secure communication context

"secure" communication

mostly talk about on network

between *principals* $\approx$ people/servers/programs

but same ideas apply to, e.g., messages on disk

communicating with yourself

A to B

running example: A talking with B

- maybe sometimes also with C

attacker E — eavesdropper

- passive
- gets to read all messages over network

attacker M — machine-in-the-middle

- active
- gets to read and replace and add messages on the network

privileged network position

intercept radio signal?

control local wifi router?

  may doesn't just forward messages

compromise network equipment?

send packets with 'wrong' source address

  called "spoofing"

fool DNS servers to 'steal 'name?

fool routers to send you other's data?

possible security properties? (1)

what we'll talk about:

confidentiality — information shared only with those who should have it

authenticity — message genuinely comes from right principal (and not manipulated)

possible security properties? (2)

important ones we won't talk about...:

repudiation — if A sends message to B, B can't prove to C it came from A

(takes extra effort to get along with authenticity)

forward-secrecy — if A compromised now, E can't use that to decode past conversations with B

anonymity — A can talk to B without B knowing who it is

...

secrets

if A is talking to B are communicating,
what stops M (machine-in-the-middle) from pretending to be B?

assumption: B knows some *secret information* that M does not

secrets

if A is talking to B are communicating,
what stops M (machine-in-the-middle) from pretending to be B?

assumption: B knows some *secret information* that M does not

start: assume A and B have a *shared secret* they both know
(and attackers do not)

(later: easier to setup assumptions)

bad ways to use shared secret

A $\rightarrow$ B: What's the password?

B $\rightarrow$ A: It's '`Abc$xyM$e`'.

A $\rightarrow$ B: That's right! Here's my confidential information.

bad ways to use shared secret

A $\rightarrow$ B: What's the password?

B $\rightarrow$ A: It's '`Abc$xyM$e`'.

A $\rightarrow$ B: That's right! Here's my confidential information.

well, this doesn't really help:

against E (eavesdropper), who can read the password AND confidential info

against M (machine-in-the-middle), who can also pretend to be A for B

symmetric encryption

some magic math!

we'll be given two functions by expert:

encrypt: $E(\text{key}, \text{message}) = \text{ciphertext}$

decrypt: $D(\text{key}, \text{ciphertext}) = \text{message}$

key $=$ shared secret

ideally small (easy to share) and chosen at random

unsolved problem: how to share it?

symmetric encryption properties (1)

our functions:

encrypt: $E(\text{key}, \text{message}) = \text{ciphertext}$
decrypt: $D(\text{key}, \text{ciphertext}) = \text{message}$

knowing E and D, it should be hard to
learn anything about the message from the ciphertext without key

"hard" $\approx$ would have to try every possible key

symmetric encryption properties (1)

our functions:

encrypt: $E(\text{key}, \text{message}) = \text{ciphertext}$
decrypt: $D(\text{key}, \text{ciphertext}) = \text{message}$

knowing E and D, it should be hard to
learn anything about the message from the ciphertext without key

"hard" $\approx$ would have to try every possible key

secrecy properties

actually that's not secret enough, usually want to resist recovery of info about message or key even given...

partial info about the message, or

lots of other (message, ciphertext) pairs, or

"known plaintext"

lots of (message, ciphertext) pairs for *other messages the attacker chooses*, or

"chosen plaintext"

lots of (message, ciphertext) pairs encrypted under similar keys, or

"related key"

secrecy properties

actually that's not secret enough, usually want to resist recovery of info about message *or key* even given...

partial info about the message, or

lots of other (message, ciphertext) pairs, or

"known plaintext"

lots of (message, ciphertext) pairs for *other messages the attacker chooses*, or

"chosen plaintext"

lots of (message, ciphertext) pairs encrypted under similar keys, or

"related key"

secrecy properties

actually that's not secret enough, usually want to resist recovery of info about message or key even given...

partial info about the message, or

lots of other (message, ciphertext) pairs, or

"known plaintext"

lots of (message, ciphertext) pairs for *other messages the attacker chooses*, or

"chosen plaintext"

lots of (message, ciphertext) pairs encrypted under similar keys, or

"related key"

secrecy properties

actually that's not secret enough, usually want to resist recovery of info about message or key even given...

partial info about the message, or

lots of other (message, ciphertext) pairs, or

> "known plaintext"

lots of (message, ciphertext) pairs for *other messages the attacker chooses*, or

> "chosen plaintext"

lots of (message, ciphertext) pairs encrypted under similar keys, or

> "related key"

secrecy properties

actually that's not secret enough, usually want to resist recovery of info about message or key even given...

partial info about the message, or

lots of other (message, ciphertext) pairs, or

"known plaintext"

lots of (message, ciphertext) pairs for *other messages the attacker chooses*, or

"chosen plaintext"

lots of (message, ciphertext) pairs encrypted under similar keys, or

"related key"

using?

in advance: A and B share encryption key

A computes E(key, 'The secret formula is...') $=$ ***

send on network:
A $\rightarrow$ B: ***

using?

in advance: A and B share encryption key

A computes E(key, ‘The secret formula is…’) $=$ ***

send on network:
A $\rightarrow$ B: ***

B computes D(key, ***) $=$ ‘The secret formula is …’

encryption is not enough

if B receives an encrypted message from A, and...

it makes sense when decrypted, why isn't that good enough?

problem: an active attacker M
can *selectively* manipulate the encrypted message

simple encryption idea (1)

suppose encrypting message

one possible idea: generate unique number N (e.g. counter)

combine N and key to produce message size-bit bitstring Y

say Y=f(X, key) where f is some 'secure' function:

- f is something like a hash function, but supports arbitrary size output
- f is effectively irreversible
- f is effectivel yequally/unpredictably distributed

use Y XOR message as encrypted value

simple encryption idea (2)

E(K, message) = (N, f(N, key) XOR message) = (N, C)

If we know (N, C) and don't know key, can we figure out anything about *message*?

