT0774d0614dy9m+6
c014yhwmAyjQSCFZaui+1BrLSNWgnV4UNuoN9i04NBVain7/k1BQILJQY/ZeKw
MPCJdj8jz64FtDg91HnmimIlly7DtzItRbF2Mz0LCuU8ELc3ye+wJJO5QXGCS9Nx
dOEwIPSt8fnLh4KWABthceuIXSk80cnEngMq6co7+x5hVRDpnnkhv8zKXzmAKkKQ
AvpZ2rG1FA8GeUdFkres
=ZXI0

-----END PGP SIGNATURE-----
Where is PGP used?
Where is PGP used?

pacman/Package signing

To determine if packages are authentic, `pacman` uses GnuPG keys in a web of trust model. The current Master Signing Keys are found here. At least three of these Master Signing Keys are used to sign each of the Developer's and Trusted User's own keys which then in turn are used to sign their packages. The user also has a unique PGP key which is generated when you set up `pacman-key`. So the web of trust links the user's key to the Master Keys.

Examples of webs of trust:

- **Custom packages**: You made the package yourself and signed it with your own key.
- **Unofficial packages**: A developer made the package and signed it. You used your key to sign that developer's key.
- **Official packages**: A developer made the package and signed it. The developer's key was signed by the Arch Linux master keys. You used your key to sign the master keys, and you trust them to vouch for developers.
Where is PGP used?

Debian Public Key Server

This public key server provides simple HKP lookup and add requests for Debian developer and maintainer public keys.

The server may be accessed with gpg by using the --keyserver option in combination with either of the --recv-keys or --send-keys actions.

Please note that this server is meant only for basic key retrieve/update operation, and does not implement search functionality. To search for a specific DD or Developer, use the Developer LDAP Search interface.

Only keys in the Debian keyrings (ie those for DDs and DMs) will be returned by this server and only pre-existing keys will be updated, although a copy of all updates will be forwarded to the keyserver network.

You can use the keyring server for the following purposes:

Fetch a key

Once you know the key’s ID, just ask the server for it:
7.4 Git Tools - Signing Your Work

Signing Your Work

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the web or want to verify that commits are actually from a trusted source, Git has a few ways to sign a commit using GPG.

GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal signing key set up.

```bash
$ gpg --list-keys
/Users/schacon/.gnupg/pubring.gpg
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```bash
$ gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things with the `user.signingkey` config setting.

```bash
$ git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

Signing Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is run:

```bash
$ git tag
```

In your case, you can use the `--sign` option to sign the commit.

```bash
$ git tag new-release --sign
```

This will ensure that the commit is signed by your GPG key, providing a secure and verifiable way to distribute your work.
Where is PGP used?

Not really for email....
Who is Claire?

- PGP world: Claire is any other person you trust
- Keybase world: Claire is a set of
- SSL world: Claire is a Certificate Authority
Keybase world

• Challenge: verifying Alice’s key is actually Alice’s key on public directory

• Solution: Associate key with social network identifies
  ➢ Sign messages on GitHub, Twitter, Facebook, etc.

• Attacker model
  ➢ Attacher must compromise N of my account
Keybase world

Challenge: verifying Alice’s key is actually Alice’s key on public directory

Solution: Associate key with social network

➤ Sign messages on GitHub, Twitter, Facebook, etc.

• Attacker model

➤ Attacker must compromise N of my account
Verifying myself: https://gist.github.com/deian/9906028

Deian Stefan
@deiandelmars

I hereby claim:

- I am deian on github.
- I am deian (https://keybase.io/deian) on keybase.
- I have a public key whose fingerprint is A3CA DAA1 144E 5CDE B67F 37B9 5ED1 79BB 628C 02E2

