Cryptography in Applications

Secure Tokens, HMAC, SSH, SSL/TLS, and other stuff.

Rule 1:

Security by Obfuscation is not security.
(Its just a well hidden bug)

Rule 2:

If you are inventing your own security protocol, your probably doing it wrong.

Part 1:

Symmetric Cryptography (Shared Secret)

THE GOAL:

To encrypt a message that can only be decrypted by its indented recipient.

The Solution:

Encrypted Communication with a shared secret.

_ An old technique used throughout history

_ Commonly Used Modern Algorithms:

AES
Blowfish
DES
Triple DES
Serpent
Twofish

The General idea is to have two functions:

```
cipherText = encrypt(key, plainText);

plaintext = decrypt(key, cipherText);
```

This is great, but Symmetric Key Algorithms by themselves do not guarantee the plaintext was not altered in transport.

... That's a big problem brah.**

**The Hawaiian equivalent of "Bro"-- Urban Dictionary.

The Solution: MAC**

** Message Authentication Code

Goals:

(1) Validate the message is from the right person

(2) Validate the message was not changed in transit

DONT FORGET

We are not trying to encrypt the message. We only care about the above two rules.

General idea is to use a 1-way hash function to generate a hashcode:

```
hashcode = hash(stuff-to-hash);
```

The intention is that hashcode never gets decoded back into plaintext... If it does, something broke

Commonly used Hash Functions

sha1 sha256 md5

Making a MAC

(sometimes also called a secure token)

token = hash(shared_secret + time + message);

(1) shared_secret:

guarantees only the sender and receiver can validate the token.

(2) time:

guarantees the hash will expire at the agreed upon time granularity

(3) message:

guarantees the message was unchanged during transport

But thats wrong...

With most hash functions, it is easy to append data to the message without knowing the key and obtain another valid MAC ("length-extension attack").

(http://en.wikipedia.org/wiki/Length_extension_attack)

Maybe it would be better to try:

token = hash(time + message + shared_secret);

But thats wrong too...

An attacker who can find a collision in the (unkeyed) hash function has a collision in the MAC (as two messages m1 and m2 yielding the same hash will provide the same start condition to the hash function before the appended key is hashed, hence the final hash will be the same).

Maybe it would be better to try:

```
token = hash(shared_secret + time + message + shared_secret);
```

But thats wrong too...

Some smart math folk figured out some stuff that I cannot understand.

The Solution: HMAC**

** Hash Based Message Authentication Code

HMAC provides a standard way to create secure tokens that cannot be tampered with.

This strategy basically boils down to:

shared_secret => ss1 , ss2;

hash(ss1 + hash(ss2 + message + time))

Putting it Together

HMAC + Encrypted Communication utilizing a shared secret

Together, Encrypted Communication + HMAC using a shared secret gives up the following functions:

ciphertext = **encrypt(** shared_secret, plaintext + HMAC **);**

plaintext + HMAC = **decrypt(** key, ciphertext **);**

isvalid = **validate(** key, paintext, HMAC **);**

Other Applications for HMAC validation:

_HTTP Request Validation
  Oauth Systems, API Access, ECommerce

_ SSL/TLS

Part 2:

Asymmetric Cryptography (public key cryptography)

Symmetric cryptography is great if the two parties are already friends.

...But what if the secure communication is their first contact?

First contact rules out a shared secret. Is it even possible to make such communication secure?

Invented right before pacman.

_ Even though we use this stuff every day, its worth mentioning that this idea is very new.

_ First invented by the british in 1973 (think James Bond) but remained classified until 1997.

_ Also invented at MIT in 1977 and published in a paper in 1978. The algorithm became known as RSA in honor of the first letters of each name of the MIT mathematician.

Now for some math.

Every Pardot employee knows prime numbers are a big deal:

(https://gist.github.com/2267570)

Q1. What is the prime factorization of 20711?

A1. [139,149]

Q2. What is the 139 * 149?

A2. 20711

Q3. Which is easy Q1 or Q2?

A3. Q2

Let's apply some math

(1) Use 20711 as the basis for a public key.

(2) Use 139 & 149 as the basis for the private key

Disclaimer: THIS IS A GROSS AND POSSIBLY IRRESPONSIBLE OVERSIMPLIFICATION OF THE MATH INVOLVED

The goals of public key cryptography

(1) Communication - To guarantee that the recipient is the only one that will be able to understand the message.

Note: This means that neither the public nor the sender will be able to decode the message. Only the recipient can decode the message.

(2) Validation - To guarantee that a particular message did indeed come from the intended sender.

Public key cryptography gives us:

2 Communication Functions

```
ciphertext = encrypt( receivers _public_key, plaintext );

plaintext = decrypt( receivers_private_key, ciphertext );
```

Public key cryptography gives us:

2 Validation Functions

```
MAC = sign( senders_private_key, message );

isValid = verify( senders_public_key, MAC );
```

But, much like everything in this world, this is broken too.

How do I know that your public key is *really* your public key?

For SSH this is not such a big deal:

_ **Upload your public key to target machines**

(~/.ssh/authorized_keys)

_ **Keep a list of trusted server's public keys**

(~/.ssh/known_hosts)

..But for SSL, it's a big problem.

Solution:

Certificate Authorities

SSL Certificate Authority:

_ CA Issues a certificate to the website owner

_ CA Issues root authority certificates to browsers.

_ Browsers download the cert on an ssl website and verify that it validates with one of its root authority certificates. If so, it trusts that the associated public key is valid.

_ The solution is not ideal because it puts too much trust in the root authorities. This is a known open issue with SSL.

SSL/TLS Request Walkthrough:

(1) Client/Server agree on SSL version and ciphers.

(2) Server sends certificate and client authenticates it with a Root Cert Authority

(3) Client creates a “Pre-Master Secret” and sends it to the server, encrypting it with the server’s public key

(4) Client/Server each generate a “Master Secret” from the “Pre-Master Secret”

(5) Client/Server use the “Master Secret” to generate symmetric keys called “Session Keys”

(6) Client tells server all communication will now use Symmetric Keys.

(7) Server tells client all communication will now use Symmetric Keys.

(8) Handshake Complete. Its all symmetric communication for the remaining duration of the connection. The symmetric communication includes both:

Encrypted Communication AND HMAC Verification (TLS)

Questions?