Chapter 2: Web-Based Attacks – Part 2

Gonzalo De La Torre Parra, Ph.D.

Fall 2021

SQL Manipulation

- It means to manipulate and retrieve data in a relational database.
- SQL Manipulation comprises the SQL–Data change statements, which modify the stored data but not the schema or database objects.

Code Injection

- Code injection is the exploitation of computer application that is caused by processing invalid data.
- It is always used malevolently which means it is always used in an evil way to destroy a database by exploiting the other codes.

Functional Injection

- It is one of the most common type of injection technique where functions are used for injection.
- When a function call a parameter then the attacker passes a different parameter to the function resulting something different than expected.

Buffer Overflow

- It also one of the common technique used for injection at the users input side.
- It is a mechanism of injection by input of data exceeding the limits of the fields of the user input resulting an error message using which the SQL codes are injected.

SQL Injection Steps

- Input field to submit data (e.g. a login page)

SQL Injection Steps

- Check for server pages if input field is absent

  e.g. http://www.xsecurity.com/index.jsp?id=10

- In the above example attack will be like this:

  e.g. http://www.xsecurity.com/index.jsp?id=debu' or 1=1 –

- Look for errors: This can be done using single quotation mark (‘). E.g.

Microsoft OLE DB Provider for ODBC Drivers
error '80040e14'

[Microsoft][ODBC Microsoft Access Driver] Extra)
in query expression 'Userid='3306') or ('a'='a'
AND Password='".

Test for Vulnerability

Using single quote in the input

- sujit' or 1=1 --
- login: shweta' or 1=1 --
- http://search/index.asp?id=sql' or 1=1 --

Depending on the error:

- ' or 1=1 --
- " or 1=1 --
- ' or 'a' = 'a
- " or "a" = "a
- ') or ('a' = 'a)

How it Works

Original SQL Query:

```
• strQry = "SELECT Count(*) FROM Users WHERE UserName='" + txtUser.Text + "'
  AND Password='" + txtPassword.Text + "'";
```

In the case of the user entering a valid user name of "Paul" and a password of "password", strQry becomes:

```
• SELECT Count(*) FROM Users WHERE UserName='Paul' AND Password='password'
```

But when the hacker enters ' Or 1=1 -- the query now becomes:

```
• SELECT Count(*) FROM Users WHERE UserName='' Or 1=1 --' AND Password=''
```

Because a pair of hyphens designates the beginning of a comment in SQL, the query becomes simply:

```
• SELECT Count(*) FROM Users WHERE UserName='' Or 1=1
```

Examples: BadLogins.aspx.cs

```
private void cmdLogin_Click(object sender, System.EventArgs e) {
    string strCnx =
     "server=localhost;database=northwind;uid=sa;pwd=;";
     SqlConnection cnx = new SqlConnection(strCnx);
    cnx.Open();

    //This code is susceptible to SQL injection attacks.
    string strQry = "SELECT Count(*) FROM Users WHERE UserName='" +
     txtUser.Text + "' AND Password='" + txtPassword.Text + "'";
    int intRecs;

    SqlCommand cmd = new SqlCommand(strQry, cnx);
    intRecs = (int) cmd.ExecuteScalar();

    if (intRecs>0) {
        FormsAuthentication.RedirectFromLoginPage(txtUser.Text, false);
    }
    else {
        lblMsg.Text = "Login attempt failed.";
    }
    cnx.Close();
}
```

Attack Occurs Here

User-Id : srinivas

Password : mypassword

```
select * from Users where user_id= ' srinivas '
                and password = ' mypassword '
```

User-Id : ` OR 1= 1; /*

Password : */--

```
select * from Users where user_id= '` OR 1 = 1; /* '
                and password = ' */-- '
```

More on Types

1)First Order Attack:-The attacker can simply enter a malicious string and cause the modified code to be executed immediately.

2)Second Order Attack:-The attacker injects into persistent storage (such as a table row) which is deemed as a trusted source. An attack is subsequently executed by another activity.

3)Lateral Injection:-The attacker can manipulate the implicit functionTo_Char() by changing the values of the environment
variables, NLS_Date_Format orNLS_Numeric_Characters.

Injection Mechanisms

Injection through user input

Injection through cookies

Injection through server variables

Second-order injection

Injection Mechanism

First-order injection

- The application processes the input, causing the attacker's injected SQL query to execute.

Second-order injection

- The application stores that input for future use (usually in the database), and responds to the request.
- The attacker submits a second (different) request.
- To handle the second request, the application retrieves the stored input and processes it, causing the attacker's injected SQL query to execute.