violates f(N, key) being equally/unpredictably distributed

If we know (N, C) and message, can we find out about *key*

violates f(N, key) being irreversible

If we know (N, C) and message, can we decrypt (N', C')?

remember: each encrypted message chooses unique N

not if f(N, key) and f(N', key) don't have predictable relationship

manipulating simple encryption

E(K, message) = (N, f(N, key) xor message) = (N, C)

If we know (N, C) and message, we can generate an encryption of (all zeroes):

(N, C xor message) = (N, f(N, key) xor (message xor message)) = (N, f(N, key) xor 0)

And can generate encryption of some other message Q:

(N, C xor message xor Q) = (N, f(N, key) xor Q)

manipulating messages, more generally

as an active attacker

if we know part of plaintext
can sometimes make it read anything else by flipping bits

> "Pay \$100 to Bob" $\rightarrow$ "Pay \$999 to Bob"

we can sometimes shorten

> "Pay \$100 to ABC Corp if they ..." $\rightarrow$ "Pay \$100 to ABC Corp"

we can sometimes corrupt selected parts of message and check what the response is

> e.g. what changes don't make B reject message as malformed?
> with repeated tries, often reveals part of message's values

maybe don't xor?

these XOR-based constructions are very common

- example: probably used within most connections to websites

there are other ideas, but...

but can still generate meaningful manipulated messages

- usually just need to work on larger units than bits

actual solution: additional *message authentication code*

- sometimes provided pre-combined with encryption and called *authenticated encryption*

calling things encryption

in this class, *(symmetric) encryption* means condidentiality but not authenticity

has malleability problme

matches most common thing a library calls encryption

but, sometimes encryption will be...

“authenticated encryption” = encryption + message authentication code, or

some lower level tool (similar to f function earlier) that needs extra steps to provide confidentiality

message authentication codes (MACs)

goal: use shared secret *key* to verify message origin

one function: $MAC(\text{key}, \text{message}) = \text{tag}$

knowing MAC and the message and the tag, it should be hard to:

- find the value of $MAC(\text{key}, \text{other message})$ — ("forge" the tag)
- find the key

contrast: MAC v checksum

message authentication code acts like checksum, but…

checksum can be recomputed without any key

checksum meant to protect against accidents, not malicious attacks

- checksum can be faster to compute + shorter

using without encryption?

in advance: choose + share MAC key

A prepares message:

A computes 'Please pay \$100 to M.'

A computes MAC(MAC key, 'Please pay \$100 to M.') = @@@

A $\rightarrow$ B: Please pay \$100 to M. @@@

using without encryption?

in advance: choose + share MAC key

A prepares message:

A computes 'Please pay $100 to M.'

A computes MAC(MAC key, 'Please pay $100 to M.') = @@@

A → B: Please pay $100 to M. @@@

B processes message:

B recomputes MAC(MAC key, 'Please pay $100 to M.')

rejects if it doesn't match @@@

using with encryption?

in advance: choose $+$ share encryption key and MAC key

A prepares message:

A computes E(encrypt key, 'The secret formula is...') $=$ ***

A computes MAC(MAC key, ***) $=$ @@@

A $\rightarrow$ B: *** @@@

using with encryption?

in advance: choose + share encryption key and MAC key

A prepares message:

A computes E(encrypt key, ‘The secret formula is...’) $=$ ***

A computes MAC(MAC key, ***) $=$ @@@

A $\rightarrow$ B: *** @@@

B processes message:

B recomputes MAC(MAC key, ***)

rejects if it doesn’t match @@@

B computes D(key, ***) $=$ ‘The secret formula is ...’

"authenticated encryption"

often encryption + MAC packaged together

name: authenticated encryption

exercise

suppose A, B have shared keys K_1, K_2

assume attackers do not have keys

E/D = encrypt/decrypt function

A asks B to pay Sue $100 by sending message with these parts:

"2023-11-03: pay $100"
$E(K_1,$ "2023-11-03 Sue"$)$
$MAC(K_2,$ "2023-11-03 $100"$)$

1. can eavesdropper learn: (a) who is being paid, (b) how much?

2. can machine-in-middle change: (a) who is being paid, (b) how much?

shared secrets impractical

problem: shared secrets usually aren't practical

need secure communication before I can do secure communication?

scaling problems

- millions of websites $\times$ billions of browsers $=$ how many keys?
- hard to talk to new people

shared secrets impractical

problem: shared secrets usually aren't practical

need secure communication before I can do secure communication?

scaling problems

- millions of websites $\times$ billions of browsers $=$ how many keys?
- hard to talk to new people

shared secrets impractical

problem: shared secrets usually aren't practical

need secure communication before I can do secure communication?

scaling problems

- millions of websites $\times$ billions of browsers $=$ how many keys?
- hard to talk to new people

bootstrapping keys?

will still need to have some sort of secure communication to setup!

because we need some way to know we aren't talking to attacker

bootstrapping keys?

will still need to have some sort of secure communication to setup!

because we need some way to know we aren't talking to attacker

but...

bootstrapping keys?

will still need to have some sort of secure communication to setup!

because we need some way to know we aren't talking to attacker

but...

can be broadcast communication

don't need full new sets of keys for each web browser

bootstrapping keys?

will still need to have some sort of secure communication to setup!

because we need some way to know we aren't talking to attacker

but...

can be broadcast communication

- don't need full new sets of keys for each web browser

only with smaller number of trusted authorities

- don't need to have keys for every website in advance

asymmetric encryption

we'll have two functions:

encrypt: $PE(\text{public key}, \text{message}) = \text{ciphertext}$
decrypt: $PD(\text{private key}, \text{ciphertext}) = \text{message}$

