Asymmetric Cryptography

Diffie-Hellman and RSA, TLS, X.509, etc

Reminder Time!

- Go register for the inaugural UMBC DawgCTF!
  - Saturday March 11th, from 9am-5pm
- Learning experience!
- Prizes!
- FREE FOOD!!!!!
- What more could you ask for?! (Don’t say a Ferrari, not enough blood money for that)

Problems with Symmetric Cryptography

- Keys have to remain secret
- But you need a secret key in order to safely transmit secrets
- You have a chicken-and-egg problem

- What if there were some way to communicate keys over an insecure channel

Diffie-Hellman key exchange

- Named after Whitfield Diffie and Martin Hellman
    - they also worked with Ralph Merkle, so you'll sometimes see this referred to as Diffie-Hellman-Merkle key exchange
- It's a way for two parties to securely agree on a cryptographic key using insecure communication, without anyone listening able to derive it.
- No one party can choose the key, but that's okay, they just need the same one.
- Uses number theory, resulting in some seemingly basic level math (at least at the conceptual level)
- Let's work it out:

Diffie-Hellman key exchange: *Algorithm*

- Choose two non-secret values p and g, such that p is prime, and g is a primitive root mod p
  - You don't need to know what a primitive root mod n is to understand D-H. It has something to do with finite fields, but let's ignore that for now. There's a reason we tell you not to implement it without more study :)
- Each person chooses a random integer x from {1, …, p-1}.
- They then calculate $Y = g^x$ (mod p). Y can be transmitted in the clear.
- The other person then calculates $K = Y^x$ mod p. (The other person's Y.)
- This leaves both people with the same K, which can be used as a symmetric key.

Link to Diffie and Hellman's original paper:
https://www-ee.stanford.edu/~hellman/publications/24.pdf

Diffie-Hellman key exchange: *Demo*

- Let's choose p=29 and g=10
  - (I found those numbers on Wikipedia's "Primitive root modulo n" page)
  - Normally, you would, of course, use numbers large enough to prevent easy attacks.
- Alice chooses x=9, Bob chooses x=15.
- Alice calculates $Y = g^x \pmod p = 10^9 \pmod{29} = 18$
- Bob calculates $10^{15} \pmod{29} = 19$
- They transmit 18 and 19 to each other.
- Alice then calculates $K = 19^9 \pmod{29} = 11$
- Bob calculates $K = 18^{15} \pmod{29} = 11$

Question:

We’re doing Diffie-Hellman. We’ve chosen p=23,g=5. We then chose x=6.

We sent Y=8. Our friend sent us Y=19.

What is K?

A. 3
B. 8
C. 2
D. 1

(Remember, we want Y^x (mod p)

Diffie-Hellman key exchange: *Explanation*

- Let's work out the algebra and see if this holds up
  - (of course, this is actually algebra over a finite field, but apparently it holds up enough for demonstration)
- Let's denote Alice's x and Y as x_1 and Y_1, and Bob's as x_2 and Y_2
- Alice calculates $Y_1 = g^{x1}$ (mod p), Bob calculates $Y_2 = g^{x2}$ (mod p)
- They transmit Y_1 and Y_2 to each other.
- Alice then calculates $K = Y_2^{x1}$(mod p), which is equivalent to $(g^{x2})^{x1}$ (mod p)
- That equals $g^{(x2 * x1)}$ (mod p). Bob's calculation will equal $g^{(x1 * x2)}$ (mod p)
- Since multiplication is commutative, Alice and Bob arrive at the same key

Diffie-Hellman key exchange: *Explanation*

- Why is this secure?
- An attacker has g, p, Y_1, Y_2. They need to calculate $K = Y_2^{x1} (\mathrm{mod}\ p)$.
- So they need x_1 (or x_2). They can get that with $\log_g Y_1$ (mod p).
- The cryptography community thinks that’s hard to calculate, and that idea has held up for decades.
- There is no mathematical proof that a log mod p is actually hard to solve. This is called the “discrete logarithm” problem, and a fast solution would make some waves in the security field.
- Interestingly, the field is starting to move to elliptic curve Diffie-Hellman, which is believed to be even harder to do.

RSA

- RSA: Stands for Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman, the three inventors of RSA.
- Is not necessarily a method for key exchange (although you could use it that way)
- Uses two keys: the **public key** and the **private key**
- Public key can be published to the entire world
- Private key must be kept secret.

