Key (mis)Management
Applied
Crypto and Crapto
Announcements...

- HW2 out now
  - Due Friday the 28th
- MT1 on Tuesday...
  - See Piazza for the room assignments
How Can We Communicate With Someone New?

• Public-key crypto gives us amazing capabilities to achieve confidentiality, integrity & authentication without shared secrets …

• But how do we solve MITM attacks?

• How can we trust we have the true public key for someone we want to communicate with?

• Ideas?
Trusted Authorities

• Suppose there’s a party that everyone agrees to trust to confirm each individual’s public key
  • Say the Governor of California
• Issues with this approach?
  • How can everyone agree to trust them?
  • Scaling: huge amount of work; single point of failure …
    • ... and thus Denial-of-Service concerns
  • How do you know you’re talking to the right authority??
Trust Anchors

• Suppose the trusted party distributes their key so everyone has it ...
Jerry Brown’s Public Key is 0x6a128b3d3dc67edc74d690b19e072f64.
Trust Anchors

• Suppose the trusted party distributes their key so everyone has it …

• We can then use this to bootstrap trust
  • As long as we have confidence in the decisions that that party makes
Digital Certificates

- Certificate ("cert") = signed claim about someone’s public key
- More broadly: a signed *attestation* about some claim

- Notation:
  \{ M \}_K = “message M encrypted with public key k”
  \{ M \}^{-1}_K = “message M signed w/ private key for K”

- E.g. M = “Nick's public key is $K_{Nick} = 0xF32A99B...$”
  Cert: M,
  \{ “Nick's public key … $0xF32A99B...$” \}^{-1}_{Jerry} = 0x923AB95E12...9772F
Certificate

Jerry Brown hereby asserts:
Nick’s public key is \( K_{Grant} = 0xF32A99B \ldots \)
The signature for this statement using \( K^{-1}_{Jerry} \) is \( 0x923AB95E12 \ldots 9772F \)
Certificate

Jerry Brown hereby asserts:
Nick’s public key is $K_{\text{Grant}} = 0xF32A99B...$

The signature for this statement using $K^{-1}$ is $0x923AB95E12...9772F$
Jerry Brown hereby asserts:

Nick’s public key is \( K_{\text{Grant}} = 0xF32A99B \ldots \)

The signature for this statement using \( K^{-1}_{\text{Jerry}} \) is computed over all of this

\( K^{-1}_{\text{Jerry}} = 0x923AB95E12 \ldots 9772F \)
Certificate

Jerry Brown hereby asserts:
Grant’s public key is $K_{Grant} = 0xF3A99B...$
The signature for this statement using $K^{-1}_{Jerry}$ is $0x923AB95E12...9772F$
and can be validated using:
Jerry Brown hereby asserts:
Grant’s public key is $K_{Grant}$
The signature for this statement using $K^{-1}_{Jerry}$ is $0x923AB95$
If We Find This Cert Shoved Under Our Door …

• What can we figure out?
  • If we know Jerry’s key, then whether he indeed signed the statement
  • If we trust Jerry’s decisions, then we have confidence we really have Nick's key

• Trust = ?
  • Jerry won’t willy-nilly sign such statements
  • Jerry won’t let his private key be stolen
Analyzing Certs Shoved Under Doors …

- **How** we get the cert doesn’t affect its utility
- **Who** gives us the cert doesn’t matter
  - They’re not any more or less trustworthy because they did
  - Possessing a cert doesn’t establish any identity!
- However: if someone demonstrates they can decrypt data encrypted with $K_{\text{nick}}$, then we have high confidence they possess $K^{-1}_{\text{nick}}$
  - Same for if they show they can sign “using” $K_{\text{nick}}$
Scaling Digital Certificates

• How can this possibly scale? Surely Jerry can’t sign everyone’s public key!

