



A COMPARATIVE STUDY ON  
THE SECURITY OF OPEN  
SOURCE WEB CONTENT  
MANAGEMENT SYSTEMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF  
THE REQUIREMENTS FOR THE DEGREE OF MASTER  
OF SCIENCE IN COMPUTER SCIENCE

*Author:*  
Steve Bottelbergs

*Mentor:*  
Bram Bonn 

*Promotor:*  
Prof. dr. Wim Lamotte

*Co-Promotor:*  
Prof. dr. Peter Quax

2012 - 2013

# Preface

This thesis is the result of work conducted during my final year at Hasselt University. It would never have been possible without the help of the people I would like to thank here. Firstly, I would like to thank my mentor, Bram Bonné and promotors Wim Lamotte and Peter Quax for providing me with constructive feedback and suggestions. Secondly, thanks go out to my family and friends, without whom I would never have been where or who I am today. Lastly, very special thanks go to my girlfriend Sabrina Gust, for giving me the needed support, helping me relax in stressful times, and generally keeping me sane throughout the process of writing this thesis.

# Abstract

Web content management systems are used by a lot of people who want to create their own website or blog. Using content management systems can be a very attractive solution because all low level functionality is abstracted by providing a high level interface and the ability to extend core functionality by installing added components. This also means that a typical CMS user will not be concerned with the security of the system, as long as everything works.

A major issue of using content management systems is that a large number of websites are susceptible to the same vulnerability if one is discovered. In this thesis we will provide a comparative study on the three most used web content management systems: WordPress, Joomla, and Drupal. We will compare these content management systems to each other based on a set of objective criteria. We will also provide statistics on the types of reported vulnerabilities and the number of security-related releases for each of the content management systems over the years.

# Nederlandse samenvatting

In deze thesis bekijken we de beveiliging van de drie meest gebruikte web Content Management Systemen of CMS'en. Volgens een onderzoek van W3Techs gebruikt iets meer dan 30% van de websites op het internet een of ander web content management systeem [57]. De drie meest gebruikte zijn WordPress, Joomla en Drupal. Een content management systeem biedt een zekere abstractie voor het beheren van een website, waardoor de website op hoger niveau kan worden onderhouden en er gemakkelijker inhoud kan worden toegevoegd en aangepast. Dit is zeer aantrekkelijk voor onervaren gebruikers of mensen zonder programmeerervaring omdat dit hun in staat stelt om ook een website te kunnen beheren.

Bij de keuze van een CMS wordt er vaak geen rekening gehouden met hoe veilig het systeem precies is, maar eerder naar de gebruiksvriendelijkheid en hoe goed het de wensen van de beheerder vervult. Het is bijgevolg belangrijk om te weten in welke mate deze veel gebruikte web content management systemen veilig zijn en waar er eventueel ruimte is voor verbetering. Hiervoor werd gezocht naar een aantal objectieve criteria om de systemen met elkaar te vergelijken. Deze criteria bestaan voornamelijk uit een onderzoek naar hoe ieder systeem omgaat met de veiligheidsrisico's vooropgesteld door de OWASP Top 10 [1]. Het *Open Web Application Security Project*, oftewel OWASP is een organisatie die zich bezighoudt met het verbeteren van de beveiliging van software [37]. OWASP is een online gemeenschap van security experts die hun kennis over het software beveiligingsdomein delen zodat applicaties geschreven kunnen worden met een grotere focus op veiligheid en beveiliging. Het OWASP Top 10 project is een lijst van de 10 gevaarlijkste en meest voorkomende veiligheidsrisico's op het internet, gebaseerd op meer dan 500000 kwetsbaarheden in meer dan 1000 applicaties [1].

Onze vergelijkende studie is gebaseerd op de risico's die vermeld worden in de OWASP Top 10, en de technieken die ieder content management systeem voorziet om deze te voorkomen. Verder werd er ook gebruik gemaakt van rapporten over kwetsbaarheden in de basisfunctionaliteit en toegevoegde componenten van ieder content management systeem. Deze komen voornamelijk van de *Common Vulnerabilities and Exposures* database. De informatie die deze database bevat werd gegroepeerd per jaar, er werd onderzocht over welk soort kwetsbaarheden het ging en of de kwetsbaarheid zich in de basisfunctionaliteit of toegevoegde componenten bevond. Aan de hand van deze statistieken kregen we een beter overzicht van de voornaamste kwetsbaarheden en waar deze zich

vooral bevonden. Verder hebben we ook statistieken opgemaakt die het aantal gewone en beveiligingsupdates van ieder CMS vergelijken.

## OWASP Top 10

In deze sectie zullen we beknopt de veiligheidsrisico's die omschreven worden in de OWASP Top 10 bespreken en technieken aanhalen die gebruikt kunnen worden om ze te voorkomen.

### Injection

Injection kwetsbaarheden zijn volgens de OWASP Top 10 de grootste risicofactor voor web applicaties. SQL injection doet zich voor wanneer een kwaadwillige gebruiker in staat is om een SQL query zo te manipuleren dat ze niet langer de bedoelde functionaliteit uitoefent [44]. Een voorbeeld van een succesvolle SQL injection aanval zou bijvoorbeeld kunnen zijn wanneer men in staat is om een SELECT query zo aan te passen dat de wachtwoorden van iedere gebruiker worden opgehaald in plaats van de zoekresultaten van een bepaalde zoekopdracht. Dit is mogelijk indien de query op een onveilige manier wordt opgebouwd. Als de query afhankelijk is van een of meerdere parameters die ingevuld worden door een gebruiker, kan ze eventueel vatbaar zijn voor een SQL injection aanval. De onbetrouwbare input van de gebruiker moet ofwel op voorhand gevalideerd en opgeschoond worden voor dat ze gebruikt wordt in de query, ofwel moet er gebruik gemaakt worden van zogenaamde *Prepared Statement*. In beide gevallen wordt ervoor gezorgd dat SQL injection aanvallen gemitigeerd worden.

### Broken Authentication and Session Management

Dit veiligheidsrisico houdt in dat een kwaadwillige gebruiker in staat zou kunnen zijn om de account van een andere gebruiker over te nemen doordat er gebrekkige sessiebeheer- en authenticatiemechanismen aanwezig zijn. Sessies zorgen ervoor dat gebruikers zich kunnen authenticeren en dat hun handelingen bewaard kunnen blijven gedurende een bepaalde periode [3] (bijvoorbeeld het toevoegen van producten aan een winkelwagen in een online webwinkel). Indien de aanvaller erin slaagt om de account van iemand anders over te nemen is hij in staat om alles te doen wat de gebruiker zelf zou kunnen doen. Als bijvoorbeeld de sessie van een bevoorrechte account gestolen wordt, kan de aanvaller gebruik maken van administratieve functionaliteit.

De twee volgende voorbeelden van aanvallen die gerelateerd zijn aan sessiebeheer stellen een aanvaller in staat om de sessie van een gebruiker over te nemen. Deze aanvallen zijn ook sterk gerelateerd aan het volgende veiligheidsrisico uit de OWASP Top 10: Cross-Site Scripting.

- **Session hijacking** – Bij dit type van aanval kan een aanvaller de sessie van een gebruiker overnemen door zijn sessie ID te stelen. Door deze

gestolen sessie informatie mee te sturen met ieder verzoek naar de webserver zal de webserver denken dat de verzoeken van de aanvaller afkomstig zijn van het slachtoffer.

- **Session fixation** – Bij dit type van aanval hoeft de aanvaller de sessie informatie van het slachtoffer niet te weten te komen. In plaats daarvan probeert hij ervoor te zorgen dat het slachtoffer sessie informatie gebruikt afkomstig van de aanvaller. Als het slachtoffer inlogt gebruikmakend van deze sessie informatie kan de aanvaller verzoeken doen bij de webserver in naam van het slachtoffer.

## Cross-Site Scripting (XSS)

Veel websites bevatten scripts die uitgevoerd worden door de browser. Cross-Site Scripting of XSS aanvallen houden in dat een aanvaller in staat is om zijn eigen script te injecteren in een anderzijds betrouwbare websites. De code is meestal geschreven in HTML/JavaScript, maar kan ook geschreven zijn in VBScript, ActiveX, Java, Flash, of iedere andere technologie die door de browser ondersteund wordt [58]. XSS veiligheidsrisico's doen zich voor wanneer een web applicatie onbetrouwbare gebruikersinput rechtstreeks gebruikt in de output van de website, denk bijvoorbeeld aan het herhalen van de zoekterm bij het tonen van de resultaten van een zoekopdracht. De browser heeft geen idee dat het script dat geïnjecteerd wordt niet betrouwbaar is en voert het dus gewoon uit. Omdat het script vanuit de webpagina zelf wordt uitgevoerd, heeft het toegang tot cookies, sessie tokens en alle andere objecten die bevat zitten in de HTML pagina.

Er bestaan meerdere types van XSS aanvallen. De vaakst voorkomende zijn persistente (of opgeslagen) en niet-persistente (of gereflecteerde) XSS aanvallen. Een persistente XSS aanval houdt in dat een aanvaller erin slaagt om zijn kwaadaardig script in de database terecht te krijgen. De web applicatie veronderstelt dat dit legitieme gegevens zijn en toont deze dus gewoon aan iedereen die deze gegevens kan en wil zien. Niet-persistente XSS aanvallen houden in dat het kwaadaardig script niet in de database wordt opgeslagen, maar mee terug gestuurd wordt in de respons van de webserver. Denk bijvoorbeeld aan het vorige voorbeeld waarbij de zoekterm van een zoekopdracht wordt herhaald bij het tonen van de resultaten. Als de aanvaller een script kan injecteren als zoekterm, dan wordt deze in de webpagina opgenomen en uitgevoerd.

Er bestaan een aantal methodes om XSS kwetsbaarheden tegen te gaan. Deze houden vooral in dat onbetrouwbare gegevens gefilterd moeten worden afhankelijk van de HTML context<sup>1</sup> waarin ze terecht zullen komen. Het is ook aangeraden om input te valideren aan de hand van een aantal verwachte eigenschappen. Bijvoorbeeld de lengte van een inputveld limiteren, valideren dat input die verwacht wordt numeriek te zijn dit inderdaad is en bij een voorgedefinieerde selectie valideren dat de waarde die verzonden werd effectief een van de geldige opties is.

---

<sup>1</sup>body, attribute, JavaScript, CSS of URL

## Insecure Direct Object References

Het is goed mogelijk dat niet alle objecten van een website publiek toegankelijk mogen zijn. Deze gegevens enkel verbergen van bezoekers die niet de juiste autorisatie hebben is niet voldoende. Indien een bezoeker rechtstreeks toegang probeert te krijgen tot deze objecten moet er gevalideerd worden of zij hiervoor de juiste rechten hebben. Een aanvaller mag bijvoorbeeld niet zomaar het profiel van een andere bezoeker kunnen aanpassen door de ID parameter in de URL aan te passen naar die van een andere gebruiker. Er zouden sterke toegangscontrole mechanismen aanwezig moeten zijn die de rechten van een bezoeker nagaan en die aftoetsen of hij de resource mag bekijken.

## Security Misconfiguration

Een web applicatie bestaat meestal uit een aantal samenwerkende componenten, zeker in het geval van CMS'en die uitbreidbaar zijn met plug-ins, thema's en andere toegevoegde componenten. Ieder van deze componenten maakt het mogelijk dat er veiligheidsrisico's kunnen optreden in de web applicatie. Daarom is het belangrijk dat de standaard instellingen veilig zijn, zodat het CMS meteen veilig gebruikt kan worden. Dit houdt bijvoorbeeld in dat onnodige features automatisch uitgeschakeld zijn, er geen default accounts en paswoorden aanwezig zijn en dat het tonen van foutrapporten automatisch uitgeschakeld is.

## Sensitive Data Exposure

Gevoelige gegevens enkel afgeschermd opslaan, bijvoorbeeld in een database, is niet voldoende. Het mag niet mogelijk zijn dat gevoelige gegevens gelezen kunnen worden als een kwaadwillige gebruiker op een of andere manier toegang kan krijgen tot de database. Dit betekent dat deze geëncrypteerd moeten worden opgeslagen. De gebruikte encryptie algoritmen moeten zelf ook veilig genoeg zijn zodat de originele gegevens niet achterhaald kunnen worden. De veiligheid van hash waarde die resulteert uit het encrypteren van gegevens mag niet rechtstreeks afhankelijk zijn van de originele gegevens. Bij paswoorden komt het bijvoorbeeld vaak voor dat men simpele paswoorden kiest die gemakkelijk zijn om te onthouden. Twee paswoorden die geëncrypteerd worden gebruikmakend van hetzelfde algoritme, zullen resulteren in dezelfde hash. Dit maakt het bijvoorbeeld mogelijk om hashes te vergelijken met waarden in een *rainbow table* van hashes van typische paswoorden. Om dit te voorkomen kan er bijvoorbeeld een willekeurige *salt* waarde toegevoegd worden aan de gegevens die geëncrypteerd moeten worden. Door een salt toe te voegen is het niet langer mogelijk om de hashes zomaar te vergelijken met de waarden in een rainbow table.

## Missing Function Level Access Control

Dit veiligheidsrisico is sterk gerelateerd aan *Insecure Direct Object References*, maar specifieker gericht op functionaliteit. Beide risico's kunnen echter gemitigeerd worden door gebruik te maken van goede toegangscontrole mechanismen.

Zoals eerder vermeld is enkel het verbergen van functionaliteit niet voldoende. Er moeten achterliggend in het systeem ook sterke mechanismen aanwezig zijn die nagaan welke rechten een bepaalde gebruiker heeft en of hij de juiste rechten heeft voor het uitvoeren van bepaalde acties.

## **Cross-Site Request Forgery (CSRF)**

Dit type veiligheidsrisico houdt in dat een aanvaller ervoor kan zorgen dat een gebruiker een bepaalde actie uitvoert op een betrouwbare web applicatie, zonder dat hij hier besef van heeft [41]. Hiervoor hoeft de aanvaller de sessie informatie van zijn slachtoffer niet te weten te komen, maar maakt hij misbruik van het feit dat de browser deze automatisch meestuurt met ieder verzoek. De web applicatie heeft zelf geen besef van de legitimiteit van het verzoek, dus verwerkt deze gewoon als enig ander verzoek. Als een aanvaller erin slaagt om een bevoorrechte account zijn verzoeken te laten uitvoeren, kan de hele applicatie gecompromitteerd worden. CSRF kwetsbaarheden worden meestal verholpen door zogenaamde anti-CSRF tokens, dewelke aan de gebruiker gekoppeld zijn en meegestuurd worden bij het uitvoeren van bepaalde kritieke functionaliteit. Indien een verzoek geen correct token bevat, wordt het genegeerd.

## **Using Components with Known Vulnerabilities**

Content management systemen kunnen vaak uitgebreid worden met plug-ins en toegevoegde componenten die extra functionaliteit toevoegen. Deze toegevoegde componenten kunnen op hun beurt allerlei veiligheidsrisico's veroorzaken die zich niet voordoen in de basisfunctionaliteit van het CMS. Het is dus van groot belang dat deze toegevoegde componenten worden nagezien op beveiliging voordat ze beschikbaar worden gesteld aan het publiek. Daarnaast is het ook belangrijk dat deze blijvend onderhouden worden door de ontwikkelaar, zodat eventuele veiligheidsrisico's opgelost worden. Het blijft wel aan degene die de website onderhoudt om alle toegevoegde componenten up to date te houden zodat de website veilig blijft.

## **Unvalidated Redirects and Forwards**

Gebruikers worden vaak omgeleid naar andere pagina's, bijvoorbeeld om na het inloggen op de homepage van de website terecht te komen. Er moet voorzichtig met deze omleidingen omgegaan worden, zeker indien de omleidingslocatie gebaseerd is op gegevens die door de gebruiker kunnen worden aangepast (bijvoorbeeld in een GET parameter van de URL). Er moet vooral opgepast worden indien deze omleidingslocatie een extern domein betreft. De webpagina op deze externe locatie zou mogelijk gegevens kunnen stelen van de gebruiker. Verder moet er opgepast worden dat de gebruiker de juiste rechten heeft om naar de pagina omgeleid te mogen worden. Anders zou het bijvoorbeeld mogelijk kunnen zijn dat de toegangscontrole mechanismen omzeild kunnen worden en er rechtstreeks toegang verstrekt kan worden tot administratieve functionaliteit.



## Resultaten

Door de source code en API documentatie van ieder CMS te doorzoeken werd duidelijk dat er voor ieder van de veiligheidsrisico's uit de OWASP Top 10 wel een of ander mechanisme aanwezig is om deze risico's te voorkomen. Ondanks dat WordPress, Joomla en Drupal allen op PHP gebaseerd zijn, zijn er soms toch opmerkelijke verschillen. Joomla en Drupal gebruiken bijvoorbeeld een *Database Abstractie Laag* (DBAL) zodat de CMS'en gebruikt kunnen worden met meerdere types van databases. WordPress vereist daarentegen dat een MySQL database gebruikt wordt. Als SQL injection preventietechniek gebruiken WordPress en Joomla een soortgelijke aanpak door onbetrouwbare input te valideren en op te schonen voordat ze in de query geplaatst wordt en de query uitgevoerd wordt. Drupal maakt gebruik van prepared statements en plaatst gewoon de ruwe data in de database. In Drupal moet deze data dus worden opgeschoond voordat ze getoond wordt op een webpagina.

Terwijl Joomla en Drupal gebruikmaken van PHP sessies en sessie cookies, maakt WordPress enkel gebruik van cookies. Zelfs tussen Joomla en Drupal zijn er verschillen in de voorziene sessiebeheertechnieken aangezien Joomla gebruik maakt van standaard PHP sessies en sessie ID's, terwijl Drupal zelf sessie ID's genereert en deze opslaat in de sessie. Basisfunctionaliteit en toegevoegde componenten van WordPress en Joomla up to date houden is ook aanzienlijk gemakkelijker dan in Drupal. Drupal vereist dat de beheerder manueel alle oude bestanden verwijdert en vervangt door de nieuwe, terwijl WordPress en Joomla hiervoor geautomatiseerde systemen voorzien.

Om XSS tegen te gaan voorzien ze alle drie verschillende filters, gebaseerd op de HTML context waarin de gegevens terecht zullen komen. WordPress en Drupal gebruiken standaard white-list input validatie, terwijl Joomla gebruik maakt van black-list input validatie. Verder voorzien ze ook mechanismen waarbij de rechten van de al dan niet geregistreerde bezoekers kunnen worden bepaald. WordPress voorziet een systeem van *Roles* en *Capabilities*, Joomla een systeem van *User Groups*, *Access Levels*, *Actions* en *Permissions* en Drupal een systeem van *Roles* en *Permissions*. Deze mechanismen zijn standaard vrij uitbreidbaar en aanpasbaar in Joomla en Drupal, maar niet in WordPress. WordPress voorziet standaard zes gebruikersgroepen waar bepaalde gebruikers aan gekoppeld kunnen worden. Deze mechanismen zorgen er in ieder CMS voor dat objecten en functionaliteit niet vrij toegankelijk zijn voor iedereen, maar toegang beperkt kan worden afhankelijk van de rol van een gebruiker binnen de website.

Al de besproken CMS'en voorzien ook paswoord encryptiealgoritmen zodat deze veilig kunnen worden bewaard in de database. WordPress en Drupal maken gebruik van het *Portable PHP Password Hashing* (phpass) framework, dat technieken bevat zoals salting en stretching. Stretching zorgt ervoor dat het encryptiealgoritme vaker wordt uitgevoerd en dat brute force attacks bijgevolg dus minder snel werken. Joomla gebruikt een eigen encryptiealgoritme dat ook gebruik maakt van salting. Een opmerkelijk feit is wel dat zowel WordPress als Joomla de minst veilige encryptiemethode kiezen van het phpass framework. Deze methode maakt standaard gebruik van MD5. Drupal gebruikt een aangepaste versie van het phpass framework en maakt gebruik van SHA-512 in

plaats van MD5 als onderliggend encryptiealgoritme.

De CMS'en voorzien ook technieken om CSRF aanvallen tegen te gaan. Joomla en Drupal maken gebruik van anti-CSRF tokens die gekoppeld zijn aan de sessie van de gebruiker en dus opnieuw gegenereerd worden als sessie van de gebruiker vernieuwt wordt. WordPress maakt daarentegen gebruik van *nonces*, tokens die 24u geldig blijven, zelfs als de sessie van de gebruiker verloopt.

We besluiten met het feit dat al deze CMS'en veelvuldig gebruik maken van *redirects and forwards*. WordPress voorziet hiervoor meerdere functies, waarbij de `wp_safe_redirect` functie gebruikt worden voor omleidingen die afhankelijk zijn van GET parameters. Deze functie laat omleidingen naar externe webpagina's enkel toe indien ze in een lijst van toegestane domeinen voorkomen. Deze lijst bevat standaard enkel het domein waarop de website zich bevindt. Drupal laat standaard geen omleidingen toe naar externe domeinen indien deze locatie verkregen werd van een GET parameter. In Joomla is dit echter wel mogelijk, aangezien hierin de enige voorwaarde is dat de locatie in de GET parameter geëncodeerd is gebruikmakend van Base64.

## Kwestbaarheidstatistieken

Zoals vermeld in de inleiding hebben we informatie van de *Common Vulnerabilities and Exposures* (CVE) database gebruikt om erachter te komen welk type kwetsbaarheden zich het vaakst voordoen en of deze zich over het algemeen vaker voordoen in de basisfunctionaliteit dan wel in toegevoegde componenten. Door de kwetsbaarheidsrapporten te groeperen per jaar en de omschrijvingen te analyseren, slaagden we voor ieder CMS erin om een overzicht te krijgen van deze factoren. Weinig opmerkelijk is het feit dat er aanzienlijk meer kwetsbaarheden ontdekt en gerapporteerd worden in toegevoegde componenten dan in de basisfunctionaliteit.

Toegevoegde componenten worden in ieder van de CMS'en wel manueel en automatisch nagekeken, maar er kunnen altijd kwetsbaarheden over het hoofd gezien worden. De basisfunctionaliteit van de CMS'en doorstaat verscheidene ontwikkelingsfasen waardoor het minder waarschijnlijk is dat er hierbij veel kwetsbaarheden achter zullen blijven in de finale release.

In WordPress zijn er een aanzienlijk aantal XSS kwetsbaarheden in zowel basisfunctionaliteit als in toegevoegde componenten. In Joomla overheersten SQL injection kwetsbaarheden vooral in vroegere jaren. Tegenwoordig zijn kwetsbaarheidsrapporten over Joomla schaarser in de CVE, maar steken XSS kwetsbaarheden meer de kop op. Voor Drupal hebben we niet enkele gekeken naar de gegevens in de CVE, maar ook naar Drupal's eigen *Security Advisories*. Dit zijn gegevens over kwetsbaarheden in Drupal die ze zelf publiek aankondigen. We hebben deze statistieken erbij gehaald omdat er vrij weinig kwetsbaarheidsrapporten over de basisfunctionaliteit van Drupal bevat waren in de CVE. Een van de voornaamste bronnen van kwetsbaarheden in toegevoegde componenten van Drupal zijn XSS kwetsbaarheden. Dit zou verklaard kunnen worden door het feit dat ruwe data rechtstreeks wordt opgeslagen in de database en actief opgeschoond moet worden voordat deze getoond wordt op de webpagina. Deze filters kunnen gemakkelijk vergeten worden door ontwikkelaars en

testers, waardoor het aantal XSS kwetsbaarheden aanzienlijk kan toenemen. In de vroegere jaren van het bestaan van Drupal waren voornamelijk XSS kwetsbaarheden veel voorkomend in de basisfunctionaliteit. Tegenwoordig worden vooral kwetsbaarheden opgemerkt die te maken hebben met het omzeilen van toegangscontrole.

## Conclusie

Terwijl al de besproken CMS'en verschillende mechanismen en technieken voorzien om de veiligheidsrisico's van de OWASP Top 10 te voorkomen, is het toch opmerkelijk dat er kwetsbaarheden blijven gevonden worden. Sommige technieken zijn voor verbetering vatbaar, bijvoorbeeld het gebruik van *nonces* door WordPress of het toestaan van omleidingen naar externe domeinen in *Joomla*. Hoe dan ook is het zeer belangrijk dat de website beheerder steeds alle basisfunctionaliteit en toegevoegde componenten up to date houdt, en dat hij enkel componenten gebruikt die betrouwbaar zijn. Voor onervaren gebruikers zijn daarom CMS'en zoals Joomla en WordPress meer aanbevolen, aangezien deze gebruiksvriendelijker zijn. Ze voorzien namelijk geautomatiseerde updatemechanismen waardoor de beheerder zich zelf geen zorgen hoeft te maken over de werking van het updateproces.

# Contents

<b>Preface</b>	<b>1</b>
<b>Abstract</b>	<b>2</b>
<b>Nederlandse samenvatting</b>	<b>3</b>
<b>1 Introduction</b>	<b>17</b>
1.1 Problem statement and type of thesis . . . . .	18
1.2 OWASP . . . . .	18
1.2.1 OWASP Top 10 . . . . .	18
1.3 Common Vulnerabilities and Exposures (CVE) Database . . . . .	19
<b>2 OWASP Top 10</b>	<b>21</b>
2.1 Injection . . . . .	21
2.1.1 What is SQL injection . . . . .	21
2.1.2 Dangers of SQL injection . . . . .	23
2.1.3 Preventing SQL injection . . . . .	23
2.2 Broken Authentication and Session Management . . . . .	28
2.2.1 Dangers of broken authentication and session management	28
2.2.2 Preventing authentication and session management flaws	29
2.3 Cross-Site Scripting (XSS) . . . . .	29
2.3.1 Types of XSS . . . . .	30
2.3.2 Dangers of XSS . . . . .	31
2.3.3 Preventing XSS . . . . .	33
2.4 Insecure Direct Object References . . . . .	34
2.4.1 Dangers of insecure direct object references . . . . .	34

2.4.2	Preventing insecure direct object references . . . . .	34
2.5	Security Misconfiguration . . . . .	34
2.5.1	Dangers of security misconfiguration . . . . .	35
2.5.2	Preventing security misconfiguration . . . . .	35
2.6	Sensitive Data Exposure . . . . .	35
2.6.1	Dangers of sensitive data exposure . . . . .	36
2.6.2	Preventing sensitive data exposure . . . . .	36
2.7	Missing Function Level Access Control . . . . .	37
2.7.1	Dangers of missing function level access control . . . . .	37
2.7.2	Preventing missing function level access control . . . . .	37
2.8	Cross-Site Request Forgery (CSRF) . . . . .	37
2.8.1	Dangers of cross-site request forgery . . . . .	38
2.8.2	Preventing cross-site request forgery . . . . .	38
2.9	Using Components with Known Vulnerabilities . . . . .	39
2.9.1	Dangers of using components with known vulnerabilities .	39
2.9.2	Preventing using components with known vulnerabilities .	40
2.10	Unvalidated Redirects and Forwards . . . . .	41
2.10.1	Dangers of unvalidated redirects and forwards . . . . .	41
2.10.2	Preventing unvalidated redirects and forwards . . . . .	41
2.11	Conclusion . . . . .	41
<b>3</b>	<b>Content Management Systems</b>	<b>45</b>
3.1	WordPress . . . . .	45
3.1.1	WordPress Core Functionality . . . . .	46
3.2	Joomla . . . . .	50
3.2.1	Joomla Core Functionality . . . . .	50
3.3	Drupal . . . . .	56
3.3.1	Drupal Core Functionality . . . . .	57
<b>4</b>	<b>CMSs and the OWASP Top 10</b>	<b>63</b>
4.1	WordPress . . . . .	63
4.1.1	Injection . . . . .	64
4.1.2	Authentication and session management . . . . .	66
4.1.3	Cross-site scripting . . . . .	69

4.1.4	Direct object references . . . . .	70
4.1.5	Security misconfiguration . . . . .	71
4.1.6	Sensitive data exposure . . . . .	72
4.1.7	Function level access control . . . . .	72
4.1.8	Cross-site request forgery . . . . .	73
4.1.9	Components with known vulnerabilities . . . . .	74
4.1.10	Redirects and forwards . . . . .	75
4.2	Joomla . . . . .	76
4.2.1	Injection . . . . .	76
4.2.2	Authentication and session management . . . . .	78
4.2.3	Cross-site scripting . . . . .	78
4.2.4	Direct object references . . . . .	81
4.2.5	Security misconfiguration . . . . .	82
4.2.6	Sensitive data exposure . . . . .	84
4.2.7	Function level access control . . . . .	84
4.2.8	Cross-site request forgery . . . . .	85
4.2.9	Components with known vulnerabilities . . . . .	86
4.2.10	Redirects and forwards . . . . .	87
4.3	Drupal . . . . .	89
4.3.1	Injection . . . . .	89
4.3.2	Authentication and session management . . . . .	91
4.3.3	Cross-site scripting . . . . .	92
4.3.4	Direct object references . . . . .	94
4.3.5	Security misconfiguration . . . . .	94
4.3.6	Sensitive data exposure . . . . .	96
4.3.7	Function level access control . . . . .	97
4.3.8	Cross-site request forgery . . . . .	98
4.3.9	Components with known vulnerabilities . . . . .	98
4.3.10	Redirects and forwards . . . . .	100
4.4	Conclusion . . . . .	101

<b>5</b>	<b>Comparing Content Management Systems</b>	<b>103</b>
5.1	OWASP Top 10 . . . . .	103
5.1.1	Conclusion . . . . .	111
5.2	Vulnerability Database . . . . .	112
5.2.1	Conclusion . . . . .	119
<b>6</b>	<b>Conclusion</b>	<b>121</b>
<b>A</b>	<b>Portable PHP Password Hashing Framework</b>	<b>123</b>
A.1	Validating passwords . . . . .	124

# List of Figures

3.1	WordPress 3.4 user registration form . . . . .	47
3.2	Joomla user registration form . . . . .	51
3.3	The hierarchy of Joomla User Groups . . . . .	54
3.4	An overview of Joomla extensions [27] . . . . .	56
3.5	Drupal 7.x user registration form . . . . .	58
3.6	Granting Permissions to certain Roles in Drupal 7.x . . . . .	61
4.1	WordPress update indicator for the <i>Akismet</i> plug-in . . . . .	71
4.2	WordPress update indicator for the <i>Twenty Eleven</i> theme . . . . .	72
4.3	Joomla Extension Manager . . . . .	86
4.4	Drupal Warning System for important issues . . . . .	96
5.1	WordPress Vulnerabilities Types and Core/Extension Ratio (CVE)	113
5.2	WordPress Core Releases [71] . . . . .	114
5.3	Joomla Vulnerabilities Types and Core/Extension Ratio (CVE) .	115
5.4	Joomla Core Releases . . . . .	116
5.5	Drupal Vulnerabilities Types and Core/Extension Ratio (CVE) .	117
5.6	Drupal Core Security Advisories . . . . .	118
5.7	Drupal Core Releases . . . . .	119



# List of Tables

2.1	<code>mysqli_real_escape_string()</code> characters <sup>2</sup> . . . . .	26
4.1	WordPress's Data Validation Filters <sup>3</sup> . . . . .	70
4.2	Secure Conversions from One Text Type to Another <sup>4</sup> . . . . .	93
5.1	Comparing content management systems based on the OWASP Top 10 (Part 1) . . . . .	107
5.2	Comparing content management systems based on the OWASP Top 10 (Part 2) . . . . .	108
5.3	Comparing content management systems based on the OWASP Top 10 (Part 3) . . . . .	109
5.4	Comparing content management systems based on the OWASP Top 10 (Part 4) . . . . .	110
5.5	Comparing content management systems based on the OWASP Top 10 (Part 5) . . . . .	111

# Chapter 1

## Introduction

The world is becoming more and more interconnected every day through the use of the internet. Many people want their own website to promote their products or share their ideas, but not everyone wants or needs to know exactly how to build a secure website from scratch. Open source web content management systems enable anyone to build and maintain their own website, even without having any programming knowledge. Content management systems provide a basic framework for users to build upon, and are usually highly customisable to fit the user's needs. Default functionality can be extended by plug-ins, and other added components like themes and templates can be installed to give the website a totally different look and feel. These frameworks are expected to be well secured against common security risks, but as we shall see in Chapter 5, this isn't necessarily the case. New exploits in CMS core functionality are reported frequently and even more so for vulnerabilities found in added components. One of the major issues of using the same web content management systems is that a large number of websites will be vulnerable to the same security exploit if one is discovered.

