# 

Time: 90 minutes Instructions:

Answer all questions

Question 1 SQLi. [20 Marks]

Question 2 SSL/TLS [30 Marks]

Question 3 CSRF [20 Marks]

Question 4 PCI [20 Marks]

Question 5 RFI [10 Marks]

**Not bad at all**

# Questions from last year

Question 1

a) Explain the differences between reflective XSS and stored XSS. [20 Marks]

b) Outline what the OWASP cheat sheet has to say about preventing XSS.

Question 2

Outline and explain how a browser sets up a secure session with a web server in SSL/TLS. [30 Marks]

Question 3

Explain how the use of a **CSRF Token** is used to prevent CSRF., and why it prevents CSRF. Your answer should clearly demonstrate how CSRF Attacks work.

# .[SQL Injection](https://citbb.blackboard.com/bbcswebdav/pid-314404-dt-content-rid-856328_1/xid-856328_1)

SQL includes a syntax to update, insert, and delete records:

* SELECT - extracts data
* UPDATE - updates data
* INSERT INTO - inserts new data
* DELETE - deletes data

Other Important commands:

* DROP: Deletes the table
* SHUTDOWN: Stop the server
* IF: Conditional statement
* SUBSTRING: Select a part of the string – useful for blind SQLi
* WAITFOR :Causes a time delay

**Typical steps in the SQLi Pen Test Process**

* Discover the SQLi Flaw Determine the database type
* Determine the database structure, the metadata
* Query the data
* Retrieve data from the database
* Possibly run commands in the OS of the database server

**Fingerprinting a Database**

**Aka: Find what database and database engine is used**

Use error messages to determine the database type. e.g. send a single quote to the database

Oracle : ORA-02367: string not terminated

MS SQL: Incorrect syntax near '

MySQL: Syntax error

PostgreSQL: a 5-digit hex error code

**DB Admin**

* Default administrator accounts include: sa, system, sys, dba, admin, root and many others
* In MS SQL they map into dbo: The dbo is a user that has implied permissions to perform all activities in the database.
* Anything object created is auto added to the db admins

**Fingering the Metadata**

We want the names of the tables and the fields within the tables.

**Discovering SQLi**

* Find input points in the app and supply test strings
* Try the single quote and the double quote
* Look at the error messages that you get back Note that they might be in the HTML source Also, you can use a fuzzer

Tools : Burp can be used

**Tamper Data:** Allows one to change HTTP requests

**Manual Fuzzing** : "fuzzing" a system to see if the input is properly sanitized, example : admin'--

SQL reserved words with white space delimiters e.g. %09select

**Blind SQLi :**

* Sometimes the app being tested does not display back error messages
* Still vulnerable however.
* Need to use YES/NO type questions
* the attacker looks for differences between true code injections (1=1) and false code injections (1=2) in the response pages to extract data

Attacker analyzes the response pages looking for differences between “True- Answer Page” and “False-Answer Page”:

* Different hashes Different html structure
* Different patterns (keywords)
* Different linear ASCII sums “Different behavior”

**Time-based Blind SQL Injection using heavy queries**

**In scenarios with no differences between “True- Answer Page” and “False-Answer Page”, time delays can be used.**

Injection forces a delay in the response page when the condition injected is True.

Delay functions: SQL Server:

waitfor Oracle: dbms\_lock.sleep

MySQL: sleep or Benchmark Function Postgres: pg\_sleep

Ex: ;

if (exists(select \* from users)) waitfor delay '0:0:5’

**Evasion**

Input validation, IDS detection AND strong database and OS hardening must be used together

Examples

OR 'unusual' = 'unusual'

' OR 'something' = 'some'+'thing'

**MySQL Input Validation Circumvention using Char()**

Inject without quotes (string = "%"): ' or username like char(37);