• Approach #1: Introduce hierarchy via delegation

  • \{ “Janet Napolitano’s public key is 0x... and I trust her to vouch for UC” }K^{-1}_{Jerry}
  • \{ “Carol Christ’s public key is 0x... and I trust him to vouch for UCB” }K^{-1}_{Janet}
  • \{ “James Demmel’s public key is 0x... and I trust him to vouch for EECS” }K^{-1}_{Carol}
  • \{ “Nick Weaver's public key is 0x...” }K^{-1}_{Jim}
Scaling Digital Certificates, con’t

• Nick puts this last on his web page
  • (or shoves it under your door)
• Anyone who can gather the intermediary keys can validate the chain
  • They can get these (other than Jerry’s) from anywhere because they can validate them, too

• Approach #2: have multiple trusted parties who are in the business of signing certs …
  • (The certs might also be hierarchical, per Approach #1)
Certificate Authorities

- CAs are trusted parties in a Public Key Infrastructure (PKI)
- They can operate offline
  - They sign ("cut") certs when convenient, not on-the-fly (… though see below …)
- Suppose Alice wants to communicate confidentially w/ Bob:
  - Bob gets a CA to issue {Bob’s public key is $B\} K^{-1}_{CA}$
  - Alice gets Bob’s cert any old way
  - Alice uses her known value of $K_{CA}$ to verify cert’s signature
  - Alice extracts $B$, sends $\{M\}K_B$ to Bob
Is this really Bob?

\[ \{\text{Bob: } B\}_{K^{-1}}^\text{CA} \]
I’d like to talk privately with Bob

Alice

\{Bob: B\}_{K^{-1}}_{CA}

Bob
Does CA’s signature on $B$ validate?

$C_i = E(M_i, B)$
Is this really Bob?
Is this really Mal?

\[\{\text{Mal: } B^*\}^{-1}_{\text{CA}}\]
Bob

I’d like to talk privately with Bob

Mallory

Bob

Alice

{\text{Mal: } B^*}_{K^{-1}_C A}
Wait, I want to talk to Bob, not Mallory!
Revocation

- What do we do if a CA screws up and issues a cert in Bob’s name to Mallory?
I’d like to talk privately with Bob.

\[ \text{I'd like to talk privately with Bob} \]
Revocation

- What do we do if a CA screws up and issues a cert in Bob’s name to Mallory?
  - E.g. Verisign issued a Microsoft.com cert to a Random Joe
  - (Related problem: Bob realizes b has been stolen)
- How do we recover from the error?
- Approach #1: expiration dates
  - Mitigates possible damage
  - But adds management burden
    - Benign failures to renew will break normal operation
Revocation, con’t

• Approach #2: announce revoked certs
  • Users periodically download cert revocation list (CRL)
CRL = Certificate Revocation List

Time for my weekly revoked cert download
CRL = Certificate Revocation List
Revocation, con’t

- Approach #2: announce revoked certs
  - Users periodically download cert revocation list (CRL)
- Issues?
  - Lists can get large
  - Need to authenticate the list itself – how?
Time for my weekly revoked cert download

CRL = Certificate Revocation List

Bob

Mallory

Alice
Revocation, con’t

- Approach #2: announce revoked certs
  - Users periodically download cert revocation list (CRL)

- Issues?
  - Lists can get large
  - Need to authenticate the list itself – how? Sign it!
  - Mallory can exploit download lag
  - What does Alice do if can’t reach CA for download?
    - Assume all certs are invalid (fail-safe defaults)
    - Wow, what an unhappy failure mode!
    - Use old list: widens exploitation window if Mallory can “DoS” CA (DoS = denial-of-service)
The (Failed) Alternative: The “Web Of Trust”

- Alice signs Bob’s Key
  - Bob Sign’s Carol’s
- So now if Dave has Alice’s key, Dave can believe Bob’s key and Carol’s key…
  - Eventually you get a graph/web of trust…
- PGP started out with this model
  - You would even have PGP key signing parties
  - But it proved to be a disaster: Trusting central authorities can make these problems so much simpler!
The Facebook Problem: Applied Cryptography in Action