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Algorithm*

- There will be a substantial amount of handwaving in this explanation to try to emphasize the points that will be helpful to understand.
- Key generation:
  - Generate 2 **random prime numbers**, normally denoted **p** and **q**. They must be kept secret.
  - Calculate **n** = p * q. n will be part of your public key.
  - Choose a **public exponent e**. Today's software uses **e=65537** (0x10001) to make math easier.
  - Calculate the **private exponent d** so that e is congruent to d mod (p-1) * (q-1)
    - Here's a substantial amount of handwaving.
  - Your **public key is n and e** (remember, basically everyone uses e=65537
  - Your **private key is n and d** (you generally save p and q as well)

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Algorithm*

- Encryption:
  - Convert your message to a number between 0 and n-1, we'll call it m.
    - There are intricacies here that are critical for security. We're skipping them for clarity.
  - Ciphertext $C = m^e \pmod{n}$ [Remember e and n were our public key]
- Decryption
  - Plaintext $m = C^d \pmod{n}$ [Remember d and n were our private key]
  - Convert m back into your message
- Proof
  - The proof is based on number theory, and requires advanced enough properties that we won't prove it in class.
  - You're welcome to read the paper, though, the proof is in section 6 (VI).

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Demo*

- Key Generation:
  - Choose **p** = 43, **q** = 59.
  - Let **n** = p * q = 43 * 59 = 2537
  - Choose **e** = 67. Calculate **d** = 1927.
  - **Public key:** n = 2537, e = 67
  - **Private key:** n = 2537, d = 1927
- Encryption
  - Let's encrypt **42**. Let m = 42
  - $C = m^e \pmod{n} \Rightarrow C = \mathbf{42}^{67} \pmod{2537} = \mathbf{1332}$.
- Decryption
  - $M = c^d \pmod{n} \Rightarrow C = \mathbf{1332}^{1927} \pmod{2537} = \mathbf{42}$.
- Want to test this? Use Python! It handles largeish numbers.
  - I used SageMath, a math program based on Python to compute this.

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Explanation*

- Why is this secure?
- You're probably wondering why you can't just factor n into p and q, and calculate the private key.
- We think factoring large products of large primes is hard. Again, there's no proof of this. It's getting easier with modern computers, that's why we use longer and longer keys.

Link to the original RSA paper:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

RSA: *Uses*

- Encryption (obviously)
- Signing
  - Interestingly, the encryption and decryption operations are inverses of each other, so if something is encrypted with the private (*normally decryption*) key, it can be decrypted with the public key.
  - This has the interesting property that since only the person holding the private key can encrypt with it, it follows that if something can be decrypted with the public key, it was encrypted by the person holding the private key.
  - This is a method of accomplishing **digital signatures**. Typically you take a hash of a message you want to sign and encrypt it with the private key. To verify, you decrypt the signature using the public key and see if the hash matches what it should be.

RSA: *Demo*

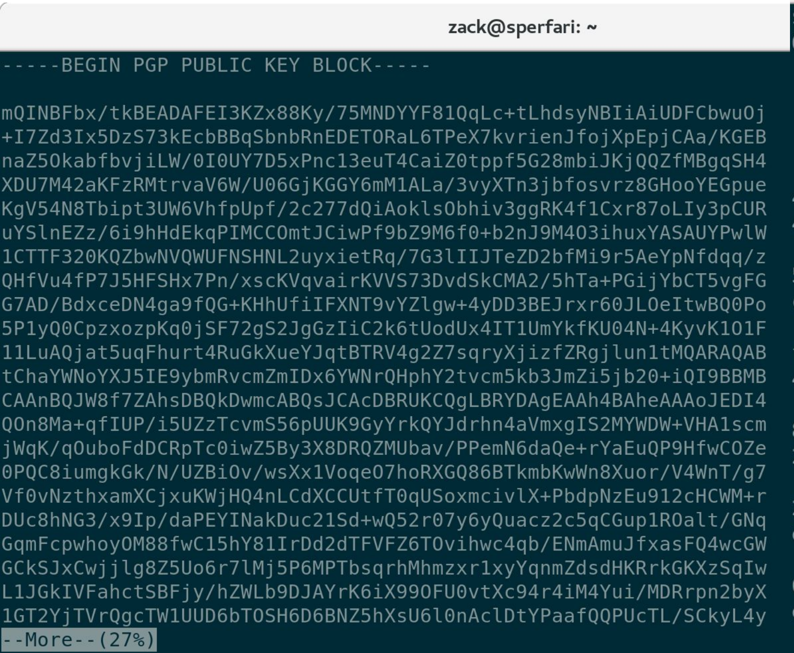
- Let’s use the openssl tool to take a brief look at how one might go about generating an RSA key
- openssl is a cryptographic library used by many major applications, such as the Apache web server and the nginx web server.
- It also comes with a command line tool

```
openssl genrsa -aes256 -out privkey.key 4096

openssl rsa -text -in privkey.key
```