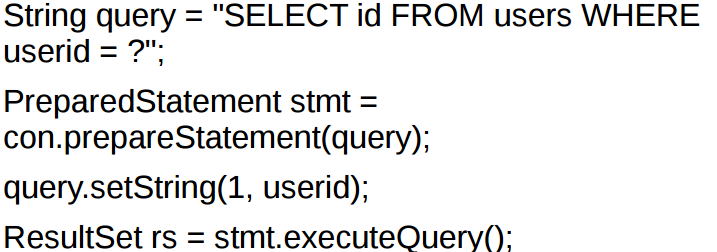
Inject without quotes (string = "root"): ' union select \* from users where login = char(114,111,111,116);

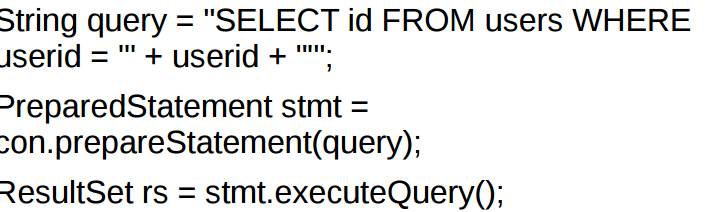
SQL Injection Defence

* Use parameterized queries (Prepared Statements)
* Use of stored procedures
* Escape all user supplied Input

Parameterized Queries

* The use of parameterized queries (preped statements) is how all developers should be taught how to write database queries.
* They are simple to write and easy to understand
* Parameterized queries force developers to first define all the SQL code, then pass in each parameter in at a later stage.
  + This style of coding allows databases to distinguish between what is code and what is data regardless of what the user inputs
* These queries ensure that attackers are not able to change the intent of the query even if they insert commands into the SQL.
  + Example - Even if attacker were to enter the userId of tom’ or ‘1’=’1
  + The query would not be vulnerable and would instead look for a username with the value of the screen literally as “tom’ or ‘1’=’1.
* Java has a prep statement class in its implementation of parameterized queries
  + It can be used in the following manner
* Good Example



* Bad Example
* Parameterized queries can also add performance benefits to your querying
* Input still needs to be valid… Queries will still input malicious input into the database
* Second order issues - Take data from database and use it dynamically.
  + This means that the SQL injection will not be when the data goes into the database but instead go in at a later stage
* Very Hard to find with Black Box Penetration testing
* Easier to find with a code review
* Persistent XSS, unsanitized data is being injected and used somewhere else
* Rarely SQL is used in a way that cannot be parameterised
  + Applications where the designer/developers have decided to dynamically supply SQL keywords should not do this.
* There is certain situations where parameterized queries can cause harm to performance
  + Especially in certain versions of oracle where building large complex queries against a large set of data.
  + This is an incredibly extreme case
* Here is when you should just encode the variables individually before putting them into an SQL string
* Any data going into the database could be potentially malicious. Always treat it as if it is, this is the best way to protect yourself from SQLi injection

Stored Procedures

* Stored procedures have the same effect as the use of Parameterized Queries when implemented safely
* Requires the developer to define the SQL code first, then pass in parameters after
* The difference between prepared statements and stored procedures is that SQL code for a stored procedure is defined and stored in the DB itself and then called by the application
* Both techniques have the same effectiveness in preventing SQL injection so your organization should choose which approach makes the most sense for your application
* Stored Procedures are also vulnerable to SQL injection if not coded correctly
* In Java, stored procedures are called using the CallableStatement class
* It may appear that you’re not creating dynamic queries in your java code, but if the backend stored procedure code is via an “execute” call, to the OS, then you’re still vulnerable to SQL injection.

Escaping all user Input

* Escape user input before putting it into any query
* This method can be frail compared to using something like parameterized queries so it is not guaranteed it will prevent all SQL Injection for every situation
* This technique will work only in the following
  + Each DBMS supports one or more character escaping scheme for each kind of query
  + If you escape all user supplied input using proper escaping for the database being used the DBMS will not confuse the input with SQL code written by developers thus avoiding any SQL injection vulnerabilities

Input Validation

* Define data types for each field
  + Make sure input meets passes a whitelist criteria check
* How to deal with customer comments?
  + Implement stringent “Allow only good” filters
    - If the input is suppose to be numeric, use only a numeric variable in your script to store it
* Reject any bad input values rather than attempting to escape or modify it.
* Implement stringent “known bad” filters
  + Examples
    - Reject “select”, “insert”, “update”, “shutdown”, “delete”, “drop”, “--” “”
* Disallow all ‘ and “
* This won’t stop all though.
  + Names like O’Reilly
  + The input could be escaped (Put slash before the input) but that would depend on the server type.
  + This means developers will need to write input validation every time.