To claim this, I am signing this object:

```json
{
    "body": {
        "key": {
            "fingerprint": "a3cadaaa144e5cde67f37b95ed179bb628c02e2",
            "host": "keybase.io",
            "key_id": "5ED179BB628C02E2",
            "uid": "56845ebd23c5b6f304a3da75554ff00",
            "username": "deian"
        },
        "service": {
            "name": "github",
            "username": "deian"
        },
        "type": "web_service_binding",
        "version": 1
    },
    "ctime": 1396315705,
    "expire_in": 157680000,
    "prev": "70543d8dd91a406196f3addec945da82b05094c5a38b899b9664aadcd6b4d5bc",
    "seqno": 2,
    "tag": "signature"
}
```

with the PGP key whose fingerprint is A3CA DAA1 144E 5CDE B67F 37B9 5ED1 79BB 628C 02E2 (captured body.key.fingerprint), yielding the PGP signature:

```plaintext
-----BEGIN PGP MESSAGE-----
Version: GnuPG v2.0.22 (GNU/Linux)

owGbwMvMvMQyD7Fyd1IP0yPG6weeJ3EEW4lZVsI5adUKlllVK2WngqmoZLz01KRCosy8EiUrRTJ5MSuxERDQx0TNVPk1NQjM/M0Y/MkS9PUFENzy6Q/KMyOLZA0jYcMlHaw/MGKQdQaXyNf9q+FqXz+UAIICc+MwUoaqrqATvSARU72xg5APXwqRMLXpMwMU1N7sEy67ZNMkssN7AvsWv0STQ3NTU15UszmAAPLE4tykVM7QWkTzkNIMXqTtVRAgVb
```

Sign me on:
- GitHub
- Twitter
- Facebook

Attacker model:
- Attacker must compromise N of my account

> Sign me on:
- GitHub
- Twitter
- Facebook

> Attacker model:
- Attacker must compromise N of my account
Challenge: verifying Alice's key is actually Alice's key on public directory

Solution: Associate key with social network

➤ Sign messages on GitHub, Twitter, Facebook, etc.

Attacker model
➤ Attacker must compromise N of your accounts
Who is Claire?

- PGP world: Claire is any other person you trust
- Keybase world: Claire is a set of
- SSL world: Claire is a Certificate Authority
Announcements

• Last PA due Tue/Sat
• Final: March 20th
• Review: Monday 4PM in 1202
• Today: Rehash certificates
  ➤ Then: A: General network security
  B: DNS security
  C: Review
  ➤ Then: sphiel about ethics etc.
Certificate Authorities

- **Certificate Authority:** Trusted authority
  - Signs keys after checking some kind of identity (e.g., you own www.google.com)
  - Server presents signed key (cert) to clients
  - Browsers and OSes ship with public keys of trusted CAs
Certificate Authorities

Browser (Alice)

Server (Bob)

CA

PK_{CA}

PK_{CA}

SK_{CA}
Certificate Authorities

Browser (Alice)

Server (Bob)

PK<sub>CA</sub>

choose (SK, PK)

PK<sub>CA</sub>

PK and proof “I am Bob”

CA

SK<sub>CA</sub>
Certificate Authorities

Browser (Alice)

PK_{CA}

Server (Bob)

choose (SK, PK)

PK_{CA}

PK_{CA} and proof "I am Bob"

issue Cert with SK_{CA}:

Bob’s key is PK

CA

SK_{CA}
Certificate Authorities

Browser (Alice)

\[ PK_{CA} \]

Verify cert

\[ PK_{CA} \]

Server (Bob)

choose (SK, PK)

CA

Check proof

\[ SK_{CA} \]

PK and proof “I am Bob”

issue Cert with \[ SK_{CA} \]:

