Insecure Implementation of Security Practices

Karmendra Kohli

Director, Co-founder

SecurEyes

Infusing Security

Agenda

- Talk about insecure implementation of security practices for
  - Salted Hashing
  - CAPTCHA
  - Browser Caching

Why do we need authentication

- Abstraction of Lock and Key
  - Humans are used to and comfortable with the concept
- Explicit Authentication happens only once
  - Implicit authentication with each request

Threats to passwords

- In the Host
  - Can be found in Memory or Browser History
- Network
  - Can be sniffed or found in proxy logs
- Database
  - If stored in clear-text, compromise of DB leads to complete compromise
  - Backup tapes etc

Securing Credentials

- Denote password by ******** in the form field
  - Can be guessed
- Implement strong password policy
  - Can be stolen
- Base64 Encoding
  - Can be decoded
- Encrypting the password
  - Can be decrypted
- Hashing the password
  - Our story begins from here…

Hashing

- Cryptographic Hash Function is a transformation that takes an input and returns a fixed-size string, which is called the hash value. – From Wikipedia
- One way hashes cannot be reversed, they can only be compared
  - Just like finger prints
- A strong one-way hash will have minute probability of collision

Use of hashing

- Authentication
- Message Integrity
- Digital Signatures

Is just a hash secure?

- Hash can be stolen from the
  - memory
  - browser history
  - network by sniffing
  - proxy logs
- We can replay the hash instead of the clear text password
  - Does not need to be reversed for authentication

Hash with a pinch of salt

- Salted hash
  - Aims at implementing a one time password scheme
  - Each time the password of the user transmitted from the client is different
  - Even if stolen cannot be reused/replayed

  Well That Is The Aim!!

The Working Steps

1. With each request to the login page by the user, the server generates a random number, the salt, and sends it to the client along with the login page.
2. A JavaScript code on the client computes the hash of the password entered by the user.
3. The salt is concatenated with the computed hash of the password to re-compute the hash of the concatenated string (let this salted hash be ‘A’).
4. The result i.e. ‘A’ generated in the previous step is sent to the server along with the username.

The Working Steps

5. On the server side - the application retrieves the hashed password for the corresponding username from the database.
6. The hashed password fetched from the database is concatenated with the salt (which was earlier sent to the client) stored at the server. Hash of the concatenated string is calculated at the server (let this salted hash be B).
7. If the user entered the correct password then these two hashes (A & B) should match. The server compares these two hashes and if they match, the user is authenticated.
8. The salt is then deleted from the server side

Insecure Implementations

1. Generating salt on client side

```
function sendhash()
{
    retFlag = false;
    var hashrand=TestRand();
    hash = hex_sampling(document.login.Password.value);
    hash = hex_sampling(hash+hashrand);
    document.login.Password.value = hash;
    document.login.UserId.value = userid;
    document.login.salt.value = hashrand;
    retFlag = true;
    return retFlag;
)
```

Generating salt on client side

```
POST /auth/jsp/UserValidate.jsp HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-powerpoint, application/vnd.ms-excel,
application/msword, application/x-shockwave-flash, */*
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Proxy-Connection: Keep-Alive
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)
Host: 192.168.12.73:9954
Pragma: no-cache
Content-Length: 138

salt=04156206651622044345505453456421&Password=e2e1f726939332a00b4b9d78293c3e3f&UserId=Anant1950
```

Generating salt on client side

- So we have all values that server needs to calculate the salted hash
- It is enough to replay the values
- Beats goal of using salts

2. Using a limited set of salts

- The number of salts being used by the server is fixed
- Above replay technique works
  - We just need to try multiple number of times

3. Using same salt for same user

- In this case the salted hash was stored by the developer in the DB
  - Gives a ‘feeling’ of more security
- If salted hash is stored as password, then salt too needs to be stored ☺
- Hence same salt is sent each time - replay

4. Not re-initializing the salt for login pages

- Multiple logins from the same browser resulted in same salt being used
- Salt not deleted from the server side on successful or unsuccessful authentication
- Problem is also related to not re-initializing the session after successful or unsuccessful authentication
- Attack window is small, but still…

5. Transmitting clear text password along with salted hash

- Many implementations observed
- The salted hash is sent along with clear text form input fields
- The salted hash is computed using different set of hidden variables