In a study by W3Techs about the usage statistics of web content management systems [57], WordPress, Joomla and Drupal came out on top as the three most used web content management systems available today. This is why we will analyse the security for these CMSs based on a set of objective criteria. In the next subsection, we will provide an introduction to *The Open Web Application Security Project* (OWASP), an organisation that focusses on improving the security of software. OWASP provides the OWASP Top 10 project, which is a list of the ten most critical web application security risks. We provide a more in depth overview of the security risks on the list in Chapter 2, together with common best practices and mitigation techniques. This overview will help us assess the security of the content management systems' core functionality.

Lastly in this introductory chapter, we will take a look at the Common Vulnerabilities and Exposures Database [36], which contains a large number of real-world security vulnerabilities reports. We have scraped and analysed the information concerning the web content management systems we will describe in Chapter 3 from this database. The results of this analysis can be found in

Chapter 5, in which we will compare the content management systems to each other based on the criteria set forth in Chapter 2. A final conclusion will be provided in Chapter 6.

## 1.1 Problem statement and type of thesis

This thesis tries to answer the question "How safe are open source web content management systems". We hope to answer this question by providing a *comparative study* between the three most used open source web content management systems. By basing our research on the OWASP Top 10, vulnerability reports, and core release statistics, we have provided a solid set of objective criteria to form the basis of this comparative study.

## 1.2 OWASP

The *Open Web Application Security Project* (OWASP) is an organization that focusses on improving the security of software. They provide information so that individuals and organizations can make informed decisions about software risks. OWASP is a free and open community of security experts sharing their knowledge of the software security domain, dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted [37]. OWASP supports many initiatives, including the WebGoat project<sup>1</sup>; a deliberately insecure web application to teach web application security lessons, the Zed Attack Proxy Project<sup>2</sup>; an integrated penetration testing tool for finding vulnerabilities in web applications, and the OWASP Top 10; an awareness document listing the 10 most critical web application flaws<sup>3</sup>. The OWASP Top 10 will be used as a basis of comparison for the different web content management systems that will be discussed. In the next subsection, we will provide more information about the OWASP Top 10, why it exists and why we will use it as a basis of comparison.

### 1.2.1 OWASP Top 10

The OWASP Top 10 is a list of the ten most critical web application security risks. The most recent version of the OWASP Top 10 was released in June of 2013. The current release is based on 8 datasets from 7 firms that specialize in application security [1]. This data spans over 500000 vulnerabilities across hundreds of organisations and thousands of applications. The items in the Top 10 are selected and prioritized according to this prevalence data, in combination with consensus estimates of exploitability, detectability, and impact estimates [1]. The Top 10 provides techniques and best practices to protect against high risk problem areas. The primary aim of the OWASP Top 10 is to

---

<sup>1</sup>[https://www.owasp.org/index.php/Category:OWASP\\_WebGoat\\_Project](https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project)

<sup>2</sup>[https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)

<sup>3</sup>[https://www.owasp.org/index.php/Category:OWASP\\_Top\\_Ten\\_Project](https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project)

educate people about the consequences of the most important web application security weaknesses.

Each new release contains release notes, describing which changes were made compared to the previous version, and why the change was made. Usually these changes are limited to reordering certain items on the list, and broadening certain other areas to include more use cases. For instance, Cross-Site Request Forgery (CSRF) attacks moved down in prevalence based on the datasets used for the previous release of the OWASP Top 10, which was released in 2010. It is believed that this is because CSRF has been in the OWASP Top 10 for 6 years, causing organisations and framework developers to have focussed on it significantly, reducing the number of CSRF vulnerabilities in web applications [1]. The current Top 10 consists of the following security risks:

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

In Chapter 2, we will go into more detail about each of these security risks, providing a definition, the dangers, and ways to mitigate each risk. We will use this as a basis of comparison in Chapter 4, and find out which techniques and best practices are upheld by each of the content management systems described in the chapter.

### 1.3 Common Vulnerabilities and Exposures (CVE) Database

The Common Vulnerabilities and Exposures (CVE) Database is a dictionary of publicly known information security vulnerabilities and exposures [36]. It contains examples of real-world security issues detected in all sorts of software, including web applications. We chose this database because other vulnerability databases, such as the Exploit Database<sup>4</sup>, often refer to the CVE, and it was the largest repository of vulnerability reports for each CMS. We have used this

---

<sup>4</sup><http://www.exploit-db.com/>

database to obtain an overview of which and how many security issues are detected and reported in the open source content management systems described in Chapter 3. These vulnerability reports encompass both core and added components.

Vulnerabilities in the CVE have a certain identifier which contains the year in which the exploit was reported. This allowed us to group the vulnerability reports by year and get a decent overview of the yearly number of reported vulnerabilities. By analysing the vulnerability descriptions, we were able to identify which types of vulnerabilities were reported, and if the vulnerability concerned functionality in CMS core or added components. The results of this analysis can be found in Chapter 5.

## Chapter 2

# OWASP Top 10

In the introductory chapter, we gave more information about the OWASP Top 10 project. In this chapter, we will take a look at each item in the Top 10 and provide more information about each of them. This section combines the OWASP Top 10 with the dangers of each flaw and a collection of mitigation and prevention techniques. This information will be necessary in Chapter 4 when we analyse the security of each of the CMSs based on these techniques and best practices.

### 2.1 Injection

Injection attacks are first on the list of the OWASP Top 10. This is due to the fact that injection vulnerabilities are quite trivial to exploit [21]. Injection attacks aren't limited to attacks on databases; everything with a command interface that combines data into a command is susceptible [44]. This includes SQL, LDAP, Operating System, XPath, XQuery, Expression Language, and many more. Cross-site scripting attacks, which will be explained in Section 2.3, are also a form of injection. In this section, however, we will focus primarily on injection attacks targeted at databases. Successful attacks are possible even when one only has basic knowledge of the database query language (e.g. SQL). Not only web applications are vulnerable to this type of injection. For instance, SQL injection can also occur in desktop applications that use SQL server backends [21].

#### 2.1.1 What is SQL injection

An SQL injection attack occurs when a malicious user is able to trick the database into executing malicious queries by inserting, or *injecting*, SQL into the application, whether directly in the input fields of the application, or by tampering with cookie data. A successful exploit can expose sensitive data from the database, modify database data or give the malicious user access to administration operations on the database [46].

An application is vulnerable to SQL injection when unsanitized user input data is put directly into a database query and then fed to the database. Example 2.1 demonstrates an unsafe database query in PHP.

```
1 $username = $_GET[ 'username' ];
2 $original_query =
3  "SELECT * FROM users WHERE username = '$username';"
```

Example 2.1: Unsanitized input data, put directly into an SQL query

In this example, the HTTP request's GET parameter is put directly into the query. This query works well as long as everyone uses the application as intended, but imagine the scenario where someone inputs the following code into the application's *username* field: ' or '1'='1. Notice that the input string starts with a single quote but ends without one. This is to align the number of quotes with the ones that are already there in the original query. Lines 5 and 6 of Example 2.2 show how the original query gets altered by the malicious user's input.

```
1 $username = $_GET[ 'username' ];
2 $original_query =
3  "SELECT * FROM users WHERE username = '$username';"
4
5 $altered_query =
6  "SELECT * FROM users WHERE username = '' or '1'='1';"
```

Example 2.2: SQL SELECT query gets altered by user input

By inputting ' or '1'='1, the query gets altered in such a way that the SQL query is still valid, but the result of the query gives the malicious user access to user details he is not allowed to see. This query will now return every row from the *users* table where the username matches with an empty string, or where the following statement is true: '1'='1'. Because this statement is always true, all rows will be returned by the database and the malicious user is able to access all of this information.

Not only SELECT queries are susceptible to SQL injection. The same problem applies to INSERT or UPDATE queries, where one can adjust data in the database for their own benefit. For instance, one could change another user's password or give themselves admin privileges through SQL injection. Imagine the case where a malicious user tries to update their profile and sets the string **Jeff**, `admin='1` as their username. Lines 5 and 6 of Example 2.3 show how the original query gets altered by the malicious user's input.

```
1 $username = $_GET[ 'username' ];
2 $original_query =
3  "UPDATE users SET name='$username' WHERE ...;"
4
5 $altered_query =
```

6 | "UPDATE users SET name='Jeff ', admin='1' WHERE ...;" |

Example 2.3: SQL UPDATE query gets altered by user input

### 2.1.2 Dangers of SQL injection

It's obvious that leaving a website vulnerable to SQL injection can be a serious problem. If an attacker is able to exploit an SQL injection flaw, he could potentially steal valuable information or deface the entire web application.

For instance, if users' passwords are stored in plain text or stored using a weak hashing algorithm, an attacker could potentially steal all passwords through an SQL injection attack. This makes protecting sensitive data very important, as will be explained in Subsection 2.6. As people often use the same password for different accounts, an attacker could use the stolen passwords to log into the victims' e-mail or other accounts. Stealing other valuable information, such as credit card information or social security numbers, are just a few examples of potentially disastrous results of leaving a web application vulnerable to SQL injection.

### 2.1.3 Preventing SQL injection

There are multiple ways SQL injection can be prevented. Most of these techniques require very little effort and take little to no time to implement. In this subsection, we will list a number of SQL injection prevention methods, which we can refer to when we investigate which – if any – of these prevention techniques are used by the developers of the content management systems that are discussed in Chapter 3. This list of techniques is based on the OWASP *SQL Injection Prevention Cheat Sheet* [47]

#### Prepared statements

Prepared statements enable a developer to first define all SQL code, and pass in each parameter to the query later. Besides the fact that using prepared statements is safer than using static queries, they are also faster and more efficient. This is due to the fact that prepared statements need only be parsed (or prepared) once, but can be executed any number of times [19]. Executing a static query requires the database to analyze, compile and optimize the query each time the query is called. For complex queries this process can take up large amounts of time and can noticeably slow down an application. Prepared statements use less resources because they only go through this process once [19].

Using prepared statements enables the database to distinguish between code and data, regardless of user input. Any user-supplied input is treated as input data only and thus will never be able to alter the query. Example 2.4 shows a comparison of a typical static query and equivalent prepared statements. If an application only uses prepared statements, one can be reasonably sure that no SQL injection will occur.



```

1 //http://example.com/users.php?name=dave' or '1'='1
2 $name = $_GET['name'];
3
4 //Static query for selecting a user from the users table
5 $static_query =
6     "SELECT * FROM users WHERE name = '$name'";
7
8 //Prepared statement using ?
9 $stmt =
10     $dbh->prepare("SELECT * FROM users WHERE name = ?");
11 $stmt->execute(array($name));
12
13 //Named prepared statement
14 $stmt =
15     $dbh->prepare("SELECT * FROM users WHERE name = :n");
16 $stmt->bindParam(':n', $name);
17 $stmt->execute();

```

Example 2.4: Selecting a user from the *users* table (static query + prepared statement)

In this example, the static query on line 5 gets altered as shown in Example 2.2, because it uses the unsanitized user input directly. The prepared statement treats the entire string as user input and queries the database for a user whose name literally matches the string `dave' or '1'='1` and thus is not susceptible to SQL injection. In the last prepared statement (starting on line 14), a named placeholder is used instead of the positional `?` placeholder. Both prepared statements are equally valid and secure, but the one using named placeholders is more readable and easier to debug. When using question mark placeholders, care must be taken that the order in which the parameters are passed to the prepared statement is the same as the order of the question marks in the query [19].

## Stored procedures

Stored procedures are comparable to prepared statements and have the same effect when implemented safely [47]. Just like prepared statements, they also require the separation of code and data. The developer defines the code first and passes in the parameters later. The main difference between stored procedures and prepared statements is where they reside in the application. The SQL code for stored procedures is defined and stored in the database itself and called from the application [20], whereas prepared statements have to be passed to the database and prepared at least once every time they are needed. Stored procedures are equally adept at preventing SQL injection as their prepared statements counterparts, so it is up to the developer which approach they will use [47].

```

1 CREATE PROCEDURE p(IN id_val INT)
2 BEGIN

```

```

3 INSERT INTO test(id) VALUES(id_val);
4 END;
5
6 CALL p(1);

```

Example 2.5: Creating and executing a stored procedure in SQL

This example creates a stored procedure on a MySQL database. The procedure is named *p* and takes one argument *id\_val* of type integer. The body of the stored procedure is defined between the **BEGIN** and **END** keywords. In this case the body of the procedure causes the value of the *id\_val* argument to be inserted into a table named *test*.

Care must be taken when creating stored procedures, because they can still be made to be insecure. Example 2.6 shows how a stored procedure is used in an insecure way.

```

1 DELIMITER //
2 CREATE PROCEDURE QueryAnyTable (IN table_name VARCHAR
3   (100))
4 BEGIN
5 SET @query = CONCAT('SELECT * FROM ', table_name);
6 PREPARE stmt FROM @query;
7 EXECUTE stmt;
8 END;//
9 DELIMITER ;
10 CALL QueryAnyTable(' (SELECT * FROM... ) ')

```

Example 2.6: Insecure use of stored procedures<sup>1</sup>

In the example above, a stored procedure is created that concatenates the value of the *table\_name* variable to a **SELECT** statement. Seeing as any value could be entered, including another **SELECT** statement, this stored procedure is vulnerable to SQL injection.

### Escaping all user-supplied input

This SQL injection prevention technique relies on actively sanitizing input data before putting it in a query. User-supplied input data is escaped using the DBMS's character escaping scheme. If done correctly, the sanitized input data will no longer be a threat, thus avoiding SQL injection vulnerabilities. Care must be taken when using this technique, however, because it is easy to forget escaping user input, making your application vulnerable to SQL injection.

<sup>1</sup>Sources: <http://thedailywtf.com/Articles/For-the-Ease-of-Maintenance.aspx> and <http://www.slideshare.net/billkarwin/sql-injection-myths-and-fallacies>, slide 31

Name	Unsafe char	Hex	Escaped char
Null	NUL	0x00	\0
Line feed	LF	0x0A	\n
Carriage return	CR	0x0D	\r
Backslash	\	0x5c	\\
Single quote	'	0x27	\'
Double quote	"	0x22	\"
Control-Z	SUB	0x1A	\Z

Table 2.1: `mysqli_real_escape_string()` characters<sup>2</sup>

The following code example, Example 2.7, shows how to perform string escaping in PHP for a MySQL database server. The first line of this code sample sanitizes the user input data supplied in the *name* parameter by replacing all instances of the unsafe characters displayed in Table 2.1 by their respective escaped versions. Under certain circumstances, the sanitized input data can now be used directly in the query, without fear of SQL injection. As we shall see in Example 2.8, however, using this method isn't always safe.

```

1 $name = mysqli_real_escape_string($_GET['name']);
2
3 $query = "SELECT * FROM users WHERE name = '$name'";

```

Example 2.7: Escaping a MySQL query in PHP

Escaping user input does not always prevent SQL injection. This issue is shown in Example 2.8.

```

1 //http://www.example.com/users.php?id=1 or 1=1
2
3 $id = $_GET['id']; //1 or 1=1
4
5 $id = mysqli_real_escape_string($id);
6
7 $query = "SELECT * FROM users WHERE id = $id";

```

Example 2.8: Even though the user input data is escaped, this query is still vulnerable to SQL injection

In the example above, a malicious user is able to provide the following id to the script: `1 or 1=1`. Because the user input is assumed to be of type integer, its value is not put between quotes. Despite the input data being escaped, this query is still prone to SQL injection.

<sup>2</sup>Source: <http://php.net/manual/en/mysqli.real-escape-string.php>

## Least privilege

One can try to minimize the impact of a successful SQL injection attack by limiting the privileges assigned to the database accounts used in the DBMS. Giving everyone all privileges is easy, but it is very dangerous and should be avoided as much as possible [47]. Limiting database user privileges is not something a CMS can enforce in itself, but we will go over the minimal required privileges for each framework and compare them with what their installation guides suggest in Chapter 4.

## White list input validation

The act of white list input validation is a great way to limit the input a user (or attacker) can insert into the application. By using input validation, an attack can be prevented at the earliest possible stage, because unauthorized input can be detected before it is processed by the application [40]. White list input validation involves tightly defining which input is allowed and authorized. By definition, all other input is not authorized. When the type of input data expected is well structured data, like dates or e-mail addresses, a developer could easily define strong validation patterns, usually based on regular expressions [40], for validating such input. Examples of such regular expressions can be found in Example 2.9. If the expected input data is less well structured, defining validation patterns can be harder, but not impossible. For instance, when the input field to validate is a *free text* field, like a *place comment* text field, one can include only printable characters or define a maximum size for the input field. Another example might be checking if the input data received from a choice-based input field (e.g. a drop-down list generated with the HTML `<select>` tags) is actually one of the provided options and did not get altered in any way.

```
Validating Data from Free Form Text Field for Zip Code
(5 digits plus optional -4):
^\d{5}(-\d{4})?$

Validating a Free Form Text Field
for allowed chars (numbers, letters, whitespace, .-):
^[a-zA-Z0-9\s\.\-]+$ (Any number of characters)
^[a-zA-Z0-9\s\.\-]{1-100}$ (Limited to 1 to 100
characters)
```

Example 2.9: White list validation regex examples<sup>3</sup>

Another technique one could use to try and validate user supplied input could be using black list validation. Instead of validating user input against allowed strings and patterns, input is checked against a list of disallowed strings and patterns. The problem with black list validation, however, is that it is impossible to account for all different ways of writing a string and all various encoding

<sup>3</sup>Source: [https://www.owasp.org/index.php/Input\\_Validation\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet)

methods. It is much more likely for a malicious user to find some way to circumvent the black-listed input validation than it is for him to circumvent white list input validation [18]. For instance, even if the string ' or '1'='1 had been black-listed, one could easily bypass the black list validation by writing ' OR '1'='1, ' oR '1'= '1 or any other permutation of capitalisation or spacing.

## 2.2 Broken Authentication and Session Management

Second on the OWASP Top 10 list is *Broken Authentication and Session Management* [1]. It talks about the possibility that a malicious user might be able to steal accounts from others or disguise their own actions through malfunctioning or badly established session management and authentication mechanisms. If these systems are flawed, a malicious user might be able to take advantage of these leaks or flaws to impersonate other users. Such flaws may allow some or even all accounts to be attacked. If an attacker is able to exploit a leak or flaw successfully, the attacker can do anything the victim could do. It is for this reason that privileged accounts are targeted frequently [1].

### 2.2.1 Dangers of broken authentication and session management

We will provide some common causes for broken authentication and session management, and the dangers associated with them:

- **Weak account management functions** – Weak account management functions like account creation, change or recover password and weak session IDs might make it possible for a malicious user to guess or overwrite other users' credentials.
- **URL rewriting** – URL rewriting can be used as a session management technique. The web server appends the session ID as a GET parameter to every link on the web page that points to another page on the website. If a user clicks on a link containing such a URL, the session ID will be sent along with the request in the GET parameter, enabling the web server to determine who made the request. However, if someone were to share the URL of a web page with their session ID embedded in the URL, anyone clicking on the link would be identified as the user who shared the web page. Anyone who clicked on the link would be able to change the user's credentials. If the web page were to belong to an online web shop, anyone would be able to use the unsuspecting user's credit card to make purchases.
- **Session ID timeout** – If an application's timeouts aren't set properly and a user forgets to log out before leaving the web page, anyone using the computer after them will find themselves logged in as the first user.

- **Credentials stored as plain text** – If the web application doesn't encrypt or hash their users' login data and an attacker is able to gain access to the system's database, all passwords will be exposed to the attacker.

The following two examples of session attacks are very dangerous and enable attackers to completely take over a victim's session. As these attacks are also strongly related to XSS attacks, we will provide more details about these attacks in Subsection 2.3. For now, we will provide a brief summary of each attack:

- **Session hijacking** – An attacker tries to take control over a victim's session by stealing their session ID. He then uses this session ID to make the server think he is the victim. In Subsection 2.3 we will explain how an attacker can hijack his victim's session using XSS.
- **Session fixation** – In contrast to session hijacking, the attacker forces the victim to use a session ID that is known to him in advance. The goal of both attack vectors is the same: to take control over a victim's session. In Subsection 2.3 we will explain how an attacker can force his victim to use a certain session ID using XSS.

## 2.2.2 Preventing authentication and session management flaws

In order to prevent authentication and session management flaws, OWASP suggests meeting all the authentication and session management requirements defined in their *Application Security Verification Standard (ASVS)* and to have a simple authentication template for developers to build upon<sup>4</sup>. The ASVS provides a basis for testing application security controls and provides a list of requirements and best practices to protect against vulnerabilities [38].

Strong efforts must be made to prevent cross-site scripting (XSS) flaws. These attacks are used frequently to steal users' session information. Cross-site scripting attacks are third on the OWASP Top 10 list, and will be explained in the next section. Session hijacking and session fixation attacks are related to authentication and session management. XSS can be used as an attack vector for these attacks, as we will explain in Section 2.3.

## 2.3 Cross-Site Scripting (XSS)

Cross-Site Scripting (or XSS) attacks are a type of injection attack, wherein an attacker is able to inject malicious scripts into an otherwise benign website, generally in the form of a browser side script [39, 3]. The code is usually written in HTML/JavaScript, but could also be written in VBScript, ActiveX, Java, Flash, or any other browser-supported technology [58]. XSS vulnerabilities can

---

<sup>4</sup>The OWASP Top 10 document suggests using the ESAPI Authenticator and User APIs templates as a basis: [http://owasp-esapi-java.googlecode.com/svn/trunk\\_doc/latest/org/owasp/esapi/Authenticator.html](http://owasp-esapi-java.googlecode.com/svn/trunk_doc/latest/org/owasp/esapi/Authenticator.html)

occur when a web application uses user input in its output without validating or encoding the input first. An attacker abuses this vulnerability to inject his own malicious script. The browser has no way of knowing that the script should not be trusted, and will execute it. Because the script is executed from within the trusted web page, it can access cookies and session tokens and can even rewrite the content of the HTML page [39]. The code basically has the ability to read, modify and transmit any sensitive data accessible by the browser [58].

### 2.3.1 Types of XSS

There are 2 main types of XSS attacks: persistent (or stored) and non-persistent (or reflected). There is a third, less well known type (DOM based XSS) which we will not discuss, but which is mentioned for completeness. In this subsection, we will provide a little more detail about each of these types of XSS attacks and give some examples.

#### Persistent XSS (stored XSS)

In a persistent XSS attack, an attacker is able to inject his script by abusing a web application's ability for users to provide their own content (e.g. message boards or a comment field). This allows the attacker to store his script on the web server persistently, causing the server to send the malicious script to users viewing the content that contains the malicious script.

The full attack scenario is as follows:

1. The attacker sends a message containing the malicious script to the web server, which stores it in its database.
2. An unsuspecting user visits a page containing the message submitted by the attacker.
3. The server responds to the user's request and returns the web page containing the malicious script.
4. The user's browser sees the script code and executes it.

#### Non-persistent XSS (reflected XSS)

A non-persistent XSS attack, or reflected XSS attack, occurs when an attacker is able to exploit a vulnerability in the way a web page handles requests. The script code is not stored on the web server but rather reflected back to the client's browser as part of the server's response. For clarity, we will provide the following example: imagine a website that provides search functionality, for instance at `http://www.example.com/search.php?q=query`. When a search query is entered, the search results page usually shows something along the lines of "Your search for 'query' resulted in the following matches". The unsafe script that generated this output is shown in Example 2.10.

```

1 $query = $_GET["q"];
2
3 echo "<p>Your search for '" . $query . "' resulted in the
   following matches.</p>";

```

Example 2.10: PHP script for search results web page

As shown in Example 2.10, the user-provided search term is not handled correctly, resulting in an attacker’s ability to exploit this vulnerability. For instance, if one were to enter script code instead of a regular search query, the code would be contained in the resulting web page, and subsequently executed by the browser. An attacker could abuse this vulnerability by crafting a URL containing the malicious script and sending it to the victim. Having script code in the URL might arouse suspicion with the victim, so it’s likely that the attacker will URL encode the script [58].

**Without URL Encoding:**

`www.example.com/search.php?q=<script>alert('XSS');</script>`

**With URL Encoding:**

`www.example.com/search.php?q=%3C%73%63%72%69%70%74%3E%61%6C%65%72%74%28%27%58%53%53%27%29%3B%3C%2F%73%63%72%69%70%74%3E`

As demonstrated, the URL encoded script code, appended to the URL looks a lot less suspicious. When the victim visits the link, the malicious script will get reflected back from the server and will be contained in the server’s response. The victim’s browser will see the code and execute it.

The full attack scenario is as follows:

1. The attacker crafts a URL containing a malicious script and sends it to the victim, trying to trick them into opening it.
2. The victim opens the link and his browser makes a request to the server.
3. The server responds with the requested page and reflects back the malicious script code.
4. The user’s browser sees the script code and executes it.

### 2.3.2 Dangers of XSS

Because XSS attacks allow an attacker to get their malicious code executed from within a trusted domain, the script can access sensitive information within that domain. This gives the attacker the ability to read information from a trusted domain and send it to his own web server, where he can access and abuse this information [35]. In this subsection, we will briefly go over a few of the most dangerous consequences of XSS vulnerabilities. Two of these issues –



session hijacking and session fixation – were briefly mentioned in Section 2.2. In this subsection, we will take a closer look at both and explain how XSS vulnerabilities can be used as an attack vector for these attacks.

### Session hijacking

In a session hijacking attack, an attacker tries to take over a victim’s session by stealing their session ID and using it to make requests to the server. The server thinks these requests are coming from the victim, which allows the attacker to make changes to the victim’s account, steal sensitive information, or read their e-mails to name a few examples. We will now explain how XSS can be used as an attack vector for a session hijacking attack.

As mentioned previously, the injected script code is executed from within a trusted domain. If we assume that the attacker used JavaScript, the malicious code has access to the `cookie` attribute via the `document.cookie` command. He can subsequently send the cookie value to himself and use it to take over the victim’s session. If the web application doesn’t use cookies to store session information, but rather uses URL rewriting, the attacker would still be able to steal the session information. The script also has access to any link on the web page. Because URL rewriting is used, these links contain the session ID, which the attacker can steal and send to himself, successfully stealing the victim’s session [3]. A JavaScript example of stealing the victim’s `cookie` information can be found in Example 2.11

```
1 var url = "http://www.example.com/logcookie.php?cookie="
  + document.cookie;
2 var xmlhttp = new XMLHttpRequest();
3 xmlhttp.open( "GET", url, false );
4 xmlhttp.send( null );
```

Example 2.11: JavaScript example of a session hijacking XSS attack

In this JavaScript XSS attack example, the attacker appends the value of the `cookie` to the URL of a web page and makes an AJAX call to that URL. The `logcookie.php` web page that is called, logs the value of the `cookie` GET parameter, which the attacker can subsequently use to take over the victim’s session.

### Session fixation

As in a session hijacking attack, the session fixation attack allows the attacker to take control over the victim’s session. Instead of capturing the victim’s session ID, however, the attacker forces the victim to use a session ID that is known to him in advance. The victim’s browser attaches this session ID to every subsequent request to the server. If the victim logs in, the session ID will now be associated with the logged in victim. The attacker can make requests using

the same session ID. The server will think the victim is making the requests, enabling the attacker to impersonate the victim.

If we again assume that the attacker is using JavaScript, he can set or replace the session cookie with the desired value using the `document.cookie` or `cookie.write()` function. If URL rewriting is used instead of session cookies, the attacker could write JavaScript code to replace every session ID in every link on the web page with his own desired value. It doesn't matter if session management is implemented with cookies or through URL rewriting. Either way, an attacker could easily craft a script that makes session hijacking or session fixation attacks possible.

