References


Before Security some Information for the assignment
Basic HTTP Authentication

To view this page, you need to log in to area “Access for /Campers” on www.scandiacampmendocino.org:80.
Your password will be sent in the clear.

Name: 
Password: 

☐ Remember this password in my keychain

Cancel    Log In
Sample Interaction

Client Request
GET /private/index.html HTTP/1.0
Host: localhost

Server Response
HTTP/1.0 401 Authorization Required
Server: HTTPd/1.0
Date: Sat, 27 Nov 2004 10:18:15 GMT
WWW-Authenticate: Basic realm="Secure Area"
Content-Type: text/html
Content-Length: 311

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/1999/REC-html401-19991224/loose.dtd">
<html>
<head>
<title>Error</title>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
</head>
<body>
<h1>401 Unauthorised.</h1>
</body>
</html>

Example from http://en.wikipedia.org/wiki/Basic_access_authentication
Sample Interaction

Client request (user name "Aladdin", password "open sesame")

GET /private/index.html HTTP/1.0
Host: localhost
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==

Server response:

HTTP/1.0 200 OK
Server: HTTPd/1.0
Date: Sat, 27 Nov 2004 10:19:07 GMT
Content-Type: text/html
Content-Length: 10476
Base64 Encoding

Encodes any byte sequence into sequence of printable characters

Encoded sequence can be decoded

Used to encode MIME contents for transport
  Email Attachments
Base 64 Algorithm

Divide input into parts each part 24 bits long (3 bytes)

Convert each 24 bit sequence as follows:

Divide the 24 bits into four groups of 6 bits

Use the table to convert each 6 bits

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>25</td>
<td>Z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>a</td>
</tr>
<tr>
<td>27</td>
<td>b</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>51</td>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>0</td>
</tr>
<tr>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>61</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>62</td>
<td>+</td>
</tr>
<tr>
<td>63</td>
<td>/</td>
</tr>
</tbody>
</table>

pad with =
Example

cats text

001111111 00111101 01001010 01001001 binary

001111 111001 111010 100101 001001 001 6 bit groups

001111 111001 111010 100101 001001 001000 6 bit groups padded

15 57 58 37 9 8 As decimal

P 5 6 I J I = = Converted
Base64 Encoding & HTTP Authentication

Use Base64 encoding for user name and password

user name "Aladdin"
password "open sesame"

Aladdin:open sesame

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
Security
Top 25 Programming Security Errors

Selected based on:

How common the error is

Consequences of error

2009 CWE/SANS Top 25 Most Dangerous Programming Errors
March 10, 2009
http://cwe.mitre.org/top25/
Top 25: Insecure Interaction Between Components

- Improper Input Validation
- Improper Encoding or Escaping of Output
- Failure to Preserve SQL Query Structure (aka 'SQL Injection')
- Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')
- Failure to Preserve OS Command Structure (aka 'OS Command Injection')
- Cleartext Transmission of Sensitive Information
- Cross-Site Request Forgery (CSRF)
- Race Condition
- Error Message Information Leak
SQL Injection

"SELECT * FROM users WHERE name = "' + userName + '";"

let username be
  a’ or ‘t’ = ‘t

SELECT * FROM users WHERE name = 'a' or 't'='t';

which is the same as

SELECT * FROM users;

SQL Injection

let username be
  a'; DROP TABLE users; Select * FROM data where name = 'a

"SELECT * FROM users WHERE name = "' + userName + "';" becomes:

SELECT * FROM users WHERE name = 'a' ;
DROP TABLE users;
Select * FROM data where name = 'a';
Preventing SQL Injection In Java

Replace

Connection con = (acquire Connection);
Statement stmt = con.createStatement();
ResultSet rset =
    stmt.executeQuery("SELECT * FROM users WHERE name = "' +
    userName + ";");

with

Connection con = (acquire Connection)
PreparedStatement pstmt =
    con.prepareStatement("SELECT * FROM users WHERE name = ?");
pstmt.setString(1, userName);
ResultSet rset = pstmt.executeQuery();

Wikipedia claims this is safe. I have not verified the claim.
OS Command Injection

Java Example
Running any Program

String script = System.getProperty("SCRIPTNAME");
if (script != null) System.exec(script);
OS Command Injection

Perl

Running a specific command

Code

use CGI qw(:standard);
$name = param('name');
$nslookup = "/path/to/nslookup";
print header;
if (open($fh, "$nslookup $name|")) {
    while (<$fh>) {
        print escapeHTML($_);
        print "<br>\n";
    }
    close($fh);
}

Input

cwe.mitre.org%20%3B%20/bin/ls%20-l

Decodes to

/path/to/nslookup cwe.mitre.org ; /bin/ls -l

So return list of current directory

Example From http://cwe.mitre.org/data/definitions/78.html
Top 25: Risky Resource Management

Failure to Constrain Operations within the Bounds of a Memory Buffer
External Control of Critical State Data
External Control of File Name or Path
Untrusted Search Path
Failure to Control Generation of Code (aka 'Code Injection')
Download of Code Without Integrity Check
Improper Resource Shutdown or Release
Improper Initialization
Incorrect Calculation
Buffer Overflow