ESAPI - [http://www.owasp.org/index.php/Category:OWAS P\_Enterprise\_Security\_API](http://www.owasp.org/index.php/Category:OWAS)

* ESAPI provides encodeForSQL method in its classes
* ESAPI currently encodes for the following databases
  + Oracle
  + MySQL

Countermeasures at the Sys Implementation Level

Countermeasure: Permissions

* Never run as a superadmin(SA) or as the root user
* Create and define users with limited rights
  + Prevents normal users from being able to run certain commands
  + If the app is simply for displaying news users they only require certain permissions
    - For example - SELECT statements on the part of the database that is specific to the news
    - Users do not need any permissions on the part of the database that deals with user info
    - This means users at worst can just download all the news stored in that part of the database (Which doesn’t matter since it’s the service you are providing them)

Countermeasure: Firewalls

* Limits uses
* Firewalls are likely to be ineffective at filtering traffic coming into the server as port 80 / 443 must be allowed to receive data.
* Egress traffic filtering will help as it will allow you to intercept an attack as they try to connect.
* Filter traffic from the web server to the database server.
* Attackers may bypass the firewall by using a server that only allows inbound traffic as a relay
* Attackers could attempt to use DNS in order to bypass any active firewalls
  + This is hard to block
* Attackers can run shell via DNS
  + Run a command on the database using SQLi
  + Get output of command by piping it into a DNS lookup for output-string.my-domain.com(where I own my-domain.com)
  + The attacker will then sniff this query and through doing this gets the output
  + Now the attacker is able to run certain commands through the database to be executed.

Countermeasure: Web App F/Ws

* IDPS but for web applications
* Only works sometimes
  + Can be bypassed
    - Through commands like SELE/\*comment\*/CT
    - SQL servers are very forgiving with comments(W.R.T)
    - Parameter pollution - Same parameter twice
      * ?id=select user&distract&id=password
    - How can the database react to the above
      * It will concat the 2 strings and use a comma to delimit it
    - Database sees it as Select user, password

Harden the Server

1. Run DB as a low privileged user account
2. Remove any unused stored procedures or functionality. That or restrict the access to them to the administrator
3. Change permissions and remove “Public” access to system objects
4. Audit password strength for all user accounts
5. Remove pre-authenticated linked servers
6. Remove Unused network protocols
7. Firewall the server so that only trusted clients can connect to it.
   1. Admin network
   2. Web Server
   3. Backup Server

Defence in Depth

* No one silver bullet
* Use as many methods as you can
* Watch your logs
* Flaws can be accidentally activated - This is why its important to check logs

<http://gizmodo.com/5498412/sql-injection-license-plate-hopes-to-foil-euro-trafficcameras> - teh lulz

<http://ferruh.mavituna.com/sql-injectioncheatsheet-oku/> - SQLI cheatsheet

# 

# 

# [CSRF](https://citbb.blackboard.com/bbcswebdav/pid-318018-dt-content-rid-869106_1/xid-869106_1)

https://www.owasp.org/index.php/Cross-Site\_Request\_Forgery\_(CSRF)

Cross-Site Request Forgery (CSRF) is an attack that forces an end user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks specifically target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request. With a little help of social engineering (such as sending a link via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing. If the victim is a normal user, a successful CSRF attack can force the user to perform state changing requests like transferring funds, changing their email address, and so forth. If the victim is an administrative account, CSRF can compromise the entire web application.

### **Synonyms**

CSRF attacks are also known by a number of other names, including XSRF, "Sea Surf", Session Riding, Cross-Site Reference Forgery, and Hostile Linking. Microsoft refers to this type of attack as a One-Click attack in their threat modeling process and many places in their online documentation.