- Facebook Messenger now has an encrypted chat option
  - Limited to their phone application
- The cryptography in general is very good
  - Used a well regarded asynchronous messenger library (from Signal) with many good properties, including forward secrecy
- When Alice wants to send a message to Bob
  - Queries for Bob's public key from Facebook's server
  - Encrypts message and send it to Facebook
  - Facebook then forwards the message to Bob
- Both Alice and Bob are using encrypted and authenticated channels to Facebook
Facebook's Unique Messenger
Problem: Abuse

- Much of Facebook's biggest problem is dealing with abuse...
  - What if either Alice or Bob is a stalker, an a-hole, or otherwise problematic?
    - Aside: A huge amount of abuse is explicitly gender based, so I'm going to use "Alex" as the abuser and "Bailey" as the victim through the rest of this example
  - Facebook would expect the other side to complain
    - And then perhaps Facebook would kick off the perpetrator for violating Facebook's Terms of Service
  - But fake abuse complaints are also a problem
    - So can't just take them on face value
  - And abusers might also want to release info publicly
    - Want sender to be able to deny to the public but not to Facebook
Facebook's Problem
Quantified

• Unless Bailey forwards the unencrypted message to Facebook
  • Facebook must not be able to see the contents of the message
• If Bailey does forward the unencrypted message to Facebook
  • Facebook must ensure that the message is what Alex sent to Bailey
• Nobody but Facebook should be able to verify this: No public signatures!
  • Critical to prevent abusive release of messages to the public being verifiable
The Protocol In Action

What Is Bailey's Public Key?
Aside: Key Transparency...

- Both Alex and Bailey are trusting Facebook's honesty...
  - What if Facebook gave Alex a different key for Bailey? How would he know?
  - Facebook messenger has a nearly hidden option which allows Alex to see Bailey's key
    - If they ever get together, they can manually verify that Facebook was honest
- The mantra of central key servers: Trust but Verify
  - The simple option is enough to force honesty, as each attempt to lie has some probability of being caught
- This is the biggest weakness of Apple iMessage:
  - iMessage has (fairly) good cryptography but there is no way to verify Apple's honesty
The Protocol In Action

{message=E(K_{pub\_b},
M="Hey Bailey I'm going to
say something abusive",
\text{k}_{\text{rand}}),
mac=\text{HMAC}(\text{k}_{\text{rand}}, M),
to=\text{Bailey},
from=\text{Alex},
time=now,
fbmac=\text{HMAC}(K_{fb}, \{\text{mac, from, 
\text{to, time}\})}
Some Notes

• Facebook *can not* read the message or even verify Alex's HMAC
  • As the key for the HMAC is in the message itself

• Only Facebook knows their HMAC key
  • And its the only information Facebook *needs* to retain in this protocol: Everything else can be discarded

• Bailey upon receipt checks that Alex's HMAC is correct
  • Otherwise Bailey's messenger silently rejects the message
    • Forces Alex's messenger to be honest about the HMAC, *even thought Facebook never verified it*

• Bailey trusts Facebook when Facebook says the message is from Alex
  • Bailey does *not verify* a signature, because there is no signature to verify…
    But the Signal protocol uses an ephemeral key agreement so that implicitly verifies Alex as well
Now To Report Abuse

{Abuse{
    M="Hey Bailey I'm going to say something abusive",
    k_{rand}},
    mac=HMAC(k_{rand}, M),
    to=Bailey,
    from=Alex,
    time=now,
    fbmac=HMAC(K_{fb},{mac, from, to, time})}
Facebook's Verification

• First verify that Bailey correctly reported the message sent
  • Verify $fbmac = \text{HMAC}(K_{fb}, \{\text{mac}, \text{from}, \text{to}, \text{time}\})$
    • Only Facebook can do this verification since they keep $K_{fb}$ secret
  • This enables Facebook to confirm that this is the message that it relayed from Alex to Bailey

• Then verify that Bailey didn't tamper with the message
  • Verify $mac = \text{HMAC}(k_{rand}, \{M, k_{rand}\})$