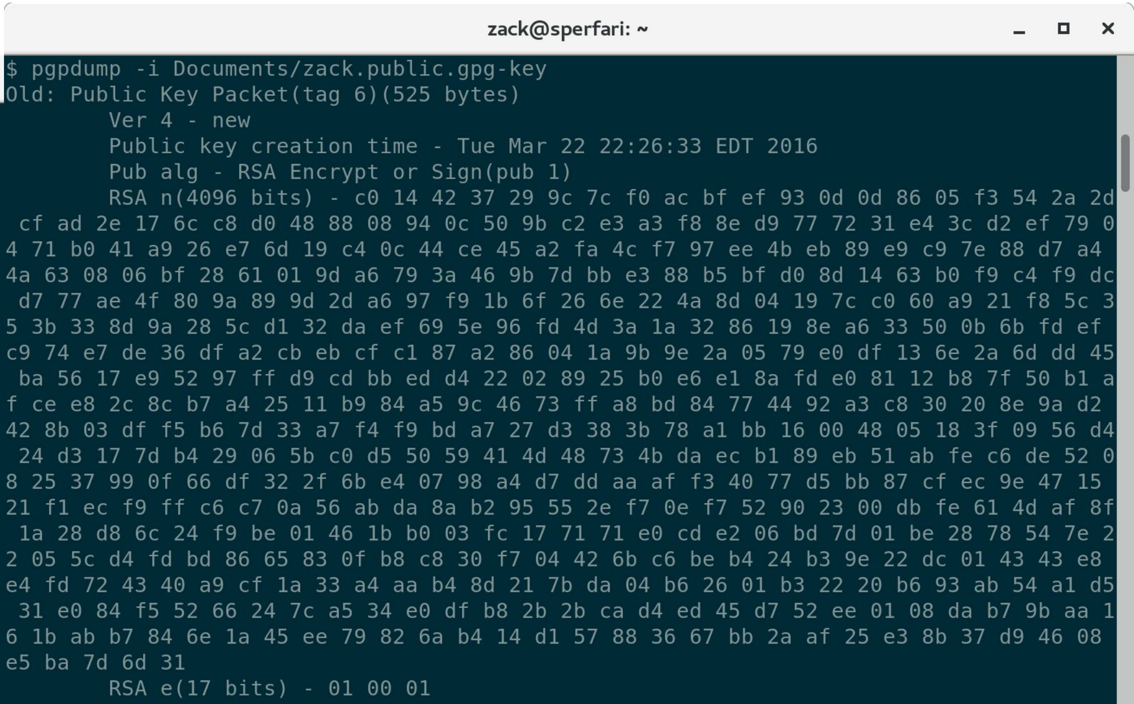
RSA: *Demo*

- Let’s look at a GPG key.



```
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQINBFbx/tkBEADAFEI3KZx88Ky/75MNDYYF81QqLc+tLhdsyNBIiAiUDFCbwuOj
+I7Zd3Ix5DzS73kEcbBBqSbnbRnEDETORaL6TPeX7kvrienJfojXpEpjCAa/KGEB
naZ5OkabfbvjiLW/0I0UY7D5xPnc13euT4CaiZ0tppf5G28mbiJKjQQZfMBgqSH4
XDU7M42aKFzRMtrvaV6W/U06GjKGGY6mM1ALa/3vyXTn3jbfosvrz8GHooYEGpue
KgV54N8Tbipt3UW6VhfpUpf/2c277dQiAoklsObhiv3ggRK4f1Cxr87oLIy3pCUR
uYSlnEZz/6i9hHdEkqPIMCCOmtJCiwPf9bZ9M6f0+b2nJ9M4O3ihuxYASAUYPwlW
1CTTF320KQZbwNVQWUFNSHNL2uyxietRq/7G3lIIJTeZD2bfMi9r5AeYpNfdqq/z
QHfVu4fP7J5HFSHx7Pn/xscKVqvairKVVS73DvdSkCMA2/5hTa+PGijYbCT5vgFG
G7AD/BdxceDN4ga9fQG+KHhUfiIFXNT9vYZlgw+4yDD3BEJrxr60JLOeItwBQ0Po
5P1yQ0CpzxozpKq0jSF72gS2JgGzIiC2k6tUodUx4IT1UmYkfKU04N+4KyvK1O1F
11LuAQjat5uqFhurt4RuGkXueYJqtBTRV4g2Z7sqryXjizfZRgjlun1tMQARAQAB
tChaYWNoYXJ5IE9ybmRvcmZmIDx6YWNrQHphY2tvcm5kb3JmZi5jb20+iQI9BBMB
CAAnBQJW8f7ZAhsDBQkDwmcABQsJCAcDBRUKCQgLBRYDAgEAAh4BAheAAAoJEDI4
QOn8Ma+qfIUP/i5UZzTcvmS56pUUK9GyYrkQYJdrhn4aVmxgIS2MYWDW+VHA1scm
jWqK/qOuboFdDCRpTc0iwZ5By3X8DRQZMUbav/PPemN6daQe+rYaEuQP9HfwCOZe
0PQC8iumgkGk/N/UZBiOv/wsXx1VoqeO7hoRXGQ86BTkmbKwWn8Xuor/V4WnT/g7
Vf0vNzthxamXCjxuKWjHQ4nLCdXCCUtfT0qUSoxmcivlX+PbdpNzEu912cHCWM+r
DUc8hNG3/x9Ip/daPEYINakDuc21Sd+wQ52r07y6yQuacz2c5qCGup1ROalt/GNq
GqmFcpwhoyOM88fwC15hY81IrDd2dTFVFZ6TOvihwc4qb/ENmAmuJfxasFQ4wcGW
GCkSJxCwjjlg8Z5Uo6r7lMj5P6MPTbsqrhMhmzxr1xyYqnmZdsdHKRrkGKXzSqIw
L1JGkIVFahctSBFjy/hZWLb9DJAYrK6iX99OFU0vtXc94r4iM4Yui/MDRrpn2byX
1GT2YjTVrQgcTW1UUD6bTOSH6D6BNZ5hXsU6l0nAclDtYPaafQQPUcTL/SCkyL4y
--More--(27%)
```



```
$ pgpdump -i Documents/zack.public.gpg-key
Old: Public Key Packet(tag 6)(525 bytes)
        Ver 4 - new