### **Prevention measures that do NOT work**

**Using a secret cookie**

Remember that all cookies, even the *secret* ones, will be submitted with every request. All authentication tokens will be submitted regardless of whether or not the end-user was tricked into submitting the request. Furthermore, session identifiers are simply used by the application container to associate the request with a specific session object. The session identifier does not verify that the end-user intended to submit the request.

**Only accepting POST requests**

Applications can be developed to only accept POST requests for the execution of business logic. The misconception is that since the attacker cannot construct a malicious POST request, a CSRF attack cannot be executed. Unfortunately, this logic is incorrect. There are numerous methods in which an attacker can trick a victim into submitting a forged POST request, such as a simple form hosted on the attacker's website composed entirely of hidden fields. This form can be triggered automatically by JavaScript or can be triggered by the victim who thinks the form will do something else.

**Exploitation using Get**

Cross-site request forgery attacks that use images often come from Internet forums where users can post images but not JavaScript.

For example, an attacker can upload an image with the following tag to a forum:

**<img src="http://www.mybank.com/MoneyTransfer.cgi? AccountFrom=1234&AccountTo=4321& EuroAmount=10000 width="0" height="0" border="0">**

If a user still has an active login session with a Bank and a browser requests the image on a web forum, then a request to www.mybank.com transfers €10,000 into a malicious account.

1. When a page is initially requested, the browser does not know that the page may have an image.
2. Only when you parse the HTML within the response does the browser know that the page contains an image.
3. The image is requested using a standard GET request.

**Exploitation using POST**

The only difference between GET and POST attacks is how the attack is being executed by the victim. Let's assume the bank now uses POST and the vulnerable request looks like this:

**POST http://www.mybank.com/MoneyTransfer.cgi HTTP/1.1 AccountFrom=1234&AccountTo=4321&EuroAmount=10000**

Such a request cannot be delivered using standard A or IMG tags, but can be delivered using a FORM tag:

<form action=" http://www.mybank.com/MoneyTransfer.cgi" method="POST">

<input type="hidden" name="AccountFrom" value="1234"/>

<input type="hidden" name="AccountTo" value="4321"/>

<input type="hidden" name="EuroAmount" value="10000"/>

<input type="submit" value="View my pictures"/>

</form>

**Notes :**

Instead of posting on a forum, the attacker could redirect the user to his/her site by DNS poisoning

The link posted by the attacker does not have to be an image tag.

It could be

* An IFRAME
* Javascript import
* XML HTTP

If an application is vulnerable to cross-site scripting, it is also vulnerable to cross-site request forgery.

**Difference between xss and csrf**

The main difference is in how they deliver an attack

XSS works by bypassing input validation and injecting content into a page.

CSRF works by abusing existing capabilities, such as allowed page elements.

CSRF invokes common gateway interfaces (CGIs) without additional authentication.

CSRF exploits the predictability of the web application, using a browser to retrieve and execute malicious script.

**Detecting CSRF**

Fours steps:

1. Review the app logic
2. Look for functions that perform a sensitive action, and have predictable parameters
3. Create a HTML document which contains a tag which refers to the page (using an IMG or IFRAME tag)
4. Log into the app, and access your HTML page while logged in. Check if the functionality in the IMG or IFRAME executed.

**Preventing CSRF**

* Begin by preventing XSS.
* Confirmation Pages: Ask them are they sure
* REFERER (Check last url)
  + Pro’s
    - Simple to implement
    - All web languages allow us to check this header
  + Con’s
    - Referrer header is optional and arbitrary
    - Not sent on change from HTTP to HTTPS
    - May be stripped out by proxies ● Can be spoofed

**CSRFToken**

Token could be site wide, and maybe expire after a time period

Token could be session ID

Some other value with indeterminate lifetime

Add token to each request http://www.example.com/action.cgi?CSRFToken=1234&data=...