• Now Facebook knows this was sent from Alex to Bailey and can act accordingly
  • But Bailey can't prove that Alex sent this message to anyone other than Facebook
  • And Bailey can't tamper with the message because the HMAC is also a hash
Snake Oil Cryptography: Craptography

• "Snake Oil" refers to 19th century fraudulent "cures"
  • Promises to cure practically every ailment
  • Sold because there was no regulation and no way for the buyers to know
• The security field is practically **full** of Snake Oil Security and Snake Oil Cryptography
Anti-Snake Oil: NSA's CNSA cryptographic suite

• Successor to "Suite B"
  • Unclassified algorithms approved for Top Secret:
    • There is nothing higher than TS, you have "compartments" but those are access control modifiers
    • https://www.iad.gov/iad/programs/iad-initiatives/cnsa-suite.cfm
  • Symmetric key, AES: 256b keys
  • Hashing, SHA-384
  • RSA/Diffie Helman: >= 3072b keys
  • ECDHE/ECDSA: 384b keys over curve P-384

• In an ideal world, I'd only use those parameters,
  • But a lot of "strong" commercial is 128b AES, SHA-256, 2048b RSA/DH, 256b elliptic curves, plus the DJB curves and cyphers (ChaCha20)
  • NSA has a requirement where a Top Secret communication captured today should not be decryptable by an adversary 40 years from now!
Snake Oil Warning Signs...

- Amazingly long key lengths
  - The NSA is super paranoid, and even they don't use >256b keys for symmetric key or >4096b for RSA/DH public key
  - So if a system claims super long keys, be suspicious

- New algorithms and crazy protocols
  - There is no reason to use a novel block cipher, hash, public key algorithm, or protocol
    - Even a "post quantum" public key algorithm should not be used alone: Combine it with a conventional public key algorithm
  - Anyone who roles their own is asking for trouble!
  - EG, Telegram
    - "It's like someone who had never seen cake but heard it described tried to bake one. With thumbtacks and iron filings."  Matthew D Green
    - "Exactly! GLaDOS-cake encryption. Odd ingredients; strange recipe; probably not tasty; may explode oven. :)") Alyssa Rowan
Lots in the Cryptocurrency Space...

- The biggest being IOTA (aka IdiOTA), a “internet of Things” cryptocurrency...
  - That doesn’t use public key signatures, instead a hash based scheme that means you can never reuse a key...
    - And results in 10kB+ signatures! (Compared with RSA which is <450B, and those are big)
  - That has created their own hash function...
    - That was quickly broken!
  - That is supposed to end up distributed...
    - But relies entirely on their central authority
  - That uses trinary math!?!?
    - Somehow claiming it is going to be better, but you need entirely new processors...
Snake Oil Warning Signs...

- "One Time Pads"
  - One time pads are secure, if you actually have a true one time pad
  - But almost all the snake oil advertising it as a "one time pad" isn't!
  - Instead, they are invariably some wacky stream cypher

- Gobbledygook, new math, and "chaos"
  - Kinda obvious, but such things are never a good sign

- Rigged "cracking contests"
  - Usually "decrypt this message" with no context and no structure
    - Almost invariably a single or a few unknown plaintexts with nothing else
  - Again, Telegram, I'm looking at you here!
Unusability: No Public Keys

- The APCO Project 25 radio protocol
  - Supports encryption on each traffic group
    - But each traffic group uses a single *shared* key
- All fine and good if you set everything up at once...
  - You just load the same key into all the radios
  - But this totally fails in practice: what happens when you need to coordinate with someone who doesn't have the same keys?
- Made worse by bad user interface and users who think rekeying frequently is a good idea
  - If your crypto is good, you shouldn't need to change your crypto keys
- "Why (Special Agent) Johnny (Still) Can't Encrypt"
  - http://www.crypto.com/blog/p25
Unusability: PGP

• I *hate* Pretty Good Privacy
  • But not because of the cryptography...