### 2.3.3 Preventing XSS

In this subsection, we will go over the most important XSS prevention techniques. The most important way XSS can be prevented is by escaping all untrusted data based on the HTML context<sup>5</sup> that the data will be placed into [1]. The OWASP XSS Prevention Cheat Sheet [48] provides a list of rules of how and when to escape data in order to prevent XSS vulnerabilities:

1. Never insert untrusted data except in allowed locations.
2. HTML escape before inserting untrusted data into HTML element content (tags such as `div`, `p`, `td`, etc.)
3. Attribute escape before inserting untrusted data into HTML common attributes (attributes such as `width`, `name`, `value`, etc.)
4. JavaScript escape before inserting untrusted data into JavaScript data values
5. CSS escape and strictly validate before inserting untrusted data into HTML style property values
6. URL escape before inserting untrusted data into HTML URL parameter values
7. Sanitize HTML markup with a library designed for the job
8. Prevent DOM-based XSS

It usually won't be necessary or possible to implement all these rules as not every use case will be applicable to all web applications. For instance, if a certain web application never uses untrusted data in CSS elements, rule #5 will not be applicable to that web application. For more information on these rules, visit the OWASP XSS Prevention Cheat Sheet<sup>6</sup>.

<sup>5</sup>body, attribute, JavaScript, CSS, or URL

<sup>6</sup>The OWASP XSS Prevention Cheat Sheet can be found at [https://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

## 2.4 Insecure Direct Object References

Sometimes developers wrongly assume that once a user is logged in and shown only the resources they are allowed to access, also known as Presentation Level Access Control, they will not be able to access resources they aren't authorized to see. If there are flawed resource restriction mechanisms in place, anyone could be able to access any resources, whether they are authorized to access them or not. Malicious users might be able to access administrative resources or view confidential information.

### 2.4.1 Dangers of insecure direct object references

One example of insecure direct object references might be a certain web application's functionality for users to edit their profile, for instance at `http://www.example.com/profile.php?id=ID`, where *ID* stands for their own unique ID number. If no proper checks are in place, one would be able to change *ID* to someone else's ID and view/edit their profile in stead.

Another example of an insecure direct object reference flaw could be exploited by using browser history. Assume that a user logged in to a web application on a public computer. After viewing some restricted resources they are authorized to see, the user logs out. Further assume that another user uses the computer afterwards and wants to access the same web application. When they start typing the URL of the web application, the browser will show suggestions based on the letters the user typed in. If some of the restricted resources are shown as suggestions and there are flawed resource restriction mechanisms in place, the new user will be able to access these resources directly, without proper authorization.

### 2.4.2 Preventing insecure direct object references

To prevent insecure direct object references, one must verify that all object references have appropriate defences [1]. Two techniques mentioned by the OWASP Top 10 are:

- **Per user or session indirect object references** – This prevents attackers from directly targeting unauthorized resources [1]. Direct object references are mapped to a set of indirect references that are safe to use publicly. This can be used to help protect database keys, filenames, and other types of direct object references.
- **Check access** – Care must be taken to ensure that users are authorized to access the resources they have requested.

## 2.5 Security Misconfiguration

A web application is usually comprised of a lot of components across the application stack. Therefore it is important that proper security hardening should be

performed across the entire application stack [1]. All components, including operating system, web/app server, DBMS, applications and code libraries should be kept up to date as frequently as possible, so as to not be susceptible to known vulnerabilities in these components. Furthermore, every unnecessary resource should be disabled, removed, or not installed (e.g. ports, services, pages, default accounts and passwords, privileges) [1].

### 2.5.1 Dangers of security misconfiguration

Security misconfiguration may result in attackers gaining unauthorized access to system data or functionality. This data could potentially be stolen or modified over time. Problems with security misconfiguration can occur across the entire application stack, including the platform, web server, application server, framework, and custom code [1]. Automated scanners and *Google Dorks* can be used for detecting misconfigurations, use of default accounts and passwords, unnecessary services, etc.

### 2.5.2 Preventing security misconfiguration

Security misconfigurations can be prevented by implementing a repeatable hardening process. Patches should be deployed in a timely manner and all software and code libraries should be updated regularly. Running scans or doing audits can help detect or prevent misconfigurations or missing patches, and a strong application architecture should be upheld with a good separation and security between components [1]. All settings should be set to be as secure as possible by default, so that someone using the system's defaults shouldn't worry about leaving their website open to security risks.

## 2.6 Sensitive Data Exposure

Sensitive data exposure can occur when sensitive data or backups of sensitive data are stored or transmitted unencrypted or using a weak encryption algorithm. For instance, MD5 is a one way, 128-bit cryptographic hash function that was a widely used as a means to encrypt data, such as passwords. The hashed password would be stored in the database together with other credentials. Because there is no way to decrypt an MD5 hash, when a user logged in, the password they entered would be hashed with MD5 and compared with the hashed value in the database. If both hashes were identical, the user had successfully logged in. MD5 has now been proven not to be collision resistant, which means that there are multiple input strings that share the same hash as output [53]. This, together with the fact that most people use weak passwords that are easy to remember, has made MD5 unsuitable for encrypting passwords and other sensitive data. If a hacker is able to gain access to the database and read the MD5 hashed passwords, they can compare these hashes with values in a rainbow table of precalculated hashes to effectively figure out the user's password. Furthermore, because MD5 is a very fast algorithm, brute force attacks might be able to break weak passwords quickly.

## 2.6.1 Dangers of sensitive data exposure

OWASP considers this vulnerability difficult to exploit, but a successful exploit can have severe impact on the exposed data. Typically this information includes sensitive data such as health records, credentials, personal data, and credit card numbers [1] and should be protected at all cost. Consider a site that doesn't use SSL for all authenticated pages. An attacker could then simply monitor the network, like an open wireless network, and perform a session hijacking attack by stealing the victim's cookie and sending it to the vulnerable web application, making it think that the attacker is the victim. The attacker is now able to access all the victim's private data on the web application.

## 2.6.2 Preventing sensitive data exposure

Sensitive data exposure can be prevented in a number of ways. In this section, we will provide a list of guidelines, rules, and best practices developers should keep in mind when writing the web application [43, 45].

- Disable autocomplete and caching on forms collecting sensitive data. For instance, form input fields that collect credit card information should not automatically fill in the credit card number anyone else may have typed in before.
- It's important that no sensitive data is stored unnecessarily. If the data doesn't get stored, it can't get stolen.
- If storing data is insurmountable, it is important that only strong cryptographic algorithms such as AES, RSA public key cryptography, and SHA-256 are used. These are proven to be strong cryptographic algorithms.
- Passwords should be stored hashed and salted. A salt is a fixed-length cryptographically-strong random value that is appended to the credential data. Salts serve two purposes:
  1. **Prevent encrypted data from revealing identical credentials**  
If two pieces of unsalted data are encrypted using the same encryption algorithm, they will result in the same hash. This can cause an attacker to figure out the unencrypted data.
  2. **Make sure the the entropy fed to the protecting function is high enough, without relying on credential complexity**  
As mentioned previously, it's not uncommon for users to use simple passwords such as *password*, *pass123*, or *password1*, that are easily remembered. Brute force attacks would be able to crack unsalted passwords such as these in an instant.
- Ensure that the cryptographic protection remains secure even if access controls fail. This rule supports the principle of defence in depth. Storage encryption should remain secure, even if an attacker is able to gain access to the database.

- Any secret key should always be protected from unauthorized access. This includes defining a key life cycle, storing unencrypted keys away from encrypted data and using independent keys when multiple keys are required.

## 2.7 Missing Function Level Access Control

Missing function level access control flaws could potentially enable attackers to access administrative functionality that is left unprotected, whether by misconfiguration (see Section 2.5) or by a developer who forgot to include the proper code checks. Hiding certain functionality from unauthorized users, also known as *Presentation Level Access Control*, is not enough (as explained in Subsection 2.4). If attackers are able to find out or guess how to access these hidden, unprotected functions, for instance by changing URL parameters, they will have full control of these functions, without proper authorization or authentication.

### 2.7.1 Dangers of missing function level access control

As administrative functionality is often targeted, attackers could potentially change security settings, install vulnerable components or open backdoors, giving them even more access to data and resources.

### 2.7.2 Preventing missing function level access control

Make sure that there are proper access control checks in place so that hidden functionality is inaccessible by unauthorized people. Presentation Level Access Control alone doesn't actually provide any protection. The authorization process should deny all access by default, requiring explicit grants of access to specific roles only.

## 2.8 Cross-Site Request Forgery (CSRF)

Cross-site request forgery (CSRF) is an attack which forces a user to execute actions on a trusted web application [41] for which the user is currently authenticated. In contrast to session hijacking and session fixation attacks (see Section 2.3), an attacker doesn't try to completely take over the victim's session, but instead abuses the victim's browser's implicit authentication to make requests in the victim's name [3]. This is achieved by making the victim's browser issue a request to a web application where the user has previously logged in. If the targeted victim is the administrator account, this can compromise the entire web application [41].

### 2.8.1 Dangers of cross-site request forgery

To shed light on the dangers of CSRF, we present a popular example of a possible attack scenario of CSRF: the *transfer funds* functionality of a bank's web banking service [41]. Assume that this bank's web application supports transferring money to another account from the following URL: `http://www.example.com/transfer.php?amount=AMOUNT&destAcc=ACCOUNTNUMBER`. When an authenticated user visits this page, *AMOUNT* will be transferred to the destination *ACCOUNTNUMBER*. An attacker could construct a URL so that the victim transfers a desired amount to the attacker's bank account. The attacker then needs to find a way to get the victim's browser to issue the request. If the user is still logged into the bank's web application, the browser will attach the victim's session cookie to the request, causing the request to validate at the server and the funds to be transferred to the attacker's account.

Forcing the browser to make requests in the victim's name can happen in a number of ways. All the attacker has to do is find a way to make the victim's browser issue a request to the desired URL, for instance by setting the desired URL as the `src` attribute of an `img` tag. When the victim visits a page with the `img` tag embedded, their browser will issue a request to the URL in the `src` attribute, thinking it's loading the image. This method only works when trying to issue a request to a web application that works with GET parameters.

If the web application uses POST parameters, the attacker can create an HTML form containing the desired elements and values as POST parameters. The attacker then has to either trick the victim into submitting the form, or use JavaScript to submit the form automatically.

Another way the attacker can get the victim's browser to issue a request to the URL is by performing an XSS attack, using the `XMLHttpRequest` object. The GET parameter variant of this attack is comparable to the session hijacking attack code example, Example 2.11. This example can be altered to accommodate for POST parameters.

### 2.8.2 Preventing cross-site request forgery

A common method for preventing CSRF attacks is to include a unique token to be sent in the body of the HTTP request. These unique *challenge* tokens are associated with the user's current session and are inserted within HTML forms and links associated with sensitive server-side operations. It is then the responsibility of the server application to verify the existence and correctness of the token. Using challenge tokens helps mitigate CSRF attacks, as these tokens are generated randomly and successful exploitation assumes the attacker knows the randomly generated token for the victim's session. The challenge token needs only be generated once per session and is utilized until the session expires. Example 2.12 shows an HTML POST form with a hidden field containing the value of the challenge token.

```
1 <form action="/transfer.do" method="post">
```

```
2 <input type="hidden" name="CSRFToken"
3   value="OWY4NmQwODE4ODRjN2Q2NTlhMmZlYWUwYzU... ">
4   ...
5 </form>
```

Example 2.12: HTML form containing a hidden input field with a unique, random CSRF challenge token<sup>8</sup>

If the form above were to be submitted, the value of the challenge token would automatically be contained in the request sent to the server. If the challenge token is wrong or absent from the request, it should be aborted, the token reset and the event logged as a potential CSRF attack [41].

## 2.9 Using Components with Known Vulnerabilities

Using vulnerable components is not unusual when content management systems are concerned and can leave a web application seriously compromised. Exploit databases get updated daily with new vulnerabilities that are found in core functionality, plug-ins, and other added components. When new exploits are discovered, it takes a while for the developers to fix the issue and release a new version, if at all. Even then, it's up to the systems maintainer to update the vulnerable module to the latest release, something that doesn't happen frequently enough in real life. Automated tools are a dangerous asset in aiding attackers to find websites that contain these vulnerabilities. Components with known vulnerabilities could open up the web application to a full range of weaknesses, including injection, broken access control, and XSS [1].

### 2.9.1 Dangers of using components with known vulnerabilities

We will illustrate the dangers of using components with known vulnerabilities by providing some real-world cases of vulnerable plug-ins, modules and extensions for each of the web content management systems we will discuss in Chapter 3, namely WordPress, Drupal and Joomla. These examples are taken from the vulnerability databases<sup>9</sup> mentioned in Section 1.3.

The first example comes from the WordPress *Spicy Blogroll* plug-in<sup>10</sup>. It was recently discovered that this plug-in contains a file inclusion vulnerability. In one of the PHP scripts, certain GET variables are obtained from the URL and used in the `require_once` function that includes an external file in the current script. Attackers can abuse this vulnerability to access any file on the system. The Spicy Blogroll plug-in has been downloaded at least 2500 times.

<sup>8</sup>Source: [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

<sup>9</sup>More information about these exploits can be found at: <http://cve.mitre.org/>

<sup>10</sup><http://wordpress.org/plugins/spicy-blogroll/>



The second example comes from the Drupal WYSIWYG<sup>11</sup> CKEditor<sup>12</sup> module. Versions below 4.1 of this module contained a persistent XSS flaw. If the WYSIWYG editor was enabled for posting comments or adding content, it was possible for an attacker to inject a hidden `iframe` which could execute unrestricted JavaScript when viewed in the edit mode, making it possible to access the victim's session cookie. This attack vector was targeted at privileged users who are able to edit posts. This vulnerability was fixed in version 4.1 of the module. This module was downloaded approximately 12120 times, and is in use by more than 1500 sites.

The third and final example comes from the Joomla Highlight plugin. User input passed through the *highlight* parameter was not properly sanitized before being passed to the *unserialize* method. This could be exploited to inject arbitrary PHP objects into the application. This plugin is installed by default. Versions of Joomla earlier than 2.5.8, and 3.0.2 are susceptible to this vulnerability.

## 2.9.2 Preventing using components with known vulnerabilities

The best way to deal with this risk is to ensure that all components are kept up to date and only components that are compatible with the application are installed. This means that vulnerability databases, project mailing lists, and announcements regarding vulnerabilities must be checked regularly, though this is no guarantee that component vulnerabilities will be fixed. Many developers do not create vulnerability patches for older versions, but instead simply fix the problem in the next version.

For instance, at any given time there two supported versions of Drupal. The current supported versions of Drupal are version 6.x and version 7.x. Core updates for both versions are released on a regular basis until Drupal version 8.0 will be released. Once this version is released, only versions 7.x and 8 will be supported. The reason for this is that module developers sometimes lag behind on Drupal releases, causing some required functionality to not be available for the latest version of Drupal. Having to support two releases of Drupal means that module developers must maintain and update their code for both versions of Drupal. This may cause developers to only update the 7.x version of the module, leaving the 6.x version broken and vulnerable. While anyone using the 7.x version of Drupal might be safe, people using version 6.x will still be vulnerable and have no way of installing the fixed, safe version of the module without first updating their entire Drupal installation. In our review of the Drupal content management system, we will only take the latest version, version 7.x, into account.

---

<sup>11</sup>What You See Is What You Get

<sup>12</sup>[https://drupal.org/project/wysiwyg\\_ckeditor](https://drupal.org/project/wysiwyg_ckeditor)

## 2.10 Unvalidated Redirects and Forwards

Web applications frequently use redirects and forwards to redirect users to other pages. If the target page is specified in an unvalidated parameter, this could potentially be exploited by an attacker to redirect the victim to a malicious page, or gain access to administrative functions.

### 2.10.1 Dangers of unvalidated redirects and forwards

Unvalidated redirects and forwards can have serious consequences. Once the user is on the attacker's domain, they are susceptible to any number of attacks. The attacker may attempt to install malware or trick the victim into disclosing passwords or other sensitive information [1]. For instance, the user may be persuaded into clicking on `http://www.example.com/redirect.jsp?url=evil.com` because `http://www.example.com` is a trusted web page. The victim will then be redirected to the malicious `evil.com` website, which may be designed to look like the login page of the application. When the user tries to log in, the attacker is able to steal their credentials [1].

Another way an attacker could abuse unvalidated forwards is to bypass the application's access control check to gain access to unauthorized functions. For instance, assume that an application uses a parameter to indicate where a user should be sent if a transaction is successful. The attacker could craft a URL like `http://www.example.com/forward.jsp? fwd=admin.jsp` that bypasses access control checks and forwards the attacker to an administrative function they aren't authorized to see. [1].

### 2.10.2 Preventing unvalidated redirects and forwards

The first way unvalidated redirects and forwards can be prevented is by simply not using redirects and forwards. If using redirects and forwards is insurmountable, they should not be based on user parameters, care must be taken that the supplied value is valid, and the user is authorized to be redirected there.

## 2.11 Conclusion

A lot must be kept in mind by a developer who wants to build a secure web application from the ground up. While not impossible, it is less likely that a single developer will be able to write a fully secure web application from scratch. Certainly someone with limited or no programming knowledge will not be very concerned with security. This is why content management systems are very attractive solutions. Typical users of content management systems are usually not concerned with the inner workings of the system. They just want everything to work, and assume that everything is implemented securely.

This is why it is worth finding out if these content management systems are indeed secure. In the Chapter 4, we will investigate the three most used

open source web content management systems, namely WordPress, Joomla, and Drupal. For each of these content management systems, we will attempt to find answers for the following questions and guidelines, based on the security risks described in the OWASP Top 10:

### 1. Injection

- Which SQL injection prevention techniques are used and how do they work?
- Are these techniques used in a secure way?
- Is least privilege encouraged?

### 2. Broken Authentication and Session Management

- Is the Authentication mechanism secure?
- Is a strong password policy<sup>13</sup> enforced?
- How does the Session Management mechanism work?
- Are Session IDs sufficiently random?
- Are Session IDs sufficiently long?
- Are Session IDs set to expire?
- Is there a password recovery mechanism in place?
- Does the password recovery system send passwords in plain text?

### 3. Cross-Site Scripting (XSS)

- Does the CMS contain any input/output filtering methods?
- Are there different filtering methods based on the HTML context of the input data?

### 4. Insecure Direct Object References

Preventing Insecure Direct Object References is strongly related to Missing Function Level Access Control, as both security risks are mitigated by strong access control. Therefore, we will split up our analysis of the access control mechanism in two parts. In the section about Missing Function Level Access Control, we will talk more in depth about how access control is implemented in each system. In the section about Insecure Direct Object References, we will provide a higher level overview of the access control mechanism.

### 5. Security Misconfiguration

- How difficult is it to keep core functionality up to date?
- Are there automated update processes in place?
- Are default settings set securely? This includes checking whether or not displaying error messages is turn off, and which privileges are granted to registered users by default.

---

<sup>13</sup>The definition of a strong password policy can be found at [http://www.sans.org/security-resources/policies/Password\\_Policy.pdf](http://www.sans.org/security-resources/policies/Password_Policy.pdf)

6. **Sensitive Data Exposure** When comparing web content management systems on Sensitive Data Exposure mitigation techniques, we will investigate if and how any of the guidelines mentioned in Subsection 2.6 are implemented and maintained.

7. **Missing Function Level Access Control**

As explained previously, both Insecure Direct Object References and Missing Function Level Access Control security risks can be mitigated by strong access control. In the sections about Missing Function Level Access Control, we will review the access control system at a lower level.

- How is access control enforced?
- Is directory browsing denied by default?
- Does the access control mechanism fail securely?

8. **Cross-Site Request Forgery (CSRF)**

- Are any efforts made to prevent CSRF attacks?
- Does the content management system core contain any token generation mechanisms?
- How do these mechanisms work, and are they secure?
- Are they easy to use by developers?
- Does the anti-CSRF token get regenerated if a request is received containing a wrong value?

9. **Using Components with Known Vulnerabilities** The web content management systems we will discuss in Chapter 3 have their own plug-in, or add-on databases where users can search for and download components to extend the core functionality of their desired CMS. When discussing this security risk, we will investigate the following:

- Is there a versioning system in place where the current version of an added component can easily be identified?
- Are security issues disclosed publicly?
- Is there a project or security mailing list in place to warn users about new vulnerabilities?
- Are there any manual or automated checks in place in order to detect vulnerabilities before the components are released publicly?
- Is there any indication or warning given when searching for or trying to install components with known vulnerabilities?
- How are vulnerability reports dealt with and how quickly do flaws get fixed?
- Are updates difficult to install?

Some of these criteria are also applicable to flaws in CMS core functionality. While *Using Components with Known Vulnerabilities* may allude to plug-ins and added components, it will be worth investigating if and how these criteria are dealt with when core functionality is concerned.

10. **Unvalidated Redirects and Forwards** We will investigate if the content management systems use redirects and forwards in core functionality. We will check if any of the following rules and guidelines suggested in the OWASP Top 10 are considered:

- Don't use redirects and forwards.
- Don't base redirects and forwards on user parameters.
- Make sure that the supplied value is valid.
- Make sure the user is authorized to be redirected to that location.

## Chapter 3

# Content Management Systems

In this chapter we will take a look at the three most used open source content management systems, namely WordPress, Joomla, and Drupal [57]. These open source CMSs are designed in a modular way, supporting added components to extend core functionality and the ability for users to customise the website to their own taste. They are assumed to be secure, because they have been through many iterations, with a strong focus on security. However, these open source web content management systems are maintained by people, and people are not perfect. A strong disadvantage of everyone using the same content management systems is that a lot of websites will be susceptible to the same exploit if one is discovered. It is therefore very important that core and added functionality be kept up to date as much as possible. In this Chapter we will provide a more general overview of each content management system before going into more detail in Chapter 4

### 3.1 WordPress

WordPress is a content management system that was first developed back in 2001. WordPress is primarily considered a blogging framework, but has pushed to become a fully fledged content management system. Whether or not they succeeded is up for debate. While many people agree that WordPress is a blogging tool first and a CMS second, others strongly disagree and say that it's a CMS with blogging capabilities [55]. WordPress started out as *b2 cafelog*, a simple blogging tool. More functionality has been added through the years, including a plug-ins and themes system, and customization capabilities.

WordPress is very focused on backwards compatibility and being universally deployable. Up to WordPress version 3.2, released in 2011, the earliest supported version of PHP was 4.3.x, which is very dated. In trying to ensure usability, sometimes choices are made in the codebase that are not necessarily the most secure, but provide good backwards compatibility. Examples of this

include enforcing PHP's *Magic Quotes GPC* functionality, even though this has been deprecated since PHP version 5.3.0 and removed since version 5.4.0 [52], and using hashing methods such as MD5<sup>1</sup> for compatibility. We will focus on WordPress version 3.5.2, being the latest version of WordPress at the time. This version was released in June of 2013.

WordPress consists of two distinct areas: a public front facing area and an administrative area. The first area contains all the content that is available to registered and unregistered users alike. In the front facing area, only limited functionality is allowed, including reading posts and posting comments. All administrative functionality, including adding new posts and pages, editing profiles, and installing themes and plug-ins is found in the administrative area. As we will see in the rest of this chapter, these areas are kept separate for security reasons.

In a study by W3Techs [57], a trusted source for web technology surveys, on the usage of content management systems for websites, WordPress came out on top, being considered the most used content management systems. WordPress has a 57.1% CMS market share, while approximately 18.9% of all websites use WordPress. A list of websites that use WordPress as the underlying framework include:

- MSNBCS TV (<http://tv.msnbc.com/>) – The website of a major American news network.
- Variety (<http://variety.com/>) – The website of a large entertainment news source.
- The Rolling Stones (<http://www.rollingstones.com/>) – Website for the legendary rock band "The Rolling Stones".

### 3.1.1 WordPress Core Functionality

In this subsection, we will take a look at some of the functionality provided by WordPress core. Most of the provided information will be more high level, but sometimes technical details will be provided. We will start by giving an overview of the database system, and continue on with a quick overview of the authentication and session management mechanism, including the password recovery system. Then we will take a look at WordPress's mechanism to distinguish user privileges and finally, we will talk about ways in which WordPress functionality can be extended using themes and plug-ins.

#### Database

WordPress has two minimum requirements that must be met before it can be used [72]:

1. PHP version 5.2.4 or greater must be used

---

<sup>1</sup>MD5 has been proven to be insecure, see Section 2.6

2. MySQL version 5.0 or greater must be used

This means that WordPress cannot be run on any other database system other than MySQL. The two other content management systems we will review, namely Joomla and Drupal, both provide Database Abstraction Layers (DBAL), enabling them to be run on different sorts of databases. More of these technical details will be given in Subsection 4.1.1 of the next chapter.

### User Registration and Authentication

User registration requires that a new user enters a unique username and a valid e-mail address. An e-mail will be sent containing a clear text password that was generated by WordPress. A user can change this generated password in their profile settings. Figure 3.1 shows the default user registration form on a WordPress 3.4 website.

The image shows a screenshot of the WordPress 3.4 user registration form. At the top left is the WordPress logo, a circular 'W' icon, followed by the word 'WORDPRESS' in a blue serif font. Below this is a yellow rectangular box with the text 'Register For This Site'. Underneath is a white registration form with a light gray border. It contains two text input fields: the first is labeled 'Username' and the second is labeled 'E-mail'. Below the 'E-mail' field, there is a small blue text note that says 'A password will be e-mailed to you.'. At the bottom right of the form is a blue rounded rectangular button with the word 'Register' in white text.

Figure 3.1: WordPress 3.4 user registration form

When a user changes their password, password strength is assessed as it is being typed in. Passwords are classified in four categories, using a scoring system based on the password length and a number of other criteria:

- **Password contains numbers** – Add 10 points to the overall points.
- **Password contains lower case letters** – Add 26 points to the overall points.



- **Password contains upper case letters** – Add 26 points to the overall points.
- **Password contains punctuation marks** – Add 31 points to the overall points.

The final score is then obtained by dividing the logarithm of the overall points, raised to the power of the password’s length by the natural log of 2.

$$score = \frac{\log(points^{password\_length})}{\ln(2)}$$

The resulting score is then used to classify passwords into the following categories:

- *Very weak* – If the password contains less than 4 characters.
- *Weak* – If the password is equal to username, or if the overall score is less than 40.
- *Medium* – If the overall score is less than 56.
- *Strong* – If the overall score is more than, or equal to 56.

Providing the password scoring system can give users an incentive to use longer and more secure passwords, making brute force attacks harder. Unfortunately, WordPress doesn’t enforce any minimum password requirements. For instance, the current password system allows passwords of just one character, while it states that a minimum of seven characters is recommended. We will provide more information about WordPress’s password hashing algorithm in Subsection 4.1.6: Sensitive data exposure.

Typing in the current password is not required when changing a password. If an attacker were to gain access to someone else’s account, they would be able to change their password without having to provide the current password. For instance if the attacker were somehow able to steal a user’s *Logged in* and *Admin* cookie, they would be able to change all the victim’s credentials in the administrative area. We will provide more information about WordPress sessions in Subsection 4.1.2.

## Password Recovery

WordPress provides a password recovery system. When a user indicates that she has forgotten her password, an activation key will be generated and associated with the user’s credentials in the database. An e-mail will then be sent to the user’s e-mail address containing a link to a password reset page with the activation key and the user’s username in GET parameters. If the user clicks on the link, the application will check if the activation key matches the one associated with the username in the database. If the activation key matches the one in the database, the user can enter a new password. At no point in

time are passwords stored in plain text. However, as mentioned previously, a random password will be generated for the new user when they register for a new account. An e-mail will be sent to the new user's e-mail address containing the plain text password. An attacker eavesdropping on the network might be able to intercept this e-mail and obtain the user's initial password.

## User Privileges

WordPress provides a system of *Roles* and *Capabilities*, whereby a *Role* defines which *Capabilities*<sup>2</sup> a user is allowed to perform. These capabilities can vary from writing and editing posts to managing plug-ins or other user. They define a user's responsibilities within the site. By default, WordPress provides six predefined user roles. An authenticated user can have only one *Role* assigned to them at any given time. These user roles are named as follows:

1. Super Admins
2. Administrators
3. Editors
4. Authors
5. Contributors
6. Subscribers

In Subsection 4.1.4, we will see which privileges are associated with each of these *Roles*.

## Added Components

WordPress core functionality can be extended by installing Plug-ins and Themes. WordPress provides a plug-in directory, which contains over 26000 Plug-ins [68]. Plug-ins can range from adding functionality to share posts on social media websites, to extending administrative functionality to enable an administrator to edit user *Roles*.

Plug-ins uploaded to be hosted on the WordPress website will be manually checked by code reviewers before being allowed. Reviewers will check if plug-in code is up to par with WordPress coding standards and uses the correct API functions to defend against attacks. Plug-in developers can be e-mailed and asked to provide more information about their plug-in prior to the plug-in being accepted. After approval, the developer will be allowed to upload the plug-in to a Subversion repository, after which the plug-in will appear in the plug-ins browser automatically [69].

---

<sup>2</sup>A full list of the default capabilities provided by WordPress can be found at: [http://codex.wordpress.org/Roles\\_and\\_Capabilities](http://codex.wordpress.org/Roles_and_Capabilities)

## 3.2 Joomla

Joomla is an open source web content management system that was first released in 2005. Joomla reached thirty million downloads in March of 2012 [24], and is estimated to be the second most used CMS on the internet, after WordPress [57]. We will focus only on the latest version of Joomla at the time, being Joomla version 3.1.5, which was released in August of 2013. Joomla is written in PHP and contains a database abstraction layer, which makes it possible to be used in a multitude of software environments.

Like WordPress, Joomla is comprised of separate front end web interface and a back end administrative interface. Unlike WordPress, however, users have to log in to each interface separately. The front end web interface has some limited functionality for certain users, like creating new articles and posting comments. Regular users are not allowed to access the back end interface, as this functionality is reserved for users with higher privileges.