(public key, private key) $=$ "key pair"

key pairs

'private key' = kept secret

usually not shared with *anyone*

'public key' = safe to give to everyone

usually some hard-to-reverse function of public key

concept will appear in some other cryptographic primitives

asymmetric encryption properties

functions:

encrypt: $PE(\text{public key}, \text{message}) = \text{ciphertext}$
decrypt: $PD(\text{private key}, \text{ciphertext}) = \text{message}$

should have:

knowing PE, PD, the public key, and ciphertext shouldn't make it too easy to find message
knowing PE, PD, the public key, ciphertext, and message shouldn't help in finding private key

secrecy properties with asymmetric

not going to be able to make things as hard as "try every possibly private key"

but going to make it impractical

like with symmetric encryption want to prevent recovery of *any info about message*

also have some other attacks to worry about:

- e.g. no info about key should be revealed based on our reactions to decrypting maliciously chosen ciphertexts

using asymmetric v symmetric

both:

- use secret data to generate key(s)

asymmetric (AKA public-key) encryption

- one "keypair" per recipient
- private key kept by recipient
- public key sent to all potential senders
- encryption is one-way without private key

symmetric encryption

- one key per (recipient + sender)
- secret key kept by recipient + sender
- if you can encrypt, you can decrypt

using?

in advance: B generates private key $+$ public key

in advance: B sends public key to A (and maybe others) securely

A computes PE(public key, 'The secret formula is...') $=$ *******

send on network:
A $\rightarrow$ B: *********

B computes PD(private key, *******) $=$ 'The secret formula is ...'

digital signatures

symmetric encryption : asymetric encryption ::
message authentication codes : digital signatures

digital signatures

pair of functions:

- sign: $S(\text{private key}, \text{message}) = \text{signature}$
- verify: $V(\text{public key}, \text{signature}, \text{message}) = 1$ ("yes, correct signature")

(public key, private key) = key pair (similar to asymmetric encryption)

- public key can be shared with everyone
- knowing S, V, public key, message, signature
- doesn't make it too easy to find another message + signature so that $V(\text{public key}, \text{other message}, \text{other signature}) = 1$

using?

in advance: A generates private key + public key

in advance: A sends public key to B (and maybe others) securely

A computes S(private key, ‘Please pay ...’) = *******

send on network:
A $\rightarrow$ B: ‘Please pay ...’, ********

B computes V(public key, ‘Please pay ...’, *******) = 1

tools, but...

have building blocks, but less than straightforward to use

lots of issues from using building blocks poorly

start of art solution: formal proof sytems

- mathematical proof that attacker doing X implies encryption/MAC/etc. broken
- ideally a somewhat machine-checkable proof

(we aren't going to be that formal...)

replay attacks

A→B: Did you order lunch? [signature 1 by A]

signature 1 by A = Sign(A's private signing key, "Did you order lunch?")
will check with Verify(A's public key, signature 1 by A, "Did you order lunch?")

B→A: Yes. [signature 1 by B]

signature 1 by B = Sign(B's private key, "Yes.")
will check with Verify(B's public key, signature 1 by B, "Yes.")

A→B: Vegetarian? [signature 2 by A]
B→A: No, not this time. [signature 2 by B]

...

A→B: There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature N by A]

replay attacks

A$\rightarrow$B: Did you order lunch? [signature 1 by A]
B$\rightarrow$A: Yes. [signature 1 by B]
A$\rightarrow$B: Vegetarian? [signature 2 by A]
B$\rightarrow$A: No, not this time. [signature 2 by B]
...
A$\rightarrow$B: There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature ? by A]

how can attacker hijack the reponse to A's inquiry?

replay attacks

A→B: Did you order lunch? [signature 1 by A]
B→A: Yes. [signature 1 by B]
A→B: Vegetarian? [signature 2 by A]
B→A: No, not this time. [signature 2 by B]
...
A→B: There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature ? by A]

how can attacker hijack the reponse to A's inquiry?

as an attacker, I can copy/paste B's earlier message!

- just keep the same signature, so it can be verified!
- Verify(B's public key, "Yes.", signature 2 from B) $= 1$

nonces (1)

one solution to replay attacks:

A→B: *#1* Did you order lunch? [signature 1 from A]
signature from A = Sign(A's private key, "*#1* Did you order lunch?")

B→A: *#1* Yes. [signature 1 from B]
A→B: *#2* Vegetarian? [signature 2 from A]
B→A: *#2* No, not this time. [signature 2 from B]
...
A→B: *#54* There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature ? from A]

(assuming A actually checks the numbers)

nonces (2)

another solution to replay attacks:

B→A: [next number *#91523*] [signature from B]
A→B: *#91523* Did you order lunch? [next number *#90382*] [signature from A]
B→A: *#90382* Yes. [next number *#14578*] [signature from B]
…
A→B: *#6824* There's a guy at the door, says he's here to repair the AC. Should I let him in? [next number *#36129*][signature from A]

(assuming A actually checks the numbers)

replay attacks (alt)

M→B: #50 Did you order lunch? [signature by M]
B→M: #50 Yes. [signature intended for M by B]

A→B: #50 There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature ? by A]

how can M hijack the reponse to A's inquiry?

replay attacks (alt)

M→B: #50 Did you order lunch? [signature by M]
B→M: #50 Yes. [signature intended for M by B]

A→B: #50 There's a guy at the door, says he's here to repair the AC. Should I let him in? [signature ? by A]

how can M hijack the reponse to A's inquiry?

as an attacker, I can copy/paste B's earlier message!