Bob’s key is PK

Bob’s key is PK
## Details

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject Name</td>
<td>cse.ucsd.edu</td>
</tr>
<tr>
<td>Country</td>
<td>US</td>
</tr>
<tr>
<td>Postal Code</td>
<td>92093</td>
</tr>
<tr>
<td>State/Province</td>
<td>CA</td>
</tr>
<tr>
<td>Locality</td>
<td>La Jolla</td>
</tr>
<tr>
<td>Street Address</td>
<td>9500 Gilman Drive</td>
</tr>
<tr>
<td>Organization</td>
<td>University of California, San Diego</td>
</tr>
<tr>
<td>Organizational Unit</td>
<td>UCSD</td>
</tr>
<tr>
<td>Common Name</td>
<td>cse.ucsd.edu</td>
</tr>
<tr>
<td>Issuer Name</td>
<td></td>
</tr>
<tr>
<td>Country</td>
<td>US</td>
</tr>
<tr>
<td>State/Province</td>
<td>MI</td>
</tr>
<tr>
<td>Locality</td>
<td>Ann Arbor</td>
</tr>
<tr>
<td>Organization</td>
<td>Internet2</td>
</tr>
<tr>
<td>Organizational Unit</td>
<td>InCommon</td>
</tr>
<tr>
<td>Common Name</td>
<td>InCommon RSA Server CA</td>
</tr>
<tr>
<td>Serial Number</td>
<td>36 F6 DC 47 6F 09 25 8E 94 EF BF 36 65 4F E8 98</td>
</tr>
<tr>
<td>Version</td>
<td>3</td>
</tr>
<tr>
<td>Signature Algorithm</td>
<td>SHA-256 with RSA Encryption</td>
</tr>
</tbody>
</table>

This certificate is valid.
Who are we trusting?
Who are we trusting?

Who is this cert for?
<table>
<thead>
<tr>
<th>Key ID</th>
<th>Subject Alternative Name (2.5.29.17)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical</td>
<td>NO</td>
</tr>
<tr>
<td>DNS Name</td>
<td>cse.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>cs.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>www-cs.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>www-cse.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td><a href="http://www.cs.ucsd.edu">www.cs.ucsd.edu</a></td>
</tr>
<tr>
<td>DNS Name</td>
<td><a href="http://www.cse.ucsd.edu">www.cse.ucsd.edu</a></td>
</tr>
</tbody>
</table>

Extension: Certificate Policies (2.5.29.32)
Critical: NO
Policy ID #1: 1.3.6.1.4.1.5923.1.4.3.1.1
Qualifier ID #1: Certification Practice Statement (1.3.6.1.5.5.7.2.1)
CPS URI: https://www.incommon.org/cert/repository/cps_ssl.pdf
Policy ID #2: 2.23.140.1.2.2

Extension: CRL Distribution Points (2.5.29.31)
Critical: NO
URI: http://crl.incommon-rsa.org/InCommonRSAServerCA.crl

Extension: Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
Critical: NO
Method #1: CA Issuers (1.3.6.1.5.5.7.48.2)
URI: http://crt.usertrust.com/InCommonRSAServerCA_2.crt
Method #2: Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
URI: http://ocsp.usertrust.com
Who is this cert for?