A = 3 byte string
B = integer

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Write 'cate' in A

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>e</td>
</tr>
<tr>
<td>e</td>
<td>3</td>
</tr>
</tbody>
</table>
What to use Buffer Overflow for

- Overwrite local variable to change program's behavior
- Overwrite the return address in a stack frame
- Overwrite a function pointer or exception handler
Buffer Overflow Solution 1

Check the Buffer Size

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
    char buffer[10];
    if (argc < 2)
    {
        fprintf(stderr, "USAGE: %s string\n", argv[0]);
        return 1;
    }
    strncpy(buffer, argv[1], sizeof(buffer));
    buffer[sizeof(buffer) - 1] = '\0';  /* explicitly write a string terminator */
    return 0;
}
```
Buffer Overflow Solution 2

Use a language that checks for array out-of-bounds errors
Java
Smalltalk
Ruby
Python
etc.
Top 25: Porous Defenses

- Improper Access Control (Authorization)
- Use of a Broken or Risky Cryptographic Algorithm
- Hard-Coded Password
- Insecure Permission Assignment for Critical Resource
- Use of Insufficiently Random Values
- Execution with Unnecessary Privileges
- Client-Side Enforcement of Server-Side Security
Hard-Coded Password

```java
DriverManager.getConnection(url, "admin", "secret");
```

Decompile byte code to see strings

```
javap -c ConnMngr.class
22: ldc #36; //String jdbc:mysql://ixne.com/rxsql
24: ldc #38; //String admin
26: ldc #17; //String secret
```

Example from http://cwe.mitre.org/data/definitions/259.html
Hard-Coded Password - Some Solutions

Store passwords outside of the code in a strongly-protected, encrypted configuration file or database

For inbound authentication apply strong one-way hashes to your passwords
Client-Side Enforcement

Server-side

