6.824 Lecture 18: Certificate Transparency, Equivocation

why are we looking at Certificate Transparency?
  so far we have assumed closed systems
    where all participants are trustworthy
    e.g. Raft peers
  what if the system is open -- anyone can use it?
  and there's no universally-trusted authority to run the system?
  you have to build useful systems out of mutually suspicious pieces
    this makes trust and security top-level distributed systems issues
  the most basic is "am I talking to the right computer?"
    this is the (almost unsolvable) problem that CT helps address
  this material ties backwards to consistency
    since CT is all about ensuring all parties see the same information
  this material ties forwards to block-chain and Bitcoin
    CT is a non-crypto-currency use of a block-chain-like design
    and it's deployed and used a lot

before certificates (i.e. before 1995)
  man-in-the-middle (MITM) attacks were a concern on the web
  [browser, internet, gmail.com, fake server, stolen password]
  not too hard to redirect traffic:
    DNS isn't very secure, can fake DNS information for gmail.com to browser
    network routers, routing system, WiFi not always very secure

basic certificate and CA scheme
  maps DNS names to public keys
    user is assumed to know the DNS name of who they want to talk to
  [Google, CA, gmail.com server, browser, https connection]
  a certificate contains:
    DNS name e.g. "gmail.com"
    public key of that server
    identity of CA
    signature with CA's private key
  browsers contain list of public keys of all acceptable CAs
  when browser connects with https:
    server sends certificate
    browser checks CA signature (using list of acceptable CA public keys)
    browser challenges server to prove it has private key
  now MITM attacks are harder:
    assume user clicks on https://gmail.com
    attacker's fake gmail.com server must have certificate for gmail.com

why certificates haven't turned out to be perfect
  it's not clear how to decide who owns a DNS name
    if I ask CA for a certificate for "x.com", how does CA decide?
    turns out to be hard, even for well-known names like microsoft.com
  worse: there are over 100 CAs in browsers' lists
    not all of them are well-run
    not all of them employ only trustworthy employees
    not all of them can resist demands from country's government
  and: any CA can issue a certificate for any name
    so the overall security is limited by the least secure CA
  result:
    multiple incidents of "bogus" certificates
    e.g. certificate for gmail.com, issued to someone else
    hard to prevent

hard even to detect that a bogus certificate is out there!

why not an online DB of valid certs?
DB service would detect and reject bogus certificates
browsers would check DB before using any cert
- how to decide who owns a DNS name?
if you can't decide, you can't reject bogus certificates.
- must allow people to change CAs, renew, lose private key, &c
these all look like a 2nd certificate for an existing name!
- who would run it?
there's no single authority that everyone in the world trusts.

how does Certificate Transparency (CT) approach this problem?
it's really an audit system; it doesn't directly prohibit anything
makes sense since the name ownership rules aren't well defined
the main effect is to ensure that the existence of all certificates is public
[gmail.com, CA, CT log server, browser, gmail's monitor]
the basic action:
gmail.com asks CA for a certificate
CA issues cert to gmail.com
CA registers certificate with CT log server (typically more than one)
log server adds certificate to log
browser connects to gmail.com
gmail.com provides certificate to browser
browser asks CT log server if cert is in the log
meanwhile:
gmail.com's Monitor periodically fetches the entire CT log
scans the log for all certs that say "gmail.com"
complains if there are other than the one it knows about
since those must be bogus
thus:
if browsers and monitors see the same log,
and monitors raise an alarm if there's a bogus cert in the log,
and browsers require that each cert they use is in the log,
then browsers can feel safe using any cert that's in the log.

the hard part: how to ensure everyone sees the same log?
when the log operator may be malicious,
and conspiring with (other) malicious CAs
critical: no deletion (even by malicious log operator)
otherwise: log could show a bogus cert to browser,
claim it is in the log, then delete after browser uses it,
so monitors won't notice the bogus cert.
(really "no undetected deletion")
critical: no equivocation (i.e. everyone sees same content)
otherwise: log server could show browser a log with the
bogus cert, and show the monitor a log without
the bogus cert.
(really "no undetected equivocation")
then
if a CA issues a bogus certificate,
it must add it to central log in order for clients to accept it,
but then it can't be deleted,
so name owner's Monitor will eventually see it.

how can we have an immutable/append-only fork-free log w/ untrusted server?

step one: Merkle Tree
[logged certs, tree over them, root]
log server maintains a log of certificates
let's pretend always a power of two log entries, for simplicity
H(a,b) means cryptographic hash of a+b
key property: cannot find two different inputs that produce the same hash
binary tree of hashes over the log entries -- Merkle Tree
for each hash value, only one possible sequence of log entries
if you change the log in any way, the root hash will change too
STH is Signed Tree Head -- signed by log server
so it can't later deny it issued that root hash
once a log server reveals an STH to me,
it has committed to specific log contents.