* Check token value before allowing the action
* Attacker needs to know and add this token, as it is random, and has a lifetime
* Browser does not automatically add to request

It should be difficult for the attacker to find out or predict the value of the CSRFToken

Impact

* Minimal. If using session ID nothing extra to store
* Can be modified to the “double submit cookie” approach

Pro’s

* Very simple to implement and check
* Relies on the cross-domain protection of the browser

Con’s

* Attacker only has to discover one CSRF token
* Thus, you have to have sensible expiry/rotation of the token
* Issues with using the session ID as the token (double submit cookie)

**CSRF Protection – Action + Session hash**

* Each page has a value that is designated to that page
* e.g login.html has the value 123
* We also have a secret value on the server which we use as the salt for the hash
* As our value is being created, we add in the session identifier
* SHA1( page-value + salt + session-ID )
* As a user clicks on a link, that value is sent to the server, and the server recalculates the hash value
* If it matches the value in the token, we know that the request was initiated by the user

**Pro’s**

* Strong protection
* Leakage of a hash doesn’t affect the rest of the app or users – only that user on that page

**Con’s**

* Computational overhead

**CSRF Protection: Unique Token**

Generate a unique token for every operation that will occur on the app.

As a page gets served up, we generate some random data, and we store it in a database along with the session token for that user.

That gives us one or more tokens that we append to each form or each link as appropriate

**Impact**

* High
* Storage requirement for tokens on page
* Computational requirement for generating random data
* Latency on reading/writing to database

**Pro’s**

* Very strong protection
* Doesn’t matter if token(s) are lost
* One-time use

**Con’s**

* Overhead

**CSRF Protection –“Secret” information**

**Eg use a CAPTCHA**

**Impact**

Small for application, possibly large for user

**Pro’s**

For passwords/user-known info – no technical hack

Easy to put in before “high value” operation

**Con’s**

Breaks the users flow

Biggest issue : Techniques/systems exist for defeating CAPTCHA’s : Bots and Outsourced Labour

# [PCI-DSS](https://citbb.blackboard.com/bbcswebdav/pid-319428-dt-content-rid-875460_1/xid-875460_1)

**What is PCI DSS?**

Payment Card Industry (PCI) Data Security Standard (DSS)

All member organisations that issue or acquire information from cards with the Visa, MasterCard, American Express and Discover logos are required to comply with a range of information security requirements.

**Question are added to the top**

**Where does it apply?**

Applies to organisations where cardholder data is stored, processed, or transmitted.

**PCI DSS How does it works?**

The PCI DSS standard sets common requirements for securing card information, and lays out a range of controls relating to auditing, scanning and assessment.

**Why is it needed?**

Encourage and enhance cardholder data security

Facilitates the broad adoption of consistent data security measures globally.