DNS Name: cse.ucsd.edu
Issuer Name
Country US
State/Province MI
Locality Ann Arbor
Organization Internet2
Organizational Unit InCommon
Common Name InCommon RSA Server CA
Serial Number 36 F6 DC 47 6F 09 25 8E 94 EF BF 36 65 4F E8 98
Version 3
Signature Algorithm SHA-256 with RSA Encryption
(1.2.840.113549.1.1.11)
Parameters None
Not Valid Before Thursday, January 4, 2018 at 4:00:00 PM Pacific Standard Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1)
Parameters None
Public Key 256 bytes: FA F9 1A 08 92 86 9C 7B ...
Exponent 65537
Key Size 2,048 bits
Key Usage Encrypt, Verify, Wrap, Derive
Signature 256 bytes: 6F 62 36 46 B7 43 28 04 ...
Extension Key Usage (2.5.29.15)
Critical YES
Usage Digital Signature, Key Encipherment
<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Issuer Name</strong></td>
<td>Country US</td>
</tr>
<tr>
<td><strong>State/Province</strong></td>
<td>MI</td>
</tr>
<tr>
<td><strong>Locality</strong></td>
<td>Ann Arbor</td>
</tr>
<tr>
<td><strong>Organization</strong></td>
<td>Internet2</td>
</tr>
<tr>
<td><strong>Organizational Unit</strong></td>
<td>InCommon</td>
</tr>
<tr>
<td><strong>Common Name</strong></td>
<td>InCommon RSA Server CA</td>
</tr>
<tr>
<td><strong>Serial Number</strong></td>
<td>36 F6 DC 47 6F 09 25 8E 94 EF BF 36 65 4F E8 98</td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>3</td>
</tr>
<tr>
<td><strong>Signature Algorithm</strong></td>
<td>SHA-256 with RSA Encryption (1.2.840.113549.1.1.1)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Not Valid Before</strong></td>
<td>Thursday, January 4, 2018 at 4:00:00 PM Pacific Standard Time</td>
</tr>
<tr>
<td><strong>Not Valid After</strong></td>
<td>Monday, January 4, 2021 at 3:59:59 PM Pacific Standard Time</td>
</tr>
<tr>
<td><strong>Public Key Info</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
<td>RSA Encryption (1.2.840.113549.1.1)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>None</td>
</tr>
<tr>
<td><strong>Public Key</strong></td>
<td>256 bytes : FA F9 1A 08 92 86 9C 7B ...</td>
</tr>
<tr>
<td><strong>Exponent</strong></td>
<td>65537</td>
</tr>
<tr>
<td><strong>Key Size</strong></td>
<td>2,048 bits</td>
</tr>
<tr>
<td><strong>Key Usage</strong></td>
<td>Encrypt, Verify, Wrap, Derive</td>
</tr>
<tr>
<td><strong>Signature</strong></td>
<td>256 bytes : 6F 62 36 46 B7 43 28 04 ...</td>
</tr>
<tr>
<td><strong>Extension</strong></td>
<td>Key Usage (2.5.29.15)</td>
</tr>
<tr>
<td><strong>Critical</strong></td>
<td>YES</td>
</tr>
<tr>
<td><strong>Usage</strong></td>
<td>Digital Signature, Key Encipherment</td>
</tr>
</tbody>
</table>

CSE’s pub key info
Extension: Subject Alternative Name (2.5.29.17)
Critical: NO
DNS Name: cse.ucsd.edu
DNS Name: cs.ucsd.edu
DNS Name: www.cs.ucsd.edu
DNS Name: www-cs.ucsd.edu
DNS Name: www-cse.ucsd.edu
DNS Name: www.cs.ucsd.edu
DNS Name: www.cse.ucsd.edu

Extension: Certificate Policies (2.5.29.32)
Critical: NO
Policy ID #1: (1.3.6.1.4.1.5923.1.4.3.1.1)
Qualifier ID #1: Certification Practice Statement (1.3.6.1.5.5.7.2.1)
CPS URI: https://www.incommon.org/cert/repository/cps_ssl.pdf
Policy ID #2: (2.23.140.1.2.2)

Extension: CRL Distribution Points (2.5.29.31)
Critical: NO
URI: http://crl.incommon-rsa.org/
InCommonRSAServerCA.crl

Extension: Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
Critical: NO
Method #1: CA Issuers (1.3.6.1.5.5.7.48.2)
URI: http://crt.usertrust.com/
InCommonRSAServerCA_2.crt
Method #2: Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
URI: http://ocsp.usertrust.com
Extension: Subject Alternative Name (2.5.29.17)
Critical: NO
DNS Name: cse.ucsd.edu
DNS Name: cs.ucsd.edu
DNS Name: www.cs.ucsd.edu
DNS Name: www-cse.ucsd.edu
DNS Name: www.cs.ucsd.edu
DNS Name: www.cse.ucsd.edu