> just keep the same signature, so it can be verified!
> Verify(B's public key, "#50 Yes.", signature intended for M by B) $= 1$

confusion about who's sending?

in addition to nonces, either

- write down more who is sending + other context so message can't be reused and/or
- use unique set of keys for each principal you're talking to

with symmetric encryption, also "reflection attacks"

- A sends message to B, attacker sends A's message back to A as if it's from B

other attacks without breaking math

TLS state machine attack

from `https://mitls.org/pages/attacks/SMACK`

protocol:

step 1: verify server identity
step 2: receive messages from server

attack:

if server sends "here's your next message",
instead of "here's my identity"
then broken client ignores verifying server's identity

Matrix vulnerabilties

one example from `https://nebuchadnezzar-megolm.github.io/static/paper.pdf`

system for confidential multi-user chat

protocol + goals:

- each device (my phone, my desktop) has public key
- to talk to me, you verify one of my public keys
- to add devices, my client can forward my other devices' public keys

bug:

- when receiving new keys, clients did not check who they were forwarded from correctly

on the lab

getting public keys?

browser talking to websites
needs public keys of every single website?

not really feasible, but…

certificate idea

let's say A has B's public key already.

if C wants B's public key and knows A's already:

A can generate "certificate" for B:

- "B's public key is XXX" AND
- Sign(A's private key, "B's public key is XXX")

B send copy of their "certificate" to C (most common idea)

if C trusts A, now C has B's public key

- if C does not trust A, well, can't trust this either

certificate idea

let's say A has B's public key already.

if C wants B's public key and knows A's already:

A can generate "certificate" for B:

- "B's public key is XXX" AND
- Sign(A's private key, "B's public key is XXX")

B send copy of their "certificate" to C (most common idea)

if C trusts A, now C has B's public key

- if C does not trust A, well, can't trust this either

certificate idea

let's say A has B's public key already.

if C wants B's public key and knows A's already:

A can generate "certificate" for B:

> "B's public key is XXX" AND
> Sign(A's private key, "B's public key is XXX")

B send copy of their "certificate" to C (most common idea)

if C trusts A, now C has B's public key

> if C does not trust A, well, can't trust this either

certificate authorities

websites (and others) go to *certificates authorities* (CA) with their public key

certificate authorities sign messages like:
"The public key for foo.com is XXX."

signed message called *certificate*

send certificates to browsers to verify identity

website can forward certificate instead of browser contacting CA directly

certificate authorities

websites (and others) go to *certificates authorities* (CA) with their public key

certificate authorities sign messages like:
"The public key for foo.com is XXX."

signed message called *certificate*

send certificates to browsers to verify identity

website can forward certificate instead of browser contacting CA directly

example web certificate (1)

```
    Version: 3 (0x2)
    Serial Number: 7b:df:f6:ae:2e:d7:db:74:d3:c5:77:ac:bc:44:bf:1b
    Signature Algorithm: sha256WithRSAEncryption
    Issuer:
        countryName               = US
        stateOrProvinceName       = MI
        localityName              = Ann Arbor
        organizationName          = Internet2
        organizationalUnitName    = InCommon
        commonName                = InCommon RSA Server CA
    Validity
        Not Before: Apr 25 00:00:00 2023 GMT
        Not After : Apr 24 23:59:59 2024 GMT
    Subject:
        countryName               = US
        stateOrProvinceName       = Virginia
        organizationName          = University of Virginia
        commonName                = canvas.its.virginia.edu
....
    X509v3 extensions:
....
        X509v3 Subject Alternative Name: DNS:canvas.its.virginia.edu
```

example web certificate (2)

```
....
    Subject Public Key Info:
        Public Key Algorithm: rsaEncryption
            RSA Public-Key: (2048 bit)
            Modulus:
                00:a2:fb:5a:fb:2d:d2:a7:75:7e:eb:f4:e4:d4:6c:
                94:be:91:a8:6a:21:43:b2:d5:9a:48:b0:64:d9:f7:
                f1:88:fa:50:cf:d0:f3:3d:8b:cc:95:f6:46:4b:42:

....
Signature Algorithm: sha256WithRSAEncryption
Signature Value:
    24:3a:67:c8:0d:ef:eb:8c:eb:ba:8f:d5:11:d2:1e:ea:44:eb:
    fe:af:93:7d:d9:4a:2b:44:a3:7f:47:50:aa:d1:b3:9c:a8:a8:
....
```

certificate chains

That certificate signed by "InCommon RSA Server CA"

CA = certificate authority

so their public key, comes with my OS/browser?

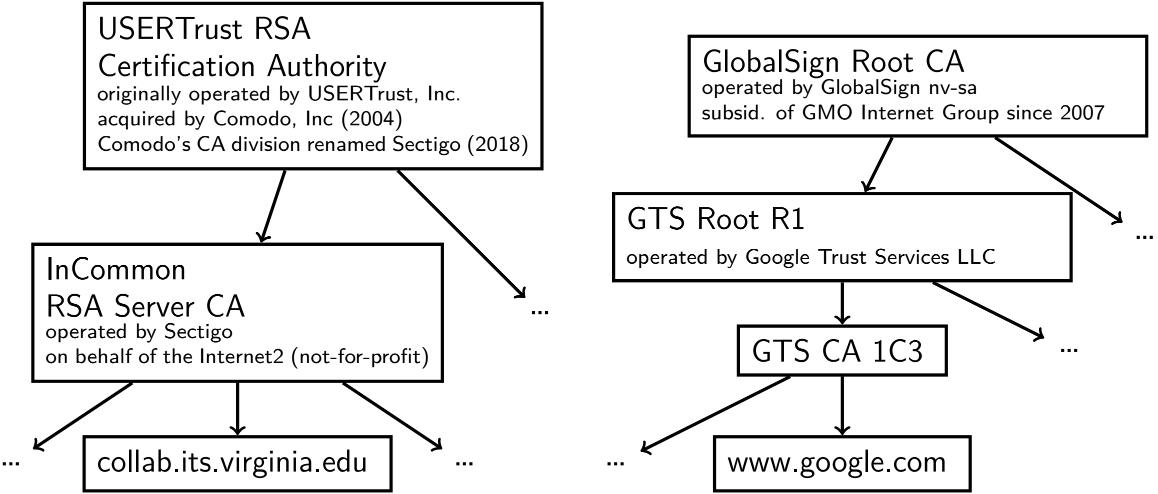
not exactly...

they have their own certificate signed by "USERTrust RSA Certification Authority"

and their public key comes with your OS/browser?