In a study by W3Techs [57] on the usage of content management systems for websites, Joomla came up second to WordPress. Joomla has a 9.9% CMS market share, while approximately 3.3% of all websites use Joomla. A list of websites that use Joomla as the underlying framework include:

- Harvard University - The Graduate School of Arts and Sciences (<http://gsas.harvard.edu/>) – Website for the academic unit responsible for many post-baccalaureate degree programs offered through the Faculty of Arts and Sciences at Harvard University.
- The Guggenheim Museum (<http://variety.com/>) – Website of an internationally renowned art museum.
- Times Square (<http://timesquare.com/>) – Website containing the happenings in Times Square New York.

More examples of websites that use Joomla as the underlying content management framework can be found in the Joomla Showcase<sup>3</sup>.

### 3.2.1 Joomla Core Functionality

In this subsection, we will take a look at some of the functionality provided by Joomla core. Most of the provided information will be more high level, but sometimes technical details will be provided. We will start by giving an overview of the database system, and continue on with a quick overview of the authentication and session management mechanism, including the password recovery system. Then we will take a look at Joomla's mechanism to distinguish user privileges and finally, we will talk about ways in which Joomla functionality can be extended using added components like plug-ins, modules, and templates.

---

<sup>3</sup>The Joomla Showcase contains more than 3900 websites and can be found at <http://community.joomla.org/showcase/sites.html>

## Database

Joomla provides a Database Abstraction Layer (DBAL) to simplify the usage of database queries [26]. Using a DBAL ensures that Joomla supports multiple kinds of SQL database systems. Syntactical differences between SQL database queries will be handled by the DBAL. As we shall see in the next Chapter, Joomla's DBAL makes it possible to create dynamic queries by using a process called *query chaining*, but also supports regular string queries. Untrusted user data must be actively sanitised by a developer before it is used in the query.

## User Registration and Authentication

User registration requires that a user enters their name, unique username, password, and a valid, unique e-mail address. A confirmation mail will be sent to the e-mail address, containing a registration token. This token is associated with the new user in the database, and is generated by hashing a 16 character application secret using salted MD5. The application secret is randomly generated when Joomla is installed. The salt value used is a randomly generated string of 8 characters, which is appended to the application secret before being hashed. Even after a new user confirms their registration, their account must still be approved by an administrator before it is finally activated. Figure 3.2 shows the default Joomla user registration form.

The image shows a screenshot of the Joomla user registration form. At the top, the title "User Registration" is displayed. Below the title is a legend: "\* Required field". The form consists of six input fields, each with a label and an asterisk indicating it is required: "Name: \*", "Username: \*", "Password: \*", "Confirm Password: \*", "Email Address: \*", and "Confirm email Address: \*". At the bottom of the form, there are two buttons: a blue "Register" button and a grey "Cancel" button.

Figure 3.2: Joomla user registration form

Unlike in WordPress, there is no password strength indicator shown when setting or changing passwords. There is a minimum password requirement, however, as passwords are required to be at least 4 characters long. Other requirements include the fact that a password may not begin or end with a space, and that

passwords cannot be over 100 characters long. Longer passwords can limit the success rate of brute force attacks but as Joomla core doesn't contain any login attempt limitations, brute force attacks are left possible.

User's passwords are stored hashed and salted, using MD5 as the encryption algorithm. More information about how Joomla's password hashing algorithm works will be given in Subsection 4.2.6.

## Password Recovery

Joomla provides a password recovery system. When a user indicates that they have forgotten their password, an e-mail will be sent to their e-mail address containing a verification code and a link to a password reset page. A code snippet of how the verification code is generated is shown in Example 3.1

```
1 ...
2 $token = JApplication::getHash(
3     JUserHelper::genRandomPassword());
4 $salt = JUserHelper::getSalt('crypt-md5');
5 $hashedToken = md5($token . $salt) . ':' . $salt;
6
7 $user->activation = $hashedToken;
8 ...
```

Example 3.1: Code snippet from Joomla's `processResetRequest` function<sup>4</sup>

The token is generated using the `getHash` function, which will return a salted MD5 hash of a 16 character long secret value that was generated when Joomla was installed. The return value of the `genRandomPassword` function, which generates an 8 character long random string, is used as salt for the `getHash` function. This is the token that is sent to the user.

In order to safely associate the token with the user, it will first be hashed again before being stored in the database. The `getSalt` function generates a salt by hashing the result of the `mt_rand` PHP function using MD5. As described earlier, the `mt_rand` function generates a random integer value between 0 and the value returned by `mt_getrandmax`. In this case only the first 8 characters of the resulting hash are used as the final salt value. Finally, the generated token is concatenated to the salt value and hashed using MD5. The salt value is then appended to the resulting hash and associated with the user.

The user has to manually copy the verification code from the e-mail and paste it in the password reset form, together with their username. The verification token will be checked against the hashed version associated with the user in the database. The token is hashed using the same method as described above, using the salt obtained from the hashed version. If the two hashes match, the user will be able to enter a new password, and the reset request will be logged in the database.

<sup>4</sup>Source: `components/com_users/models/reset.php`

## User Privileges

Joomla contains a very extensive system for defining user privileges. This system is actually divided into two systems. One system controls what users can view, the other system controls what users can do. The first system is comprised of *User Groups* and *Access Levels*, while the second system is based on *User Groups*, *Actions* and *Permissions*. In this subsection, we will provide some basic information on these systems. In Subsection 4.2.4 in the next chapter, we will provide more in depth information on each of these systems. First we will take a look at the system that defines what a user can view. Then we will review the system that defines what a user can do.

### User Groups and Access Levels

Defining what users can view is implemented in a hierarchical system of *User Groups* and *Access Levels*. Access Levels define which Groups of users can view certain kinds of content. A Group can be assigned to multiple Access Levels, making members of that Group able to view content with those specific Access Levels. There are four Access Levels defined by default.

1. **Public** – Open to all visitors of the website.
2. **Guest** – Restricted to non authenticated visitors of the website.
3. **Registered** – Restricted to all registered, authenticated users.
4. **Special** – Restricted to all User Groups, except Guest and Registered.

New content and items are assigned one Access Level, limiting who will be able to view that content or item. By default, there are nine User Groups, which reside in a hierarchical system whereby each Group can have a parent and child Groups. A user can be a member of multiple Groups at the same time. The hierarchy of User Groups is visualised in Figure 3.3.

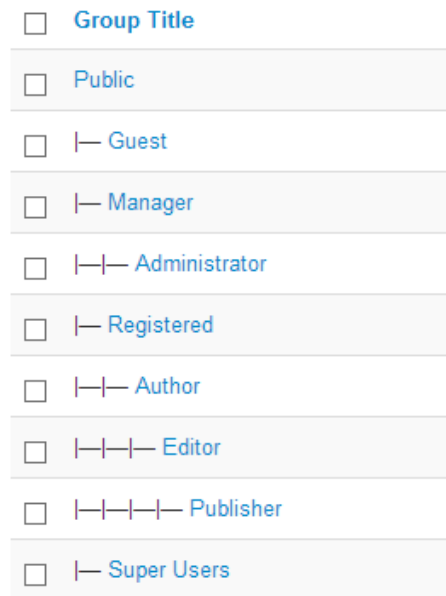


Figure 3.3: The hierarchy of Joomla User Groups

### Actions and Permissions

The system that defines what a user can do uses the same User Groups, but to a different end. The system that defines a user's privileges is comprised of User Groups, Actions and Permissions. In the following summary, we will explain how these components fit together:

- **Groups** – *Groups* are comprised of a list of *Actions* and *Permissions*. Permissions are set on the Actions of a certain Group. A Group can be set to inherit Permissions from the parent Group, or define different Permissions altogether.
- **Actions** – *Actions* define which actions are possible on a certain item. For instance creating, editing, or deleting content.
- **Permissions** – *Permissions* define which Actions are allowed and which are not. Each Action can have only one Permission defined on itself. The four possible Permissions for any given Action are:
  - **Deny** – Denies this Action for the Group and any child of this Group at all levels in its hierarchy. Even if a child Group sets the Permission to *Allow*, this will not have any effect, as only the Permission set in the parent Group will be considered.
  - **Not set** – Defaults to *Deny*, but can be overridden by a Permission in a child Group. This only applies to the main *Public* Group at the top of the hierarchy.
  - **Inherit** – Inherits the Permission from the parent Group or a higher level in the hierarchy.

- **Allow** – Allows this action to be allowed for the current Group and any child Group. A Group in a lower level of the hierarchy can override this Permission by setting it to *Deny*.

### Added Components

Joomla contains five types of extensions, each handling specific functionality [27]. These five extension types are as follows, and are visualised in Figure 3.4:

1. **Components** – Components are the largest and most complex extensions. Most components have two parts: a site part and an administrator part. A Component is called to render the body of a page, thus contains most of the content on the page. Examples of Components include Contact, News Feeds, and Web Links Components.
2. **Modules** – Modules are smaller components used for page rendering. Modules can be thought of as *boxes* that are arranged around a Component and contain certain lightweight functionality. Examples of modules include a Login Module or the *Who's Online* Module. Modules can also contain static HTML or text.
3. **Plug-ins** – Plug-ins are in essence event handlers. In the execution of any part of Joomla, an event can be triggered, and a Plug-in register to handle certain events. For example, a Plug-in could be used to intercept user-submitted articles and filter out bad words.
4. **Templates** – A Template can be thought of as the design and layout of the website. Templates change the look and feel of a website. Templates define fields in which a page component or modules will be shown.
5. **Languages** – Language packages consist of *key/value pairs* and can be used to translate the website to a different language. These *key/value pairs* contain the translation of static text strings which are used by the application to build the interface.



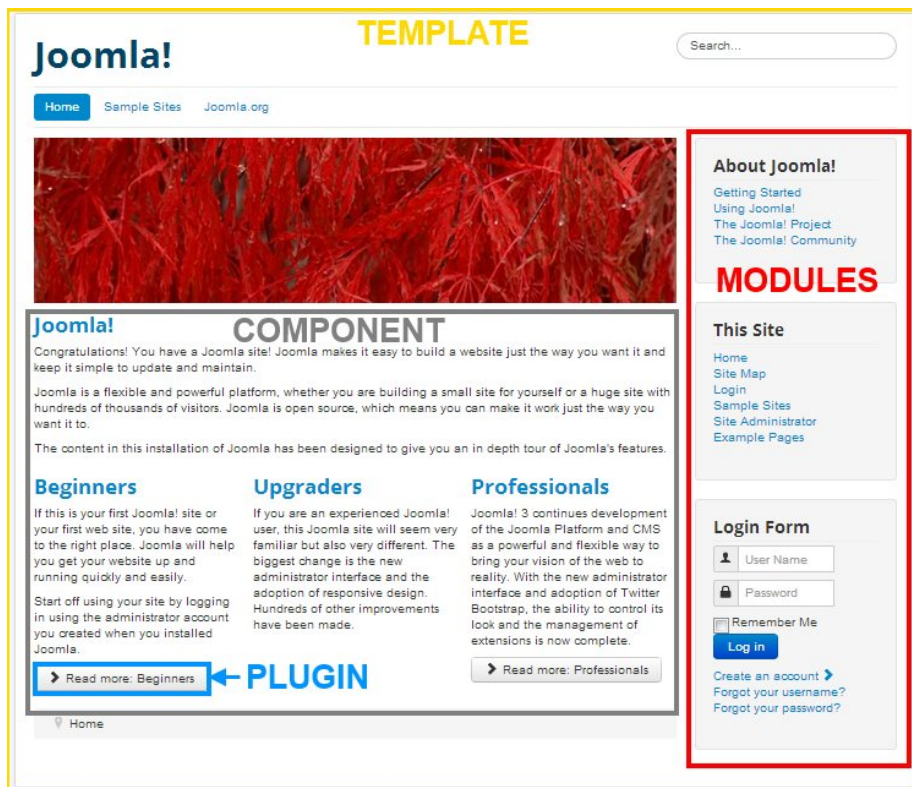


Figure 3.4: An overview of Joomla extensions [27]

Before extensions can be uploaded and used by other users, they are manually reviewed by code reviewers. The added extension will be checked according to four checklists [30]:

1. Submission Checklist
2. Trademark Checklist
3. License Checklist
4. Installation and Functionality Checklist

These checklists are available publicly, and can be checked by the developer before submission to speed up the approval process.

### 3.3 Drupal

Drupal is an open source web content management system that was first released in 2001. Drupal is used by approximately 876000 websites [17], and is estimated to be the third most used CMS on the internet, after Joomla [57]. Drupal

is written in PHP and contains a database abstraction layer, which makes it possible to use Drupal in a multitude of software environments. There are two supported versions of Drupal at any given time. At this time, Drupal versions 6.x and 7.x are the supported releases. Once Drupal version 8.0 is released, version 6.x will no longer be supported. In this section, we will focus on Drupal 7.x primarily, as this is the latest stable release.

Unlike WordPress and Joomla, Drupal doesn't contain a separate front- and back end. Administrative and site viewing functionality are rather combined in a hybrid view. Registered users are allowed to post and edit comments, but creating posts and pages is reserved for more privileged users. Unregistered users are only allowed to view articles and comments, but are not allowed to post comments of their own.

According to a study by W3Techs [57] on the usage of content management systems for websites, Drupal is the third most used web content management system. Drupal has a 6.0% CMS market share, while approximately 2.0% of all websites use Drupal. A list of websites that use Drupal as the underlying framework include:

- The Economist (<http://www.economist.com/>) – A weekly news and international affairs publication.
- Twitter Developers (<http://dev.twitter.com/>) – The website of the developer community for the Twitter platform.
- FedEx (<http://about.van.fedex.com/>) – The "about" portion of the website of an American global courier delivery services company.

More examples of websites that use Drupal as the underlying content management framework can be found in the Drupal Showcase<sup>5</sup>.

### 3.3.1 Drupal Core Functionality

In this subsection, we will take a look at some of the functionality provided by Drupal core. Most of the provided information will be more high level, but sometimes technical details will be provided. We will start by giving an overview of the database system, and continue on with a quick overview of the authentication and session management mechanism, including the password recovery system. Then we will take a look at Drupal's mechanism to distinguish user privileges and finally, we will talk about ways in which Drupal functionality can be extended using themes and modules.

#### Database

Drupal uses a Database Abstraction Layer (DBAL), based on PHP's Data Objects (PDO), to help prevent SQL injection.

---

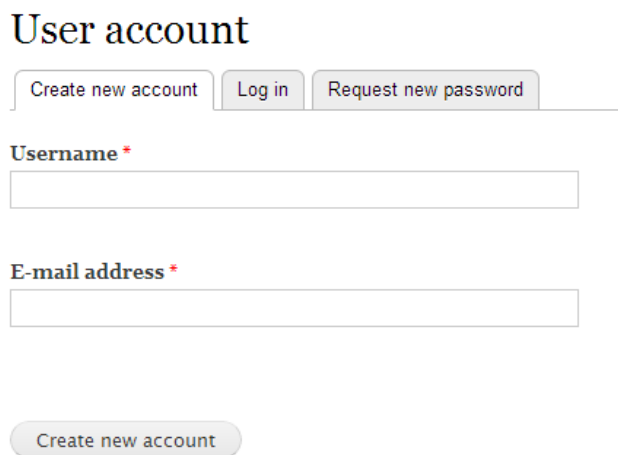
<sup>5</sup>The Drupal Showcase contains more than 3900 websites and can be found at <http://www.drupalshowcase.com/>

*The intent of this layer is to preserve the syntax and power of SQL as much as possible, but also allow developers a way to leverage more complex functionality in a unified way. It also provides a structured interface for dynamically constructing queries when appropriate, and enforcing security checks and similar good practices.* – Drupal API: Database abstraction layer

This means that Drupal can be used with a number of different types of databases. The DBAL will ensure that the correct syntax is used for querying the database. As we shall see in Subsection 4.3.1 on Drupal’s SQL injection mitigation techniques, Drupal makes use of prepared statements with slight syntactical differences to regular prepared statements.

## User Registration and Authentication

Based on the default user registration conditions, user registration requires that new users must validate their e-mail address and be approved by administrators. When a new user registers, they have to enter a unique username and a valid e-mail address. A random password will be generated, but not disclosed to the user. If no default password were generated, anyone could log into the new account, without providing a password. The other reason is that the one-time login token generation mechanism uses a snippet of the current password hash to create the token. If no password were set, this mechanism would fail. The new user will be able to change this password later.



The image shows a web form titled "User account". At the top, there are three buttons: "Create new account", "Log in", and "Request new password". Below these buttons, there are two input fields. The first is labeled "Username \*" and the second is labeled "E-mail address \*". At the bottom of the form, there is a "Create new account" button.

Figure 3.5: Drupal 7.x user registration form

When approved by an administrator, an e-mail will be sent to the user, containing their username and a unique one-time login link that enables the user to log in and change their password. We will go into more detail about this one-time login link in the next subsection, concerning password recovery. Password strength is assessed as it is being typed in. Passwords are classified in four

categories, using a scoring system based on the password length and a number of other criteria. Every password starts with a strength of 100. If the password is shorter than 6 characters, the strength indicator will deduct 5 points for every character less than 6, plus a 30 point penalty:

$$\begin{aligned} strength &= 100 \\ strength &= strength - ((6 - password\_length) * 5) + 30 \end{aligned}$$

Then, each time any of the following criteria is true, it will be counted as an extra weakness:

- Password doesn't contain any numbers.
- Password doesn't contain any lower case letters.
- Password doesn't contain any upper case letters.
- Password doesn't contain any punctuation marks.

If one weakness is detected, 12.5 points will be deducted from the current password strength. If two weaknesses are detected, 25 points will be deducted from the current password strength. If three or more weaknesses are detected, 40 points will be deducted from the current password strength. Finally, the password strength indicator will check if the password is equal to the username. If this is the case, password strength will be set to 5, because passwords that are the same as the username are considered very weak.

The resulting score is then used to classify passwords into the following categories:

- *Weak* – Password strength < 60.
- *Fair* –  $60 \geq$  Password strength < 70.
- *Good* –  $70 \geq$  Password strength < 80.
- *Strong* –  $80 \geq$  Password strength  $\leq$  100.

Providing the password scoring system can give users an incentive to use longer and more secure passwords, making brute force attacks harder. Unfortunately, Drupal doesn't enforce any minimum password requirements. For instance, the current password system allows passwords of just one character, while the password scoring system states that a minimum of six characters is recommended. We will provide more information about Drupal's password hashing algorithm in Subsection 4.3.6. Another brute force attack mitigation technique used by Drupal is enforcing a maximum number of authentication attempts. Drupal provides a mechanism that automatically blocks an IP address that's flooding the system with login attempts with incorrect credentials. The defaults are set to 5 attempts every 6 hours for a single IP address and a single account, and 50 attempts per hour coming from the same IP address, trying to log into any

account. No login attempts coming from a blocked IP address will be allowed, even if the login credentials are correct<sup>6</sup>.

Typing in the current password is not required when changing a password. If an attacker were to gain access to someone else's account, they would be able to change the victim's password and other credentials without having to provide the current password. For instance, if the attacker were somehow able to steal a user's session cookie, they would be able to change all the victim's credentials. When a user changes passwords, their session will be regenerated. This means that if an attacker were able to perform a Session Hijacking or Session Fixation attack, and change the victim's e-mail address and password, the victim's session would be invalidated. They would be unauthenticated with no means of being able to log back in as the password recovery system sends a one-time login link to the specified e-mail address. We will provide more information about Drupal sessions in Subsection 4.3.2, but first we will take a look at Drupal's password recovery system.

## Password Recovery

Drupal provides a password recovery system. When a user indicates that they have forgotten their password, a one-time login token will be generated and an e-mail will then be sent to the user's e-mail address containing a link to a password reset page. The one-time login token is only valid for 24 hours, after which it will expire and is no longer usable. This token is generated by the `user_pass_rehash` function, which can be seen in Example 3.2.

```
1 function user_pass_rehash($password, $timestamp, $login)
2 {
3   return drupal_hmac_base64($timestamp . $login,
4     drupal_get_hash_salt() . $password);
}
```

Example 3.2: Drupal's `user_pass_rehash` function

The one-time login token is based on the user's current password hash, the timestamp of the password reset request, and the timestamp of the user's last login time, which is obtained from the database. Using the timestamp of the user's last login time to calculate the hash is of significance when the token is regenerated in the verification step of the password reset procedure. The `user_pass_rehash` function creates this one-time login token by concatenating the timestamp of the password reset request to the timestamp of the user's last login time and uses these as the data to be hashed by the `drupal_hmac_base64` function. This function hashes the provided data in SHA-256, using the HMAC method. The key needed by the hash function is generated by concatenating the *Drupal Hash Salt* value, which is generated when Drupal is installed, to the user's current password hash. The resulting hash is encoded in Base64.

---

<sup>6</sup>[https://api.drupal.org/api/drupal/modules!user!user.module/function/user\\_login\\_authenticate\\_validate/7](https://api.drupal.org/api/drupal/modules!user!user.module/function/user_login_authenticate_validate/7)

The e-mail that's sent to the e-mail address of the user for whom a password request was issued, contains a link to a password reset page. This link is made up of GET parameters, which contain the user's ID, the timestamp on which the password reset request was issued, and the one-time login token. If the user clicks on the link, the application will first check if the one-time login token hasn't expired. If the token is still valid, another token will be generated by using the provided GET parameters and the timestamp of the user's last login, which is obtained from the database. If this timestamp is different from the one used when generating the original token, it can be assumed that the user has logged in since the password reset request was issued. This means that the user either remembered their password, or someone else made the password reset request in their place. The application checks if the newly generated token matches the one that's obtained from the GET parameters. If the tokens match, the user will be logged in, and the one-time login token expires. The user will then be redirected to their profile page where they can change their password. At no point in time are passwords stored in plain text.

## User Privileges

Drupal contains a system of user *Roles* and *Permissions*. Each of these *Roles* can be granted certain *Permissions* such as creating and editing content, or the ability to make configuration changes. This functionality is visualised in Figure 3.6.

PERMISSION	ANONYMOUS USER	AUTHENTICATED USER	ADMINISTRATOR
<b>Block</b>			
Administer blocks	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<b>Comment</b>			
Administer comments and comment settings	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
View comments	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 3.6: Granting Permissions to certain Roles in Drupal 7.x

Two types of users are recognized by default: authenticated users (those who are logged in) and anonymous users (those who are not). A third type of user, the administrator, is created when installing Drupal. By default, this is the only user who can edit user Roles and Permissions and perform any update to the site (content creation, installing new modules, configuration changes, etc.). New users can register freely, but administrative approval is required. Anonymous and authenticated users' Permissions are very limited by default. Both can view comments and published content, but only authenticated users

can post comments. Trying to access any unpublished content or unauthorized functionality will result in the *Access denied* error mentioned previously.

### **Added Components**

Drupal core functionality can be extended by installing Modules and Themes. Drupal provides a Modules directory which contains over 23000 modules [15]. Modules can range from extending controls over how certain content gets shown on a web page, to adding new text editors for writing posts or comments.

A new Module will be reviewed manually by code reviewers before it is allowed to be released. These code reviewers will examine the code and provide feedback to the developer. Any requested changes must be implemented before the module can be accepted. Once the module has been fully approved, it may be uploaded to the Drupal Module repository [15] and can be downloaded and installed by others.

## Chapter 4

# CMSs and the OWASP Top 10

As concluded in Chapter 2, writing secure web applications requires a lot of effort, and many different aspects must be considered. Using web content management systems is a popular solution for website developers and maintainers because a website no longer needs to be built from the ground up. Web content management systems enable inexperienced users to set up and maintain their own website, even without having any programming knowledge.

This chapter is structured so that each section talks about a different content management system. Each section is divided up into subsections that correspond to the OWASP Top 10 security risks described in Chapter 2. Each subsection contains a detailed description of the techniques used to mitigate these security risks. We will focus on the criteria set forth in the conclusion of Chapter 2 to review these open source content management systems in a well-structured, and objective way. We can then use the results of this research to compare content management systems with each other and try to figure out which is the most secure at which fields. We can assume that if these mitigation techniques are implemented correctly, the application will be relatively well protected against the most common security vulnerabilities. If no citation is given for a certain example, this means that the provided information was obtained from directly analysing the source code of the application. In most cases, the source code was analysed to ensure that the provided documentation was not out of date or contained wrong information.

### 4.1 WordPress

In Section 3.1, we provided a general introduction to the WordPress framework. In this section, we will provide more technical details concerning the techniques implemented to ensure safety against the security risks mentioned in the OWASP Top 10 [1].



### 4.1.1 Injection

In this subsection, we will continue our review of WordPress’s database system. In Subsection 3.1.1, we had already mentioned that WordPress doesn’t contain a Database Abstraction Layer and requires a MySQL database. As will see further on in this subsection, WordPress does provide some level of abstraction for different types of queries. First we will take a look at the general way of executing queries, after which we will take a look at the abstraction layer. Formatting SQL statements in WordPress requires two steps:

- Actively sanitizing the input variables, preferably by using the `prepare` function.
- Executing the query using the `query` function.

WordPress’s `query` function doesn’t use prepared statements, but rather assumes that the provided query string has been sanitized beforehand. The `query` function doesn’t sanitize input data automatically, making it more likely that an unsafe query could get executed (for instance if a developer forgets to perform the extra sanitisation step). The `prepare` function *prepares* the query for execution and escapes all input data by feeding it through either the `addslashes` function, or through the deprecated – and soon to be removed [51] – `mysql_real_escape_string` function. It does this based on whether a MySQL database handle<sup>1</sup> is available and on a boolean value that’s set to `false` by default, causing the `addslashes` method to be used primarily.

Using the `prepare` function is comparable to PHP’s `sprintf`<sup>2</sup> function. Its function signature and an example of its usage can be found in Example 4.1

```
1 function prepare( $query , $args = null );  
2  
3 wpdb::prepare( "SELECT * FROM 'table' WHERE 'column' = %s  
    AND 'field' = %d", 'foo', 1337 );
```

Example 4.1: WordPress’s `prepare` function signature and usage<sup>3</sup>

The `query` parameter contains the SQL statement to be sanitized, which can contain the following `sprintf`-like directives:

- `%d` – for decimal numbers
- `%f` – for floating-point numbers
- `%s` – for strings
- `%%` – literal percentage sign, no argument needed

<sup>1</sup>The object returned by the `mysql_connect` function, representing the database connection

<sup>2</sup>More information about the `sprintf` function can be found at: <http://php.net/manual/en/function.sprintf.php>

<sup>3</sup>More information about the `prepare` function can be found at: [http://codex.wordpress.org/Class\\_Reference/wpdb](http://codex.wordpress.org/Class_Reference/wpdb)

This behaviour emulates that of prepared statements, where the actual values are entered separately from the query. The `prepare` function sanitizes each variable and replaces the corresponding directive in the query. The query is then safe to be executed by the `query` function. As mentioned previously, WordPress contains a certain level of abstraction, as it provides different functions for different types of queries. The `query` function, however, can be used to perform any type of query on the database [59]. For clarity, we provide Example 4.2, which shows the function signature of WordPress’s `insert` function and an example of its usage.

```
1 function insert( $table , $data , $format );
2
3 $wpdb->insert (
4     'table' ,
5     array (
6         'column1' => 'value1' ,
7         'column2' => 123
8     ) ,
9     array (
10        '%s' ,
11        '%d'
12    )
13 );
```

Example 4.2: WordPress’s `insert` function signature and usage<sup>4</sup>

The first parameter of the `insert` function is the name of the table on which the insertion will occur. The `data` parameter is an array of *column, value* pairs that are to be inserted. The WordPress API reference [59] clearly states that these values should not be sanitized prior to being supplied to the `insert` function as sanitisation will occur when the `prepare` function is called internally. This means that `GET` and `POST` variables must be actively stripped of slashes before inserting them in the database. This is true because WordPress automatically escapes `GET`, `POST`, and `COOKIE` values. We will provide more information about this behaviour in the next paragraph. The final (optional) parameter is an array of formats to be mapped to each of the values in the `data` parameter. Specifying the format can provide an extra level of security as string values will be filtered from variables expected to be numbers. If the format parameter is omitted, all variables will be assumed to be of type `string`. The `insert` function will then transform the list of arguments into an `INSERT` statement with placeholder directives where the values are to be inserted. This query is then fed through the `prepare` function together with the array of *column, value* pairs, and finally the query is executed using the `query` function.

WordPress emulates PHP’s `magic_quotes_gpc` functionality by first stripping any slashes from `GET`, `POST`, and `COOKIE` values that might have been added, for instance those inserted by `magic_quotes_gpc`. It then actively sanitizes these and the `SERVER` *superglobal* variables by escaping their values using

<sup>4</sup>Source: [http://codex.wordpress.org/Class\\_Reference/wpdb](http://codex.wordpress.org/Class_Reference/wpdb)

the `addslashes` function. The `magic_quotes_gpc` function has been deprecated as of PHP version 5.3.0 and was removed as of PHP version 5.4.0 [52]. When enabled, this function would automatically escape single and double quotes, backslashes and NULs from `GET`, `POST`, and `COOKIE` variables by adding a backslash in front of those characters. WordPress forcing this behaviour on all `GET`, `POST`, `REQUEST`, `COOKIE`, and `SERVER` variables means that they must first be stripped of slashes before using them in the `prepare` function. Not removing the slashes added by the magic quotes functionality could potentially corrupt the data, as unsafe characters will be sanitised again by the `prepare` function. Furthermore, automatically escaping data can give developers a false sense of security, causing them to use these variables directly in a query, bypassing the `prepare` function. For instance, in Example 2.8 we saw that escaping input expected to be a number, will not prevent SQL injection.

On a final note, the *WordPress Coding Standards Handbook* [65] recommends that developers should avoid touching the database directly, and states that:

*If there is a defined function that can get the data you need, use it. Database abstraction (using functions instead of queries) helps keep your code forward-compatible and, in cases where results are cached in memory, it can be many times faster.*

## Least privilege

Although WordPress core requires only the `SELECT`, `INSERT` and `UPDATE` privileges, sometimes changes have to be made to the database when upgrading WordPress. Therefore it is recommended that `CREATE` and `ALTER` privileges are also enabled [5]. Some plugins might require `CREATE`, `DROP` or `DELETE` privileges, but as these are not required by WordPress core, they needn't be enabled by default.