Help prevent breaches of card data like the famous TJ MAX case

**Compliance**

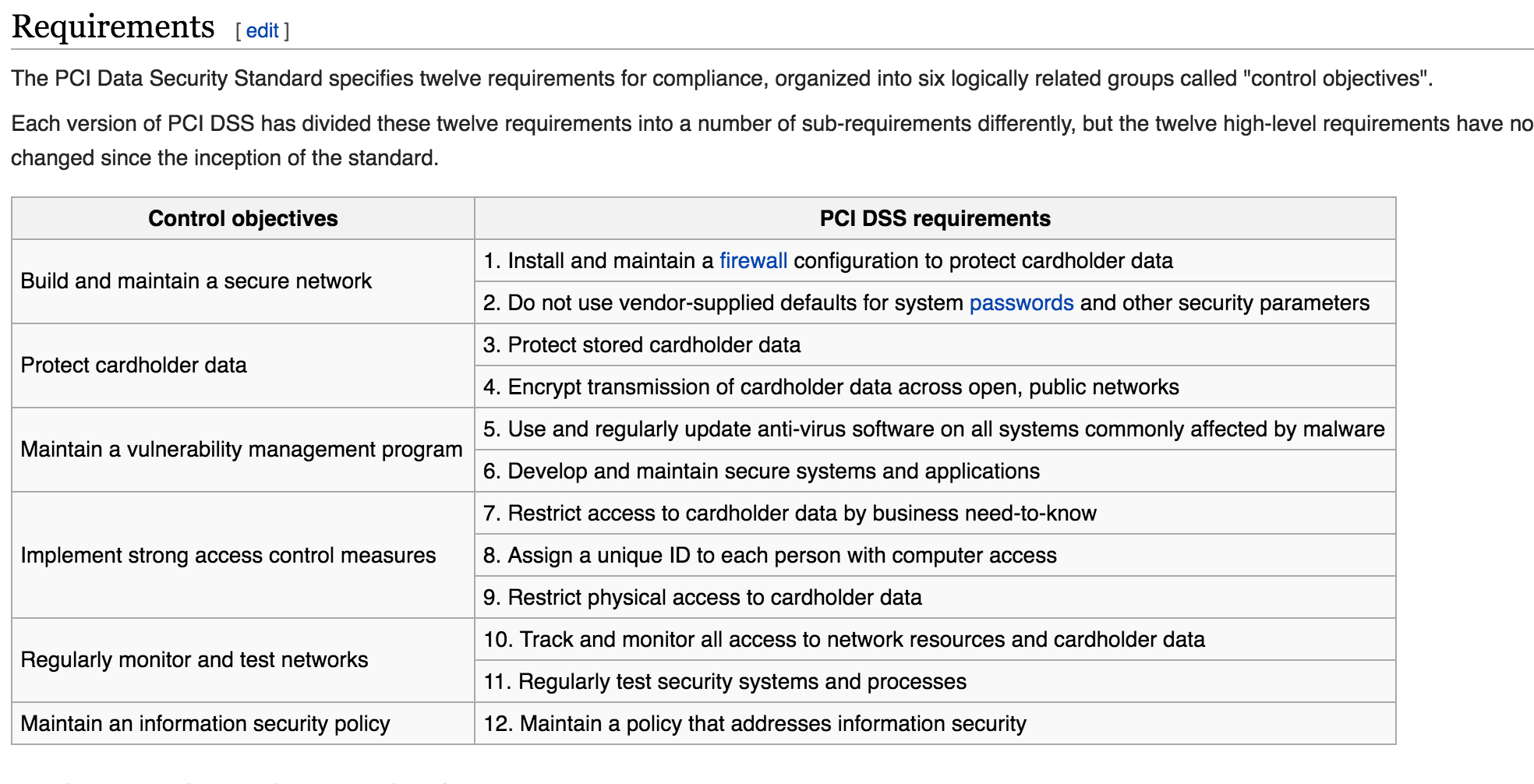
PCI Security Standards Council sets the requirements, but each card association implements and enforces the standard, fines/fees, and compliance levels and deadlines.

**PCI Levels**

Merchants fall under four categories of PCI compliance, depending on the number of transactions they process each year, and whether those transactions are performed from a brick and mortar location or over the Internet.

Unfortunately, each payment card brand (Visa, MasterCard, etc.) has their own requirements and definitions of PCI compliance levels. Even though the PCI Security Standards Council (PCI SSC) developed these standards, compliance is actually mandated by the individual payment card brands ­ Visa, MasterCard, American Express, Discover and JCB International. • The following are Visa's PCI compliance level definitions:

* Level 1: over 6 Million transactions annually
* Level 2: Between 1 Million and 6 Million transactions annually
* Level 3: Between 20,000 and 1 Million transactions annually
* Level 4: Less than 20,000 transactions annually



# [SSL/TL](https://citbb.blackboard.com/bbcswebdav/pid-320535-dt-content-rid-880057_1/xid-880057_1)S

https://www.owasp.org/index.php/Transport\_Layer\_Protection\_Cheat\_Sheet

**Cryptography**

* Cryptography is the science and study of secret(crypto) writing (graphy)
* A **cipher** is a secret method by which to communicate plaintext using an **algorithm** that transforms the **plaintext** into **ciphertext** through the process of **encryption**.
* The plaintext can be recovered from the ciphertext through the process known as **decryption**

**Cryptography Jargon**

* Encryption
  + Also known as **encipherment**(No one really calls it this) or **encoding** (its not really the same thing)
  + The process of encryption involves the use of a **key.**
    - The key is a piece of critical information used in the cipher known only to the sender & receiver of the message.
* Two main types of cryptography
  + Conventional
  + Public Key