Transmitting clear text along with salted hash

```
function sendhash()
{
    retFlag = false;
    hash = hex_sampling(document.login.Password1.value);
    hash = hex_sampling(hash+hashrand);
    document.login.Password.value = hash;
    document.login.UserId.value = userid;
    document.login.salt.value = hashrand;
    retFlag = true;
    return retFlag;
}
```

6. Not re-initializing the java script variables

- Uname = Document.form1.textbox1.value
- Pwd = Document.form1.textbox2.value
- Use the and ‘pwd’ variable for salted hash calculation
  - Hash = Hfx (pwd)
  - Hash = Hfx (Hash + salt)
- The uname and pwd can be found as clear text in memory

Not re-initializing the java script variables

```
function sendhash()
{
     retFlag = false;
     //added by santy 05/09/2008
     var hashrand=TestRand();
     var userid = document.login.UserId1.value;
     var pwd = document.login.Password1.value;
     hash = hex_sampling(pwd);
     hash = hex_sampling(hash+hashrand);
     document.login.Password.value = hash;
     document.login.UserId.value = userid;
     document.login.salt.value = hashrand;
     document.login.UserId1.value ='';
     document.login.Password1.value ='';
     retFlag = true;
     return retFlag;
)
```

7. Storing password in clear text in database

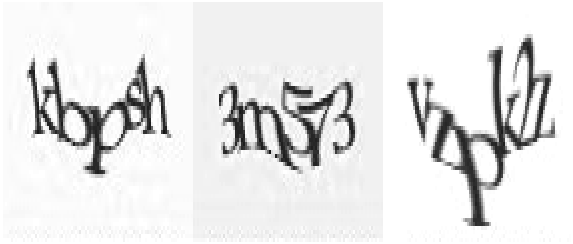
- Hash = hash (password + salt)
- Server does not store salt in the database
- Clearly implies that server also does not store hash of password in database

Salted Hash Best Practices

- Salt
  - Generated on server side
  - Must be highly random. Must use cryptographically strong PRNG algorithm
  - Must never be stored in the database
  - Must be re-generated after each successful /unsuccessful attempt
  - All temporary Form Field or JS variables must be set to null before submitting the request
- Passwords must be stored in hashed format in the database

CAPTCHA

- Completely Automated Public Turing Tests to Tell Computers and Humans Apart



- Aim is to introduce an entity that requires human intervention
- Protection against DOS or Brute force or Misuse
  - Avoiding automating submissions

Typical uses

- Public Forms
  - Registration
  - Feedback / Rating
  - Survey
  - Blog post
  - Appointment/Slot booking

CAPTCHA - Implementation

- Initiate a session for the page rendered
- Request for image
- Generate unique string randomly on server side
- Generate image with string embedded
- Remember the string in Session Object and send CAPTCHA to client
- For each client request, validate the sent value with the one in the session object
- Invalidate session after each attempt
  - Both successful and unsuccessful
- Generate new CAPTCHA for each new page request

Insecure Implementation

- We are not focusing weakness of CAPTCHA generation
  - It's a different ball game
- We will discuss common implementation flaws of using CAPTCHA's

1. Verifying CAPTCHA on client side

- The client side JavaScript verifies the CAPTCHA value
- Server trusts the decision of client

  strPassFlag=True;
- Completely bypass enforcement
- Automate submission

2. Having a limited set words

- Small fixed pool of CAPTCHA words
- Automated tools will have high probability of hits
- Bypass goal of implementing CAPTCHA

What better than a COMBO

```
function chk_text(strPicture)

(
        if (strPicture.value!="AN7df5")
        (
                if (strPicture.value!="Xd7Gf5")
                (
                        if (strPicture.value!="AY7Tf5")
                        (
                                if (strPicture.value!="PAN45T")
                                (
                                        alert(" Please Provide Value As Seen in the picture");
                                        return false;
                                }
                                else
                                (
                                        return true;
                                )
                        else
                        (
                                return true;
                        )
                else
                (
                        return true;
                )
        else
        (
                return true;
        )
)
```

Limited pool of CAPTCHA values with client side verification

3. Validate CAPTCHA on server but value sent from client

- Actual value of CAPTCHA sent as hidden variable
  - Hmmmmm, “hidden” does not mean “secure”
- Replay request with value of our choice