(but both CAs now operated by UK-based Sectigo)

certificate hierarchy



certificate hierarchy

USERTrust RSA
Certification Authority
originally operated by USERTrust, Inc.
acquired by Comodo, Inc (2004)
Comodo's CA division renamed Sectigo (2018)

GlobalSign Root CA
operated by GlobalSign nv-sa
subsid. of GMO Internet Group since 2007

GTS Root R1
operated by Google Trust Services LLC

...

InCommon
RSA Server CA
operated by Sectigo
on behalf of the Internet2 (not-for-profit)

...

GTS CA 1C3

...

...

some "trust anchors" included with browsers and OSes
(for GTS Root R1, only more recent browsers/OSes)

how many trust anchors?

Mozilla Firefox (as of 27 Feb 2023)

155 trust anchors
operated by 55 distinct entities

Microsoft Windows (as of 27 Feb 2023)

237 trust anchors
operated by 86 distinct entities

public-key infrastructure

ecosystem with certificate authorities
and certificates for everyone

called "public-key infrastructure"

several of these:

- for verifying identity of websites
- for verifying origin of domain name records (kind-of)
- for verifying origin of applications in some OSes/app stores/etc.
- for encrypted email in some organizations
- …

exercise

exercise: how should website certificates verify identity?

how do certificate authorities verify

for web sites, set by CA/Browser Forum

organization of:

- everyone who ships code with list of valid certificate authorities
  - Apple, Google, Microsoft, Mozilla, Opera, Cisco, Qihoo 360, Brave, ...
- certificate authorities

decide on rules (“baseline requirements”) for what CAs do

BR domain name identity validation

options involve CA choosing random value and:

sending it to domain contact (with domain registrar) and receive response with it, or

observing it placed in DNS or website or sent from server in other specific way

exercise: problems this doesn't deal with?

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

    some browsers check these, sometimes

certificate transparency

    public logs of every certificate issued
    some browsers reject non-logged certificates
    so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

    can indicate which CAs are allowed to issue certificates in DNS
    (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked certificates*

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- *public logs* of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate which CAs are allowed to issue certificates in DNS
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

some other things public CAs do

keep their private keys in tamper-resistant hardware

maintain publicly-accessible database of *revoked* certificates

- some browsers check these, sometimes

certificate transparency

- public logs of every certificate issued
- some browsers reject non-logged certificates
- so you can tell if bad certificate exists for your website

'CAA' records in the domain name system

- can indicate *which CAs are allowed to issue certificates in DNS*
- (but CAs apparently not required to use DNSSEC (certificate infrastructure for signing domain name records) when looking this up)

additional crypto tools

cryptographic hash functions (summarize data)

'secure' random numbers

key agreement

motivation: summary for signature

digital signatures typically have size limit

...but we want to sign very large messages

solution: get secure "summary" of message

cryptographic hash

hash(M) $=$ X

given X:

hard to find message other than by guessing

given X, M:

hard to find second message so that hash(second message) $=$ X

example uses:

substitute for original message in digital signature

building message authentication codes

password hashing

cryptographic hash functions need (basically) guessing to 'reverse'

idea: store cryptographic hash of password instead of password

- attacker who gets hash doesn't get password
- but can still check entered password is correct

password hashing

cryptographic hash functions need (basically) guessing to ‘reverse’

idea: store cryptographic hash of password instead of password

- attacker who gets hash doesn’t get password
- but can still check entered password is correct

problem: with fast hash function, can try lots of guesses fast

password hashing

cryptographic hash functions need (basically) guessing to 'reverse'

idea: store cryptographic hash of password instead of password
- attacker who gets hash doesn't get password
- but can still check entered password is correct

problem: with fast hash function, can try lots of guesses fast

fix: special slow/resource-intensive cryptograph hash functions
- Argon2i
- scrypt
- PBKDF2

random numbers

need a lot of keys that no one else knows

common task: choose a *random* number

question: what does *random* mean here?

cryptographically secure random numbers

security properties we might want for random numbers:

attacker cannot guess (part of) number better than chance

knowing prior ‘random’ numbers shouldn’t help predict next ‘random’ numbers

compromising machine now shouldn’t reveal older random numbers

exercise: how to generate?

/dev/urandom

Linux kernel random number generator

collects "entropy" from hard-to-predict events

- e.g. exact timing of I/O interrupts
- e.g. some processor's built-in random number circuit

turned into as many random bytes as you want

turning ‘entropy’ into random bytes

lots of ways to do this; one (rough/incomplete) idea:

internal variable *state*

to add ‘entropy’

$\text{state} \leftarrow \text{SecureHash}(\text{state} + \text{entropy})$

to extract value:

$\text{random bytes} \leftarrow \text{SecureHash}(1 + \text{state})$

give bytes that can’t be reversed to compute state

$\text{state} \leftarrow \text{SecureHash}(2 + \text{state})$

change state so attacker can’t take us back to old state if compromised

just asymmetric?

given public-key encryption + digital signatures...

why bother with the symmetric stuff?

symmetric stuff much faster

symmetric stuff much better at supporting larger messages

key agreement

problem: A has B's public encryption key
wants to choose shared secret

some ideas:

- A chooses a key, sends it encrypted to B
- A sends a public key encrypted to B, B chooses a key and sends back

key agreement

problem: A has B's public encryption key
wants to choose shared secret

some ideas:

- A chooses a key, sends it encrypted to B
- A sends a public key encrypted to B, B chooses a key and sends back

alternate model (not needed, but usually used by TLS, SSH, …):

- both sides generate random values
- derive public-key like “key shares” from values
- use math to combine “key shares”
- kinda like A + B both sending each other public encryption keys

Diffie-Hellman key agreement

A and B want to agree on shared secret

A chooses random value Y

A sends public value derived from Y (“key share”)

B chooses random value Z

B sends public value derived from Z (“key share”)