### Suggested by the WordPress installation guide<sup>5</sup>

In step 2 of the WordPress installation guide, *Create the Database and a User*, they offer instructions for three different database/user creation methods. However, the creation methods that are mentioned (namely cPanel, phpMyAdmin and MySQL command line), suggest granting `ALL PRIVILEGES` to the database user. Given that only `SELECT`, `INSERT` and `UPDATE` privileges are required for WordPress core to function correctly, granting the database user `ALL PRIVILEGES` is highly unnecessary.

## 4.1.2 Authentication and session management

In this subsection, we will provide more information about how WordPress's authentication and session management mechanism works. In Subsection 3.1.1 we took a look at *User registration and credentials*, providing more information about the registration procedure. Then we talked about how the *Password recovery* mechanism works. We will now continue on, explaining more about

---

<sup>5</sup>[http://codex.wordpress.org/Installing\\_WordPress](http://codex.wordpress.org/Installing_WordPress)

WordPress *Sessions*. We will use elements from the OWASP ASVS Authentication Verification Requirements and Session Management Verification Requirements as a benchmark to test how secure these mechanisms are.

## Sessions

WordPress's session management mechanism uses cookies only. By default, a cookie's expiration date will be set so the cookie gets deleted once the browser closes. If the *Remember me* option was checked, the cookie will expire after 2 weeks. Up to four cookies will be set at a time, each with a specific purpose, and each with the `http only` flag set to true, making it harder for attackers to gain access to the session cookie value. We will now provide more information on each of the cookies that are set by WordPress when a user logs in.

1. Plugins cookie: `wordpress_[siteurl-hash]`<sup>6</sup> (always)
  - **Contains:** Clear text username|expiration date|random hash.
  - **Applies to:** `/[wordpress-root]/wp-content/plugins`
  - **Purpose:** Unknown. No information concerning the purpose of this cookie could be obtained. Removing the cookie did not seem to have an effect on any functionality.
2. Admin area cookie: `wordpress_[siteurl-hash]` (always)
  - **Contains:** Clear text username|expiration date|same hash as plugins
  - **Applies to:** `/[wordpress-root]/wp-admin`
  - **Purpose:** This cookie applies only to the administrative area and allows a user to change their settings, view and edit their post, or in the case of more privileged users: allow them to view, edit and approve other users' posts. Because contributed posts are always shown sanitized in the administrative area, it is more unlikely that this cookie could get stolen. Posts by users with the *Editor* user role or higher have the `unfiltered_html` capability, which means that their posts will be shown unsanitized in front facing web pages. In the administrative area, however, even posts by these users are shown sanitized [73]. Only users with valid *Logged in* and *Admin* cookies are allowed access to the administrative area. If an attacker is somehow able to steal the *Logged in* cookie from the front facing web pages, they still will not be able to access the administrative area for that particular user.
3. Logged in cookie: `wordpress_logged_in_[siteurl-hash]` (always)
  - **Contains:** Clear text username|expiration date|different hash
  - **Applies to:** `/[wordpress-root]/`

---

<sup>6</sup>`[siteurl-hash]` is an MD5 hash of the site URL

- **Purpose:** This cookie applies to the WordPress root directory and allows the application to determine which user is logged in. All front facing pages are kept completely separated from any administrative functionality. The administrative area requires both the *Logged in* and *Admin* cookies to be validated before access is granted to the administrative area. This is possible because the *Logged in* cookie applies to the WordPress root directory, meaning it will also be available in the administrative area. In case either of the cookies is invalid, a user will automatically be logged out and redirected to the login page [63].
4. Logged in cookie: `wordpress_logged_in_[siteurl-hash]` (under certain circumstances)
- **Contains:** Clear text `username|expiration date|same hash as logged in`
  - **Applies to:** `/[wordpress-home]/`
  - **Purpose:** This cookie is set in case the home page differs from the install directory. For instance, the site home page might be located at `http://www.example.com/`, while WordPress is installed in `http://www.example.com/wordpress`. By default, `[wordpress-root]` and `[wordpress-home]` are the same.

Other cookies that are set by WordPress are a `wordpress_test_cookie` and a `wp-settings-time-[UID]`<sup>7</sup> cookie. The first cookie is used to check if the browser supports setting cookies. The second cookie is used to customize the user's view of the admin and main site interface [63].

### Cookie values

Authorization cookie hashes are generated by using the `wp_generate_hash` function, which uses a keyed Hash Message Authentication Code (HMAC). Message Authentication Codes are generally used to verify the data integrity and the authenticity of a message [2]. It involves using a cryptographic hash function in combination with a secret cryptographic key. In WordPress, authentication cookies are generated in three steps:

1. A key is generated by concatenating the user's username, a fragment of their password, and the expiration time of the cookie. The result of this concatenation is hashed using the `hash_hmac` PHP function. The key used during this step is generated by concatenating a secret `AUTH_KEY` and `AUTH_SALT`, which are generated during installation. The cryptographic function used is MD5.
2. A cookie hash is generated by concatenating the user's username with the expiration time of the cookie. The result is then hashed using `hash_hmac` with the key generated in Step 1. The cryptographic function used is MD5.

---

<sup>7</sup>[UID] is the user's unique user ID

3. The final cookie value is obtained and set by concatenating the user's username, the expiration time of the cookie, and the hash generated in Step 2.

Because a secret key is used in conjunction with a cryptographic function, it's unlikely that an attacker would be able to guess the cookie value. For instance, in [2] Mihir Bellare et al. provided an example of the infeasibility of a Birthday Attack on the HMAC method when MD5 is used as the cryptographic function. It would require an attacker to obtain the authentication of  $2^{64}$  blocks of data using the same key. Furthermore, it would take 250000 years on a 1 Gbit/sec communication link to process all the data required by such an attack.

### 4.1.3 Cross-site scripting

WordPress's approach to mitigating cross-site scripting attacks works on multiple levels. Whether or not filters are applied is dependent on the *user role* of the user creating the content. Users with the *Editor* role or higher have the `unsanitized_html` capability, which allows them to post content (pages, posts, links, comments, etc.) that contain HTML and JavaScript code. This means that data from these users is neither sanitized when being inserted into the database, nor when it is retrieved and displayed on front facing pages. Contributed content from users with less privileged roles will get properly sanitized, both on insertion and retrieval. As suggested in the *OWASP XSS Prevention Cheat Sheet*, WordPress provides multiple filtering methods to be used based on the type of data and the HTML context in which it is used [61]. These filtering methods are listed in Table 4.1.

We will provide some more information about the `wpkses` filter, whose function signature can be found in Example 4.3.

```
1 wpkses( $fragment , $allowed_html , $protocols = null );
```

Example 4.3: WordPress's `wpkses` function signature<sup>8</sup>

This function will sanitize the value in the first parameter, based on the allowed HTML tags provided in the second parameter. An array of allowed HTML tags based on a given context can be retrieved using the `wpkses_allowed_html` function, which will also supply a list of allowed attributes for each tag. The third parameter is an optional array of allowed protocols. If this parameter is not set, WordPress will use a default set of allowed protocols, namely: `http`, `https`, `ftp`, `ftps`, `mailto`, `news`, `irc`, `gopher`, `nntp`, `feed`, `telnet`, `mms`, `rtsp`, `svn`, `tel`, `fax`, and `xmpp`.

The `wpkses` function will first remove all NULL characters from the input. It will then remove any HTML JavaScript entities found in early versions of Netscape 4, using the `wpkses_js_entities` function, after which it will normalize HTML entities. The second to last step is to call `wpkses_hook`, allowing

<sup>8</sup>More information about the `wpkses` function can be found at: [http://codex.wordpress.org/Function\\_Reference/wpkses](http://codex.wordpress.org/Function_Reference/wpkses)

any other sanitisation filters registered to the `pre_kses` hook to run. As a final step, all HTML tags, including malformed HTML tags, will be split into an element and an attribute list, after which all illegal attributes will be removed.

Source Format	Target Format	Function	What It Does
Plain text	HTML	<code>esc_html</code>	Encodes special characters into HTML entities and validates string as UTF-8
		<code>esc_attr</code>	Same as <code>esc_html</code> . Used when escaping HTML attribute values
		<code>esc_textarea</code>	Encodes text for use inside a <code>&lt;textarea&gt;</code> element
		<code>sanitize_text_field</code>	Sanitizes a string from user input or from the database. Checks for invalid UTF-8, converts single <code>&lt;</code> characters to entity, strips all tags, removes line breaks, tabs and extra white space, and strips octets
HTML	HTML	<code>wp_kses</code>	Makes sure that only the allowed HTML element names, attribute names, and attribute values will occur. Any slashes from PHP's magic quotes must be removed before calling this function
Plain text	JavaScript	<code>esc_js</code>	Escapes text strings for echoing in JavaScript. It is intended to be used for inline JavaScript
Plain text	URL	<code>urlencode_deep</code>	Encodes an array of strings for use in a URL. Calls the <code>urlencode</code> PHP function on each element of the array.
URL	HTML	<code>esc_url</code>	Encodes characters as HTML entities and strips out harmful protocols, such as <code>javascript:</code>

Table 4.1: WordPress's Data Validation Filters<sup>9</sup>

#### 4.1.4 Direct object references

As mentioned in Chapter 3, WordPress contains a system of *Roles* and *Capabilities*. The default user *roles* are listed below and are ordered from most to least allowed *Capabilities*:

<sup>9</sup>Source: [http://codex.wordpress.org/Data\\_Validation](http://codex.wordpress.org/Data_Validation)

- **Super Admins** can access all features. This user role is only available when running a network of WordPress sites.
- **Administrators** can access all the administration features within a single site.
- **Editors** can publish and manage posts, including other users' posts.
- **Authors** can only publish and manage their own posts.
- **Contributors** can write and manage their own posts, but cannot publish them.
- **Subscribers** can only manage their profile.

### 4.1.5 Security misconfiguration

#### Default settings

There are no default accounts or passwords used by WordPress. WordPress comes installed with two default plug-ins, and two default themes. Neither of the plug-ins are active by default, and only one of the two themes is active. By default, all PHP or MySQL error reporting is turned off during the bootstrapping phase of a page load, so that no information is leaked through error messages. If the `WP_DEBUG` constant (global variable) is set to true, all error messages and warning will be displayed. By default, this constant is set to false [64].

#### Updating core, plug-ins, and themes

When a new version of WordPress core is available, all authenticated users will get a notification when they visit the administrative area. Anyone except a user with the *Administrator* role will get a message to notify the site administrator about the update. An administrator is able to update the site either manually or automatically. Updating automatically will replace all files except the `wp.config.php` file containing the site's configuration preferences [62]. It's recommended to back up the database and files prior to updating.

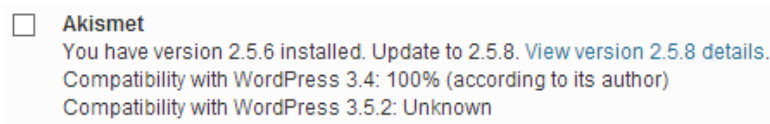


Figure 4.1: WordPress update indicator for the *Akismet* plug-in



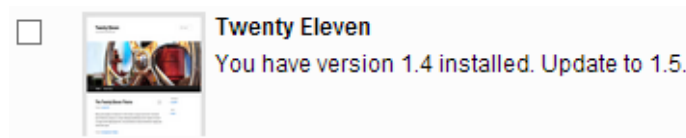


Figure 4.2: WordPress update indicator for the *Twenty Eleven* theme

### WordPress release cycle

WordPress is developed using a release cycle which lasts about 4 months. This means that there should be about 3 planned releases per year. In reality, however, there is an average of about 10 core releases per year, and just over 34 days between releases. These in-between releases fix reported bugs and security issues and will be discussed further in Chapter 5. A release cycle follows the following pattern [70]:

- Phase 1: Planning and securing team leads
- Phase 2: Development work begins
- Phase 3: Beta
- Phase 4: Release Candidate
- Phase 5: Launch.

## 4.1.6 Sensitive data exposure

### User credentials

User credentials are stored in two different tables of the database. The `wp_users` table stores user ID, username, password, *nice name*, e-mail address, website and display name together with the date on which the user registered, an activation key in case a password recovery request was made, and their current status. The `wp_usermeta` table stores meta data about each user like preferences and user role (see Section 4.1.4). Passwords are hashed using the portable fallback MD5 algorithm in the Portable PHP Password Hashing Framework (phpass, see Appendix A). WordPress uses phpass’s final MD5 fallback by default in order to remain compatible with older PHP versions [66]. Passwords are salted using an 8 byte string of random characters and stretched by performing 8192 iterations. All other credentials are saved in plain text.

### 4.1.7 Function level access control

Before accessing certain sensitive resources, the `current_user_can` function is called. Its function signature can be found in Example 4.4.

```
1 function current_user_can( $capability , $args );
```

Example 4.4: WordPress’s `insert` function signature and usage<sup>10</sup>

The `current_user_can` function will check if the current user has the appropriate capability or role to access the requested resource. It takes two arguments, the first of which is a string containing the necessary capability. The second argument is optional and is capability-dependent. For instance, if one wants to find out if a certain user is allowed to edit a post with ID 123, the `current_user_can` function will be called as follows: `current_user_can('edit_post', 123)`. If an unauthenticated user tries to access a restricted page, they will first be redirected to the login page. After successfully logging in, they will be redirected to the resource they were trying to access, whereupon user access will be checked.

### 4.1.8 Cross-site request forgery

In order to prevent Cross-site request forgery exploits, WordPress uses a form of CSRF challenge tokens, called *nonce* [67], included in a hidden form field. This token is valid for up to 24 hours and is generated by the `wp_create_nonce` function, which is shown in Example 4.5.

```
1 function wp_create_nonce($action = -1) {  
2     $user = wp_get_current_user();  
3     $uid = (int) $user->ID;  
4  
5     $i = wp_nonce_tick();  
6  
7     return substr(wp_hash($i . $action . $uid, 'nonce'),  
8                 -12, 10);  
9 }
```

Example 4.5: WordPress's `wp_create_nonce` function

A nonce tick is equal to `ceil(time() / ( 86400 / 2 ))`<sup>11</sup>. The hash function that's used is the same HMAC hash function used to generate cookie values, which uses secret keys generated during installation (see Subsection 4.1.2). Because of the workings of these functions, nonces are unique to the WordPress install, to the user, to the action, to the object of the action, and to the time of the action. If any of these things variables changes, the nonce is considered invalid. If it is considered invalid, the nonce value will not be reset, however, as suggested in the OWASP *Cross-Site Request Forgery Prevention Cheat Sheet*.

Another way that WordPress's nonces are not up to par with OWASP's recommendations is that they are not associated with the user's current session [42], nor are they invalidated if the user's session ends. WordPress nonces are valid for a 24 hour period, meaning that if a user logs out and back in within a 24 hour period of the nonce's creation, it will still have the same value as before<sup>12</sup>.

<sup>10</sup>Source: [http://codex.wordpress.org/Function\\_Reference/current\\_user\\_can](http://codex.wordpress.org/Function_Reference/current_user_can)

<sup>11</sup>86400 seconds = 24 hours

<sup>12</sup>For a full security advisory of this vulnerability, visit <http://www.webapp-security.com/wp-content/uploads/2012/04/Wordpress-3.3.1-Multiple-CSRF-Vulnerabilities6.txt>

### 4.1.9 Components with known vulnerabilities

WordPress will show notifications to all registered users who enter the administrative area, when updates are available to either core or plug-ins. Less privileged users will be prompted to report this to an administrator, while more privileged users will be able to install the updates from the administrative interface. This subsection is structured somewhat differently from the others, as we will answer the criteria set forth in the conclusion of Chapter 2 directly:

- **Is there a versioning system in place where the current version of an added component can easily be identified?**

There are clear versioning systems in place indicating to the administrator or anyone with the appropriate privileges which version of a certain plug-in is currently in use, and the version number of the latest version. There is also an indication that specifies to what extent the plug-in is compatible with the current and latest version of WordPress core (Figure 4.1). There is a comparable indication for any theme that might be installed. Plug-ins and themes can also be automatically updated from the *Plugin* and *Themes* pages in the administrative area.

- **Are security issues disclosed publicly?**

Security issues are not disclosed publicly, but there is a security issue reporting system available where security issues in WordPress core can be reported. Security issues found in plug-ins should be e-mailed to a special e-mail address.

- **Is there a project or security mailing list in place to warn users about new vulnerabilities?**

No, there is not. The only announcement given is when there is a new version of WordPress core available.

- **Are there any manual or automated checks in place in order to detect vulnerabilities before the components are released publicly?**

Any plug-in uploaded to be hosted on the WordPress website, but will be manually checked before being allowed. Plug-in developers can be e-mailed and asked to provide more information. After approval, the developer will be allowed to upload the plug-in to a Subversion repository, after which the plug-in will appear in the plug-ins browser automatically [69].

- **Is there any indication or warning given when searching for or trying to install components with known vulnerabilities?**

There is no indication when searching for and installing the plug-in from the administrative area. A known vulnerable plug-in can be installed just as easily as any other. There is a warning given, however, when visiting the plug-in page in a browser. For instance, in the case of the *Spicy Blogroll* example provided in Section 2.9, the following warning is given:

*This plugin hasn't been updated in over 2 years. It may no longer be maintained or supported and may have compatibility issues when used with more recent versions of WordPress.* -  
Source: <http://wordpress.org/plugins/spicy-blogroll/>

- **How are vulnerability reports dealt with and how quickly do flaws get fixed?**

Fixing vulnerabilities in plug-ins is up to its developer. A vulnerability report should be sent to both a special e-mail address for vulnerabilities in plug-ins and to the developer, if their e-mail address is available. Otherwise, issues can be reported in the support forum of the plug-in page. Reported vulnerabilities in WordPress core should get reviewed within 24 hours.

- **Are updates easy to install?**

Yes, updates to either core, plug-ins, or themes are installed automatically by the click of a button.

#### 4.1.10 Redirects and forwards

WordPress often uses redirects and forwards to redirect users after performing certain actions. For instance, when an unauthenticated user tries to access a page that requires authentication, they will be redirected to the login page. The URL they were trying to access is appended to the current URL in a GET parameter. After successfully logging in, they will be redirected to the page they were trying to access. Before rendering the page, however, the access control mechanism will check if the user has the appropriate capabilities to access the requested page. If no redirect parameter is supplied on the login page, the user will be redirected to the administrative dashboard.

WordPress provides two redirect functions that can be used to redirect users to different pages, in response to their actions on the site: `wp_redirect` and `wp_safe_redirect`. The `wp_redirect` function can be used to redirect users both internally and externally to a different domain. The `wp_safe_redirect` function can also be used to redirect users to internal locations, but only allows redirects to external domains if they are in a white-list of accepted domains. By default, this list contains only the domain of the current website, meaning that `wp_safe_redirect` doesn't allow redirects to any other domain.

In the conclusion of Chapter 2, we supplied multiple best practices to prevent flaws in redirects and forwards. We will now show how WordPress lives up to these best practices.

- **Don't use redirects and forwards** – WordPress uses redirects and forwards extensively throughout the application's core.
- **Don't base redirects and forwards on user parameters** – WordPress rarely bases the redirect location on user parameters. When it does, the `wp_safe_redirect` function is used to prevent redirecting users to external domains.
- **Make sure that the supplied value is valid** – `wp_sanitize_redirect` is used for both redirect methods. This function sanitizes a URL for use in a redirect by removing invalid characters. When `wp_safe_redirect` is used, the domain of the redirect URL will be checked against a white-list of trusted domains.

- **Make sure the user is authorized to be redirected there** – Access control is not checked before a user is redirected, but rather when the request to the redirect location is being processed. WordPress’s access control mechanism is discussed in subsection 4.1.4.

## 4.2 Joomla

In Section 3.2, we provided a general introduction to the Joomla framework. In this section, we will provide more technical details concerning the techniques implemented to ensure safety against the security risks mentioned in the OWASP Top 10 [1].

### 4.2.1 Injection

In Subsection 3.2.1, we explained that Joomla provides a Database Abstraction Layer. We also briefly mentioned that although string queries are supported, the preferred method of building queries is by *query chaining*. Query chaining consists of connecting a number of methods, with each method returning an object that can support the next method. The Joomla developers claim that this improves readability and simplifies the code. Joomla queries do not use prepared statements, but rather expect developers to actively sanitise untrusted data. An example of query chaining can be found in Example 4.6.

```

1 $db = JFactory::getDbo();
2
3 $query = $db->getQuery(true);
4
5 $query
6     ->select(array('user_id', 'profile_key', '
7         profile_value', 'ordering'))
8     ->from('#__user_profiles')
9     ->where('profile_key LIKE \'custom.%\')
10    ->order('ordering ASC');
11 $db->setQuery($query);
12
13 $results = $db->loadObjectList();

```

Example 4.6: Joomla query chaining<sup>13</sup>

First a database connection is set up, by calling the `JFactory::getDbo` function. This function will return an instance of the `JDatabaseDriver` object, which is used as an abstraction for the actual database connection. This object is then used to create a new query. If the boolean value fed to the `getQuery` function

<sup>13</sup>Source: [http://docs.joomla.org/J3.1:Accessing\\_the\\_database\\_using\\_JDatabase](http://docs.joomla.org/J3.1:Accessing_the_database_using_JDatabase)

were false, the last query that was created will be returned in stead of a new, blank query.

Each `query` object is made up of multiple `query elements`. Query elements are an abstraction for SQL statements, clauses and keywords. When a query is executed, all of these query elements will be put together to create a regular, valid SQL query. Each method fills in a specific query element. When executing a query, these query elements will be transformed into corresponding statements, clauses and keywords. For instance, in the example above, the `select` method is used to supply an array of column names to the `SELECT` query element of the query. The `from` method will add a table name to the `FROM` clause of the query, while the `where` method adds a conditional clause to the query. Finally, the `order` method specifies the order in which to present the results. When executed, the query in the example above will select all records from the user profile table where *key* begins with *custom..* All types of queries, including `INSERT`, `SELECT`, `UPDATE`, `DELETE`, etc. use the same kind of query object, but use their own specific object methods to transform the query object into the desired query type.

Unfortunately, the Joomla API documentation is not consistent in the information they provide to developers. For instance, the example above makes proper use of the Joomla DBAL, using query chaining to build and execute queries [26]. On the other hand, examples shown in the Joomla documentation on secure coding guidelines [32], are provided as shown in Example 4.7.

```
1 //casting expected integer values
2 $query = 'SELECT * FROM #__table WHERE 'id=' . (int) $id
3       ;
4 //escaping string values
5 $query = 'SELECT * FROM #__table WHERE 'field' = ' . $db
        ->quote( $field );
```

Example 4.7: Joomla secure coding guidelines: Constructing SQL queries<sup>14</sup>

This is inconsistent and could cause developers to write insecure code. Properly escaping user data is not mentioned in the Joomla DBAL documentation [26], which could cause developers to assume that proper sanitisation is being handled automatically. As this is not the case, developers must make sure to escape all untrusted data themselves, for instance by casting expected integer values to `int`, and using the `quote` function. This function will add quotes around string values and escape the contents of the string dependent on the underlying database escaping scheme.

Because both the query chaining and regular query methods are allowed, developers could potentially use queries in an insecure way. Example 4.14 about using the Drupal `db_query` in an insecure way is also applicable here. An equally insecure SQL query could be fed into Joomla's `setQuery` method, leaving no reason and no way for the DBAL to block such a query.

<sup>14</sup>Source: [http://docs.joomla.org/J3.1:Accessing\\_the\\_database\\_using\\_JDatabase](http://docs.joomla.org/J3.1:Accessing_the_database_using_JDatabase)

## Least privilege

Joomla core requires only the following privileges: SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, INDEX, ALTER, CREATE TEMPORARY TABLES, LOCK TABLES [50].

### Suggested by Joomla! installation guide<sup>15</sup>

Joomla doesn't provide any official instructions for creating a database or database user in their installation guide. Instead they refer their inexperienced users to the web server provider. Joomla used to provide official instructions<sup>16</sup> but these were considered redundant and removed as of April 2013.

## 4.2.2 Authentication and session management

In Subsection 3.2.1, we provided more information about Joomla's authentication and session management systems. We took a look at Joomla's default user registration procedure, and which user credentials are obtained. Then we analysed the password recovery system, to make sure that this is also implemented securely. In this subsection, we will continue our analysis, taking a closer look at Joomla's session management mechanism.

### Sessions

Joomla uses PHP sessions and session cookies. It uses default PHP session IDs, which are stored in the database and associates with the user. Before a new session is started, any existing session data is destroyed. Privileged users can change the session lifetime from within the administrative interface. Sessions are set to invalidate after 15 minutes of inactivity by default. Session cookies are destroyed when a user closes her browser.

Joomla is comprised of two distinct areas: a public front facing interface and a back end administrative interface. The front facing interface provides some limited functionality to certain users, like creating, editing, or publishing posts, while the back end interface provides administrative functionality to more privileged users. A user has to explicitly authenticate in order to gain access to the administrative interface. A different session will be started, and a different session cookie will be set. This means a user could be logged in to the front facing interface with one account and to the administrative interface with another. This also means that a user has to explicitly log out of the administrative interface, as only logging out on the front facing interface will not cause the user to be logged out of the administrative interface.

## 4.2.3 Cross-site scripting

Joomla provides filtering techniques for both input and output filtering. Finding the correct information on Joomla's filtering techniques proved to be quite cum-

<sup>15</sup><http://help.joomla.org/content/view/37/132/>

<sup>16</sup><http://docs.joomla.org/Goals:Installation>

bersome. The primary search results always returned an outdated API document [32], containing filtering techniques using a deprecated Joomla class, called `JRequest`. This class accounted for the fact that PHP's `magic_quotes_gpc` functionality might have been enabled. It is for this reason that all core components in Joomla 2.5.x still use `JRequest`. As of Joomla 3.0+ magic quotes is required to be disabled. Now the `JInput` class is used for sanitising untrusted input data instead [31]. The `JInput` class is a wrapper around PHP globals<sup>17</sup>, providing automatic input sanitisation, based on a requested filter. The way a value is retrieved using `JInput` is shown in Example 4.8.

```
1 $jinput = JFactory::getApplication()->input;
2
3 $jinput->get('varname', 'default_value', 'filter');
```

Example 4.8: Joomla `JInput` usage

The first parameter of the `get` function is the name of the variable to be retrieved. The second parameter is a default value to be returned if the variable to be retrieved doesn't exist. The final parameter dictates which filter must be used. By default, the filter is set to `CMD`, but can be set to any of the following [31]:

- **INT** or **INTEGER** – Returns the first positive or negative integer value found in the input data, ignoring any non-integer values. If multiple integer values are contained in the same input variable, for instance divided by spaces, only the first integer will be returned. The value is casted to `int` prior to being returned.
- **UINT** – Does the same as the **INT** or **INTEGER** filter, but returns the modulus of the integer value, ensuring only positive integers are returned.
- **FLOAT** or **DOUBLE** – Returns the first positive or negative floating point value found in the input data, ignoring any non-numeric characters except the floating point. If multiple floating point values are contained in the same input variable, for instance divided by spaces, commas, or dots, only the first matched floating point value will be returned. The value is casted to `float` prior to being returned.
- **BOOL** or **BOOLEAN** – Casts the input to a boolean value. Non boolean input will automatically be converted into a `true` or `false` value.
- **WORD** – Makes sure that the input data is a word. Only alphabetic characters<sup>18</sup> and underscores are allowed. All other characters will be removed from the input data.
- **ALNUM** – Makes sure that the input data is alphanumeric. Only alphabetic characters and numbers are allowed. All other characters will be removed from the input data.

---

<sup>17</sup>GET, POST, COOKIE, etc.

<sup>18</sup>characters a through z, and A through Z



- **CMD** – Removes any non-alphanumeric characters, except underscores, dots, and hyphens from the input data. Whitespaces at the beginning of the value are trimmed prior to being returned.
- **BASE64** – Makes sure that the input data is a valid Base64 encoded value. Any non-alphanumeric characters, except "=" will be removed.
- **STRING** – Tries to convert the input data to plain text before removing all disallowed and malformed HTML tags and attributes from the input.
- **HTML** – Removes all disallowed and malformed HTML tags and attributes from the input data.
- **ARRAY** – Casts the input data to an array.
- **PATH** – Makes sure that the input data is a valid path.
- **RAW** – No filtering is applied. The original input data is returned as is.
- **USERNAME** – Makes sure the input data is a valid username. All disallowed ASCII characters will be removed from the input data<sup>19</sup>.

Most of the filtering techniques rely on matching the input data against a regular expression, either ignoring or removing disallowed characters. The filtering techniques used for string and HTML filtering, however, use input sanitisation techniques based on black-listing disallowed tags and attributes. Different kinds of users have different privileges when it comes to input validation. In Subsection 4.2.4, we will provide more information about these different kinds of users, called User Groups. For now, we will just list the kinds of filter types available in Joomla by default, and which User Groups they apply to by default:

- **Default Black List** – Input data is filtered using a default black list of disallowed HTML tag and attribute names. Default tags include `applet`, `iframe`, `script`, etc. and will be removed from the input data. Default black listed attributes include `action`, `background`, `codebase`, etc. Additional black listed tags can be added to Groups by more privileged users. User Groups that use this filtering type by default are *Guest*, *Manager*, *Author*, *Editor*, and *Publisher*.
- **Custom Black List** – Input data is filtered using a custom black list of disallowed tag and attribute names. Custom tags and attributes can be supplied in the same way as the additional black list items for the Default Black List, but now the tags and attributes in the default blacklist will be ignored. No User Groups use this type of filtering by default.
- **White List** – Input data is filtered using a custom white list of supplied tags and attributes. These tags and attributes must be supplied by a more privileged user. Items in the Default Black List will be ignored. There is no Default White List. No User Groups use this type of filtering by default.