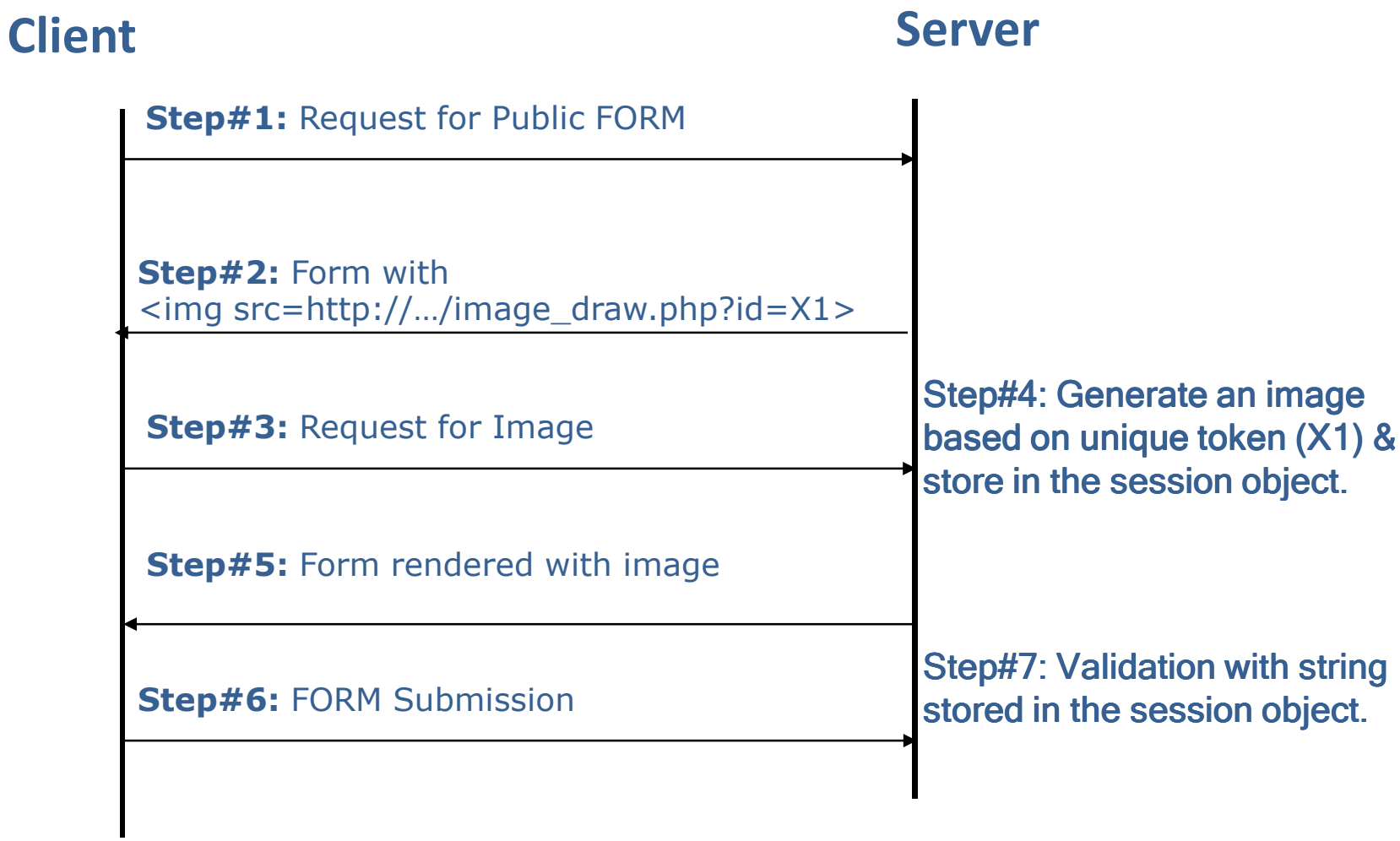
4. Session is not re-initialized

- Session not re-initialized after form submission
- Refresh on submitted page works
- Automate replay of requests using assigned session ID
- Multiple registrations
  - Treasure trove for spammers

5. Image ID can be replayed

- The request for image is of the form
- <img src = http://www.testserver.com/image_draw.jsp?ID=23erdfret45ffrsd4g3fd3sa3a4d5a
- The image_draw.jsp decrypts the value to get the word
- For the same value the same word is rendered ALWAYS
- Replay image request

Image ID can be replayed



- Step 2 can be replayed

6. CAPTCHA is not an image

- One application had a ‘feeling’ of an image
- Java Script used for rendering the text on the page in a “different” font
- Tools can read such text easily

