Engineering Secure Software

APPLIED CRYPTOGRAPHY

Networks, Crypto, & You.

- Most application developers…
  - Don't implement networking protocols
  - Don't implement encryption algorithms
- Knowing how to safely *deploy* them, however, is paramount
  - Different situations call for different techniques
    - Types of Authentication
    - One-way digests (hashes)
    - Symmetric-key vs. Public-key
    - Trusting public keys
  - Know Thy Algorithms instead of "just use crypto"

The Basic Problems

- The internet is a scary, scary place
  - Anyone can join
  - Anyone can sniff
- BUT! “distrust everything all the time” is not feasible
- Authentication: are you who you say your are?
  Trust must be built *somehow*
- Encryption: can someone else listen in?
- Authentication & Encryption overlap in techniques
  - How do we encrypt data for someone we do not trust?
  - How do we know nobody else has the key?
  - How do authenticate this *machine*?

Multi-Factor Authentication

- Security experts recommend that we utilize three types of authentication…
- Something you *know*
  - e.g. passwords
  - Can be guessed, though
- Something you *have*
  - Maybe a physical item
  - Maybe a one-time randomly generated key
  - e.g. both: pre-seeded secure PRNG key fob
- Something you *are*
  - Biometrics? Tons of false positives
  - Easier for humans, at least right now (e.g. face recognition)

Hash Digests

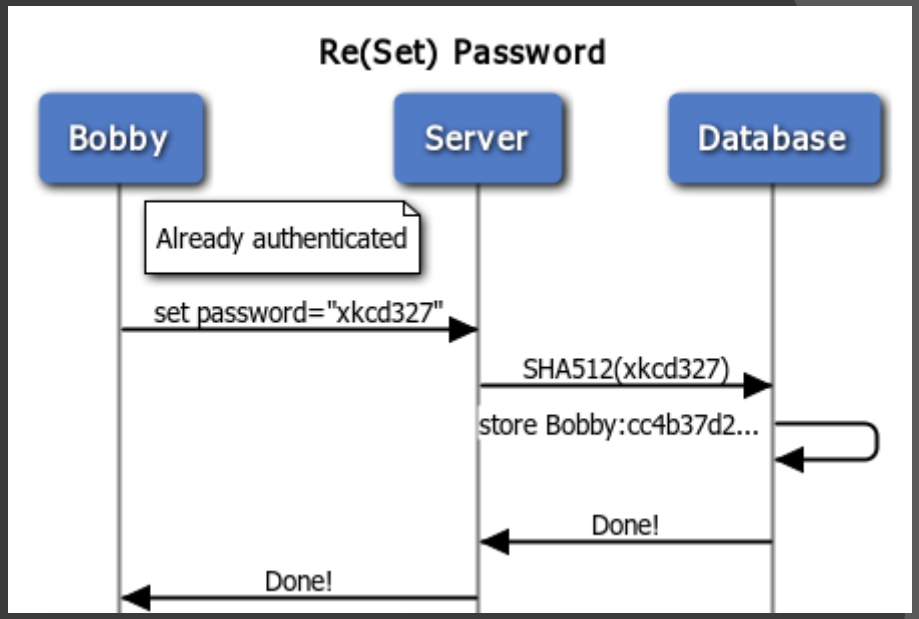
- Problem: sensitive data needs to be identified *only* by the original user, *nobody* else

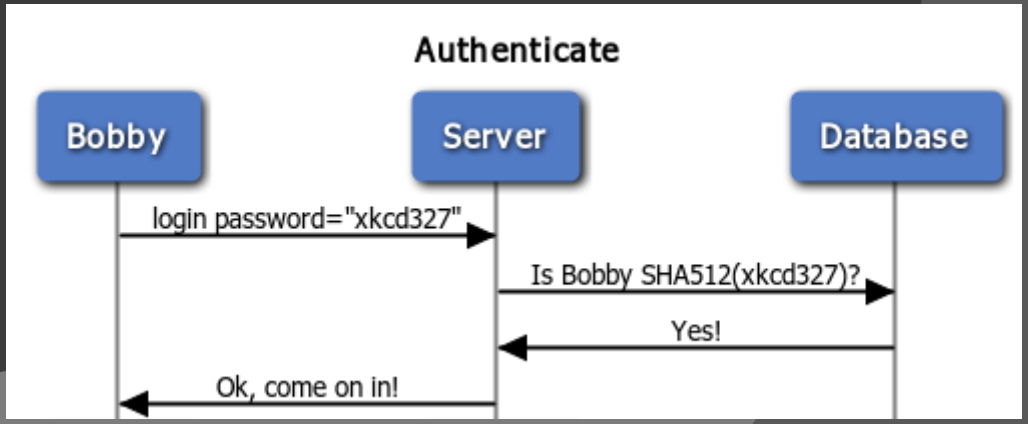
  e.g. user wants to authenticate, but we don't want to store passwords in plain text in case an attacker breaks in