A combines Y with public value from B to get number

B combines Z with public value from A to get number

and b/c of math chosen, both get same number

Diffie-Hellman key agreement (details, if needed)

math requirement:

some f, so $f(f(X,Y),Z) = f(f(X,Z),Y)$
(that's hard to invert, etc.)

choose X in advance and:

A randomly chooses Y	B randomly chooses Z
A sends $f(X,Y)$ to B	B sends $f(X,Z)$ to A
A computes $f(f(X,Z),Y)$	B computes $f(f(X,Y),Z)$

Diffie-Hellman key agreement (details, if needed)

math requirement:

some f, so $f(f(X,Y),Z) = f(f(X,Z),Y)$
(that's hard to invert, etc.)

choose X in advance and:

A randomly chooses Y	B randomly chooses Z
A sends $f(X,Y)$ to B	B sends $f(X,Z)$ to A
A computes $f(f(X,Z),Y)$	B computes $f(f(X,Y),Z)$

example $f(a,b) = a^b \pmod{p}$

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

PE(public key, message) $=$

generate random value Z
combine with public key to get shared secret
use symmetric encryption $+$ MAC using shared secret as keys
output: (key share generated from Z) (sym. encrypted data) (mac tag)

key agreement and asym. encryption

can construct public-key encryption from key agreeement

private key: generated random value Y

public key: key share generated from that Y

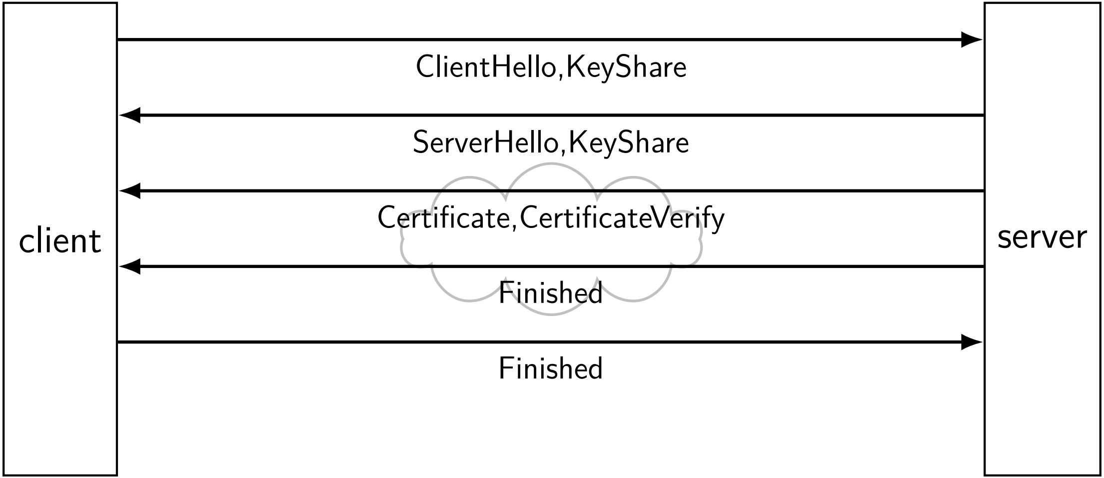
PE(public key, message) $=$

generate random value Z
combine with public key to get shared secret
use symmetric encryption $+$ MAC using shared secret as keys
output: (key share generated from Z) (sym. encrypted data) (mac tag)

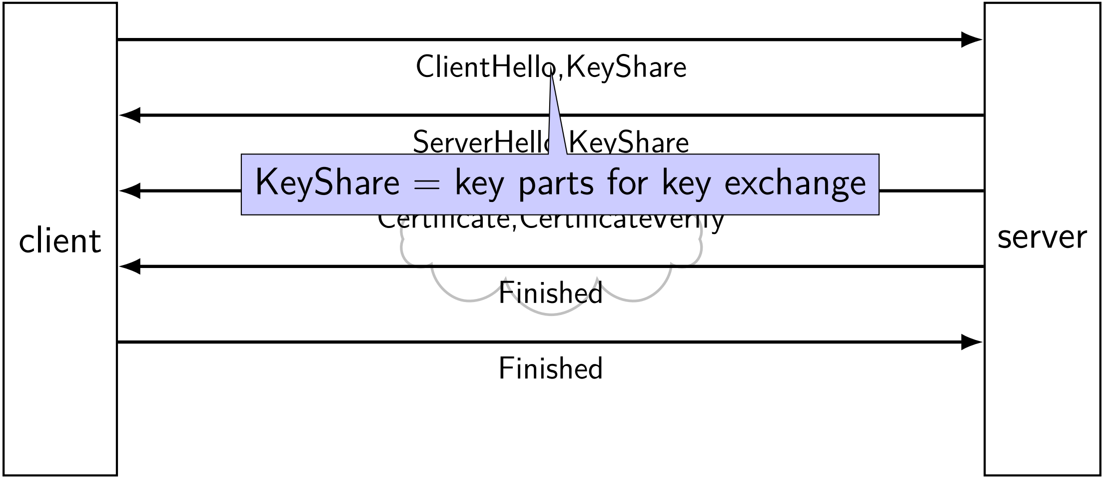
PD(private key, message) $=$

extract (key share generated from Z)
combine with private key to get shared secret, ...