Text can be selected ☺

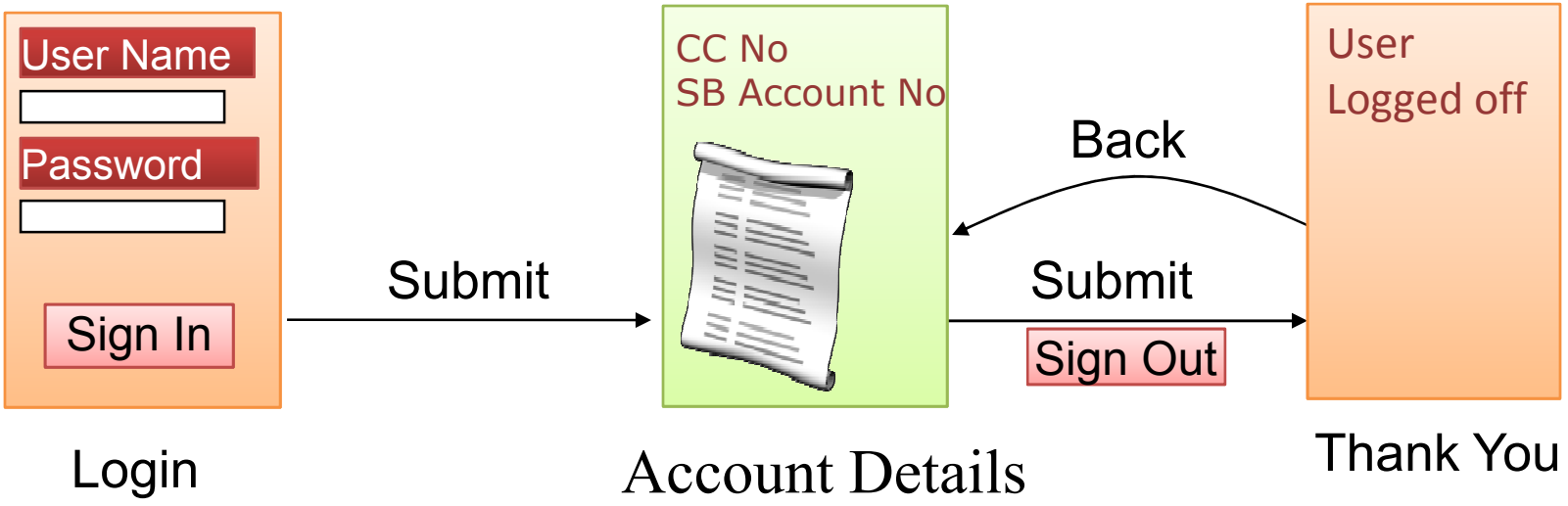
CAPTCHA implementation best practices

CAPTCHA:

- Generated on server side
- Verified on server side
- Strings must be randomly generated
- Images must not be generated based on token from ‘hidden’ client side
- Must be bound to a session
- Must be re-generated after each successful /unsuccessful attempt

Caching Issues

- Unauthorized access to information



Browser Cache

- A local store
  - Aims to improve performance
  - Renders pages locally, fast user experience
  - Support across browsers
- Controlled by a lot many directives
  - Pragma: no-cache
  - Cache-control: no-cache
  - Expires: Mon, 03 Jul 2000 9:30:41 GMT

Caching

- Relevant HTTP fields
  - Request
    - If-Modified-Since
    - Pragma: no-cache
  - Response
    - Last-Modified
    - Expires: Mon, 03 Apr 2009 9:30:41 GMT
    - Cache-Control: no-cache, no-store

Caching

- Relevant HTTP fields
  - Request
    - If-Modified-Since – Recheck for fresh content
    - Pragma: no-cache – HTTP 1.0
  - Response
    - Last-Modified – Confirm fresh content
    - Expires: …… – HTTP 1.0
    - Cache-Control: no-cache – Pages will not be displayed, but will still be stored
    - Cache-Control: no-store – Pages will not be stored

Caching best practice

- For critical applications
  - Cache-Control: no-cache and Cache-Control: no-store must be set for ALL pages

Example Implementation flaw

- Banking application
  - Ideally all pages must be fetched fresh
  - Clicking on ‘back’ must not render page
  - All pages set with Cache-Control: no-cache, no-store
    - excepting logout page
  - On logout, an intermediate proxy used to serve logout page
  - Session was not invalidated as request did not reach server

Thank you

karmendra.kohli@secureyes.net