- Solution: hash digest algorithms
  - Compute a very large number based on a chunk of data
    - The more numbers it can map to, the better (e.g. 2^{128})
    - Similar chunks of data should not compute to the same hash
  - Same number? ***Highly*** probable it's the same data

Authentication with Hashes

- Use Case: (re)set password
  - User inputs password
  - Server hashes pw
  - Stores the hash
- Abuse Case: Break-in
  - Attacker steals plaintext passwords from Database
  - Harm done: can authenticate as any user
  - Mitigation: can't reverse the hashes
- Use Case: Authenticate
  - User inputs password
  - Server computes hash
  - Checks the hashes





Abuse Case: Rainbow Tables

- What if an attacker steals the hashes?
  - common passwords + common digests = common hashes
  - Thus, attackers have large databases of pre-computed hashes called *rainbow tables*
- Solution: hashing with *salt*
  - Today's VotD

Hash Collisions

- By definition, hash digests cannot *uniquely* map data to a hash
  - Thus, many pieces of data map to the same hash
  - A collision is two known pieces of data that map to the same hash number
  - Can be used to “spoof” a password
- MD4, MD5 & SHA1 now considered “broken”
  - Colliding digests can be manufactured
  - http://www.phreedom.org/research/rogue-ca/
  - Still cannot be reversed, and probably won’t be

Symmetric-Key Cryptography

- Encrypt key == Decrypt key
  - So keep that key a secret!!
  - Traditional arrangement
- Modern algorithms
  - AES
  - Blowfish
  - 3DES
- Traditional usage
  - Encryption of data storage: backups, hard drives, etc.
  - Not typically for networking situations
    - Both parties need the same key
    - Can't send that key in the open over the wire
    - Could hard-code the keys ahead of time, but what if we need to change the key??

Public-Key Cryptography

- Encrypt key is *public,*
  Decrypt key is *private*
  - Anyone in the world can encrypt data and send it do you
  - But they can't decrypt any other messages sent to you
- Most popular modern algorithm: RSA
  - Factorization of two prime numbers
  - Public/private keys generated from computing two very large prime numbers
- RSA has never been cracked, although…
  - The algorithms for generating very large primes have been cracked/poorly implemented many times
  - Result of poor PRNG practices (bad algorithms & bad seeds)
- Traditional usage: networking (SSH, SSL, PGP)

Public-key Authentication

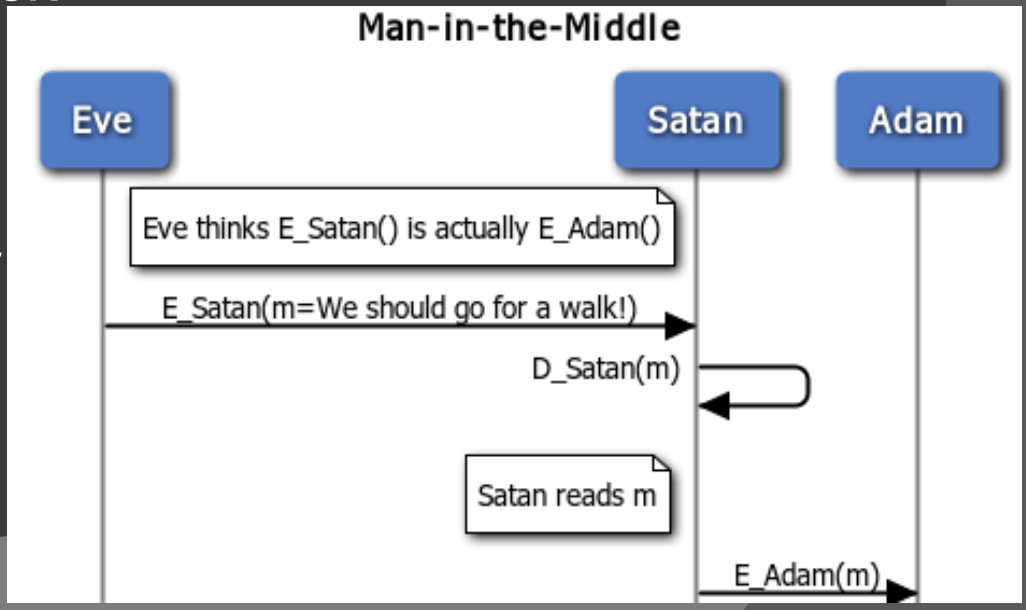
- E[..] is “encrypt” (public)
  D[..] is “decrypt” (private)
  - D[E[m]]=m is *encrypt then decrypt m (normal usage)*
  - D[m] is *use the decryption on plain text m (strange, but legal)*