        Public key creation time - Tue Mar 22 22:26:33 EDT 2016
        Pub alg - RSA Encrypt or Sign(pub 1)
        RSA n(4096 bits) - c0 14 42 37 29 9c 7c f0 ac bf ef 93 0d 0d 86 05 f3 54 2a 2d
 cf ad 2e 17 6c c8 d0 48 88 08 94 0c 50 9b c2 e3 a3 f8 8e d9 77 72 31 e4 3c d2 ef 79 0
4 71 b0 41 a9 26 e7 6d 19 c4 0c 44 ce 45 a2 fa 4c f7 97 ee 4b eb 89 e9 c9 7e 88 d7 a4
4a 63 08 06 bf 28 61 01 9d a6 79 3a 46 9b 7d bb e3 88 b5 bf d0 8d 14 63 b0 f9 c4 f9 dc
 d7 77 ae 4f 80 9a 89 9d 2d a6 97 f9 1b 6f 26 6e 22 4a 8d 04 19 7c c0 60 a9 21 f8 5c 3
5 3b 33 8d 9a 28 5c d1 32 da ef 69 5e 96 fd 4d 3a 1a 32 86 19 8e a6 33 50 0b 6b fd ef
c9 74 e7 de 36 df a2 cb eb cf c1 87 a2 86 04 1a 9b 9e 2a 05 79 e0 df 13 6e 2a 6d dd 45
 ba 56 17 e9 52 97 ff d9 cd bb ed d4 22 02 89 25 b0 e6 e1 8a fd e0 81 12 b8 7f 50 b1 a
f ce e8 2c 8c b7 a4 25 11 b9 84 a5 9c 46 73 ff a8 bd 84 77 44 92 a3 c8 30 20 8e 9a d2
42 8b 03 df f5 b6 7d 33 a7 f4 f9 bd a7 27 d3 38 3b 78 a1 bb 16 00 48 05 18 3f 09 56 d4
 24 d3 17 7d b4 29 06 5b c0 d5 50 59 41 4d 48 73 4b da ec b1 89 eb 51 ab fe c6 de 52 0
8 25 37 99 0f 66 df 32 2f 6b e4 07 98 a4 d7 dd aa af f3 40 77 d5 bb 87 cf ec 9e 47 15
21 f1 ec f9 ff c6 c7 0a 56 ab da 8a b2 95 55 2e f7 0e f7 52 90 23 00 db fe 61 4d af 8f
 1a 28 d8 6c 24 f9 be 01 46 1b b0 03 fc 17 71 71 e0 cd e2 06 bd 7d 01 be 28 78 54 7e 2
2 05 5c d4 fd bd 86 65 83 0f b8 c8 30 f7 04 42 6b c6 be b4 24 b3 9e 22 dc 01 43 43 e8
e4 fd 72 43 40 a9 cf 1a 33 a4 aa b4 8d 21 7b da 04 b6 26 01 b3 22 20 b6 93 ab 54 a1 d5
 31 e0 84 f5 52 66 24 7c a5 34 e0 df b8 2b 2b ca d4 ed 45 d7 52 ee 01 08 da b7 9b aa 1
6 1b ab b7 84 6e 1a 45 ee 79 82 6a b4 14 d1 57 88 36 67 bb 2a af 25 e3 8b 37 d9 46 08
e5 ba 7d 6d 31
        RSA e(17 bits) - 01 00 01
```