Extension: Certificate Policies (2.5.29.32)
Critical: NO
Policy ID #1: (1.3.6.1.4.1.5923.1.4.3.1.1)
Qualifier ID #1: Certification Practice Statement (1.3.6.1.5.5.7.2.1)
CPS URI: https://www.incommon.org/cert/repository/cps_ssl.pdf
Policy ID #2: (2.23.140.1.2.2)

Extension: CRL Distribution Points (2.5.29.31)
Critical: NO
URI: http://crl.incommon-rsa.org/InCommonRSAServerCA.crl

Extension: Certificate Authority Information Access (1.3.6.1.5.5.7.1.1)
Critical: NO
Method #1: CA Issuers (1.3.6.1.5.5.7.48.2)
URI: http://crt.usertrust.com/InCommonRSAServerCA_2.crt
Method #2: Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)
URI: http://ocsp.usertrust.com
Where we should check for revocation information

<table>
<thead>
<tr>
<th>Key ID</th>
<th>1E 05 A3 77 8F 6C 96 E2 5B 87 4B A6 B4 86 AC 71 00 0C E7 38</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extension</td>
<td>Subject Alternative Name (2.5.29.17)</td>
</tr>
<tr>
<td>Critical</td>
<td>NO</td>
</tr>
<tr>
<td>DNS Name</td>
<td>cse.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>cs.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>www-cs.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td>www-cse.ucsd.edu</td>
</tr>
<tr>
<td>DNS Name</td>
<td><a href="http://www.cs.ucsd.edu">www.cs.ucsd.edu</a></td>
</tr>
<tr>
<td>DNS Name</td>
<td><a href="http://www.cse.ucsd.edu">www.cse.ucsd.edu</a></td>
</tr>
<tr>
<td>Extension</td>
<td>Certificate Policies (2.5.29.32)</td>
</tr>
<tr>
<td>Critical</td>
<td>NO</td>
</tr>
<tr>
<td>Policy ID #1</td>
<td>1.3.6.1.4.1.5923.1.4.3.1.1</td>
</tr>
<tr>
<td>Qualifier ID #1</td>
<td>Certification Practice Statement (1.3.6.1.5.5.7.2.1)</td>
</tr>
<tr>
<td>CPS URI</td>
<td><a href="https://www.incommon.org/cert/repository/cps_ssl.pdf">https://www.incommon.org/cert/repository/cps_ssl.pdf</a></td>
</tr>
<tr>
<td>Policy ID #2</td>
<td>2.23.140.1.2.2</td>
</tr>
</tbody>
</table>

| Extension                    | CRL Distribution Points (2.5.29.31)                        |
| Critical                     | NO                                                          |
| URI                          | http://crl.incommon-rsa.org/InCommonRSAServerCA.crt        |

| Extension                    | Certificate Authority Information Access (1.3.6.1.5.5.7.1.1) |
| Critical                     | NO                                                          |
| Method #1                    | CA Issuers (1.3.6.1.5.5.7.48.2)                            |
| URI                          | http://crt.usertrust.com/InCommonRSAServerCA_2.crt         |
| Method #2                    | Online Certificate Status Protocol (1.3.6.1.5.5.7.48.1)    |
| URI                          | http://ocsp.usertrust.com                                  |
Revocation

• Problem: keys get compromised
  ➢ Attacker with a key can impersonate you and read messages encrypted to you

• Key expiration helps with this but not enough

• CA and PGP PKIs support revocation
  ➢ “I, Alice, revoke my public key ... do not use it.”
  ➢ Signs revocation with her private key
  ➢ Others can verify Alice’s signature, stop using key
Revocation

• In CA model, Alice asks CA to revoke certificate
  ➤ Alice does not need private key to do this
  ➤ CAs publish a Certificate Revocation List (CRL)

• In PGP model, only Alice can revoke her own key
  ➤ If Alice loses her private key, she can’t revoke
  ➤ Do not lose private key
  ➤ Option: generate revocation with key, store in secure place
Revocation

- CRL: Certificate Revocation List
  - CA publishes list of revoked certs
  - Client downloads the list
  - Problem: Only care about few certs.
  - Problem: What if CRL server is down?
Revocation

- **OCSP: Online Certificate Status Protocol**
  - Query CA about cert when you get it from server
  - Problem: Revealing visited sites to CA.