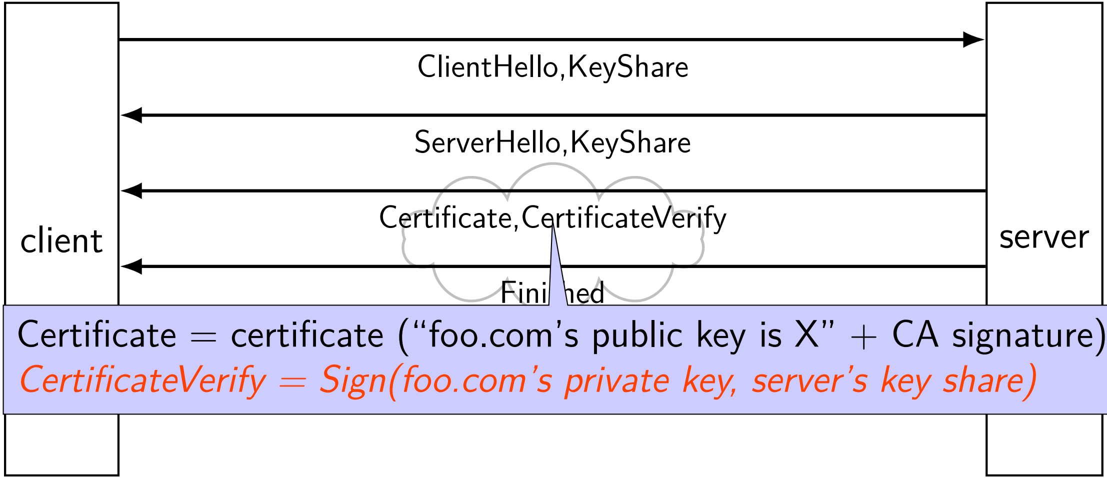
typical TLS handshake



typical TLS handshake



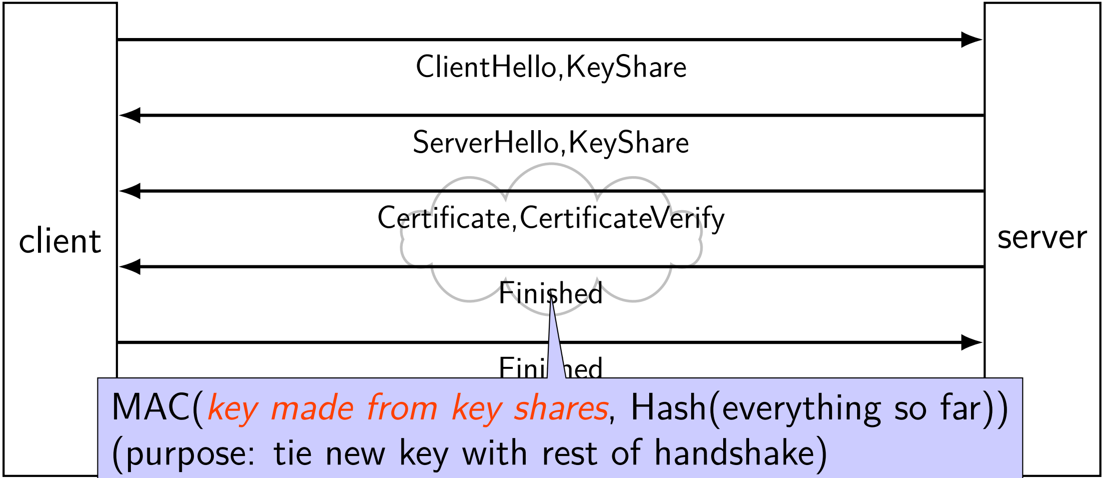
typical TLS handshake



Certificate = certificate (“foo.com’s public key is X” + CA signature)

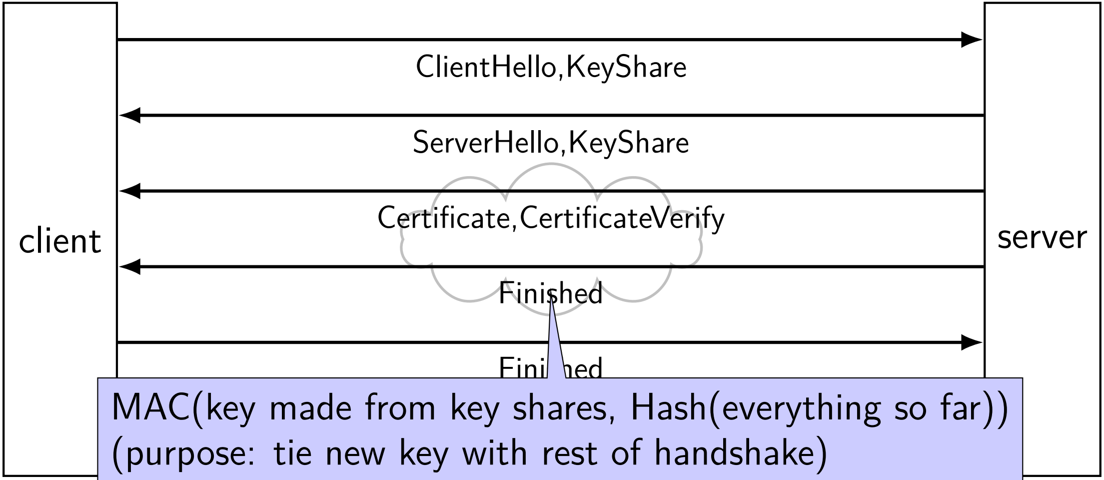
CertificateVerify = Sign(foo.com’s private key, server’s key share)