Quantum Computing and Cryptography

- If a **sufficiently large** quantum computer is ever built:
  - RSA and Diffie-Hellman are **completely broken** by an algorithm called Shor's algorithm
  - The **bit length** of symmetric ciphers is **effectively halved**. I.E. if it would previously required 2^{128} computations to crack something, it would require 2^{64} quantum computations.
  - Hash functions - in general preimage resistance is halved, and collision resistance is decreased from $2^{n/2}$ computations to $2^{n/3}$ quantum computations.
- More info: post by cryptographer Thomas Pornin on Stack Exchange:
  - http://security.stackexchange.com/questions/48022/what-kinds-of-encryption-are-not-breakable-via-quantum-computers/48027#48027
- Cryptographers are working on encryption algorithms that are not as vulnerable to quantum cryptography.

Man-in-the-middle attack (**MITM** attack)

- This attack only works if the attacker has the ability to see/modify the traffic between the two parties that are communicating.
- It involves the attacker either intercepting information they need or modifying the communication to suit their needs.
- For example: If Alice asks Bob for money, an attacker can substitute his/her own bank account number for Alice’s, and then Bob will end up sending the attacker money.

Man-in-the-middle attack (**MITM** attack)

- Diffie-Hellman and RSA, in and of themselves, do not mitigate MITM attacks.
- For D-H, the attacker can simply sit in the middle and negotiate a key with each side himself. To Bob, he pretends to be Alice; to Alice, he pretends to be Bob. As they communicate, the attacker simply decrypts the message and reencrypts it with the other key, so that neither party can notice the difference.
- For RSA, the attacker does the same thing, only he swaps in his own public key and performs the same reencryption process as the parties communicate.

Key Distribution

- Wait -- didn’t asymmetric encryption solve the problem of **key distribution**?
- Nope -- we just went over that. We have a new problem, which I’m calling key *authenticity*
- Yes, with asymmetric encryption, we can establish shared secrets through encrypting them with a public key or using Diffie-Hellman, but a MITM attacker can ruin the whole scheme! What do we do?
- Typically, we find some out-of-band way to verify we have the other party’s public key, and then everything is either signed or encrypted.

Key Distribution

- But how do we get that key?
- There are a couple different ways:
    - Manually verify the key, i.e. over the phone by voice or something
        - Not typically used in Internet protocols… for obvious reasons
    - **Web of trust**
        - Used by PGP/GPG.
    - **Public Key Infrastructure (PKI)**
        - Used by SSL,**TLS**,S/MIME
            - That includes **HTTPS**, we'll get to that.

Public Key Infrastructure (**PKI**): *Analogy*

- We'll start with studying uses of PKI, as it's used most commonly.
- It's best explained by analogy.
- Let's say I need to turn in my homework encrypted with RSA
- I need my professor's public key
- I could ask Dr. Hrabowski (whose public key is on the UMBC website)
    - Okay, not really, but this is an example, okay!
- And if he gives me my professor's key, I can turn in my homework and all is well.
- However, I'm going to guess Dr. Hrabowski has better things to do with his time -- and he probably also wants to sleep. This solution isn't scalable.

Public Key Infrastructure (**PKI**): *Analogy*

- What if my professor got Dr. Hrabowski to sign a message saying that “Dr. Soandso’s key is “YWJjMTIzCg==” and put it on the syllabus?
  - Then if I had Dr. Hrabowski’s key, I could get a copy of the syllabus and verify the signature.
  - I could then turn in my homework on time, way too late at night.
- One more concept -- this also isn’t scalable. Dr. Hrabowski doesn’t have time to sign each faculty member’s key.
- So instead each department asks Dr. Hrabowski to sign the department’s key, and then the department signs the professor’s key.
- My professor then includes Dr. Hrabowski’s signature of their department’s key, and the department’s signature of his key in the syllabus.

Public Key Infrastructure (**PKI**): *Explanation*

- Let’s map our analogy to reality now
- In a real PKI, we call signed statements about keys “X.509 certificates”,
  - Normally just called “**certificates**”
  - X.509 is some seemingly arcane standard that dictates how most PKIs work
- And an entity that can sign them is called a certification authority (**CA**)
- In our example, Dr. Hrabowski represents a **Root CA**, and each department is an **intermediate CA**.
- The operating system or browser vendor sets the default list of trusted Root CAs.
  - You can change them if you really want, and know what you’re doing.
- My computer trusts 173 (!) Root CAs.

X.509 Certificates

- Most people resort to copy/pasting `openssl` commands from the internet.
- We're going to take a minute and discuss the theory behind the system, that way you know which commands to copy/paste.
    - It's not that complicated!