- **OCSP stapling:** Web server includes recent OCSP cert
Sometimes CAs go wrong

- CAs get hacked or do the wrong thing:
  - 2011: Comodo and DigiNotar CAs hacked, issue certs for Gmail, Yahoo! Mail, ...
  - 2013: TurkTrust issued cert for Gmail
  - 2014: Indian NIC issue certs for Google and Yahoo!
  - 2016: WoSign issues cert for GitHub

- Solution: Certificate transparency!
  - CAs have to publish certs they issue
<table>
<thead>
<tr>
<th>Category</th>
<th>Disclosure Required?</th>
<th># of CA certs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disclosure Incomplete</td>
<td>Yes!</td>
<td>1 + 6 Summary</td>
</tr>
<tr>
<td>Unconstrained Trust</td>
<td>Yes!</td>
<td>1 + 15 Summary</td>
</tr>
<tr>
<td>Unconstrained, but all unexpired observed paths Revoked</td>
<td>Unknown</td>
<td>46</td>
</tr>
<tr>
<td>Unconstrained, but zero unexpired observed paths</td>
<td>Unknown</td>
<td>1640</td>
</tr>
<tr>
<td>Expired</td>
<td>No</td>
<td>4437</td>
</tr>
<tr>
<td>Technically Constrained (Trusted)</td>
<td>Maybe soon?</td>
<td>56</td>
</tr>
<tr>
<td>Technically Constrained (Other)</td>
<td>No</td>
<td>40</td>
</tr>
<tr>
<td>Disclosed as Revoked, but Expired</td>
<td>Already disclosed</td>
<td>156</td>
</tr>
<tr>
<td>Disclosed as Revoked and in OneCRL</td>
<td>Already disclosed</td>
<td>575</td>
</tr>
<tr>
<td>Disclosed as Revoked (but not in OneCRL)</td>
<td>Already disclosed</td>
<td>32</td>
</tr>
<tr>
<td>Disclosed as Parent Revoked (so not in OneCRL)</td>
<td>Already disclosed</td>
<td>142</td>
</tr>
<tr>
<td>Disclosed, but Expired</td>
<td>Already disclosed</td>
<td>466</td>
</tr>
<tr>
<td>Disclosed, but zero unexpired observed paths</td>
<td>Already disclosed</td>
<td>496</td>
</tr>
<tr>
<td>Disclosed (as Not Revoked), but in OneCRL</td>
<td>Already disclosed</td>
<td>28</td>
</tr>
<tr>
<td>Disclosed, but Technically Constrained</td>
<td>Already disclosed</td>
<td>240</td>
</tr>
<tr>
<td>Disclosed, but with Errors</td>
<td>Already disclosed</td>
<td>0</td>
</tr>
<tr>
<td>Disclosed (as Not Revoked), but Revoked via CRL</td>
<td>Already disclosed</td>
<td>9</td>
</tr>
<tr>
<td>Disclosed (as Not Revoked) and &quot;Unrevoked&quot; from CRL</td>
<td>Already disclosed</td>
<td>2</td>
</tr>
<tr>
<td>Disclosed</td>
<td>Already disclosed</td>
<td>3170</td>
</tr>
<tr>
<td>Unknown to crt.sh or Incorrectly Encoded</td>
<td>Already disclosed</td>
<td>6</td>
</tr>
</tbody>
</table>
How is asymmetric key crypto used?
How are these used in TLS 1.3?