---

<sup>19</sup>Any character from the input data that matches to the following regular expression is removed: `/[\x00-\x1F\x7F<>'"%&]/i`

- **No HTML** – Input data is stripped of all HTML tags when it is saved. This filtering type applies to the *Registered* User Group.
- **No Filtering** – Raw input data will be used without being filtered at all. This filtering type applies to the *Administrator* and *Super Users* User Groups.

Tag and attribute names are converted to all lower case characters before being compared to the disallowed tags and attributes in the black list, and the allowed tags in the white list. This makes sure that a malicious user will not be able to trick the system by randomly capitalising certain characters in the tag or attribute names. Attribute values are stripped of disallowed characters and checked for malformedness.

#### 4.2.4 Direct object references

In Subsection 3.2.1 of the previous chapter, more general information was provided about Joomla's mechanism to distinguish user privileges from one another. In this subsection, we will provide even more information, specifically concerning the hierarchical system of User Groups.

##### More on User Groups

By default, Joomla contains nine User Groups. Which Actions are allowed is dependent on the Permissions defined on each Action of a certain Group and its parent. Access Levels are automatically inherited down the hierarchy. For example, a user of the *Manager* User Group and its children are allowed to see an overview of all registered members. This is defined as an Access Level. The *Administrator* User Group, who is the child of the *Manager* User Group, inherits the Access Levels from its parent, allowing its members to view the list of registered users. The *Administrator* User Group also inherits Permissions from the *Manager* User Group, but redefines certain Permissions allowing Administrators to not only view a list of registered users, but to edit these user's credentials as well. By default, Joomla contains the following hierarchy of Groups [33]:

- **Public** – Defines the top level of the hierarchy. This top level User Group is assigned to the *Public* Access Level, making all *Public* content visible to all Groups.
- **Guest** – The Guest Group is assigned to the *Guest* Access Level. Unauthenticated users are considered Guests.
- **Manager** – A member of the Manager Group can access the back end administrative area through the administrator interface. Their Permissions and Access Levels are restricted to content management: they can create or edit any content, and access back end features such as adding, deleting, and editing *Sections* and editing the *Front Page*. Managers are not allowed access to any user management controls,

and are denied the ability to install components and modules. If a manager logs in through the front end interface, they are considered Publishers, with the same Permissions and Access Levels.

- **Administrator** – This Group is allowed access to administrative functionality. Because an Administrator inherits from a Manager, they have at least the same Permissions in the back end interface as a Manager. Administrators, however, can also set options, install and delete components, and have access to user management controls. When an Administrator logs in through the front end interface, they will be treated like a Publisher, just like a Manager would.
- **Registered** – Members of the Registered User Group are allowed to log in through the front end interface. Registered users can't contribute content, but may be allowed access to areas such as a forum or a download section. Authenticated users are placed in the Registered Group by default.
  - **Author** – A member of the Author Group can post content through the front end interface, but is not allowed access to the back end administrative interface. They can submit content, but cannot directly publish any content. Content submitted by Authors must be approved by more privileged users. Authors can edit their own posts, but only after they have been published.
  - **Editor** – This Group allows its members to post and edit any content from the front end interface, including other people's content. They can also edit unpublished content, but cannot publish any content.
  - **Publisher** – This Group allows its members to post, edit and publish any content from the front end interface, including other people's content. Publishers can review and edit content, change publishing options, and determine when an article may be published. Publishers have the highest privileges of all front end users, without having access to the back end interface.
- **Super Users** – Super Users are allowed access to all administrative functionality, and are granted full access to all areas and content. Super Users can block users and change users credentials. When a Super User logs in through the front end interface, they are considered Publishers.

## 4.2.5 Security misconfiguration

### Default settings

As a security feature, Joomla requires the installation folder to be removed after installation. The installation procedure will not continue as long as the installation folder is not removed. If someone were able to access the installation

folder, they could potentially overwrite the installation by running the installer again [23].

Joomla doesn't contain any default accounts or passwords. A primary Super User account is created when Joomla is installed, but the person installing Joomla has full control over their username, password, and other credentials. By default, Joomla comes installed with a whole range of extensions, ranging from login Modules to layout Templates.

Joomla contains a Debug System which displays diagnostic information, language translation, and SQL errors. By default, the Debug System is disabled, ensuring that no information can leak through error messages. Joomla recommends leaving this option disabled on sites that are live and accessible by others, but provides the option to display debug information for development purposes.

As mentioned previously in Subsection 4.2.4, Joomla contains a system of *User Groups*, *Actions*, and *Permissions* that define which types of users can perform certain actions on the website. These settings are set securely by default, but they are highly customisable. For instance, care must be taken that no administrative functionality is granted to certain User Groups by accident.

## Updating core, plug-ins, and themes

Extensions can be updated from the administrative area by users with the right Permissions. The *Update* section of the *Extensions Manager* will display a list of Extensions that require updating, and allows one to perform an in-place update without having to upload and install the updated files directly. This system makes keeping core and extensions up to date significantly easier for experienced and inexperienced users alike.

### Joomla release cycle

Joomla is developed using a fixed release cycle. Every six months, the Joomla project releases a new minor or major version of Joomla [34]. Each released version is supported for a limited amount of time, whereby each fourth release is assigned long term support:

- **STS (short term support)** releases are supported for seven months. Their support ends one month after the next release of Joomla is released. They are one click upgrades to the next STS or LTS version.
- **LTS (long term support)** releases are supported for twenty-seven months.

STS versions are released in between LTS releases and introduce new features and changes that potentially break compatibility with the previous LTS. The LTS release that's released at the end of the cycle finalizes the work of the three STS releases. Extra updates and released versions are considered maintenance updates and include fixes for security issues. Usually, multiple vulnerability issues are fixed in a single maintenance update. There is an average of about 7.5 releases per year, and just above 42 days between releases of Joomla core, including maintenance updates.

## 4.2.6 Sensitive data exposure

### User credentials

Joomla stores users' password using salted MD5 hashes. Other user credentials including full name, username, e-mail address, etc. are stored together with the password in the `users` table. The primary *Super User* account does not have ID 1, but rather is given a random ID. Both Drupal's and WordPress's primary account have ID 1, which makes these account easier for attackers to target, as they know for certain which ID is associated with the primary administrator account.

A code snippet that shows how plain text passwords are hashed is shown in Example 4.9

```
1 ...
2 $salt = JUserHelper::genRandomPassword(32);
3 $crypt = JUserHelper::getCryptedPassword($array['password
4     '], $salt);
5 $array['password'] = $crypt . ':' . $salt;
```

Example 4.9: Code snippet of Joomla's password hashing algorithm<sup>20</sup>

Firstly, a salt value with a length of 32 characters is randomly generated, using the `genRandomPassword` function. The `getCryptedPassword` function will append the salt to the plain text password, and hash the result using MD5. Finally, the salt is appended to the resulting hash, separated with a ":" symbol, to indicate where the hash ends and the salt begins. This result of this concatenation is then stored in the database.

As Joomla core contains no mechanism to limit the number of login attempts, and seeing as MD5 is a very fast hashing algorithm, this method of password hashing is very prone to brute force attacks. Unlike Drupal or WordPress, Joomla doesn't use a password hashing framework like `phpass`, which uses stretching in conjunction to salting to make brute force attacks slower (see Appendix A). This means that brute force attacks could crack a weak password very quickly.

## 4.2.7 Function level access control

As mentioned in Subsection 4.2.4, Joomla contains a very extensive access control mechanism comprised of *User Groups*, *Access Levels*, *Actions*, and *Permissions*. Access Levels define which items<sup>21</sup> a user can view, while Actions and Permissions define what a user can do. Any time a user wants to view a certain item, the application checks whether or not the user has the right privileges

<sup>20</sup>Source: `libraries/joomla/user/user.php`

<sup>21</sup>Articles, menu items, modules, etc.

and acts accordingly. Checking if a user has the right privileges is done as follows [25]:

- Create a list of all the Access Levels that the user has access to, based on all Groups that the user belongs to. Also, if a Group has a parent Group, Access Levels for the parent Group are also included in the list.
- Check whether the Access Level for the item is on that list. If yes, then the item is displayed to the user. If no, then the item is not displayed.

The *Permissions* system defines what a user can do. When a user wants to initiate a specific action against an item, the system checks the permission for this combination of user, item, and action. If it is allowed, then the user can proceed. Otherwise, the action is not allowed [25]. Access control is implemented using the `authorise` function, whose function signature can be seen in Example 4.10.

```
1 function authorise($action , $assetname = null);
```

Example 4.10: Joomla access control function signature: `authorise`

The first parameter is the name of the action to check for permission. The second parameter is optional and defines the name of the asset on which to perform the action. The function will first check if the current user is the primary Super User. If this is the case, it will return true no matter what. Otherwise, it will check if the current user has the appropriate Permission defined on the requested Action. If the requested Action doesn't exist, or the user doesn't have the appropriate Permissions, access will be denied by default, making sure that the access control mechanism fails securely.

## 4.2.8 Cross-site request forgery

Joomla attempts to protect against CSRF by inserting an anti-CSRF token into each POST form and each GET query string that is able to modify something in the Joomla system [28]. Tokens change with each session and each user. A token is created using the `getFormToken` function. The code that is used to generate an anti-CSRF token is shown in Example 4.11

```
1 $hash = JApplication::getHash($user->get('id', 0) .  
    $session->getToken($forceNew));
```

Example 4.11: Joomla anti-CSRF token generation

The `getToken` function will either create a new form token, or reuse an old one, based on the value of the `forceNew` variable. The token generated by this function is created by concatenating a randomly generated 32 character string to the current session name, and hashing the results using MD5. This generated token is then concatenated to the current user's ID. It is then used as the seed

value for the `getHash` function. This function will concatenate the seed value to a random application secret, which is generated when Joomla is installed. The result of the concatenation is hashed using MD5 and returned. After all these steps are performed, the form token will be ready to use.

As suggested by the OWASP *Cross-Site Request Forgery Prevention Cheat Sheet*, Joomla's anti-CSRF tokens are associated with the current session. If the token is incorrect, the request will be ignored but the token will not be invalidated or regenerated.

#### 4.2.9 Components with known vulnerabilities

As mentioned in Subsection 3.2.1, Joomla contains five types of extensions. Each of these types of extensions could potentially open security holes in the web application. It's important that these extensions can be kept up to date easily, so that the website's maintainers are encouraged to do so. We will now answer the criteria set forth earlier in the conclusion of Chapter 2.

- **Is there a versioning system in place where the current version of an added component can easily be identified?**

The Joomla administrative area contains an *Extension Manager*. This Extension Manager provides a full overview of all installed Components, Modules, Plug-ins, Templates, Languages, etc. with their version numbers clearly visualised. Figure 4.3 shows part of the Extension Manager.




<input type="checkbox"/>	Name <sup>▲</sup>	Location	Status	Type	Version
<input type="checkbox"/>	Authentication - LDAP	Site		Plugin	3.0.0
<input type="checkbox"/>	Banners	Site		Module	3.0.0
<input type="checkbox"/>	Banners	Administrator		Component	3.0.0

Figure 4.3: Joomla Extension Manager

- **Are security issues disclosed publicly?**

All known vulnerabilities are listed in the Joomla Vulnerable Extensions List (VEL) [29]. This list contains vulnerability reports for Joomla core and extensions, but does not claim to be an up to date or complete list. The Joomla Vulnerable Extensions List states the following:

1. *We do NOT promise to test or validate these reports.*
2. *We do NOT guarantee the quality or effectiveness of any updates reported to us or listed here.*

Information is gathered from other sources, like vulnerability databases (see Section 1.3) and compiled into the Joomla VEL.

- **Is there a project or security mailing list in place to warn users about new vulnerabilities?**

People can register to the *Joomla! Security - Vulnerable Extensions* or *Joomla! Security - Recently Resolved Vulnerable Extensions* mailing lists in order to stay informed about recent vulnerabilities, or receive notices when vulnerabilities have been resolved.

- **Are there any manual or automated checks in place in order to detect vulnerabilities before the components are released publicly?**

The added extension will be checked according to four checklists. More information on this process can be found in Subsection 3.2.1.

- **Is there any indication or warning given when searching for or trying to install components with known vulnerabilities?**

Vulnerable components are removed from the extensions repository until the vulnerability has been resolved. Once it has been resolved, the extension can be resubmitted.

- **How are vulnerability reports dealt with and how quickly do flaws get fixed?**

Joomla has a *Security Strike Team* which focusses solely on managing and improving Joomla security. Once a vulnerability has been confirmed, a rough timeline will be sent to the person reporting the vulnerability, and they will be asked not to disclose the vulnerability to anyone else. A *go-public* date will be determined for announcing the vulnerability and the fix, before publicly announcing it to the world.

- **Are updates easy to install?**

As mentioned earlier, Joomla contains an Extension Manager, which makes installing updates much easier. Extensions that require updating will be shown in a list, where they can be updated with the click of a button.

#### 4.2.10 Redirects and forwards

Joomla makes use of redirects and forwards to direct users to different parts of the website. For instance, after successfully logging in, a user will be redirected to the home page of the website. If an unauthenticated user tries to access a resource that requires authentication, they will be prompted to log in before being redirected to the resource they were trying to access. Before the resource is shown, however, Joomla's Access Levels mechanism (see Subsection 4.2.4) will first check if the user has the appropriate privileges to view the requested resource and act accordingly.

Joomla's redirect functionality is implemented in the aptly named `redirect` function. This function can redirect users to both internal and external URLs. The function will first check if the URL is a relative internal link. If this is the case, the relative internal link will be transformed into an absolute link before



proceeding. The URL will be checked for validity. If the URL is invalid, the function will attempt to fix the URL. If the URL is valid, the user will be redirected to that location.

- **Don't use redirects and forwards** – Joomla uses redirects and forwards extensively throughout the application.
- **Don't base redirects and forwards on user parameters** – Redirects are sometimes based on GET parameters in the URL. The only requirement is that they are encoded in Base64, but redirection to an external domains is not blocked.
- **Make sure that the supplied value is valid** – The supplied URL will be checked for validity in the `redirect` function, prior to being used. Relative internal URLs will be transformed into full absolute URLs, and an attempt will be made to fix malformed URLs.
- **Make sure the user is authorized to be redirected there** – Access control is not checked before a user is redirected, but rather when the request to the redirect location is being processed. Joomla's access control mechanism is discussed in Subsection 4.2.7.

## 4.3 Drupal

In Section 3.3, we provided a general introduction to the Drupal framework. In this section, we will provide more technical details concerning the techniques implemented to ensure safety against the security risks mentioned in the OWASP Top 10 [1].

### 4.3.1 Injection

In Subsection 3.3.1 of the previous chapter, we mentioned that Drupal contains a Database Abstraction Layer to ensure that Drupal can be used with multiple kinds of database systems. In this subsection we will provide more technical information about Drupal’s database API and how security risks concerning *injection attacks* are handled.

#### Prepared statements

Most Drupal SELECT queries either use `db_query()` or `db_query_range()` [10]. These queries are based on PHP prepared statements (Section 2.1.3), making their usage comparable to prepared statements, but with some minor differences. In Example 4.12, two ways of selecting a list of the most recent 10 nodes authored by a given user are shown. Line 1 show the use of an unsafe static query, while line 6 shows the use of Drupal’s `db_query_range()` method.

```
1 $basic_sql_query =
2   "SELECT n.nid , n.title , n.created
3   FROM node n
4   WHERE n.uid = $uid LIMIT 0 , 10;"
5
6 $result = db_query_range(
7   'SELECT n.nid , n.title , n.created
8   FROM {node} n
9   WHERE n.uid = :uid ',
10  0 , 10 ,
11  array( ':uid ' => $uid ));
12 }
```

Example 4.12: Usage of Drupal’s `db_query_range()` method

We will now explain each parameter of this method in some more detail. The first parameter is the SELECT statement and is quite self-explanatory. Apart from a specific syntactical difference, Drupal’s method is identical to a normal prepared statement. Drupal’s method uses curly braces around the *node* table name in order to provide table prefixing, for when multiple installations of Drupal are run from the same database [8].

Instead of inserting the *uid* variable directly into the query, a named placeholder (`:uid`) is used, which will be filled in later. The second and third arguments are in place to provide an abstraction to the LIMIT syntax, as this may vary between database servers [10]. The last parameter is an array of all the named placeholders and the values that should be inserted into these placeholders. The database driver handles inserting the values into the query in a secure fashion, which means a value should never be quoted or escaped (Section 2.1.3) before being inserted into the query [10].

INSERT, UPDATE, and DELETE queries need special care in order to behave consistently across all different databases. Therefore, they use a special object-oriented API<sup>22</sup> for defining a query structurally [10]. Example 4.13 shows the usage of the `db_insert` method. UPDATE or DELETE statements use a similar pattern, but the `db_update()` and `db_delete()` methods are used instead respectively.

```

1 $basic_sql_query =
2   "INSERT INTO node (nid, title, body)
3   VALUES (1, 'my title', 'my body');"
4
5 $fields = array(
6   'nid' => 1,
7   'title' => 'my title',
8   'body' => 'my body');
9 db_insert('node')->fields($fields)->execute();

```

Example 4.13: Usage of Drupal's `db_insert()` method

Unfortunately, there is nothing stopping a developer from using the `db_query()` and `db_query_range()` methods in an insecure way. This issue is one of the main causes for SQL injection vulnerabilities in 3rd party modules and will be explored further in Section 4.3.9. Example 4.14 shows an insecure – but valid – way of using the `db_query()` method.

```

1 $searchterm = $_GET['searchterm'];
2
3 $result = db_query(
4   "SELECT nid, title
5   FROM {node}
6   WHERE title LIKE '%$searchterm%'" )->fetchAssoc()
7   ;

```

Example 4.14: Insecure usage of Drupal's `db_query()` method

Because this is a valid SQL statement, there is no reason and no way for Drupal's DBAL to block the execution of this query, despite it being susceptible to SQL injection. Internally, the query will still get translated into a prepared statement, but as it has already been altered before it gets to this point in execution, turning

<sup>22</sup><https://drupal.org/developing/api/database>

the query into a prepared statement no longer provides any security benefits. For instance, injecting the following string will enable an attacker to access all usernames and passwords from the *users* table: `'' UNION SELECT name, pass FROM users --` (notice the trailing space).

### Least privilege

Drupal core requires only the `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE`, `DROP`, `INDEX`, `ALTER`, `LOCK TABLES` and `CREATE TEMPORARY TABLES` privileges [14].

### Suggested by the Drupal installation guide<sup>23</sup>

Drupal differentiates between certain database/user creation methods. When using phpMyAdmin they suggest granting `ALL PRIVILEGES` to the database user and even propose just using the root user's credentials for the new database. However, in the instructions for creating a new database/user from the MySQL command line, they suggest limiting the required privileges to the ones stated above.

## 4.3.2 Authentication and session management

In Subsection 3.3.1 we provided more information about Drupal's authentication and session management systems. First we took a look at Drupal's default user registration procedure, and which user credentials are obtained. Then we examined the password recovery system, to make sure that this is also implemented securely. In this subsection, we will take a look at Drupal's session management mechanism.

### Sessions

Drupal's session management system uses PHP sessions and session cookies, using the database as session handler. Session lifetime is set to 200000 seconds, or just over 55 hours. This means that if a user is inactive for over 55 hours, the session will expire. The cookie lifetime is set to 2000000 seconds, just over 23 days, which is the maximum lifetime of a session. These numbers can be changed in the `settings.php` file, but are not configurable through the administrative interface. After the maximum lifetime has elapsed, the session will expire and the user will have to re-authenticate. The `httponly` flag is set, making it harder for attackers to gain access to the session cookie value. Sessions are generated upon login and destroyed upon logout [22] or if a user gets blocked. Session IDs are destroyed and regenerated when a user changes their password or when user permissions change.

### Session ID

The session ID is generated by appending a string of 55 randomized bytes, generated by Drupal's `drupal_random_bytes` function, to the return value of PHP's `uniqid` function. The `uniqid` method is supplied with a random prefix

---

<sup>23</sup><http://drupal.org/documentation/install>

generated by PHP's `mt_rand` function, which generates a random integer value between 0 and the value returned by `mt_getrandmax`<sup>24</sup>. The `more_entropy` boolean is set to `TRUE`, causing the `uniqid` function to generate a 23 character long unique identifier, rather than the 13 character long identifier that's generated by default. This identifier is appended to the supplied prefix. The result of concatenating the return values of the `uniqid` and `drupal_random_bytes` functions is finally hashed using SHA-256 and encoded in Base64, using the `drupal_hash_base64` function. The code used to generate the session ID is shown in Example 4.15.

```
1 $session_id = drupal_hash_base64(  
2  uniqid(mt_rand(), TRUE) . drupal_random_bytes(55));
```

Example 4.15: Session ID generation in Drupal 7.x

Because the session ID is generated randomly, it should be different each time a user logs in. Drupal sets only one cookie that's used on both front facing web pages and the administrative area. This is in contrast to WordPress, for example, which sets multiple cookies, each serving a certain purpose (see Section 4.1.2). Drupal's administrative area is not kept separate from the front facing web pages, but rather integrated in a hybrid view of administrative functionality and front facing web pages. This makes it impossible for Drupal to implement a system whereby an *Admin* cookie could be set in conjunction with a *Logged in* cookie, like in WordPress.

### 4.3.3 Cross-site scripting

Drupal's approach to mitigating cross-site scripting attacks can be summarized as *store the original; filter on output* [54]. This means that it is false to assume that the data stored in the database is safe and that it should not be trusted. Data stored in the database must first be explicitly filtered before rendering it in the web page. This could prove extremely dangerous as a core or module developer could potentially forget to do proper sanitizing on the returned data. Drupal contains several API functions for filtering output to prevent XSS attacks, each applying to a different use case (HTML context), and each using white-list input validation or replacement patterns. These filters can be found in Table 4.2. Reflected XSS attacks can be mitigated using the same filters, but could be easier to forget as properly handling `GET` and `POST` variables is not mentioned in the API.

---

<sup>24</sup>The return value of `mt_getrandmax` on both 32- and 64-bit versions of OSX, Linux, and Windows 7 is equal to 2147483647 ( $\approx 2^{31}$ ). Source: <http://be2.php.net/manual/en/function.mt-getrandmax.php>

Source Format	Target Format	Function	What It Does
Plain text	HTML	<code>check_plain</code>	Encodes special characters into HTML entities and validates strings at UTF-8
HTML	HTML	<code>filter_xss</code>	Removes characters and constructs that can trick browsers. Makes sure that all HTML entities are well formed. Makes sure that all HTML tags and attributes are well formed, and makes sure that no HTML tags contain URLs with a disallowed protocol (e.g. JavaScript)
Rich text	HTML	<code>check_markup</code>	Runs text through all enabled filters
Plain text	URL	<code>drupal_encode_path</code>	Encodes a Drupal path for use in a URL
URL	HTML	<code>check_url</code>	Strips out harmful protocols, such as <code>javascript:</code>
Plain text	MIME	<code>mime_header_encode</code>	Encodes non-ASCII, UTF-8 encoded characters

Table 4.2: Secure Conversions from One Text Type to Another<sup>25</sup>

These filters align with the rules set forth in the OWASP Cross-site Scripting Prevention Cheat Sheet, meaning they should be used based on the HTML context the user supplied data will appear in. We will take a closer look at the `filter_xss` function, as this filter was built specifically to prevent XSS attacks coming from input that's expected to contain HTML content. The function signature of `filter_xss` is as follows:

```
1 filter_xss($string, $allowed_tags = array('a', 'em', 'strong', 'cite', 'blockquote', 'code', 'ul', 'ol', 'li', 'dl', 'dt', 'dd'))
```

Example 4.16: Drupal's `filter_xss` function signature<sup>26</sup>

<sup>25</sup>Source: Pro Drupal 7 Development, third Edition

<sup>26</sup>More information about the `filter_xss` function can be found at: [https://api.drupal.org/api/drupal/includes%21common.inc/function/filter\\_xss/7](https://api.drupal.org/api/drupal/includes%21common.inc/function/filter_xss/7)

The `allowed_tags` variable is an array of allowed tags that the value of the `string` variable will be checked against. The `filter_xss` function performs a number of operations to ensure that XSS is not possible. The most important operation is removing all tags from the input `string` that are not in the whitelist `allowed_tags` array and removing any tag attribute starting with `on`<sup>27</sup> as these are most likely JavaScript event-handler definitions [54]. It further makes sure the text being filtered is valid UTF-8<sup>28</sup> (to avoid a bug in Internet Explorer 6), removes odd characters (such as NULL) and makes sure that HTML entities (such as `&amp;`;) are well-formed [54].

Another thing worth mentioning is that, when identified, common developer errors that lead to XSS vulnerabilities are mitigated by building safer defaults. For example, a page title function in Drupal 6 is the source of many XSS holes due to a lack of proper escaping. In Drupal 7 this function escapes output by default [22].

#### 4.3.4 Direct object references

Drupal often provides direct object references in the URL, such as unique identifiers of user accounts or content. For instance, the original administrator account always has ID #1. This means anyone could try to access the profile page of the administrator account by going to `http://www.example.com/user/1`, but anyone other than the administrator will get an *Access denied - You are not authorized to access this page* error. This is due to Drupal's access control mechanism preventing unauthorized requests. This mechanism is based on the system of *Roles* and *Permissions*, which was mentioned in Subsection 3.3.1.

Unauthorized functionality is hidden from unauthorized users through presentation level access control. Drupal doesn't only provide this level of access control, however. In Subsection 4.3.7, we will take a closer look at Drupal's function level access control mechanism and how it's used to stop unauthorized requests.

#### 4.3.5 Security misconfiguration

##### Default settings

Drupal doesn't use any default accounts or passwords. An administrator account is created when Drupal is installed. Account credentials are supplied by the person installing the system. Drupal comes installed with a variety of modules. Most of these modules are enabled by default, and are required to use and customize the CMS. Pre-installed modules include a *Search module*, for providing search functionality, an *Update manager module*, which can be used to securely install or update modules and themes via a web interface, and a *User module*, which can be used to manage the user registration and login system. By default, only two of the four pre-installed themes are enabled. The first is

---

<sup>27</sup>e.g. `onclick` or `onblur`

<sup>28</sup>UTF-8 is a widely used variable-width encoding that can represent every character in the Unicode character set

the site's default theme, used for front facing web pages. The second theme is used to show administrative functionality to more privileged users.

By default, Drupal is set so that all error messages are displayed, but recommend that sites running on a production environment do not display any errors. Users with the appropriate permissions can change which error messages are displayed, or can disable displaying error messages completely. This can be done in the *Configuration* administration settings.

As mentioned in Section 4.3.4, Drupal provides a *Roles and Permissions* system that enables an administrator to define which users have which privileges. These settings are set securely by default, but the *Roles and Permissions* system is highly customisable, enabling an administrator to change any permission for any type of user. For instance, settings could be changed so that unauthenticated users have access to administrative functions or to full HTML capabilities when entering comments.

### Updating core, plug-ins, and themes

During installation, an option can be checked whether or not Drupal should check for updates automatically, and if e-mail notifications should be sent if an update is available to either Drupal core or modules.

Drupal update instructions state that core updates must occur manually by deleting the old files and replacing them with the new. The *Drupal Upgrading Handbook* states that installing minor updates is not required, but if a security update is released, it should be installed as soon as possible. Updates that fix security issues and bugs are considered minor updates. Minor updates do not require that each individual update in between versions should be installed. For instance, one could update directly from Drupal version 7.1 to 7.16 without having to install intermediate updates [6], making the update procedure somewhat less cumbersome. More information about how Drupal handles security issues will be addressed in Section 4.3.9.

Drupal contains a Warning System for displaying warnings when important updates to core and modules are detected, or if there are problems with the current configuration. When Administrators visit the *Configuration* part of the administrative area and there are important issues that require the Administrator's attention, the Warning System will display a message comparable to the one shown in Figure 4.4.



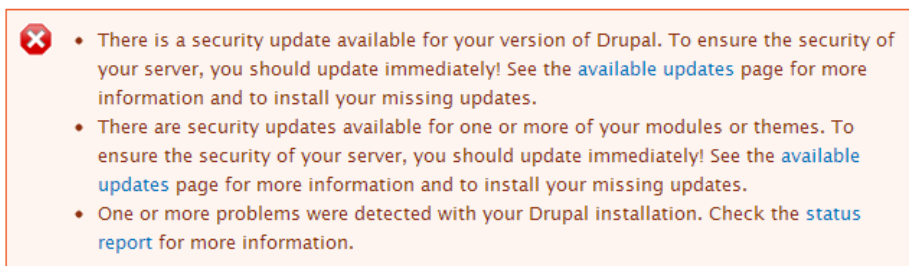


Figure 4.4: Drupal Warning System for important issues

### Drupal release cycle

New stable releases of Drupal core are scheduled to occur within particular release windows [11]:

- A bug fix release window is on the first Wednesday of each month.
- A security release window is on the third Wednesday of each month.

A release window doesn't necessarily mean that a release will be made on that date, but it gives Drupal users a general idea of when to look out for new releases. Exceptions to the schedule may be made in case of severely critical security vulnerabilities being actively exploited. In reality, there is an average of about 9 core releases per year, and just under 40 days between releases.

### 4.3.6 Sensitive data exposure

#### User credentials

Drupal versions up to 6 used unsalted MD5 to store users' passwords. From Drupal version 7 onwards, a more secure password encryption method is used. Drupal's current password encryption method is based on the Portable PHP Password Hashing Framework, or *phpass* (see Appendix A), but uses SHA-512 instead of MD5 as the underlying cryptographic function. The resulting hash is encoded using Base64 and appended to the initial salt value, which starts with a `$$` identifier to indicate the use of SHA-512. The creator of *phpass* argues that Drupal's decision to use SHA-512 was unnecessary and doesn't provide any additional technical or security benefits, but rather was changed for political reasons due to MD5's bad reputation [49]. Indeed it is true that MD5 is not collision resistant and is therefore considered unsafe, but the main problem comes from the fact that MD5 is a very fast algorithm, which makes brute force attacks easier to perform. By using the stretching technique in conjunction with a random salt value, a brute force attack is made harder because it would take a much longer time to perform. A snippet of the code needed to generate the *salted* and *stretched* password hash is shown in Example 4.17.

```

1 ...
2 $hash = hash($algo , $salt . $password , TRUE);
3 do {
4   $hash = hash($algo , $hash . $password , TRUE);
5 } while (--$count);
6 $len = strlen($hash);
7 $output = $setting . _password_base64_encode($hash , $len)
8   ;
9 ...