What’s in a certificate? (simplified)

- The person or thing the certificate is for
- Their public key
- Whether they’re a CA or not
- Serial number
- Validity times (Not Before and Not After)
- Entity issuing the certificate
- How to tell if the certificate is revoked
    - Unfortunate truth -- this never works as intended
    - Browsers have awful hacks to work around that fact

What’s in a certificate? (slightly less simplified)

- The person or thing the certificate is for
    - [Subject Distinguished Name and Subject Alternative Names]
- Their public key
- Whether they’re a CA or not
    - [Basic Constraints, CA:TRUE or CA:FALSE]
- Serial number
    - Usually some random hex value
- Validity times (Not Before and Not After)
- Entity issuing the certificate
- How to tell if the certificate is revoked
    - CRLs/OCSP endpoints
    - Doesn’t work super well

Question:

Can a CA backdate a cert (i.e. today issue a cert that looks like it was issued on Jan 1)?

A. Yes
B. No

Distinguished Name (**DN**)

- This concept is used in LDAP directories as well as certificates
- Example: the (simplified) DN for the certificate used at https://umbc.edu/
  - (remember they're public)
- cn=*.umbc.edu, o=University of Maryland Baltimore County, ou=UnixInfra, l=Baltimore, st=MD, c=US
- As you can see, it's made up of multiple parts, separated by commas
- Here's another one:
  - CN=*.ycombinator.com, OU=PositiveSSL Wildcard, OU=Domain Control Validated
- Notice any differences?

Distinguished Name (**DN**): *Parts*

- Ex: cn=*.umbc.edu, o=University of Maryland Baltimore County, ou=UnixInfra, l=Baltimore, st=MD, c=US
- CN: **C**ommon **N**ame (more on that later)
- O: **O**rganization
- OU: **O**rganizational **U**nit
- L: **L**ocality
- ST: **St**ate
- C: **C**ountry
- DC: **D**omain **C**omponent
  - (more in LDAP directories, probably not in X.509 certs)

What’s in a certificate? *Subject*

- For TLS or SSL (includes https, discussed later), the CN combined with the Subject Alternative Names (SANs) determines which domains a certificate is valid for
- SANs are sort of new, so if you want to support ancient XP installs, you can’t use them yet
- SANs are just the domain name, no DN
- There can be a wildcard at the beginning of a domain (for extra money)
  - *.google.com, for instance
  - A cert with CN=*.google.com would be valid for:
    - a.google.com
    - b.google.com
    - But NOT a.b.google.com. It’s limited to one level.

What’s in a certificate? *Basic Constraints*

- Root CAs are allowed to issue certificates for intermediate CAs.
- But the certificates given by an intermediate CA to a website shouldn’t be able to sign certificates.
- This is done with the basic constraints section. The certificate can have either CA=TRUE or CA=FALSE set.
- There are other basic constraints; they are not widely used.

Question:

If you have a certificate with the following DN and SANs, which host would **NOT** be valid?

DN: cn=chiapet.net,o=UMBCCD Enterprises,ou=Lab,st=MD,c=US
SAN: *.chiapet.net
SAN: *.dog.chiapet.net
SAN: chiapet.net

A. chiapet.net
B. dog.chiapet.net
C. cat.chiapet.net
D. cat.dog.chiapet.net

Certificate Revocation

- The original system had CAs publish a signed list of revoked certificates, called **CRLs**
  - Unfortunately, this proves impractical to check.
- The next system has browsers check the CAs API to see if a certificate is valid. Called Online Certificate Status Protocol (**OCSP**)
  - This is slightly less impractical to check
- The current gold standard is called **OCSP stapling**, but many servers don't support it yet
  - The web server gets a signed OCSP statement from the CA, and sends it with its certificate.
  - The OCSP response expires much sooner than the certificate

Certificate Revocation

- The way it works in practice is that each browser has its own list of revoked certificates, and it checks that. It might check OCSP or OCSP staples, depending on configuration.
- If a certificate is misissued for a high-valued site, everyone freaks out and blacklists the cert.
  - Sometimes they even blacklist the CA…

Public Key Infrastructure (**PKI**): *Conclusion*

- As long as you trust your CAs, PKI solves the key-distribution problem
  - Why?
- Whether we can trust CAs is currently a somewhat debatable question in the security community
- However, it’s the best system we have at the moment, as applied to TLS

Transport Layer Security (**TLS**)

- TLS is the main protocol in mainstream use today for securing data in transit.
- It’s used by HTTPS, SMTP (sometimes), IMAPS, POPS, and other protocols.
- You may have heard of “Secure Sockets Layer” or SSL. That’s basically an earlier version of TLS.
    - Colloquially, you’ll hear TLS referred to as SSL.
    - If you’re actually using SSL 3.0 though, you’re doing something wrong.