Blind SQL Injection

The information must be inferred from the behavior of the page by asking the server true/false questions

```
SELECT accounts FROM users WHERE login='legalUser'
and 1=0 -- ' AND pass='' AND pin=0

SELECT accounts FROM users WHERE login='legalUser'
and 1=1 -- ' AND pass='' AND pin=0
```

Timing Attacks

Gain information by observing timing delays in the response of the database

```
SELECT accounts FROM users WHERE login='legalUser' and
ASCII(SUBSTRING((select top 1 name from sysobjects),1,1))
> X WAITFOR 5 -- ' AND pass='' AND pin=0
```

Alternate Encoding

Employ alternate methods of encoding attack strings

```
SELECT  accounts FROM users
WHERE  login =  'doe'; exec(char(0x73697574646f776e)) --'
          AND pass = '' AND pin =
```

SQL Injection Countermeasures

- Minimize the Privilege of Database Connection
- Disable Verbose Error Message
- Protect the system account “SA”

- Audit Source Code:
  - Escape Single Quotes
  - Input Validation
  - Reject Known Bad Input
  - Input Bound Checking
  - All user inputs should be filtered

SQL Injection Detection and Blocking Tools

SQLBlock

SQLBlock is an ODBC/JDBC driver with a patent pending SQL injection prevention feature

It works as an ordinary ODBC/JDBC data source, and it monitors every SQL statement being executed

If the client application tries to execute any un-allowed SQL statements, SQLBlock will block the execution and will send an alert to the administrator

DMT SQL Block Monitor 1.5
File Run View Tools Help
Clear Start SQL
Blocking Process
Process ID Host name Blocked Duration CPU Login name Login time Last batch
52 HOST2 HOST1 (51) 01:20 78 sa 2007-5-13 15:10:09 2007-5-13 15
51 HOST1 04:10 15 LICHENG\A... 2007-5-13 15:06:21 2007-5-13 15
Host Statistics
HostName Wait Blocked
HOST2 11:17 53:47
HOST1 11:17 53:47
History
Time Connect Max duration
2007-5-13 15:10:27 2/19 03:01
2007-5-13 15:10:32 2/19 03:05
2007-5-13 15:10:37 2/19 03:10
2007-5-13 15:10:42 2/19 03:15
2007-5-13 15:10:47 2/19 03:20
2007-5-13 15:10:52 2/19 03:25
2007-5-13 15:10:57 2/19 03:30
2007-5-13 15:11:02 2/19 03:36
2007-5-13 15:11:07 2/19 03:40
Process ID Host name Blocked Duration
52 HOST2 HOST1 (51) 00:11 78
51 HOST1 03:01 15
Monitor
Connected To (local)

SQL Injection Wrap-Up

Now a days SQL injection is one of the biggest nightmare among Database administrators. Though we have a lot of way for its prevention but still today's most website suffer from this attack.

Cross Site Scripting (XSS)

What is XSS?

Cross Site Scripting

XSS is a vulnerability which when present in websites or web applications, allows malicious users (Hackers) to **insert their client side code** (normally JavaScript) in those web pages. When this malicious code along with the original webpage gets displayed in the web client (browsers like IE, Mozilla etc), allows Hackers to **gain greater access** of that page.

Effects

stealing other user's cookies
stealing their private information
performing actions on behalf of other users
redirecting to other websites
Showing ads in hidden IFRAMES and pop-ups

How XSS Works

Web server gets data from web client (POST, GET, COOKIES etc) with the request. So a malicious User can **include client side code snippets** (JavaScript) into the data. For example :

```
Amit<script>alert ('this site
has been hacked') ;</script>
```

Note: This image has been created using Firebug and this XSS hole is not present in google.com

XSS Cont.

Let's assume Web server performs **no validation** or filtration on this data.
Now web server either saves this data + XSS code to some persistent storage (like database) or print this data back in the HTML.
When this XSS code, comes from server along with HTML into the web client (Browser) and **executes as server's own code**, it gets access whole HTML document, page URL, cookies etc.

XSS Cont.