- Scenario: Adam and Eve
  - Adam’s public and private: $E_{Adam}[..]$ and $D_{Adam}[..]$
  - Eve’s public and private: $E_{Eve}[..]$ and $D_{Eve}[..]$
  - They know each others’ public keys
    - Adam has access to $E_{Eve}[..]$
    - Eve has access to $E_{Adam}[..]$

- Adam wants to ensure Eve that the message came from him,
  - So he does: m=“This should be bubbles: D_{Adam}[bubbles]”
  - Sends E_{Eve}[m] to Eve – only Eve can read the message with D_{Eve} [m]
  - Eve checks $E_{Adam}[D_{Adam}$[bubbles]]=bubbles so that the message came from Adam, and not, say, her son Cain.

Drawbacks of Public Key

- Implementation issues
  - Tends to be slower than traditional symmetric key
  - Generating primes with 50+ digits is hard
- How do we trust the public key?
  - What if Eve confuses E_{Adam} with E_{Satan}?
  - Man-in-the-middle attack
    - Satan intercepts it
    - Decrypts it
    - Reads it
    - Re-encrypts it properly
    - Sends it off to Adam



SSH

- **S**ecure **Sh**ell
  - Used for remote access into machines
  - Ubiquitous for Unix-like systems
  - Uses passwords by default
- SSH and public keys
  - Key pairs have a one-time PRNG built in
  - Private key
    - Encrypted with a symmetric cipher
    - Requires a “passphrase” to unlock
  - Trust the public keys? `authorized_keys`
  - Trust the host? `known_hosts`

e.g. SSH Key pairs

```
me@client$ ssh-keygen –t rsa
Generating public/private rsa1 key pair...
Enter file in which to save the key (~/.ssh/identity):
Enter passphrase:
Enter same passphrase again:
Your public key has been saved in ~/.ssh/id_rsa.pub
Your private key has been saved in ~/.ssh/id_rsa
The key fingerprint is:
   22:bc:0b:fe:f5:06:1d:c0:05:ea:59:09:e3:07:8a:8c
```

- Untrusted public keys?
  - SSH-enabled servers don't trust any public keys initially
  - Need to copy your public key to the authorized_keys file on the server

```
me@client$ scp ~/id_rsa.pub me@server.edu:~
me@client$ ssh me@server.edu
me@server$ cat id_rsa.pub >> ~/.ssh/authorized_keys
```

SSH and known_hosts

- When I SSH into nitron, how do I know that this isn't a malicious server who changed his network address?
- Answer: your known_hosts file
  - Every server has a unique fingerprint
  - First time you sign in, trust the key and add the key to your known_hosts cache
  - Host changes? You'll get a warning like this:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@ WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED! @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-
the-middle attack)! It is also possible that the RSA host
key has just been changed. The fingerprint for the RSA key
sent by the remote host is
8b:ff:a1:b5:08:2f:8f:fd:2e:2f:67:80:9e:ba:8d:ff. Please
contact your system administrator. Add correct host key in
/home/bob/.ssh/known_hosts to get rid of this message.
Offending key in /home/bob/.ssh/known_hosts:2 RSA host key
for 192.168.1.100 has changed and you have requested strict
checking. Host key verification failed.
```

Next time...

- More common algorithms
  - Public key: SSL
  - Combination of Public and Symmetric: PGP
- Some common cryptanalysis techniques