TLS: *Overview*

- Most of this info is condensed from the TLS 1.2 standard (RFC 5246).
  - https://tools.ietf.org/html/rfc5246
- TLS provides security to higher layer protocols.
  - Such as HTTP, IMAP, POP, SMTP, and others
- A couple features:
  - Data sent over TLS is **private**.
    - Only the two parties to the connection can read the data being sent.
  - Data sent over TLS is “**reliable**” (i.e. it can’t get silently corrupted)
  - Optionally, one of the parties to the connection may **prove its identity**. In typical use, that’s the server, but for something like a VPN it may go both ways.
    - This is done using **certificates**, like we just discussed

TLS: *Overview*

- To accomplish its goals, TLS specifies a **handshake** process to:
  - Negotiate the cryptographic algorithms to use
  - Verify each other’s identity
- We’re going to step through the handshake process and talk about the important stuff
  - We’re omitting some details, as they are not necessary to understand the general process.
  - We’re basically going through section 7.4 of the TLS 1.2 standard.
- Please don’t memorize the names of all the types of packet. Try to understand what’s going on and why.

TLS: *Cipher Suites*

- A cipher suite is simply a collection of cryptographic algorithms that together can fulfill the following roles:
  - Key exchange
  - Encryption
  - Message authentication
  - Pseudo-random function
    - Used to generate keys from the master secret, we’ll discuss this later

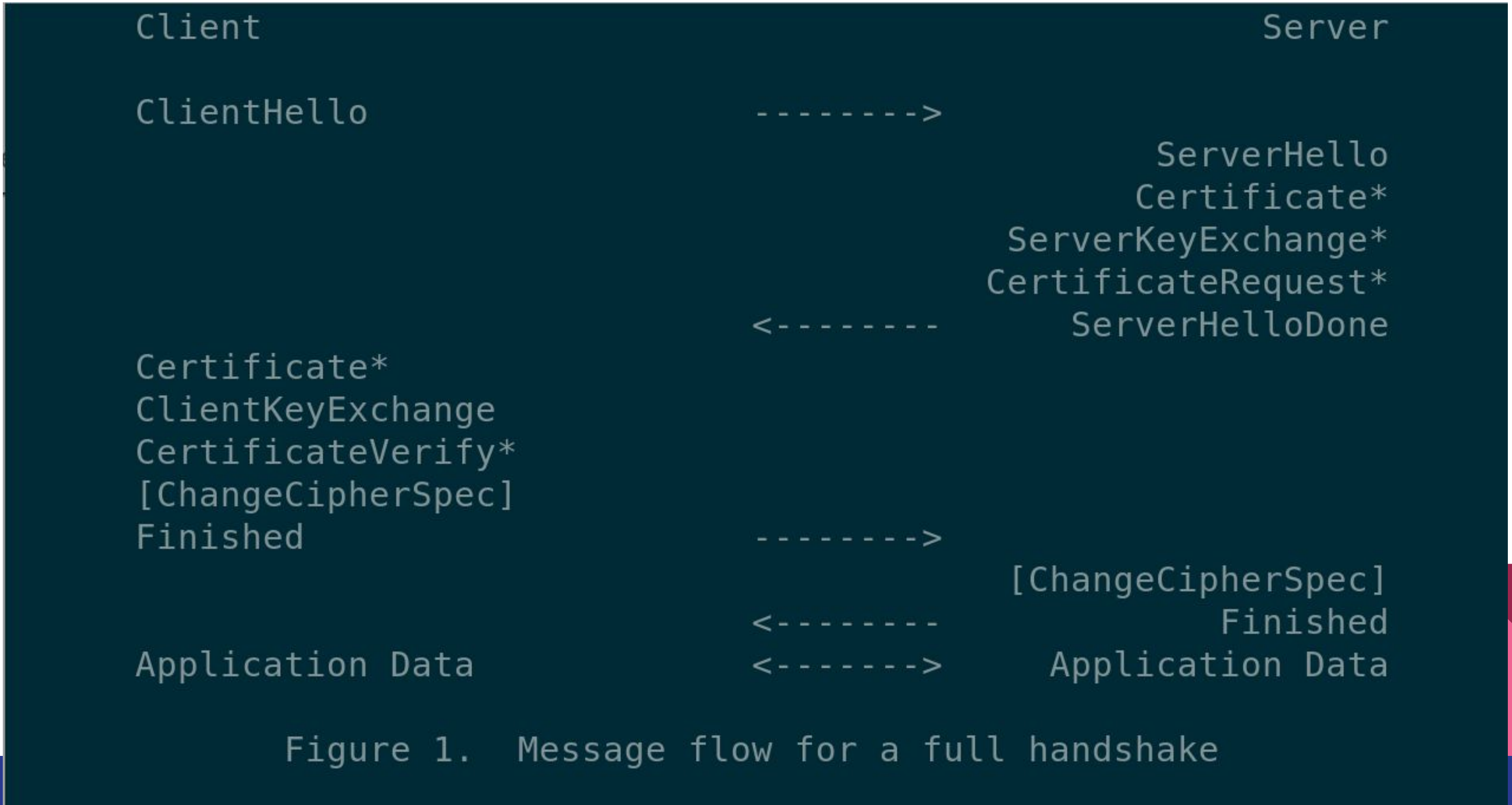
TLS: *Cipher Suites*

```
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
TLS_RSA_WITH_AES_256_CBC_SHA256
```