how log server appends records to a Merkle Tree log
assume N entries already in the log, with root hash H1
log server waits for N new entries to arrive
hashes them to H2
creates new root H3 = H(H1, H2)

how a log server proves that a certificate is in the log under a given STH
"proof of inclusion" or "RecordProof()" or "Merkle Audit Proof"
since browser doesn't want to use the cert if its not in the log
since then Monitors would not see it
and there's be no protection against the cert being bogus
the proof shows that, for an STH, a certificate, and a log position, that
the specified certificate has to be at that position.
the browser asks the log server for the current STH.
(the log server may lie about the STH; we'll consider that later.)
consider a log with just two records, a and b.
STH = H(H(a), H(b))
initially browser knows STH but not a or b.
browser asks "is a in the log?"
server replies "0" and z=H(b) -- this is the proof
browser checks H(H(a), z) = STH
browser asks "is x in the log?" -- but we know x is not in the log under STH
log server wants to lie and say yes.
it wants to do this to trick the browser into using
bogus cert x without a Monitor seeing x in the log.
browser knows STH.
so log server needs to find a y such that
H(H(x), H(y)) = STH = H(H(a), H(b))
for x != a
but cryptographic hash guarantees this property:
infeasible to produce any pair of distinct messages M1 â‰ M2 with
identical hashes H(M1) = H(M2).
you can extend this to bigger trees by providing the "other"
hashes all the way up to the root.
the proofs are smallish -- log(N) hashes for a log with N elements.
important; we don't want browsers to have to e.g. download the whole log.
there are millions of certificates out there.

what if the log server cooks up a *different* log for the browser,
that contains a bogus cert, and sends the STH for that log
only to the browser (not to any Monitors)?
i.e. the server lies about what the current STH is.
the log server can prove to the browser that the bogus cert

is in that log.
this is a fork attack -- or equivocation.
[diagram -- linear log, forked]
forks are possible in CT.
but that is not the end of the story.

how to detect forks/equivocation?
browsers and monitors should compare STHs to make sure they
are all seeing the same log.
called "gossip".
how can we tell if a given pair of STHs are evidence of a fork?
they can differ legitimately, if one is from a later version
of the log than the other!

merkle consistency proof or TreeProof()
given two STHs H1 and H2, is the log under H1 a prefix of the log under H2?
clients ask the log server to produce a proof
if proof works out, the log server hasn't forked them
if no proof, log server may have forked them -- shown them different logs
the proof:
for each root hash Hz = H(Hx,Hy) as the tree grew from H1 to H2,
the right-hand hash -- Hy
clients can compute H(H(H1,Hy1),Hy2) ... and check if equal to H2

why does the log consistency proof mean the clients are seeing the same log?
what if H2 is derived from different log entries in the region that H1 covers?
i.e. the log server has forked the two clients
suppose H2 is the very next larger power-of-two-size log from H1
[draw tree]
the clients, who both know H1 and H2, are expecting this proof:
x such that H2 = H(H1, x)
because the log server forked the clients, we know
H2 = H(Hz, y) where Hz != H1
so the cheating log server would have to find x such that
H(H1, x) = H2 = H(Hz, y) where H1 != Hz
but cryptographic hashes promise this isn't likely:
not practical to find two different inputs that produce the same output
so a consistency proof is convincing evidence that H1 is a prefix
of H2, and thus that the log server hasn't forked the two clients

so, if browsers and Monitors do enough gossiping and require consistency proofs,
they can be pretty sure that they are all seeing the same log,
and thus that Monitors are checking the same set of certificates that
web browsers see.

one last proof issue
what if browser encounters bogus cert C1,
gets a valid proof of inclusion from the log server,
but it's in a forked log intended to trick the browser
[diagram of fork]
browser will go ahead and use the bogus C1
but next time the browser talks to log server,
log server provides it with the main fork and its STH again
now when browser compares STHs with Monitors, it will
look like nothing bad happened

browsers use log consistency proofs to prevent switching forks

browser remembers (persistently) last STH it got from log server
each time browser talks to log server, it gets a new STH
browser demands proof that old STH represents a log prefix of new STH
thus, once log server reveals a certificate to browser,
future STHs it shows browsers must also reflect that certificate
[diagram]
so, if a log server forks a browser, it can't then switch forks
this is called "fork consistency"
once forked, always forked
so, next time browser gossips with a Monitor, the fork will be revealed
so, log servers are motivated to not fork

what should happen if someone notices a problem?
failure to provide log consistency proofs:
i.e. evidence of a fork attack
this suggests the log service is corrupt or badly managed
after investigation, the log server would probably be taken off
browers' approved lists
failure to provide proof of inclusion despite a signed SCT
this may be evidence of an attack
i.e. showing a victim browser a certificate,
but omitting it from logs where Monitors would see it.
or perhaps the log server is slow about adding certificates to its log
bogus certificate in a CT log
e.g. a certificate for mit.edu, but MIT didn't ask for it
human must talk to responsible CA and find out the story
often simply a mistake -> add cert to revocation list
if happens a lot, or is clearly malicious
browser vendors may delete CA from their approved list

how can CT be attacked?
window between when browser uses a cert and monitors check log
no-one monitoring a given name
not always clear what to do if there's a problem
who owns a given name?
maybe the bogus cert was an accident?
lack of gossip in current CT design/implementations
privacy/tracking: browsers asking for proofs from log server

what to take away
the key property: everyone sees the same log, despite malice
both browsers and DNS name owners
if a browser sees a bogus cert, so will the DNS name owner
a consistency property!
auditing is worth considering when prevention isn't possible
equivocation
fork consistency
gossip to detect forks
we will see much of this again in the next lecture, about Bitcoin