Client

- ClientHello
- [Certificate], [CertificateVerify], Finished
- ApplicationData

Server

- ServerHello, [Certificate], [CertificateVerify], [Finished]
- ApplicationData

secret key

cert_S
How are these used in TLS 1.3?
How are these used in TLS 1.3?

ClientHello: $\text{nonce}_C$, KeyShare

ServerHello: $\text{nonce}_S$, KeyShare, Enc[cert$_S$,…]

CertVerify: $\text{Enc}[\text{Sig}_S(\text{data})]$, Finished

Client

Server

secret key

cert$_S$
How are these used in TLS 1.3?

Client

Server

ClientHello: nonce<sub>C</sub>, KeyShare

ServerHello: nonce<sub>S</sub>, KeyShare, Enc[cert<sub>S</sub>,...]

CertVerify: Enc[Sig<sub>S</sub>(data)], Finished

Diffie-Hellman key exchange

secret key
cert<sub>S</sub>
How are these used in TLS 1.3?

ClientHello: $\text{nonce}_C$, KeyShare

ServerHello: $\text{nonce}_S$, KeyShare, Enc[cert$_S$, ...]

CertVerify: Enc[Sig$_S$(data)], Finished

Finished

session-keys $\leftarrow$ HKDF( DHkey, nonce$_C$, nonce$_S$ )

Diffie-Hellman key exchange
How are these used in TLS 1.3?

ClientHello: nonce<sub>C</sub>, KeyShare

ServerHello: nonce<sub>S</sub>, KeyShare, Enc[cert<sub>S</sub>,...]

CertVerify: Enc[Sig<sub>S</sub>(data)], Finished

Finished

session-keys ← HKDF( DHkey, nonce<sub>C</sub>, nonce<sub>S</sub>)

Encrypted ApplicationData

Encrypted ApplicationData

Diffie-Hellman key exchange
How are these used in TLS 1.3?

{\text{ClientHello: nonce}_C, \text{KeyShare}}

{\text{ServerHello: nonce}_S, \text{KeyShare, Enc[cert}_S,\ldots\text{]}}

\text{CertVerify: Enc[Sig}_S(\text{data})], \text{Finished}}

\text{Finished}

\text{session-keys \leftarrow HKDF( DHkey, nonce}_C, \text{nonce}_S)}

\text{Encrypted ApplicationData}

\text{Encrypted ApplicationData}

\text{TLS 1.2 handshake is way longer, requires encrypting w/ server public key, etc.}
Brief overview of TLS 1.2
Brief overview of TLS 1.2
Brief overview of TLS 1.2

- **Client**:
  - Sends `client-hello`

- **Server**:
  - Sends `server-hello`
  - Sends `server-cert (PK)`
  - Receives `client-hello`
  - Generates `SK`
Brief overview of TLS 1.2

- Browser
- Server

**Client Hello**

**Server Hello** + **Server Cert (PK)**

**Key Exchange (several options):** EC-DHE
Brief overview of TLS 1.2

Browser

Client-hello

Server-hello + server-cert (PK)

Key exchange (several options): EC-DHE

Server-key-exchange

Client-key-exchange

k

Server

Cert

SK

k
Brief overview of TLS 1.2

Browser

client-hello

| server-hello + server-cert (PK) |

key exchange (several options): EC-DHE

| server-key-exchange |

| client-key-exchange |

Finished

k

Server

cert

SK

k
Brief overview of TLS 1.2

- **client-hello**
- **server-hello + server-cert (PK)**
- **key exchange (several options): EC-DHE**
  - **server-key-exchange**
  - **client-key-exchange**
- **Finished**
- HTTP data encrypted with KDF(k)
What does TLS give you?

- **Nonces**: prevent replay of an old session
- **Forward secrecy**: server compromise does not expose old sessions
- **Some identity protection**: certificates are sent encrypted
  - SNI also encrypted so can’t see which server in clear
- **(One sided) authentication**: browser identifies server using server-cert
No excuse not use TLS!