```perl
$sock = acceptSocket(1234);
($cmd, $args) = ParseClientRequest($sock);
if ($cmd eq "AUTH") {
    ($username, $pass) = split(\s+, $args, 2);
    $result = AuthenticateUser($username, $pass);
    writeSocket($sock, "$result\n");
    # does not close the socket on failure; assumes the
    # user will try again
}
elsif ($cmd eq "CHANGE-ADDRESS") {
    if (validateAddress($args)) {
        $res = UpdateDatabaseRecord($username, "address", $args);
        writeSocket($sock, "SUCCESS\n");
    }
    else {
        writeSocket($sock, "FAILURE -- address is malformed\n");
    }
}
```

Example from http://cwe.mitre.org/data/definitions/602.html
Client Side

$client = "server.example.com";
$username = AskForUserName();
$password = AskForPassword();
$address = AskForAddress();
$sock = OpenSocket($server, 1234);
writeSocket($sock, "AUTH $username $password
";
$resp = readSocket($sock);
if ($resp eq "success") {
    # username/pass is valid, go ahead and update the info!
    writeSocket($sock, "CHANGE-ADDRESS $username $address
";
}
else {
    print "ERROR: Invalid Authentication!\n";
}

Here the client checks, but the server does not. As long as people only use the above client we are safe. While honest people will restrict them selves to the above client, hackers trying to break into your system will not.
Client-Side Enforcement - Solutions

Server repeats any security check done by client

Client side security checks help

Detect an attack

Provide helpful feedback to the user about the expectations for valid input
Some Problems Require Global Solution

Denial of Service Attacks
Cryptography
Security ≠ Cryptography

Kevin Mitnick often got people’s passwords by asking
Cryptographic Examples

Alice and Bob communicate with each other in secret

Eve wants to see Alice's and Bob's communication
One-Way Hash Functions

Let $M$ be a message (sequence of bytes)

A one-way hash function $f()$ such that:

- $f$ maps arrays of bytes to arrays of bytes
- $f(M)$ is always the same length
- Given an $M$ it is easy to compute $f(M)$
- Given $f(M)$ it is hard to compute $M$
- Given $M$ it is hard to find $N$ such that $f(M) = f(N)$

MD5 - Message Digest 5
SHA - Secure Hash Algorithm
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class OneWay
{
    public static void main(String args[])
            throws NoSuchAlgorithmException
    {
        MessageDigest sha = MessageDigest.getInstance("SHA");
        sha.update("Hi mom".getBytes());
        byte[] shaHash = sha.digest();
        System.out.println(new String(shaHash));

        MessageDigest md5 = MessageDigest.getInstance("MD5");
        md5.update("Hi mom".getBytes());
        byte[] md5Hash = md5.digest();
        System.out.println(new String(md5Hash));
    }
}
Authentication with One-Way Functions

Given
- list of passwords $P_1, P_1, \ldots, P_k$
- One-way function $f()$

In password file only store $f(P_k)$

Password File

<table>
<thead>
<tr>
<th>Name1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name2</td>
<td></td>
</tr>
<tr>
<td>Name3</td>
<td></td>
</tr>
</tbody>
</table>
Authentication with One-Way Functions

Host stores only one-way functions of passwords

Alice sends host her password

Host performs one-way function on password

Host compares result with stored value for Alice

Access to password file does not help Eve in getting Alice's password
Dictionary Attacks

Eve compiles list of N common passwords

Eve applies one-way function to all N common passwords f(P)

Eve stores all results as map f(P) -> P

For the N common passwords Eve has the inverse function

If

N = 1,000,000
f() = MD5
Passwords are about 8 bytes

Eve's map is ~ 24MB
Salt

Salt - random string

Given
  list of passwords P1, P1, ..., Pk,
  One-way function f()
  Salt S

In password file only store f(Pk + S)

<table>
<thead>
<tr>
<th>Name</th>
<th>f(Pk + S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name1</td>
<td>f(P1 + S)</td>
</tr>
<tr>
<td>Name2</td>
<td>f(P2 + S)</td>
</tr>
<tr>
<td>Name3</td>
<td>f(P3 + S)</td>
</tr>
</tbody>
</table>
Dictionary Attacks and Salt

Eve compiles list of $N$ common passwords

Eve has to compile $f(P + S)$ for each password & possible salt

Greatly increases size of Eve's reverse map file

Daniel Klein password-guessing often cracks 40% of salted passwords
SKEY Authentication

Given a one-way function $f()$

Alice enters a random number $R$

Computer computes

\[
\begin{align*}
x_1 &= f(R) \\
x_2 &= f(f(R)) = f(x_1) \\
x_3 &= f(f(f(R))) = f(x_2) \\
&\quad \ldots \\
x_{100} &= f(x_{99})
\end{align*}
\]

Alice remembers $x_1, \ldots, x_{100}$

Computer stores $x_{101}$

To log on to computer Alice enters $x_{100}$

Computer computes $f(x_{100})$ and compares to $x_{101}$

Computer stores $x_{100}$
Encryption

Traffic on a network can be sniffed

A solution is encryption of all traffic

This can be done at any layer of the protocol stack

Two basic types of encryption

Symmetric encryption
One key both encrypts and decrypts

Public/Private key encryption
One key encrypts, another decrypts
Symmetric encryption

Work fine except that both parties have to share the same key

Distributing the shared key is as hard as distributing secrets
Public/Private Key Encryption

Algorithm uses two keys

Can use any key to encrypt message

The other key will decrypt the message

The encrypt key cannot be used to decrypt message

One key is made public

Other key is kept private
Public/Private Key Encryption

Require a way to distribute public keys in open

Algorithms tend to be slow

Often used to distribute shared key for Symmetric Encryption algorithm

Common Algorithms

RSA (Rivist, Shamir, Adleman)
DSA (Digital Signature Algorithm)
Public/Private key Uses - Digital Signature

Alice has a message she wants to sign

Alice uses her private key to encrypt the message

Anyone can decrypt the message using Alice's public key

If Alice's public key decrypts the message the Alice had to encrypt it

So

We know the message came from Alice

The message was not altered

Alice cannot deny sending the message
Public/Private key Uses - Secret Messages

Alice wishes to send a message to Bob

Alice encrypts the message using Bob's public key

Bob can use his private key to decrypt the message

Eve cannot decrypt the message
RSA

Public Key

Key contains n & e

n = p*q, p & q are primes
e relatively prime to (p-1)*(q-1)

p & q must be kept secret

Private Key

Key contains d & N

d = e^{-1} \mod ((p-1)*(q-1))

that is (d*e) \mod ((p-1)*(q-1)) = 1

Encrypting

Let m be a message such that m < n

Let the encrypted message, c be

\[ c = m^e \mod n \]

Decrypting

\[ m = c^d \mod n \]
RSA Example

Alice’s Keys

Let

\[ p = 47 \]
\[ q = 71 \]
\[ e = 79 \]

Then

\[ n = p \times q = 3337 \]
\[ d = 79^{-1} \mod 3220 = 1019 \]

Public key

\[ n = 3337 \]
\[ e = 79 \]

Private key

\[ d = 1019 \]
\[ n = 3337 \]
Sending Message to Alice with RSA

Let the message, m, be 42

Compute the encrypted message

\[ c = m^e \mod n \]
\[ = 42^{79} \mod 3337 \]
\[ = 2973 \]

Let the message, m, be 42

Decrypting the message

\[ c^d \mod n = 2973^{1019} \mod 3337 \]
\[ = 42 \]
Why Public/Private Key Systems are slow

2973^{1019} contains 3,540 digits
Alice Signing a Document with RSA

Let the message, m, be 42

Encrypt the message using the Private Key

c = m^e \mod n
  = 42^{1019} \mod 3337
  = 2151

Decrypting the message using the Public key

c^d \mod n = 2151^{79} \mod 3337
  = 42

Public key
n = 3337
e = 79

Private key
d = 1019
n = 3337