Server
http request with
XSS JavaScript
http response with
XSS JavaScript
Hacker's Browser
Hacker's Browser

XSS Output

Note: This image has been created using Firebug and this XSS hole is not present in google.com

XSS Vectors

```
<SCRIPT
SRC=http://ha.ckers.org/xss.js></SCRIPT>
<IMG SRC=javascript:alert('XSS')>
<IMG SRC=javascript:alert(&quot;XSS&quot;)>
<IMG SRC=`javascript:alert("RSnake
says, 'XSS'")`>
<IMG """><SCRIPT>alert("XSS")</SCRIPT>">
<IMG
SRC=javascript:alert(String.fromCharCode(88
,83,83))>
<IMG
SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;
&#105;&#112;&#116;&#58;&#97;&#108;&#101;&#1
14;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41
;>
```

XSS Types

Non-persistent
Persistent
DOM Based

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Weakness ID: 79
Abstraction: Base

Status: Usable

Presentation Filter: Complete

Description

Description Summary

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

Extended Description

Cross-site scripting (XSS) vulnerabilities occur when:

1. Untrusted data enters a web application, typically from a web request.
2. The web application dynamically generates a web page that contains this untrusted data.
3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc.
4. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data.
5. Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain.
6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

Nature	Type	ID	Name	V
ChildOf	C	74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	699 1000 1003

http://cwe.mitre.org/data/definitions/79.html

XSS attacks and the SOP

- **Common attack pattern**
  - Malicious input is supplied to a web application server encoding an executable script (e.g. through malicious link in email).
  - The server includes the script in the dynamic generation of a web page, possibly immediately (**reflected XSS**) or later (**stored XSS**).
  - Browser renders the page and executes the script.
- **XSS and the Same Origin Policy (SOP)**
  - SOP dictates that only scripts received from the same origin as the web page have access to the web page's DOM data.
  - XSS attacks "conform" to the SOP, since malicious content is loaded from the same origin.

Reflected XSS

- Malicious code delivered to an user through a link e.g. embedded in an email, web page, …
- Server reply “reflects” malicious script that is executed on the victim’s browser.

Stored XSS

- Malicious script stored by adversary exploiting a server-side vulnerability, then propagated to client browsers.

Reflected XSS — DVWA example

browser

What's your name? <script>window.alert('he Submit

request

server

```
echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
```

response

browser

- Example above
  - DVWA set with low security level
  - Manual test illustrated, but CSRF-style malicious link could be easily crafted (note that a GET request is used).

Stored XSS — DVWA example

browser

request

server
stores script

Name *

Hello

Message *

```
<script>
window.alert(document.cookie)
</script>
```

Sign Guestbook

```
$query  = "INSERT INTO guestbook ( comment, name )
           VALUES ( '$message', '$name' );";
$result = mysql_query( $query )
```

```
$query  = "SELECT name, comment FROM guestbook";
while( $row = mysqli_fetch_row( $result ) ) {
   …
}
```

subsequent
request by browser

- Script stored in the database and echoed back by the server for execution in the browser in subsequent visits.

Famous XSS attack — Samy XSS worm

- Samy attack on MySpace — a few quotes from "Ajax prepares for battle on the dark side", by Quinn Norton, Guardian, 2006
  - "Samy created **Ajax code on his MySpace site that ran automatically when anyone looked at his profile**. Because Ajax can interact with pages users never see, his code pressed all the relevant buttons to **add Samy to the victim's friends**, and added the words "but most of all, samy is my hero" to their page. **Finally, the code pasted itself into the victim's profile, so that any MySpace user viewing the victim's page would have their page infected**. MySpace users were unaware their computers were doing anything unusual."
  - "**The code - strictly speaking, a cross-site scripting worm - spread exponentially.** Within 24 hours Samy had a million emails from MySpace users "wanting" to be his friend and to whom he was their "hero". **MySpace was forced to shut down and make changes to stop Samy's code spreading.** The MySpace Worm, as it came to be called, served as an alarming example of what malicious hackers could do, even if they only had access to your browser."

Other XSS attacks

- Hackers still exploiting eBay's stored XSS vulnerabilities in 2017, Paul Mutton, NetCraft.com, 2017
  - "*All of the attacks stem from the fact that* ***eBay allowed fraudsters to include malicious JavaScript in auction descriptions.***"
- Email attack exploits vulnerability in Yahoo site to hijack accounts, Lucian Constantin, PCWorld, 2013
  - *"The same-origin policy is usually enforced per domain. […]* ***However, depending on the cookie settings, subdomains can access session cookies set by their parent domains.*** *This appears to be the case with Yahoo, where the user remains logged in regardless of what Yahoo subdomain they visit, including developer.yahoo.com.*
  - *"The rogue JavaScript code […]* ***forces the visitor's browser to call developer.yahoo.com with a specifically crafted URL […]***"

Prevention by input validation and output encoding

```
$message = htmlspecialchars( $message );
$name = htmlspecialchars( $name );
```

```
<script> ──────────► &lt;script&gt;
```

- **Preventing XSS — Server side**
  - **Input validation**: disallow/sanitise malicious input using conventional techniques, e.g. “escape” functions.
  - **Output encoding**: server sanitizes data before sending it, employing similar techniques.
  - DVWA example: high / impossible security levels in DVWA use the htmlspecialchars PHP function to escape HTML both for input sanitization and output encoding.