typical TLS handshake



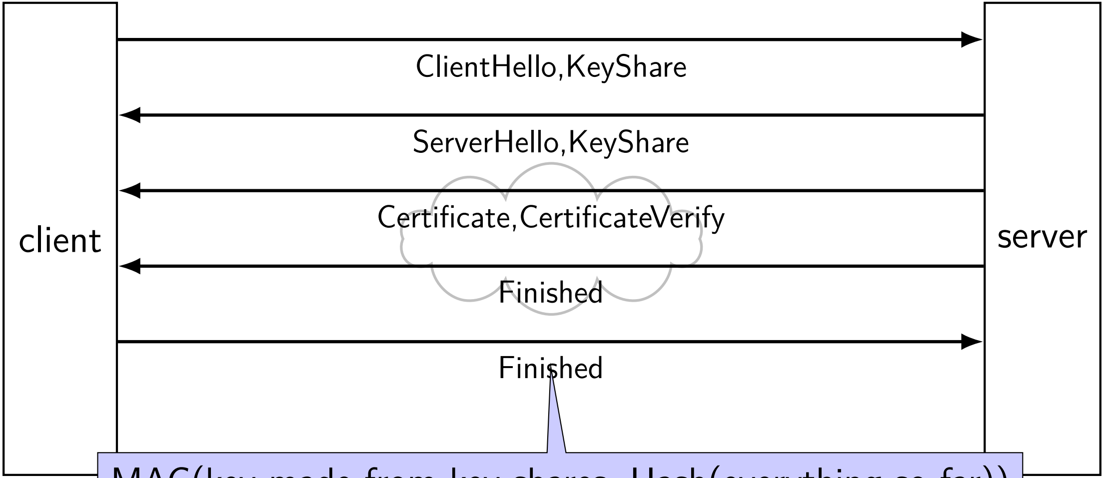
MAC(*key made from key shares*, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



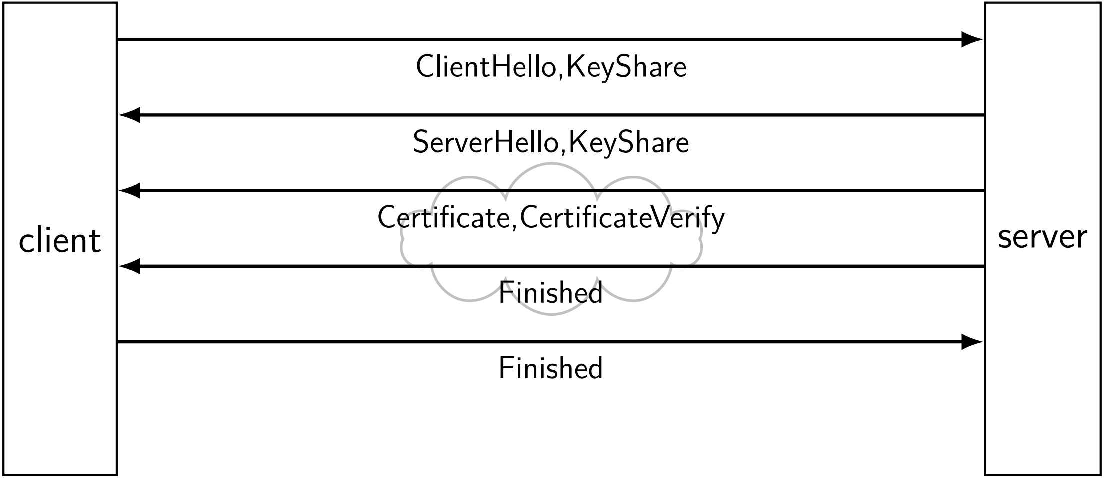
MAC(key made from key shares, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



MAC(key made from key shares, Hash(everything so far))
(purpose: tie new key with rest of handshake)

typical TLS handshake



TLS: after handshake

use key shares results to get **several** keys

take hash(something + shared secret) to derive each key

separate keys for each direction (server $\rightarrow$ client and vice-versa)

often separate keys for encryption and MAC

later messages use encryption + MAC + nonces

things modern TLS usually does

(not all these properties provided by all TLS versions and modes)

confidentiality/authenticity

- server = one ID'd by certificate
- client = same throughout whole connection

forward secrecy

- can't decrypt old conversations (data for KeyShares is temporary)

fast

- most communication done with more efficient symmetric ciphers
- 1 set of messages back and forth to setup connection

denial of service (1)

so far: worried about network attacker disrupting confidentiality/authenticity

what if we're just worried about just breaking things

well, if they control network, nothing we can do...

but often worried about less

denial of service (2)

if you just want to inconvenience...

attacker just sends lots of stuff to my server

my server becomes overloaded?

my network becomes overloaded?

but: doesn't this require a lot of work for attacker?

exercise: why is this often not a big obstacle

denial of service: asymmetry

work for attacker $>$ work for defender

how much computation per message?

- complex search query?
- something that needs tons of memory?
- something that needs to read tons from disk?

how much sent back per message?

resources for attacker $>$ resources of defender

how many machines can attacker use?

denial of service: reflection/amplification

instead of sending messages directly…attacker can send messages "from" you to third-party

third-party sends back replies that overwhelm network

example: short DNS query with lots of things in response

"amplification" =

third-party inadvertantly turns small attack into big one

firewalls

don't want to expose network service to everyone?

solutions:

- service picky about who it accepts connections from
- filters in OS on machine with services
- filters on router

later two called "firewalls"

firewall rules examples?

ALLOW tcp port 443 (https) FROM everyone

ALLOW tcp port 22 (ssh) FROM *my desktop's IP address*

BLOCK tcp port 22 (ssh) FROM everyone else

ALLOW from address X to address Y

…

network security summary (1)

communicating securely with math

- secret value (shared key, public key) that attacker can't have
- symmetric: shared keys used for (de)encryption + auth/verify; fast
- asymmetric: public key used by any for encrypt + verify; slower
- asymmetric: private key used by holder for decrypt + sign; slower

protocol attacks — repurposing encrypt/signed/etc. messages

certificates — verifiable forwarded public keys

key agreement — for generated shared-secret "in public"

- publish key shares from private data
- combine private data with key share for shared secret

network security summary (2)

TLS: combine all cryptography stuff to make "secure channel"

(things we probably didn't get to:)

denial-of-service — attacker just disrupts/overloads (not subtle)

firewalls

backup slides

backup slides

cryptographic hash uses

find shorter 'summary' to substitute for data

- what hashtables use them for, but...
- we care that adversaries can't cause collisions!

cryptographic hash uses

find shorter 'summary' to substitute for data

what hashtables use them for, but...

we care that adversaries can't cause collisions!

deal with message limits in signatures/etc.

password hashing — but be careful! [next slide]

constructing message authentication codes

hash message + secret info (+ some other details)