- Let's spell out some acronyms
- DHE - Diffie-Hellman ephemeral
  - This is Diffie-Hellman, results are destroyed after the connection (they are ephemeral)
  - This provides "**Perfect Forward Secrecy**"
- ECDHE - elliptic curve Diffie-Hellman ephemeral
  - Elliptic curve cryptography makes use of the properties of elliptic curves to produce stronger cryptography with analogous operations to the traditional algorithms. You aren't expected to understand them for this class.
- What do these cipher suites mean?

(From the TLS 1.2 standard)

TLS: *Handshake: Diagram*



```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data

             Figure 1.  Message flow for a full handshake
```

TLS: *Handshake: Client Hello*

- Includes the following information:
  - Version of TLS the client wants to use
  - Random values, for later use
  - Cipher suites the client supports and is willing to use
  - Compression methods the client supports and is willing to use
  - Other stuff we’re leaving out for clarity

TLS: *Handshake: Server Hello*

- If the server can agree on protocol and algorithms, it **chooses what it prefers to use and tells the client**
- Includes the following information:
    - Version of TLS the server chose to use. Usually the highest supported by both parties.
    - Another random value, for later use
    - The cipher suite the server picked from the client's list
    - Compression method the server picked

TLS: *Handshake: Server Certificate*

- Includes the following information:
  - Server’s **certificate**
  - Any other certificates that are needed to verify its signature. (**Certificate Chain**)
  - This allows TLS to guarantee the server’s identity.

TLS: *Handshake: Server Key Exchange*

- Only sent if using a Diffie-Hellman Ephemeral key exchange
- Includes the server’s part of the Diffie-Hellman exchange: p, g, and Y.
    - Remember those?

TLS: *Handshake: Certificate Request*

- Not used often
- If the server wants the client to present a certificate, it can request that now.
- Server can specify type of certificate and allowed CAs
- Browsers will typically then prompt the user
  - Usually really ugly

TLS: *Handshake: ServerHelloDone*

- Exactly what it says
- Server is done its part of the handshake

TLS: *Handshake: Client Certificate*

- Only sent if server requested it
- Includes client’s certificate and any certificate chain needed to verify it
- Client can leave this blank -- and server can choose to close the connection if it wants.

TLS: *Handshake: ClientKeyExchange*

- Contents depend on what type of key exchange is being used.
- If RSA:
  - Client generates a random **premaster secret** and encrypts it with the server's public key
- If Diffie-Hellman:
  - Client sends its Y value

TLS: *Handshake: CertificateVerify*

- Only used if client certificates are being used
- It contains a signature of all messages received so far, so as to prove possession of the private key for the certificate.

TLS: *Handshake: Calculating the **master secret***

- The master secret is 48 bytes (384 bits)
- It’s derived using the PRF from:
  - the premaster secret
  - the two random values from the Hello messages.
- Remember the PRF is normally based on a hash
- The premaster secret was either:
  - Just sent by the client (RSA key exchange)
  - Agreed upon by Diffie-Hellman
- The premaster secret is then erased from memory.

TLS: *Handshake: ChangeCipherSpec*

- Not strictly a handshake message, but happens whenever the cipher is changed
    - For the initial handshake, that’s from no cipher to the agreed one
- At this point, the two parties swap ChangeCipherSpec messages
    - Not much data in them

TLS: *Handshake: Finished*

- This message is actually encrypted now!
- Verifies the success of the handshake
- Contains the PRF of some information: including
  - All the handshake messages, concatenated
  - The master secret
- If something doesn’t add up here, there’s a problem

TLS: *Handshake: Conclusion*

- Conclusion isn’t a message type, we’re done!
- The client has verified the server’s identity
- The server has optionally verified the client
  - Not often
- They have agreed on ciphers to use, and a secret to derive keys from
- Now Application Data can be exchanged!

- And **YOU** understood most of what happened!