```

Example 4.17: Password hash generation in Drupal 7.x<sup>29</sup>

### 4.3.7 Function level access control

Drupal hides functionality from unauthorized users through presentation level access control. For instance, administrative functionality is not shown to users without the proper permissions. Presentation level access control alone is not enough. Malicious users might try to access certain resources directly, or by tampering with user data supplied to the web application. Fortunately, Drupal also contains function level access control, made possible by the *roles* and *permissions* system discussed in Subsection 4.3.4.

When accessing a page or resource, user permissions are checked and a decision is made whether or not to allow access to the requested page or resource. This is done by calling the `user_access` callback function, whose function signature can be seen in Example 4.18.

```

1 user_access($string , $account = NULL)

```

Example 4.18: Drupal's `user_access` function signature<sup>30</sup>

This function will return a boolean value based on whether the requested permission in the `string` variable is granted to the user requesting the resource. The second argument (`account`) can be supplied to check if a certain user (other than the current user) has permission to access the resource. Because the permission to be checked is provided in a case sensitive `string` variable, care must be taken that its name is spelled correctly, with correct capitalisation of characters. Fortunately the `user_access` function will always return `false` when dealing with unrecognised user permissions. However, if the user issuing the request is the original administrator (with user ID #1), the function will always return `true`, regardless of the permission's existence.

The OWASP Application Security Verification Standard warns to check for unauthorized directory listings. By default, Drupal core doesn't allow direc-

<sup>29</sup>This code snippet was taken from Drupal's `_password_crypt` function. More information can be found at [https://api.drupal.org/api/drupal/includes%21password.inc/function/\\_password\\_crypt/7](https://api.drupal.org/api/drupal/includes%21password.inc/function/_password_crypt/7)

<sup>30</sup>More information about the `user_access` function can be found at: [https://api.drupal.org/api/drupal/modules%21user%21user.module/function/user\\_access/7](https://api.drupal.org/api/drupal/modules%21user%21user.module/function/user_access/7)

tory browsing. A directive is put in the `.htaccess` file which will give a *403 Forbidden* error message when someone tries to access a folder [9].

### 4.3.8 Cross-site request forgery

Drupal uses several techniques in order to prevent Cross-site request forgery exploits. Drupal contains an API that must be used to generate forms. Forms generated by this forms API work with POST submissions by default [12], making CSRF attacks harder – but certainly not impossible – to perform. Aside from using POST submission by default, Drupal implements CSRF challenge tokens as described in the OWASP *Cross-Site Request Forgery Prevention Cheat Sheet* [42]. An anti-CSRF token is generated by the `drupal_get_token` function, which is shown in Example 4.19.

```
1 function drupal_get_token($value = '') {  
2   return drupal_hmac_base64($value, session_id() .  
   drupal_get_private_key() . drupal_get_hash_salt());  
3 }
```

Example 4.19: Drupal’s `drupal_get_token` function

The `value` parameter is the form’s ID, which is used to create a unique token for that form ID. The form ID is hashed with SHA-256, using the HMAC method. The resulting hash is then encoded in Base64. The key needed by the hash function is generated by concatenating the user’s session ID, a private key, based on 55 random bytes created by the `drupal_random_bytes` function, and a random hash salt that’s generated when Drupal is installed. Form token values are dependent on the form’s ID. Tokens are added to the forms automatically by the Drupal Forms API [12] and are valid for as long as the user’s session is alive. When the user’s session expires and the user logs back in, a new session ID will be generated (as explained in Section 4.3.2), which causes the token value to be different as well.

Drupal’s form API validates the form token during submission of the form by comparing it to a hash for the same form ID. If the token doesn’t validate, Drupal will stop form submission and respond with the following error message: *The form has become outdated. Copy any unsaved work in the form below and then reload this page* [7].

### 4.3.9 Components with known vulnerabilities

Drupal can be extended by themes and modules. Themes can be installed to give the website a different look, while modules extend the application’s functionality to fit the user’s needs. Modules can range from image uploaders to upload multiple images at the same time, to CAPTCHA’s<sup>31</sup> for form submissions [15]. We will now answer the criteria set forth in the conclusion of Chapter 2:

<sup>31</sup>CAPTCHA’s are a type of challenge-response test to block form submissions by spam bots

- **Is there a versioning system in place where the current version of an added component can easily be identified?**

Yes. Drupal provides an overview of all installed modules, where they can easily be activated and managed. This overview clearly shows the version number of the installed modules. It doesn't show the version number of the latest version of the module, however.

- **Are security issues disclosed publicly?**

Security issues are kept private until there is a fix, or until it is apparent that the maintainer of the module is not addressing the issue in a timely manner. Public announcements are made when a secure version is available. If the maintainer does not provide a fix, a security advisory is issued, recommending disabling the module. The project will then be marked as *unsupported* on the Drupal website, announcing to everyone that wants to install the module that it is possibly unsafe to use.

- **Is there a project or security mailing list in place to warn users about new vulnerabilities?**

Yes, Drupal provides a security mailing list where all security issues will be disclosed once certain criteria have been met. Drupal also provides a list of security advisories, in response to vulnerability reports.

- **Are there any manual or automated checks in place in order to detect vulnerabilities before the components are released publicly?**

Before a module is allowed to be released, it will be reviewed manually by reviewers, who will examine the code and provide feedback to the developer. Any requested changes must be implemented before the module can be accepted. Once the module has been fully approved, it may be uploaded to the Drupal module repository and can be downloaded and installed by others.

- **Is there any indication or warning given when searching for or trying to install components with known vulnerabilities?**

Modules with known vulnerabilities will be marked as *Unsupported* if the security issues are not addressed by the module's maintainer in due time. Others will still be able to download and install the module, but a message is shown that the module is currently unsupported.

- **How are vulnerability reports dealt with and how quickly do flaws get fixed?**

The following steps are taken directly from the Drupal *Security team* documentation page [16].

- Review the issue and evaluate the potential impact on all supported releases of Drupal. If it is indeed a valid problem, the security team mobilizes the maintainer to eliminate it (whether for core or contrib).
- New versions are created, reviewed, and tested.
- New releases are created on Drupal.org.
- When an issue has been fixed, we use all available communication channels to inform users of steps that must be taken to protect themselves.

- If the maintainer does not fix the problem within the deadline, an advisory is issued, recommending disabling the module and the project on Drupal.org is marked as unsupported.

- **Are updates easy to install?**

Updates to core and contributed modules are quite cumbersome to install. The update installation guide [6] contains the following steps to be followed when installing updated versions:

- Make a backup of your Drupal instance.
- Download the latest release of your current Drupal version.
- Extract the Drupal package.
- Set your site to maintenance mode
- Delete all the files & folders inside your original Drupal instance except for `/sites` folder and any custom files you added elsewhere.
- Copy all the folders and files except `/sites` from inside the extracted Drupal package into your original Drupal instance.
- If the update release includes changes to `settings.php` replace old `settings.php` in `.../sites/default/` with the new one, and edit site-specific entries (e.g. database name, user, and password)

#### 4.3.10 Redirects and forwards

Drupal frequently uses redirects and forwards, for instance when redirecting a user to the home page after successfully logging in. The Drupal `drupal_goto` function handles redirecting users to a different page. This function can be used to redirect users to internal or external web pages. External destinations supplied in a GET parameter are never allowed and are always sanitized using `drupal_parse_url` before being used [13]. An internal URL must be supplied as a relative path. The `drupal_goto` function will transform the relative path into an absolute path before issuing the redirect. An array of options and an HTTP response code<sup>32</sup> can be provided as well. The contents of the options array will be appended to the URL as GET parameters, while the HTTP response code<sup>33</sup> details the reason for the redirect.

- **Don't use redirects and forwards** – Drupal uses redirects and forwards extensively throughout the application's core.
- **Don't base redirects and forwards on user parameters** – The `drupal_goto` function is configured so that redirections to external URLs obtained from GET parameters are never allowed.
- **Make sure that the supplied value is valid** – Drupal checks the provided URL for validity using several functions.

<sup>32</sup>More information about Redirection response codes can be found at: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

<sup>33</sup>More information about HTTP response codes can be found at: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- **Make sure the user is authorized to be redirected there** – Access control is not checked before a user is redirected, but rather when the request to the redirect location is being processed. Drupal’s access control mechanism is discussed in Subsection 4.1.7.

## 4.4 Conclusion

In this chapter we have seen the techniques used by each of the content management systems to mitigate the security risks described in the OWASP Top 10. While all CMSs are based on PHP, there are some differences in the way certain problems are handled. For instance, WordPress and Joomla escape untrusted user input and sanitize input data before it is put into the database, while Drupal uses prepared statements and stores user data unsanitized. In the next chapter, we will compare the techniques used by the CMSs to each other to provide a better comparison of how each CMS handles a certain security issue.



## Chapter 5

# Comparing Content Management Systems

In this chapter we will compare the content management systems described in Chapters 3 and 4 to each other based on a number of criteria. In the previous chapter we provided an in depth overview of how each content management system provides ways to mitigate the security risks described in the OWASP Top 10, which we talked about in Chapter 2. To further help our analysis, we will also compare other factors of each content management system with each other. These factors include the data we scraped from vulnerability websites, and information about the update frequency of CMS core. We will use the data we scraped from vulnerability websites to try and give us an overview of how many security issues are discovered in core and added components (modules, plug-ins, themes, etc.) over a given period of time.

### 5.1 OWASP Top 10

#### 1. Injection

Joomla and Drupal contain Database Abstraction Layers in order to ensure that their frameworks work on all kinds of database systems. WordPress does not, and requires a MySQL database, stating that it is not a priority of the core development team to support additional database engines [60]. This doesn't necessarily form a problem, were it not for the fact that WordPress still uses the deprecated `mysql_real_escape_string` PHP function for escaping untrusted user data. The PHP manual clearly states that MySQLi is strongly recommended when using MySQL versions 4.1.3 and above [56], but as WordPress supports down to version 4.1.2 of MySQL [72], the MySQLi extension is not used. Because Joomla uses a Database Abstraction Layer, escaping is based on the database engine that is used. If a MySQL database is used, Joomla will also use the deprecated `mysql_real_escape_string` function to escape untrusted user data. As we have seen in Example 2.8, escaping untrusted data isn't always safe.



Drupal uses prepared statements, which are the safest option to use, if used correctly. This doesn't necessarily mean that the other frameworks are unsafe, as other techniques are used in conjunction with escaping to ensure that input data is safe, for instance casting expected numeric values to integer or float, or using regular expressions. Using Drupal's SQL injection mitigation technique in way ensures that queries cannot be altered. On the other hand raw input data is stored unsanitised in the database, which could leave the application vulnerable to other forms of attacks, like XSS attacks.

## 2. Broken Authentication and Session Management

As mentioned earlier, WordPress and Joomla are each comprised of two distinct areas: a front facing interface and a back end administrative interface. In Drupal, administrative functionality is embedded in front facing pages. Joomla's front facing interface provides some functionality like creating, editing, or publishing posts, but most administrative functionality is provided in the back end interface. WordPress provides most functionality in the back end administrative area, with the exception of writing comments on posts. In WordPress, anyone can visit the administrative interface, albeit with reduced functionality for users with limited privileges. Joomla only allows certain kinds of users to gain access to the administrative area. Another difference between the two is that a user has to explicitly authenticate in order to gain access to Joomla's administrative interface, while in WordPress a session cookie will be set for both the front facing area and the administrative area at the same time. When a user logs out of either area in WordPress, all session cookies will be destroyed. In Joomla, a user has to explicitly log out of both interfaces, which could cause a user to forget to log out of the administrative interface.

WordPress requires that both a *Logged in* and *Admin* cookie are set and are valid before access is allowed to the administrative area. Joomla requires explicit authentication before access is allowed to the administrative interface, and starts a new session. Joomla doesn't require a user to be logged in to both areas at the same time, however. Drupal starts only one session and shows and allows functionality based on the user's privileges. Only Drupal contains a mechanism to limit the number of login attempts and blocks IP addresses that try to log in more than an allowed number of times using wrong credentials.

## 3. Cross-Site Scripting

As described by the OWASP *Cross-Site Scripting Prevention Cheat Sheet* [42], all three content management systems provide different sanitisation filters based on the HTML context of the content. While WordPress and Drupal both use white-list input validation, Joomla uses black-list input validation by default in stead. However, input data is converted to UTF-8 and to lower case characters prior to being validated, making sure that different encoding or capitalisation will not affect the black-list input validation. Both WordPress and Joomla sanitize input data before placing it in the database, dependent on user privileges. Drupal on the other hand stores all raw input data unsanitized and requires that data coming from the database must be sanitized before rendering it to a web page. Figure 5.5

in the next subsection shows the ratio of the number of reported vulnerabilities in Drupal core and added components. This figure shows that most of the reported vulnerabilities in added components come from XSS vulnerabilities.

#### 4. **Insecure Direct Object References**

All three systems contain mechanisms whereby users' privileges can be set. WordPress provides a mechanism of *Roles* and *Capabilities*. Joomla provides a hierarchical mechanism of *User Groups* and *Access Levels* that define what a user can view, and a hierarchical mechanism of *User Groups*, *Actions*, and *Permissions* that define what a user can do. Lastly, Drupal provides a mechanism of *Roles* and *Permissions*. WordPress doesn't contain any functionality to add, remove, or change any Roles. Joomla's and Drupal's user privileges mechanisms are highly customisable, and allow a privileged user to tailor these mechanisms to the application's needs.

#### 5. **Security Misconfiguration**

Because these systems are highly customisable, it is important that default settings are set securely. For instance, that newly registered members get the least amount of privileges by default, and that displaying error messages is turned off automatically. While all three systems do give newly registered members the least amount of privileges, displaying error messages is not turned off by default in Drupal.

At installation, all three content management systems check if the minimum requirements are met before the system is installed. If not, the installation will not proceed. Joomla requires that the installation folder be removed before installation can be completed, as an extra security feature. This makes it impossible for an attacker to somehow overwrite the current installation.

#### 6. **Sensitive Data Exposure**

All three systems store minimal amounts of user data by default. Usernames and e-mail addresses are required, and stored as plain text in the database. Passwords are all stored encrypted, using irreversible hashing algorithms. Wordpress and Joomla both use MD5 as the underlying hashing algorithm, while Drupal uses SHA-512. Both WordPress and Drupal use the *phpass* framework, which provides salting and stretching techniques, making brute force attacks harder. Joomla uses its own password hashing algorithm, which also uses salting, but not stretching, making brute force attacks possible.

#### 7. **Missing Function Level Access Control**

All systems contain a decent system of Function Level Access Control. Access control is based on a user's privileges, which are defined by the mechanisms described in the part about Insecure Direct Object Reference. While all access control systems enforce both Function Level and Presentation Level Access Control, Joomla contains a separate system of *Access Levels* for Presentation Level Access Control, which defines what a user can see. Both Joomla's and Drupal's access control mechanisms are highly customisable, while WordPress only contains default user roles which can't be changed.

## 8. Cross-Site Request Forgery

In order to protect against CSRF attacks, both Joomla and Drupal use anti-CSRF tokens that expire together with the user's session. In both cases, this token is based on user and session data. WordPress uses *Nonces* based on a form's ID. Nonces do not expire together with the session, but rather have a predefined lifetime of 24 hours. None of these systems reset the token when a wrong value is supplied.

## 9. Using Components With Known Vulnerabilities

Joomla and WordPress contain mechanisms to update added components and extensions with the click of a button, which is a lot more user friendly than Drupal's approach. Drupal requires that the old files be replaced with the new ones manually, which is something that can scare novice users to perform updates. Unlike Joomla and Drupal, WordPress doesn't disclose security vulnerabilities publicly, but rather just releases a new update, referring to vulnerability databases such as the CVE. Joomla and Drupal both provide security mailing lists, to which people can register and be informed of important updates and new vulnerabilities. All content management systems provide manual and automated checks on added components before they are released, trying to ensure that no security issues are contained within them.

## 10. Unvalidated Redirects and Forwards

All three content management systems use redirects extensively. WordPress contains two redirect functions: `wp_redirect` and `wp_safe_redirect`. The function allows redirections to both internal and external locations. The latter function allows redirects to internal locations, but requires external locations to be contained in a white-list of accepted domains. Joomla implements redirections in the `redirect` function and allows redirections to external domains obtained from GET parameters. The only requirement is that the URL is encoded in Base64, which makes it harder for users to detect unwanted redirects. Drupal redirections are done via the `drupal_goto` function, which can be used to redirect to both internal and external locations, but only if the location of the redirection is not based on a GET parameter. If redirection is based on a GET parameter, external locations will not be allowed. All systems check the URL for validity, but none of them check if the current user is allowed to be redirected to the requested location before the redirection is actually carried out.

Table 5.1 and subsequent tables contain a summarised overview of a number of relevant security-related concerns for each of the items on the OWASP Top 10, and if and how each CMS handles them. The answers are colour-coded so that a quick overview can be obtained for each of the concerns:

- Positive
- Neutral
- Negative

1. Injection			
Criterion	WordPress	Joomla	Drupal
Prevention technique	● Escape input	● Escape input	● Prepared statements
Implemented securely?	● Uses deprecated MySQL functionality	● Dependent on back end database escaping scheme (DBAL)	● Dependent on back end database scheme (DBAL)
Automatic sanitisation	● Yes, when not using the <code>query</code> function directly	● No	● No
Least privilege encouraged	● No	● No information provided	● Sometimes
2. Broken Authentication and Session Management			
Criterion	WordPress	Joomla	Drupal
Authentication security	● Password salting and stretching	● Password salting	● Password salting and stretching
Brute force mitigation	● Password stretching	● None	● Password stretching and maximum login attempts
Enforces strong password policy?	● No	● Somewhat	● No
Session management mechanism	● Cookies only	● PHP sessions and session cookies, using the database as session handler	● PHP sessions and session cookies, using the database as session handler
Session ID randomness	● Uses the HMAC hash method with MD5 and a unique secret key	● Default PHP session ID	● Several pieces of random data, hashed using SHA-256
Maximum session lifetime	● Same as cookie lifetime	● Up to 15 minutes of inactivity	● Up to 55 hours of inactivity
Maximum cookie lifetime	● Up to 48 hours. If <i>Remember me</i> option is on, up to 2 weeks.	● Same as session lifetime	● Up to just over 23 hours
Password recovery mechanism	● Sends verification e-mail to user	● Sends verification e-mail to user	● Sends verification e-mail to user
Plain text passwords	● Upon registration	● Optionally upon registration (default: no)	● Never

Table 5.1: Comparing content management systems based on the OWASP Top 10 (Part 1)

3. Cross-Site Scripting (XSS)			
Filtering policy	● Validate input; Sanitize output	● Validate input; Sanitize output	● Store the original; Filter on output
Filtering strategy	● White list input validation	● Black list input validation	● White list input validation
Different filters based on HTML Context	● Yes	● Yes	● Yes
User data stored sanitised?	● Yes	● Yes	● No
4. Insecure Direct Object References			
Distinguishing user privileges	● Roles and Capabilities	● User Groups, Access Levels, Actions, and Permissions	● Roles and Permissions
Customisable by default?	● No	● Yes	● Yes
5. Security Misconfiguration			
Criterion	WordPress	Joomla	Drupal
Automated updates	● Yes	● Yes	● No
Displaying error messages	● Off	● Off	● On
Automatic requirements checking	● Yes	● Yes	● Yes
Minimal privileges for default registered users?	● Yes	● Yes	● Yes

Table 5.2: Comparing content management systems based on the OWASP Top 10 (Part 2)

6. Sensitive Data Exposure			
Criterion	WordPress	Joomla	Drupal
Autocomplete disabled on forms that collect sensitive data?	● Yes	● Yes	● Yes
Minimal sensitive data stored?	● Yes	● Yes	● Yes
Strong cryptographic algorithms?	● MD5	● MD5	● SHA-512
Password stored hashed and salted?	● Yes	● Yes	● Yes
Master secret stored securely?	● Yes	● Yes	● Yes
7. Missing Function Level Access Control			
Criterion	WordPress	Joomla	Drupal
Access control	● Mechanism of User Roles and Capabilities; No mechanism in WordPress core to add more User Roles.	● Hierarchical mechanism of User Groups and Access Levels for presentation level access control; Hierarchical mechanism of User Groups, Actions, and Permissions for Function Level Access Control. More User Groups and Access Levels can be defined.	● Mechanism of Roles and Permissions. More roles can be defined.
Directory browsing disabled?	● No, but each folder contains empty <code>index.php</code>	● No, but each folder contains empty <code>index.html</code>	● Yes
Does the access control mechanism fail securely?	● Yes	● Yes	● Yes

Table 5.3: Comparing content management systems based on the OWASP Top 10 (Part 3)

8. Cross-Site Request Forgery (CSRF)			
Criterion	WordPress	Joomla	Drupal
Token generation mechanism	● Nonce	● Anti-CSRF token	● Anti-CSRF token
Tokens based on user data?	● Yes	● Yes	● No
Tokens based on session data?	● No	● Yes	● Yes
Is the token different when session information changes?	● No	● Yes	● Yes
Automatically added to forms?	● No	● No	● Yes
Automatic token verification?	● No	● No	● Yes
Tokens used in GET parameters?	● No	● Yes	● No
Invalidated and regenerated when wrong token is supplied?	● No	● No	● No
9. Using Components with Known Vulnerabilities			
Criterion	WordPress	Joomla	Drupal
Versioning system?	● Yes	● Yes	● Yes
Security issues disclosed publicly?	● No	● Yes	● Yes
Security mailing list?	● No	● Yes	● Yes
Warnings when trying to install vulnerable components?	● Yes	● Vulnerable components are removed from the extensions repository	● Yes
Automated updates?	● Yes	● Yes	● No

Table 5.4: Comparing content management systems based on the OWASP Top 10 (Part 4)

10. Unvalidated Redirects and Forwards			
Criterion	WordPress	Joomla	Drupal
Are redirects and forwards used?	● Yes	● Yes	● Yes
Are redirects and forwards sometimes based on user parameters?	● Yes	● Yes	● Yes
Are redirects and forwards to external domains obtained from GET parameters allowed?	● Only if the location belongs to a trusted domain	● Yes	● No
Are URLs checked for validity?	● Yes	● Yes	● Yes
Authorisation checked before redirection?	● No	● No	● No

Table 5.5: Comparing content management systems based on the OWASP Top 10 (Part 5)

### 5.1.1 Conclusion

Although all three content management systems are written in PHP, they all contain different techniques for mitigating certain security risks. WordPress contains a lot legacy code to ensure backwards compatibility and forces outdated mechanisms such as emulating PHP's `magic_quotes_gpc` function to ensure that older code that relies on this functionality being enabled will still work. WordPress and Drupal both uses a password hashing framework that is considered one of the best for encrypting passwords, but both frameworks choose to only use the least secure final fallback method. Joomla, on the other hand, uses their own password hashing algorithm, using MD5. Another example could be that all three content management systems contain anti-CSRF mechanisms for forms. Joomla and Drupal implement anti-CSRF tokens as described by the OWASP *Cross-Site Request Forgery Prevention Cheat Sheet* [42], but WordPress uses its own variant, called nonces, which are valid for 24 hours and cannot be reset inside that time frame.

Choosing which CMS to use depends on the needs of the user. Joomla and WordPress are generally considered more user friendly [55], but WordPress core is a lot less customisable by default. For instance, WordPress doesn't allow customisation of user privileges, while Drupal and Joomla are highly customisable. Drupal core gives their users a lot of control, but has a steeper learning curve [55]. Core functionality can be considered secure for each of these content management systems, which is not unsurprising as they are the most mature and most used of all [57]. A novice user will probably be more comfortable using



either WordPress or Joomla, while a more advanced user might be better off using Drupal [55].

## 5.2 Vulnerability Database

In this section we will analyse the prevalence of the types of security issues reported for each content management system over the years. We will also provide a distinction between core and added components, and analyse the update frequency of CMS releases. We will also provide a distinction between regular and security releases. Regular releases include extra features and non-security related bug fixes. Security releases are either fixes for specific vulnerability reports, or regular updates with security updates included in them. These vulnerability reports were scraped from the CVE database and analysed according to their description. Vulnerabilities labeled *Other* include arbitrary code execution, session fixation attacks, and other attacks that were not prevalent enough to show as a label on the chart.

### WordPress

Figure 5.1 shows an overview of the ratio between the number of vulnerability reports for WordPress core and themes/plugin. We also show the prevalence of each vulnerability type in both core and added components. This data is based on 405 records from the Common Vulnerabilities and Exposures database. Cross-Site Scripting vulnerabilities are one of the most reported vulnerabilities in added components, certainly in the last years. This might be due to the fact that developers must actively sanitize user input data using one of the input filters provided by WordPress. Not including these filtered checks could also easily be missed by reviewers, allowing these vulnerabilities to exist in released added components. There are much less vulnerabilities detected in core functionality than in added components, which is not surprising. Core development goes through multiple internal and open development cycles before a new version is released, which brings us to the chart shown in Figure 5.2. This chart shows an overview of the number of regular and security-related releases of WordPress over the past years [71].

We can deduce from these statistics that there have been more security releases on average in years that have more reported vulnerabilities in WordPress core. For instance, with a peak of 36 reported vulnerabilities in WordPress core in 2007, there have been 10 security-related core releases. There were only 4 regular releases in the same year, 6 less than the number of security-related releases. A large amount of regular updates like in 2011 is due to public betas and multiple publicly released release candidates. Of a total of 20 releases in 2011, only 7 were actual full releases. These betas and release candidates are not recommended for use on production sites, but rather for WordPress core developers to get a large security and usability assessment. Early testers can give feedback, which allows the developers to further enhance WordPress.

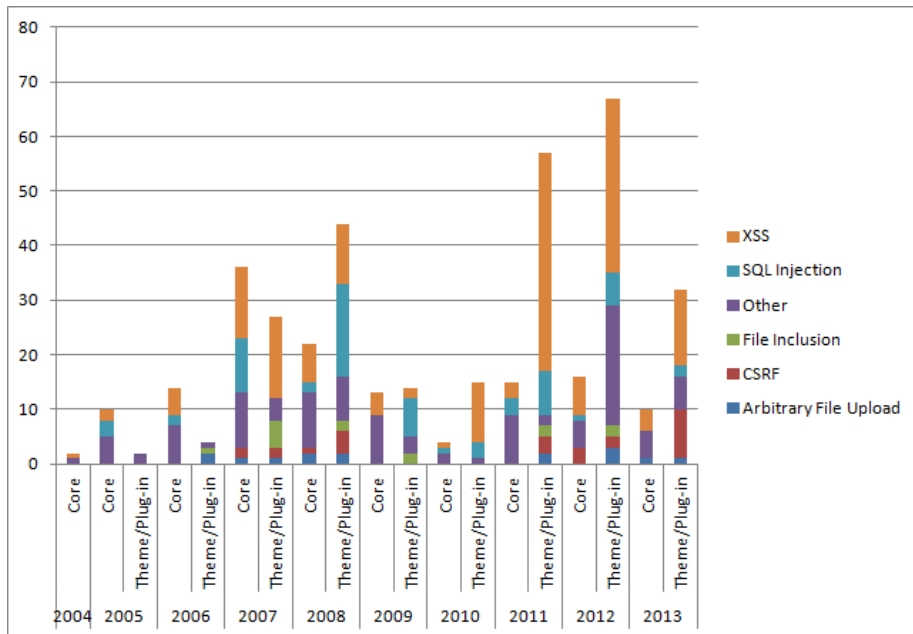


Figure 5.1: WordPress Vulnerabilities Types and Core/Extension Ratio (CVE)

Figure 5.1 shows us that most vulnerability reports concern plug-ins and themes. On average, in both core and extensions, most vulnerabilities are XSS vulnerabilities. SQL injection vulnerabilities are not as prevalent in recent years, which could be related to the fact that they are considered the most critical of all security risks according to the OWASP Top 10. This could mean that more effort is being placed in trying to prevent SQL injections, making them less prevalent in real-world applications.

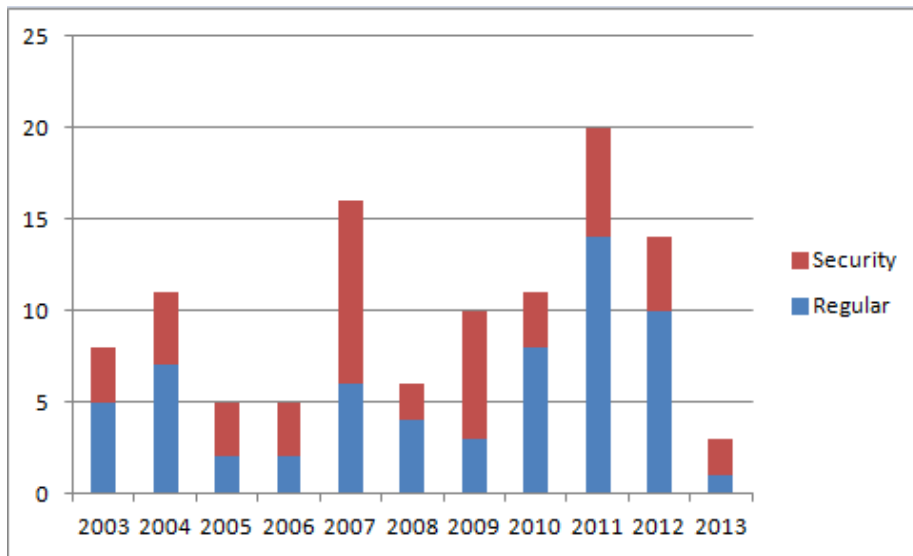


Figure 5.2: WordPress Core Releases [71]

There are many security-related releases in between regular WordPress updates. For instance, in the last year there have been more security-related releases for WordPress core than regular releases.