• The PGP cryptography is decent...
  • Except it lacks "Forward Secrecy":
    If I can get someone's private key I can decrypt all their old messages

• The metadata is awful...
  • By default, PGP says who every message is from and to
    • It makes it much faster to decrypt
    • It is hard to hide metadata well, but its easy to do things better than what PGP does

• It is never transparent
  • Even with a "good" client like GPG-tools on the Mac
  • And I don't have a client on my cellphone
Unusability:
How do you find someone's PGP key?

• Go to their personal website?
• Check their personal email?
• Ask them to mail it to you
  • In an unencrypted channel?
• Check on the MIT keyserver?
  • And get the old key that was mistakenly uploaded and can never be removed?

Search results for 'nweaver icsi eduberkeley'

<table>
<thead>
<tr>
<th>Type</th>
<th>bits/keyID</th>
<th>Date</th>
<th>User ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>pub</td>
<td>4096R/8A46A420 2013-06-20 Nicholas Weaver <a href="mailto:nweaver@icsi.berkeley.edu">nweaver@icsi.berkeley.edu</a> Nicholas Weaver <a href="mailto:n_weaver@mac.com">n_weaver@mac.com</a> Nicholas Weaver <a href="mailto:nweaver@gmail.com">nweaver@gmail.com</a></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pub</td>
<td>2048R/442CP948 2013-06-20 Nicholas Weaver <a href="mailto:nweaver@icsi.berkeley.edu">nweaver@icsi.berkeley.edu</a></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Unusability: openssl libcrypto and libssl

• OpenSSL is a nightmare...
• A gazillion different little functions needed to do anything
• So much of a nightmare that I'm not going to bother learning it to teach you how bad it is
• This is why last semester's python-based project didn't give this raw
• But just to give you an idea: The command line OpenSSL utility options:
And On To Linked Lists Blockchains And Cryptocurrencies

- “Blockchain Technology”
  - A fancy word for “Append-Only Data Structure”
    - That causes people’s eyes to glaze over and them to throw money at people
  - “Private/Permissioned Blockchain”:
    - A setup where only one or a limited number of systems are authorized to append to the log
    - AKA 20 year old, well known techniques
  - “Public/Permissionless Blockchain”:
    - Anybody can participate as appenders so there is supposedly no central authority:
      Difficulty comes in removing “sibyls”

- Cryptocurrencies
  - Things that don’t actually work as currencies…
    More on Monday (which will be a ‘fun’ lecture and not covering stuff on the midterm, but will cover all the problems with public blockchains & cryptocurrencies)
Hash Chains

- If a data structure includes a hash of the previous block of data: This forms a “hash chain”
- So rather than the hash of a block validating just the block:
The inclusion of the previous block’s hash validates all the previous blocks
- This also makes it easy to add blocks to data structures
  - Only need to hash block + hash of previous block, rather than rehash everything:
    How you can efficiently hash an "append only" datastructure
- Now just validate the head (e.g. with signatures) and voila!
  - All a “blockchain” is is a renamed hashchain!
    Linked timestamping services used this structure and were proposed back in 1990!
Merkle Trees

- Lets say you have a lot of elements
  - And you want to add or modify elements
- And you want to make the hash of the set easy to update
- Enter hash trees/merkle trees
  - Elements 0, 1, 2, 3, 4, 5...
  - H(0), H(1), H(2)...
  - H(H(0) + H(1)), H(H(2)+H(3))...
  - The final hash is the root of the top of the tree.
- And so on until you get to the root
  - Allows you to add an element and update $\lg(n)$ hashes
    Rather than having to rehash all the data
  - Patented in 1979!!
A Trivial Private Blockchain…

- We have a single server $s$, with keys $K_{pub}$ and $K_{priv}$…
  - And a git archive $g$…

- Whenever we issue a pull request…
  - The server validates that the pull request meets the allowed criteria
  - Accepts the pull request
  - Signs the head…

- And that is it!
  - Git is an append only data structure, and by signing the new head, we have the server authenticating the **entire archive!**

- This is why “private” blockchain is **not** a revolution!!!
  - Anything that would benefit from an append-only, limited writer database already has one!