**Conventional Cryptography**

* Conventional Cryptography
  + Conventional cryptography is the **same key** is used to both **encrypt** and **decrypt**
  + A decryption key can also be a simple function of the encryption key
  + It is also known as **single-key** **cryptography**, **Secret-key cryptography** or **symmetric cryptography**
  + **Alice** wants to send msg to **Bob** but to **encrypt** the msg she needs to use a **key.**
  + **Bob** must use the same **Key** as **Alice** in order to decrypt the ciphertext.
    - How do **Alice** and **Bob** communicate the **key** to each other
      * This is known as **key distribution problem**
  + Main Conventional Ciphers
    - AES
    - DES (No longer used)
    - Triple-DES (3-DES)
    - Blowfish
    - TwoFish
    - RC4 (A stream cipher)

Public Key Cryptography

* Each user has 2 keys
  + 1 Public key (**PU**) | 1 Private key (**PR**) - These keys are the inverse of each other
  + This means knowing the public key does not reveal the private key
    - If it did it wouldn’t be a very good form of encryption
  + So if **Alice** wants to send **Bob** an encrypted msg **Alice** must look up **Bob’s** PU and encrypt the msg using **Bob’s** PU.
  + Only **Bob** can decrypt this as he has his personal **PR**
* Main Public Key Ciphers
  + RSA - (Most commonly used) - Calculate n = pq
    - N = modulus for the public key and private key
    - P/Q = large random prime numbers
  + Elliptic Curve -
  + El Gamal - Not likely to come up

**Brute Force Attack**

* Also called **Exhaustive Key Attack**
* Involves trying every possible answer until the correct one is found
* In an attempt to thwart this developers have created keys with size big enough that trying every possible is unreasonable to even attempt and thus makes the attack unworthy of consideration
* 128 bits is regarded as safe in today's world
* Examples of how long brute forcing could take
  + For a 56 bit key @ 1 decryption per second it would take 833,99,931 days to decrypt
    - @ 1 per milisec -> 833,999 days to decrpyt
    - @ 1 per nanosec -> 833 days
    - @1 decrpyt per nanosec on 100 machines in parallel -> 8 days
  + Assume an attack could find a 64bit key through brute force search in 1 second.
    - For the same attacker to get a 96bit key it would take them 2^32 which is 135 years
    - For the same attack to get a 128bit key he’d need 2^32 times 135 years which is over 500,000,000,000 (thats a fuck ton of 0s)

**Key Exchange using Public Keys Cryptography**

* Diffie-Hellmann | RSA | PFS

**Key exchange using RSA**

* **Alice** wants to exchange a **session key** with **Bob**
* She invents a session key (**Ks**) - Possibly an AES key
* She looks up **Bobs** public key(PU)
* She then encrypts the session **Ks** using **Bob’s PU** to get a digital envelope

**PFS(Perfect Forward Secrecy)**

* PFS protects past sessions against future compromises of secret keys or passwords
* Protocol that achieves PFS generate a new key pair for each session and then discard them when the session ends
* The encryption is done with a key which we do not keep around and thus makes it immune to ulterior theft.
  + This only works in a setup in which we do not want to keep the data encrypted
    - Not useful for things like email (They should remain encrypted in the mailbox)

# Remote File Inclusion

https://www.owasp.org/index.php/OWASP\_Periodic\_Table\_of\_Vulnerabilities\_-\_Remote\_File\_Inclusion

**What is it?**

* RFI is a technique used to attack web applications (Especially PHP applications)
  + Done from a remote server
* RFI attacks can be extremely dangerous to the server
  + They allow a client to force vulnerable applications to run malicious code on the web server
* Attackers can do this by including a reference to their code from a URL located somewhere on the remote server
* There is many attempts to do this and is rarely successful due to server admins becoming wiser to it
* There is still plenty of software that may be vulnerable though

**Register Globals**

* These attacks require old configurations
  + Things like “Register\_globals”
* Registered globals are a feature in PHP which allow data to be passed to a PHP script via cookies or GET/POST requests in order to be made globally available in the script.
* It is a very convenient but extremely dangerous
  + It often will allow attackers to overwrite variables in a script simply by adding parameters to a request.
* Since 2002 & version 4.2.0 of PHP it has been disabled by default.
* But providers of hosting often still enable this feature to provide compatibility with old scripts.