## Joomla

Figure 5.3 shows an overview of the ratio between the number of vulnerability reports for Joomla core and extensions. This data is based on 631 entries from the Common Vulnerabilities and Exposures database. As described in the previous subsection about WordPress, it is not surprising that the most vulnerabilities are found in extensions. Extensions are checked for security issues before they are released, but as this is driven by people, sometimes certain things are overlooked. SQL injection attacks seem to be the most prevalent attacks in Joomla extensions, with the exception of the large amount of sensitive data exposure vulnerabilities in 2011. However, these are essentially all the same vulnerability, but exploited using different extensions. While many security issues were reported in earlier years, there have been much less reports in the most recent years, which could be due to many reasons. As we can see in Figure 5.4, there haven't necessarily been less security-related updates in recent years. For instance, while the CVE vulnerability database doesn't contain a lot of entries for 2012, releases of the same year have been almost exclusively security updates. The CVE isn't necessarily a complete list, but this remains a remarkable result nonetheless.

Figure 5.4 shows the ratio of regular and security releases. A release marked as *Unknown* means that there are no release notes available for that particular release. This is mostly true for earlier versions of Joomla. Almost all other

releases fix some sort of security issue. As with WordPress, regular releases include security-unrelated bug fixes and added features. Security-related updates can be either releases that only fix certain important security issues, or regular updates that include fixes to minor or major security issues.

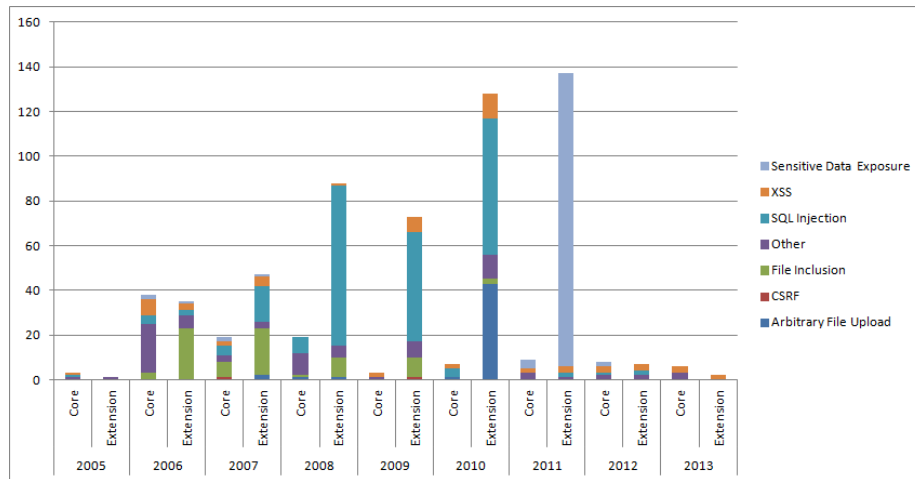


Figure 5.3: Joomla Vulnerabilities Types and Core/Extension Ratio (CVE)

One of the most prevalent types of vulnerabilities in Joomla extensions have been SQL injection vulnerabilities. However, vulnerability reports have been scarce in recent years, with XSS vulnerabilities being more prevalent in both Joomla core and extensions. The large number of *Sensitive Data Exposure* vulnerabilities are due to the same attack vector being applied to multiple extensions: directly accessing a PHP file in the extension caused an error message to disclose the installation path.

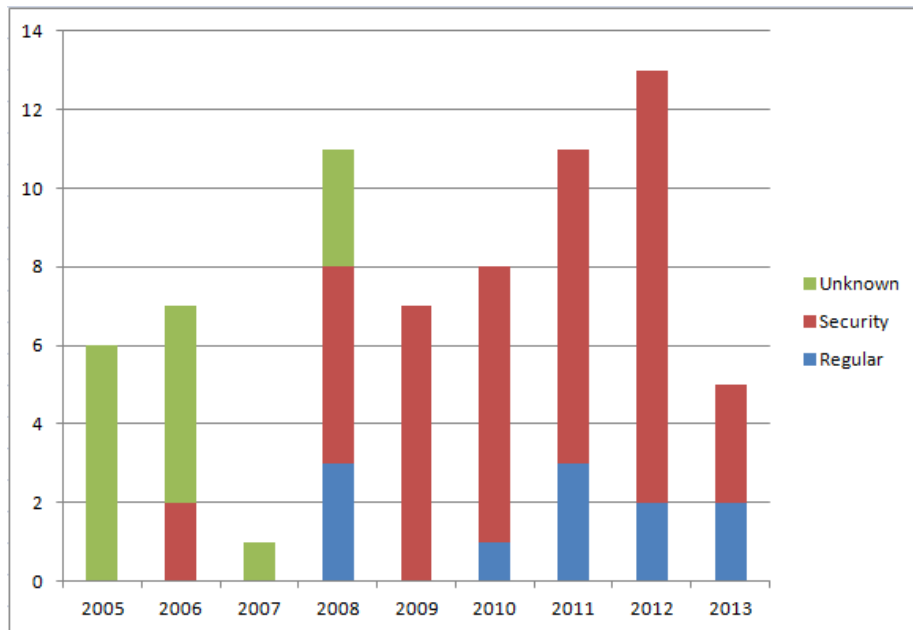


Figure 5.4: Joomla Core Releases

Figure 5.4 shows a large number of the *Unknown* release type. This is due to the fact that no release notes were available for these earlier versions. Release notes for more recent releases are available, showing the number of security-related releases often far exceeds that of regular releases. For instance, there have been exclusively security-related releases in 2009.

## Drupal

Figure 5.5 shows an overview of the ratio between the number of vulnerability reports for Drupal core and extensions. This data is based on 649 entries from the Common Vulnerabilities and Exposures database. Again, there are much more vulnerability reports for modules and themes than for Drupal Core. Because Drupal provides a Security Advisories database, where security issues are disclosed publicly once a fix has been released. We have provided an overview of the types of vulnerabilities found in these Security Advisories in Figure 5.6. What is apparent from both the CVE database results and Drupal's own security advisories is that there are barely any reported SQL Injection vulnerabilities in Drupal core, and very few have been reported for modules too. This confirms our previous statement that Drupal's SQL Injection mitigation techniques are the most secure. Many of the security issues reported and fixed in the advisories, certainly in recent years, have been *Access Bypass* vulnerabilities. The advisories also state that in some cases, these Access Bypass issues are mitigated by keeping the security settings set to their secure default values.

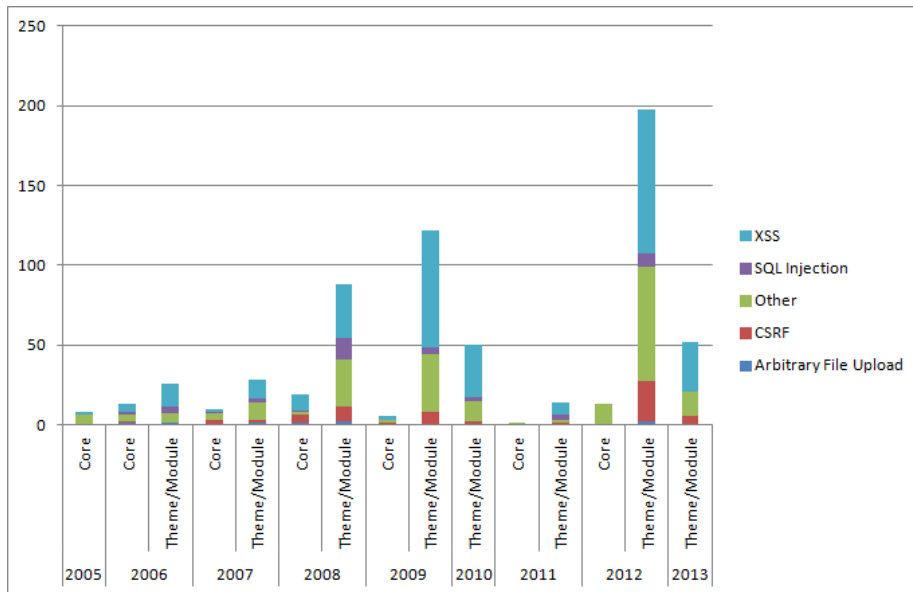


Figure 5.5: Drupal Vulnerabilities Types and Core/Extension Ratio (CVE)

The CVE contains very little data about Drupal core, which is why we have also included Figure 5.6 that shows the vulnerabilities disclosed in Drupal's Security Advisories. It is again not unsurprising that there are more reported vulnerabilities for modules and themes than for Drupal core. One of the most prevalent kinds of vulnerabilities over the years have been XSS attacks. This could be due to the fact that Drupal stores user input data unsanitised, and developers forget to sanitise the user data before rendering it to the web page.

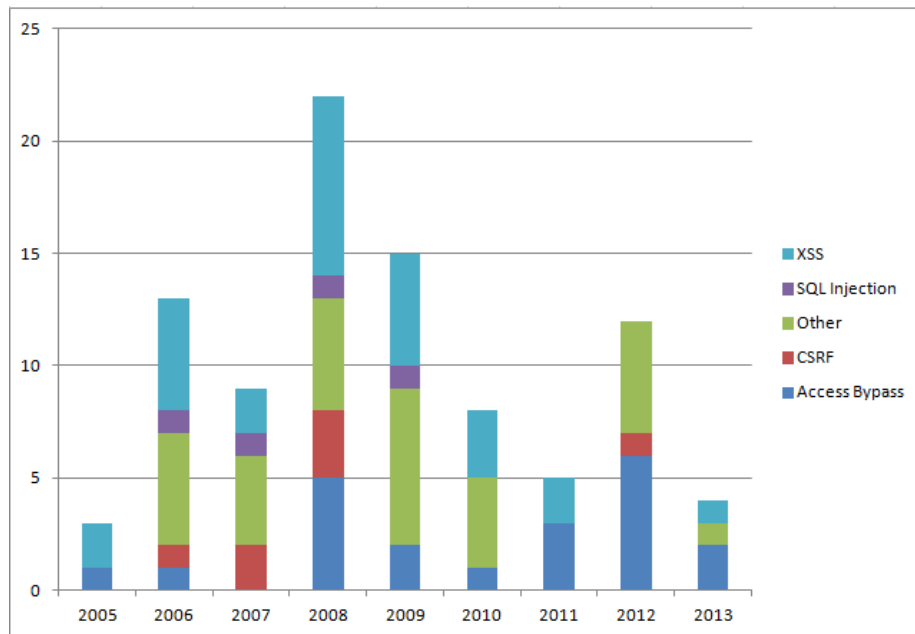


Figure 5.6: Drupal Core Security Advisories

Figure 5.6 shows an overview of the vulnerabilities disclosed by Drupal's own Security Advisories. It shows that XSS vulnerabilities have been most prevalent, even in core functionality. In more recent years, unauthorised access bypass has become a more critical issue. The *Other* vulnerability type includes session fixation and other attacks that were not prevalent enough to display on the chart. It also shows that there were a lot of reported vulnerabilities in 2008 and 2009, but that the number of reported vulnerabilities seems to be dropping in recent years.

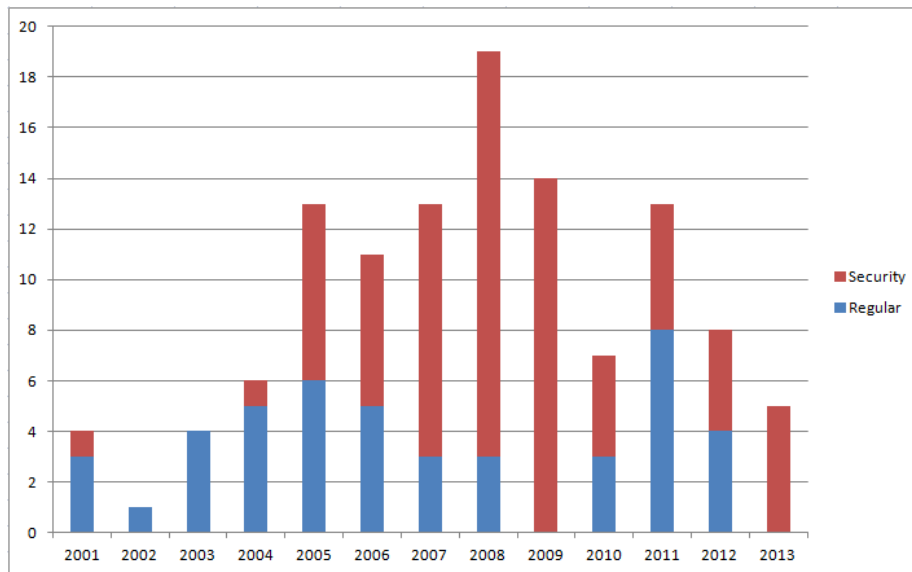


Figure 5.7: Drupal Core Releases

Figure 5.7 shows an overview of the number of regular and security-related releases of Drupal over the years. This chart shows that there have been more and more releases, and increasingly more security-related releases since the beginning. In 2009 and 2013, all new releases fixed some kind of security issue. Upgrading Drupal can be quite cumbersome because it requires manually deleting all the old files and replacing them with the new. This could potentially cause Drupal users to ignore important updates and keep using their older version, because updating would require too much work.

### 5.2.1 Conclusion

Although many efforts are made to make core functionality as secure as possible, there are still a lot of reported security issues. API documentation, manual and automated checks are in place for added components to ensure that they are implemented securely, but there are still plenty of vulnerability reports for added components. Security-related updates are released multiple times per year for each CMS in addition to regular updates and releases. In any case it is a good idea to keep a close look at new releases and updates for core and any added components that are used, and keep everything up to date as much as possible.





## Chapter 6

# Conclusion

In Subsection 1.1 of Chapter 1, we defined our problem statement as follows: "How safe are open source web content management systems". Throughout the contents of this thesis, we have tried to answer this question. In Chapter 2 we provided an overview of the OWASP Top 10, the ten most critical security risks associated with web applications, which was used as a set of objective criteria for our comparison. In Chapter 3, we provided an introduction to the three most used web content management systems at the moment: WordPress, Joomla, and Drupal. We extended our review of these systems in Chapter 4 by analysing each CMS in terms of the security risks on the OWASP Top 10 list and which mitigation techniques they implement. Lastly, in Chapter 5 we compared the results of each individual analysis with each other to provide a clearer view of the differences and similarities between each system. To further help the comparison, we used the statistics we gathered from the Common Vulnerabilities and Exposures (CVE) database, and the release statements from core releases of each CMS, to get an overview of the most reported vulnerability types and number of security-related core releases over the years.

In this chapter, we will conclude our analysis by providing a more high-level overview of our results. Each of these PHP based systems does attempt to prevent security vulnerabilities, but there are still plenty of vulnerability reports for both core and added components. These added components are not tested and retested as much as core functionality, causing the number of vulnerability reports for added components to far exceed that of core functionality. While each CMS provides a system of manual and automated reviewing of added components before they are released publicly, there still seem to be a lot of vulnerabilities that get overlooked. The fact remains that there is a great number of added components for each CMS. The number of added components for Joomla exceeds 7000, while WordPress and Drupal each host over 20000 added components. Therefore, it is not unsurprising that the number of vulnerabilities included in publicly released added components is still quite high.

By reading through the API documentation and source code of each CMS, we obtained an overview of how each CMS tackles the problems associated with web security. Designating one of these CMSs as the most secure is not

an easy task, and is not particularly relevant to the problem statement defined in the introductory section. What is clear, however, is that the open source web CMSs discussed in this thesis have a strong focus on security. Some things could be improved, for instance the fact that both WordPress and Drupal use the least secure fallback method of the phpass framework for encrypting passwords, and Joomla allowing redirects to external domains. WordPress and Joomla are considered to be more user friendly, for instance because they include automated update processes, but Drupal is perhaps more suitable for more experienced users. In the end, it depends on user preference, as all systems are secure in their own merit.

## Appendix A

# Portable PHP Password Hashing Framework

The *Portable PHP Password Hashing Framework*, or *phpass*, is a password hashing framework for use in web applications using PHP3 and above [49]. The *phpass* framework supports three different hashing methods, the last of which makes use of *salting* and *stretching* techniques to provide more security.

The preferred hashing method, and first to be tried if available, is the OpenBSD-style Blowfish-based *bcrypt*<sup>1</sup>, the second method, which is a fallback in case the first method isn't supported, is an extended DES-based hash. The third method, a final last resort fallback, uses MD5-based hashes. MD5 is included for portability, providing support for systems using PHP3 and above. This final fallback method is used in both WordPress and Drupal, and is demonstrated in Example A.1.

```
1 $hash = md5($salt . $password, TRUE);
2 do {
3     $hash = md5($hash . $password, TRUE);
4 } while (--$count);
5
6 $output = substr($setting, 0, 12);
7 $output .= $this->encode64($hash, 16);
8
9 return $output;
```

Example A.1: phpass's MD5 fallback method uses salting and stretching<sup>2</sup>

The `salt` variable contains a random salt value, which is prepended to the password and hashed using MD5. The resulting hash is then used as salt, prepended to the password, and hashed again. This process is repeated for

<sup>1</sup>For more information on *bcrypt*, visit <http://bcrypt.sourceforge.net/>

<sup>2</sup>Source: <http://joncave.co.uk/2011/01/password-storage-in-drupal-and-wordpress/>

count number of iterations [4]. This technique is called *stretching* and makes brute-force attacks on passwords harder, because calculating the hash for each attempt takes a longer amount of time.

The final hash value is encoded using the Base64 encoding scheme<sup>3</sup> and prepended by the salt value, which starts with an identifier and a single Base64 character indicating the number of iterations used. The identifier indicates which of the three encryption algorithms was used. The portable MD5 mode uses `$P$` as an identifier. Identifier `$2a$` is used to indicate that the hash was constructed using the Blowfish method.

## A.1 Validating passwords

Checking a given plain text password for validity can be achieved by using the `CheckPassword` function. Example A.2 shows the workings of this function.

```
1 function CheckPassword($password , $stored_hash)
2 {
3     $hash = $this->crypt_private($password , $stored_hash);
4     if ($hash[0] == '*')
5         $hash = crypt($password , $stored_hash);
6
7     return $hash == $stored_hash;
8 }
```

Example A.2: phpass's `CheckPassword` function

The `password` parameter contains the plain text password to be hashed and checked against the stored password, which is provided in the `stored_hash` parameter. The stored hash contains the identifier, the salt, and optionally the number of iterations if the password was hashed using the portable fallback method. The plain text password is hashed dependent on these variables and compared against the stored hash. If the two hashes match, the plain text password was correct and deemed valid.

---

<sup>3</sup>More information about the Base64 encoding scheme can be found at <https://en.wikipedia.org/wiki/Base64>

# Bibliography

- [1] *OWASP Top 10 - 2013*, author = Williams, Jeff, organization = OWASP, year = 2013, note = Available from: <http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf>,.
- [2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. <http://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf>, June 1996.
- [3] Bram Bonné. Improving session security in web applications, 2011.
- [4] Jon Cave. Drupal 7: Secure password storage by default at last. <http://joncave.co.uk/2011/01/password-storage-in-drupal-and-wordpress/>, 2011.
- [5] Daniel Chrysostomos. How to Configure Secure WordPress Database Permissions. <http://www.websitedefender.com/faq/configure-secure-wordpress-database-permissions/>, 2012.
- [6] Drupal. Drupal Upgrading Handbook. <https://drupal.org/upgrade>, 2002.
- [7] Drupal. Detailed response to publicly posted CSRF concerns in Drupal 7.12. <https://groups.drupal.org/node/216314>, 2012.
- [8] Drupal. Drupal Database API. <https://drupal.org/developing/api/database>, 2012.
- [9] Drupal. Hiding information from visitors. <https://drupal.org/node/244642>, 2012.
- [10] Drupal. Drupal API Database Abstraction Layer. <https://api.drupal.org/api/drupal/includes!database!database.inc/group/database/7>, 2013.
- [11] Drupal. Drupal Core Release Windows. <https://drupal.org/documentation/version-info>, 2013.
- [12] Drupal. Drupal Form API Reference. [https://api.drupal.org/api/drupal/developer%21topics%21forms\\_api\\_reference.html/7](https://api.drupal.org/api/drupal/developer%21topics%21forms_api_reference.html/7), 2013.
- [13] Drupal. Drupal Function Reference: Drupal goto. [https://api.drupal.org/api/drupal/includes%21common.inc/function/drupal\\_goto/7](https://api.drupal.org/api/drupal/includes%21common.inc/function/drupal_goto/7), 2013.

- [14] Drupal. Drupal Installation Guide, Step 2: Create the database. <http://drupal.org/documentation/install/create-database>, 2013.
- [15] Drupal. Drupal Modules. <https://drupal.org/project/modules>, 2013.
- [16] Drupal. Drupal Security Team. <https://drupal.org/security-team>, 2013.
- [17] Drupal. Usage statistics for Drupal core. <https://drupal.org/project/usage/drupal>, 2013.
- [18] Faronics. Blacklist-based Software versus Whitelist-based Software. Technical report, Faronics, 2011. Available from: <http://www.faronics.com/assets/blacklistvswhitelist2.pdf>.
- [19] The PHP Group. Prepared statements and stored procedures. <http://php.net/manual/en/pdo.prepared-statements.php>, 2013.
- [20] The PHP Group. Stored Procedures. <http://php.net/manual/en/mysqli.quickstart.stored-procedures.php>, 2013.
- [21] Infosec Institute. What is an SQL Injection? SQL Injection: An Introduction. <http://resources.infosecinstitute.com/sql-injections-introduction/>, 2013.
- [22] Benjamin James Jeavons and Gregory James Knaddison. Drupal Security White Paper. Technical report, Drupal, 2010. Available from: <http://drupalsecurityreport.org/sites/drupalsecurityreport.org/files/drupal-security-white-paper-1-1.pdf>.
- [23] Joomla. Delete Installation folder. [http://docs.joomla.org/Delete\\_Installation\\_folder](http://docs.joomla.org/Delete_Installation_folder), 2012.
- [24] Joomla. Leadership Highlights from March 2012. <http://magazine.joomla.org/issues/Issue-Apr-2012/item/736-Leadership-Highlights-from-March-2012>, 2012.
- [25] Joomla. Access Control List Tutorial. [http://docs.joomla.org/J2.5:Access\\_Control\\_List\\_Tutorial](http://docs.joomla.org/J2.5:Access_Control_List_Tutorial), 2013.
- [26] Joomla. Accessing the database using JFactory. [http://docs.joomla.org/J3.1:Accessing\\_the\\_database\\_using\\_JFactory](http://docs.joomla.org/J3.1:Accessing_the_database_using_JFactory), 2013.
- [27] Joomla. Extension types (general definitions). [http://docs.joomla.org/Extension\\_types\\_\(general\\_definitions\)](http://docs.joomla.org/Extension_types_(general_definitions)), 2013.
- [28] Joomla. How to add CSRF anti-spoofing to forms. [http://docs.joomla.org/How\\_to\\_add\\_CSRF\\_anti-spoofing\\_to\\_forms](http://docs.joomla.org/How_to_add_CSRF_anti-spoofing_to_forms), 2013.
- [29] Joomla. Joomla! Vulnerable Extensions List. <http://vel.joomla.org/>, 2013.
- [30] Joomla. Publishing to JED. [http://docs.joomla.org/Publishing\\_to\\_JED](http://docs.joomla.org/Publishing_to_JED), 2013.

- [31] Joomla. Retrieving request data using JInput. [http://docs.joomla.org/Retrieving\\_request\\_data\\_using\\_JInput](http://docs.joomla.org/Retrieving_request_data_using_JInput), 2013.
- [32] Joomla. Secure coding guidelines. [http://docs.joomla.org/Secure\\_coding\\_guidelines](http://docs.joomla.org/Secure_coding_guidelines), 2013.
- [33] Joomla. User Group Access Levels explained in simple terms. [http://docs.joomla.org/User\\_Group\\_Access\\_levels\\_explained\\_in\\_simple\\_terms](http://docs.joomla.org/User_Group_Access_levels_explained_in_simple_terms), 2013.
- [34] Joomla. What version of Joomla! should you use? [http://docs.joomla.org/What\\_version\\_of\\_Joomla!\\_should\\_you\\_use%3F](http://docs.joomla.org/What_version_of_Joomla!_should_you_use%3F), 2013.
- [35] Amit Klein. Cross Site Scripting Explained. <http://crypto.stanford.edu/cs155/papers/CSS.pdf>, 2002.
- [36] MITRE. Common Vulnerabilities and Exposures. <http://cve.mitre.org/>, 2013.
- [37] OWASP. About The Open Web application Security Project. [https://www.owasp.org/index.php/About\\_OWASP](https://www.owasp.org/index.php/About_OWASP), 2009.
- [38] OWASP. OWASP Application Security Verification Standard Project. [https://www.owasp.org/images/4/4e/OWASP\\_ASVS\\_2009\\_Web\\_App\\_Std\\_Release.pdf](https://www.owasp.org/images/4/4e/OWASP_ASVS_2009_Web_App_Std_Release.pdf), 2009.
- [39] OWASP. Cross-site Scripting (XSS). [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)), 2011.
- [40] OWASP. Input Validation Cheat Sheet. [https://www.owasp.org/index.php/Input\\_Validation\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet), 2012.
- [41] OWASP. Cross Site Request Forgery (CSRF). [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)), 2013.
- [42] OWASP. Cross Site Request Forgery (CSRF) Prevention Cheat Sheet. [https://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet), 2013.
- [43] OWASP. Cryptographic Storage Cheat Sheet. [https://www.owasp.org/index.php/Cryptographic\\_Storage\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Cryptographic_Storage_Cheat_Sheet), 2013.
- [44] OWASP. Injection Theory. [https://www.owasp.org/index.php/Injection\\_Theory](https://www.owasp.org/index.php/Injection_Theory), 2013.
- [45] OWASP. Password Storage Cheat Sheet. [https://www.owasp.org/index.php/Password\\_Storage\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Password_Storage_Cheat_Sheet), 2013.
- [46] OWASP. SQL Injection. [https://www.owasp.org/index.php/SQL\\_Injection](https://www.owasp.org/index.php/SQL_Injection), 2013.
- [47] OWASP. SQL Injection Prevention Cheat Sheet. [https://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet), 2013.



- [48] OWASP. XSS (Cross Site Scripting) Prevention Cheat Sheet. [https://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet), 2013.
- [49] Alexander Peslyak. Portable PHP password hashing framework. <http://www.openwall.com/phpass/>, 2010.
- [50] PhilD. MySQL user privileges. <http://forum.joomla.org/viewtopic.php?f=432&p=2692532#p2692532>, 2011.
- [51] PHP. `mysql_real_escape_string`. <http://php.net/manual/en/function.mysql-real-escape-string.php>, 2013.
- [52] PHP. PHP Runtime Configuration. <http://www.php.net/manual/en/info.configuration.php#ini.magic-quotes-gpc>, 2013.
- [53] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today. <http://www.win.tue.nl/hashclash/rogue-ca/>, 2011.
- [54] Todd Timlinson and John K. VanDyk. *Pro Drupal 7 Development*. Paul Manning, 2010.
- [55] udemy. Drupal vs Joomla vs WordPress: CMS Showdown. <https://www.udemy.com/blog/drupal-vs-joomla-vs-wordpress/>, 2013.
- [56] W3Techs. PHP Manual: Overview. <http://www.php.net/manual/en/mysqli.overview.php>, 2013.
- [57] W3Techs. Usage of content management systems for websites. [http://w3techs.com/technologies/overview/content\\_management/all](http://w3techs.com/technologies/overview/content_management/all), 2013.
- [58] WebAppSec.org. Cross site scripting. <http://projects.webappsec.org/w/page/13246920/Cross%20Site%20Scripting>, 2011.
- [59] WordPress. WordPress.
- [60] WordPress. WordPress API: Using Alternative Databases. [http://codex.wordpress.org/Using\\_Alternative\\_Databases](http://codex.wordpress.org/Using_Alternative_Databases), 2006.
- [61] WordPress. WordPress API: Data Validation. [http://codex.wordpress.org/Data\\_Validation](http://codex.wordpress.org/Data_Validation), 2013.
- [62] WordPress. WordPress API: Updating WordPress. [http://codex.wordpress.org/Updating\\_WordPress](http://codex.wordpress.org/Updating_WordPress), 2013.
- [63] WordPress. WordPress API: WordPress Cookies. [http://codex.wordpress.org/WordPress\\_Cookies](http://codex.wordpress.org/WordPress_Cookies), 2013.
- [64] WordPress. WordPress API: WP DEBUG. [http://codex.wordpress.org/WP\\_DEBUG](http://codex.wordpress.org/WP_DEBUG), 2013.
- [65] WordPress. WordPress Coding Standards Handbook. <http://make.wordpress.org/core/handbook/coding-standards/php/>, 2013.

- [66] WordPress. WordPress Function Reference: Hash Password. [http://codex.wordpress.org/Function\\_Reference/wp\\_hash\\_password](http://codex.wordpress.org/Function_Reference/wp_hash_password), 2013.
- [67] WordPress. WordPress Glossary. <http://codex.wordpress.org/Glossary>, 2013.
- [68] WordPress. WordPress Plugin Directory. <http://wordpress.org/plugins/>, 2013.
- [69] WordPress. WordPress Plugin Directory: About. <http://wordpress.org/plugins/about/>, 2013.
- [70] WordPress. WordPress Release Cycle. <http://make.wordpress.org/core/handbook/how-the-release-cycle-works/>, 2013.
- [71] WordPress. WordPress Release History. <http://wordpress.org/news/category/releases/>, 2013.
- [72] WordPress. WordPress Requirements. <http://wordpress.org/about/requirements/>, 2013.
- [73] WordPress. WordPress Security FAQ. [http://codex.wordpress.org/FAQ\\_Security](http://codex.wordpress.org/FAQ_Security), 2013.