DOM-based XSS

- Malicious code may also be delivered in several forms: untrusted Javascript library, email links, etc
- Even if server interaction may play some role in serving the adversary's purpose, **DOM-based XSS takes effect directly on the client by manipulating the DOM model — malicious code need not be emitted by the server.**

DOM — Document Object Model

Image source: W3 Schools

```
<!DOCTYPE html>
<html> event
▶ <head>…</head>
▼ <body>
  ▼ <header>
    ▼ <div class="wrapper">
      ▼ <table>
        ▼ <tbody>
          ▼ <tr>
            ▶ <td valign="top">…</td>
            ▶ <td width="30">…</td>
            ▼ <td>
                <h2>Questões de Segurança em Engenharia de Software</h2>
                <h2>2018/19</h2>
              ▼ <p>
                  <a href="http://msi.dcc.fc.up.pt">Mestrado em Segurança Informática
                  </a>
                  <br>
                  <a href="http://www.dcc.fc.up.pt">
                  Departamento de Ciência de Computadores</a>
                  <br>
                  <a href="http://www.fc.up.pt">
                  Faculdade de Ciências da Universidade do Porto</a>
```

Firefox DOM Inspector

- The Document-Object model (DOM) is a tree-abstraction for documents:
  - an HTML (but also XML, XHTML, SVG, …) document is treated as a tree structure where in each node is an object representing a part of the document.
  - tree nodes can be visited, created/added/deleted, and have associated attributes like event handlers and styles.
  - A W3C standard until 2004, now maintained by the WHATWG group — check the live document for the current DOM specification.
  - Browsers represents HTML document in an internal structure similar to the DOM - major browsers use the WebKit Webcore component for that purose

Javascript DOM API — outlook

- A brief overview of some of the functionality in the Javascript DOM API … accessible through **`document`**, the top-level object that represents the DOM:
  - Basic attributes:
    - `title referrer URL hash cookie readyState`
  - Page elements:
    - `head body forms scripts links`
    - `getElementById(elementId)`
    - `querySelector(cssSelector)`
  - Modification methods:
    - `write(anything) writeln(anything)`: append output to the document
    - `createElement()` `createEvent()` `execCommand()` `addEventListener()`
- A wide attack surface for malicious scripts!

DOM-based XSS — simple example

Malicious script

```
<script>
  // Get malicious input from query string and unescape it
  var pos = document.URL.indexOf("evil=")+9;
  var evilScript=document.URL.substring(pos,document.URL.length);
  // Make it take effect
  document.write(unescape(evilScript));
</script>
```

Code injection possible using:

```
queryStringAttack.html?evil=<script>. . . </script>
```

- Query string associated to HTML page. No server interaction for triggering the exploit.
- Anchor '#' (i.e., document.hash instead of document.URL) also exploitable in similar

DOM-based XSS — another example

```
// Store
localStorage.setItem("lastname", "Smith");
// Retrieve
document.getElementById("result").innerHTML = localStorage.getItem("lastname");
```

- HTML 5 introduced local browser storage of key-value pairs (like cookies), an extra attack surface.
  - One more facility to store sensitive or malicious data that can be controlled programatically. Above: a “predictable” usage example from w3schools … (!) How can it go wrong?
  - WebSQL and IndexedDB also allow structured databases, though the adoption of one or the other has not been peaceful (both out of HTML 5). Firefox only supports IndexedDB, Safari and Google also support WebSQL.
- Extra attack-surface: CSS-based vulnerabilities
  - CSS Exfil
  - Microsoft Internet Explorer Cascading Style Sheets Remote Code Execution Vulnerability

XSS types compared

- **Source**: Unraveling some of the Mysteries around DOM-based XSS by Dave Wichers, AppSec USA 2012

Cross Site Request Forgery (CSRF)

CSRF — general description

- CWE-352 - Cross-Site Request Forgery (CSRF)
  - "*The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.*"
- Most common attack pattern:
  - User has an authenticated session for a web application.
  - Adversary tricks user into executing some malicious action, e.g. by clicking a link sent by email or provided in a web site controlled by the adversary.
  - Malicious actions are executed in the server as if intended by the user.

XSS vs CSRF

- XSS
  - Trust relation: client trusts the server
  - Attacker tries to affect what the server sends to the client / what runs on the client.
- CSRF
  - Trust relation: server trusts the client
  - Attacker tries to affect what the client sends to the server / runs on the server.