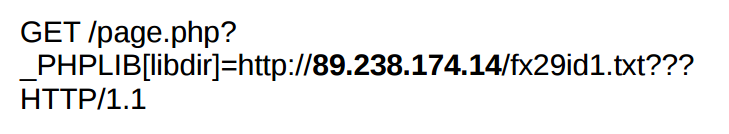
**What Conditions does it need in order to succeed?**

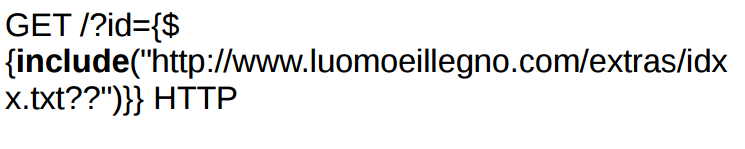
* Two main application vulnerabilities that allow RFI attacks to succeed:
  + Improper Input Handling
    - Inability to identify when a url is being passed a parameter payload
  + Application Misconfiguration
    - Configurations like PHP’s “register\_globals” being enabled

**Main Goals of RFI**

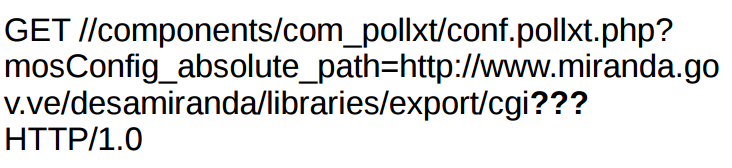
* Botnet Herding
  + Executing botnet code on a server in order for the attacker to the web server’s resources to launch a DDoS attacks.
* Malware Distribution
  + Attacker can inject malicious JS code into web pages
  + These web pages can be served to clients who will now be infected with the malicious code
    - Example - Zeus Banking Trojan

**How RFI can be carried out**

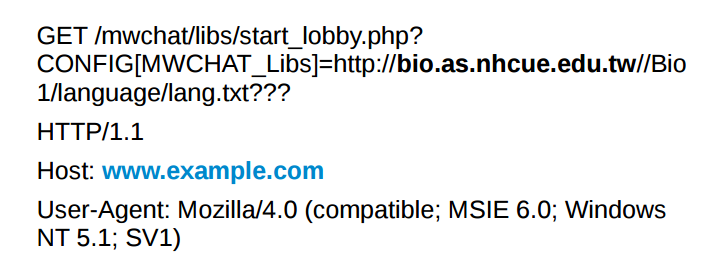
* URLs that contain an IP address
  + Most legit URL references are conducted by specifying actual domain/hostnames
  + Thus a url containing an IP address as an external link indicates that it is likely a malicious attack
  + What a typical attack using an IP address would look like
* Use of PHP functions
  + Another technique is to use Internal PHP keyword functions such as “include()” to try and trick applications into including data from an external site
  + Example below



* URLs with trailing question mark(s)
  + Appending question marks to the end of a inject RFI payload is a common technique
  + RFI attackers don’t know what the rest of the PHP code they are going to be included into.
  + So by adding a “?” at the end of theirs it allows for the remainder of the local php code to be treated as a parameter of the RFI included code
  + The RFI code simply ignores the legit code and only executes its malicious code.
  + A typical attack using question marks at the end will look something like this…



* Off-site URLs
  + An RFI attack, however would have a mismatch between the URL domain and the Host’s header



Countermeasures 1 - PHP Config

* Disabling register\_glfdAaqSDFsed RFI attempts
  + These can all be turned off via the PHP.ini file
  + The webserver will lilkely need to be restarted also.

Countermeasure 2 - Firewall

* Limit outbound traffic
  + Connections from the web server to the C&C server will fail
* Be diligent in watching your logs
  + This will aid in detecting any compromises

Countermeasure 3 - Input Validation

* Validate any filename supplied as an input
  + Check to see that it is a valid local file
* Enhance and improve the application design by making all file names require “AlphaNumeric” values
* Validation needs to be consistently done every time a file is included
* Best counter measure is to avoid user input in including filenames

# 

# Securing Web Apps - checklist