Example CSRF attacks — Gmail, 2010

- GMail Service CSRF Vulnerability (2010)
  - "*GMail is vulnerable to CSRF attacks in the "Change Password" functionality. The only token for authenticate the user is a session cookie, and this cookie is sent automatically by the browser in every request.*"
  - "*An attacker can create a page that includes requests to the "Change password" functionality of GMail and modify the passwords of the users who, being authenticated, visit the page of the attacker.*"
  - *"The attack is facilitated since the "Change Password" request can be realized across the HTTP GET method."* [it suffices to craft a malicious link with an appropriate query string]

CSRF example — DVWA

- As in the Gmail example, the password change functionality is at stake.

- Security levels:
  - **low**: no protections, a simple malicious link may be used to change the password
  - **medium**: HTTP request header **Referrer**; Referrer link may also be forged by an adversary. Alternatively, malicious link can however be accomplished by exploiting a XSS vulnerability (hint: try the message forum; more discussion on this next).
  - **high**: automatically generated **anti-CSRF token** included in hidden form field - token is attached to session but not the request itself however …
  - "**impossible**": anti-CSRF token + request for current password confirmation

Some CSRF mitigations

- Allow only POST requests
  - GET requests allows a CSRF attack through a simple link.
  - POST requests may anyway be trivially defined for instance by HTML forms that are presented to the user.
- HTTP **`Referer`** field is used to indicate origin of request. A basic protection is to check it on the server side. The value may be absent however, frequently with the good motivation of preventing user tracking or leaking data to untrusted sites.
  - **`<meta name="referrer">`** **in HTML header** or the **`Referrer-Policy`** HTTP header may inhbit it.
  - Most browsers may be configured to omit the referrer information for cross-site requests.
  - A common referrer policy is to hide the information when making HTTP requests from content loaded through HTTPS.
  - **Note**: due to a typo in the original HTTP specification, the header is called **Referer** rather than **Referrer**.
- **Hardened application logic**: re-authentication or 2FA schemes for critical operations.
- **Use of synchronization tokens (next).**

Anti-CSRF token in DVWA

```
// Check Anti-CSRF token
checkToken( $_REQUEST[ 'user_token' ],
            $_SESSION[ 'session_token' ], 'index.php' );

// Do the passwords match?
if( $pass_new == $pass_conf ) {
      . . .
}
// Regenerate Anti-CSRF token
generateSessionToken();
```

```
<form action="#" method="GET">
  . . .
  <input type='hidden' name='user_token'
    value='d842e88752fd9991fb4dbcfa35649ae4' />
</form>
```

- Expected value of anti-CRSF token is checked first. Operation does not proceed on a token mismatch.
- The token gets regenerated with a new value once operation is complete.
- Javascript / XSS-based exploit possible - check here for an example

Anti-CSRF token in DVWA (cont.)

```
function generateSessionToken() { # Generate a brand new (CSRF) token

  if( isset( $_SESSION[ 'session_token' ] ) ) {
   destroySessionToken();
  }

  $_SESSION[ 'session_token' ] = md5( uniqid() );

}
```

- The generation function is in itself weak (it would not be appropriate for session ids for example):
  - MD5 is a weak cryptographic-hash function.
  - the uniqid() PHP function is predictable and does not in fact guarantee a unique ID - “[it] gets an unique identifier based on the current time in microseconds“ and “does not guarantee uniqueness of return value”.
- However, since DVWA regenerates the token after each request and the id is based on the scale of micro-seconds, it is hard to predict it quickly enough (1 million possibilities per second).

Synchronizer token pattern

- **Synchronizer Token Pattern**
  - State changing operation uses a token (different from the session id), generated through a cryptographically-secure random generator by the server, and that is unique per session (server associates token to the session id).
  - Token value is embedded in the web page or within a cookie.
  - Token mismatch in a future request inhibits state-changing operation.
- For enhanced security:
  - Token can be regenerated after each request (as in DVWA) or at least have a short expiration time.
  - Use different tokens per request/operation rather than for the entire session. This may hinder usability though (e.g. Back button reverts to a page with a invalid token).
- The synchronizer token pattern requires a state to be maintained explicitly. Stateless alternatives exist:
  - Encrypted-token scheme (check is based on sucessful decryption).
  - “Double-submit” cookie: generate nonce and include it both in the response body and in a header field.

CSRF: further reference

- Barth et al., "Robust Defenses for Cross-Site Request Forgery", CCS'08
- OWASP Cross-Site Request Forgery Prevention