Melany Winklaar, Jordi Samson, Jeroen Hoebert & Aaik Oosters

UCloud

Hoe veilig ben jij

Inhoudsopgave

[Inleiding 1](#_Toc532483658)

[A1:2017-Injection 2](#_Toc532483659)

[Invoervelden 2](#_Toc532483660)

[Upload afbeelding 2](#_Toc532483661)

[A2:2017-Broken Authenticatie 3](#_Toc532483662)

[A3:2017-Sensitive Data Exposure 4](#_Toc532483663)

[De hash 4](#_Toc532483664)

[A4:2017-XML External Entities (XXE) 5](#_Toc532483665)

[A5:2017-Broken Access Control [Merged] 6](#_Toc532483666)

[A6:2017-Security Misconfiguration 7](#_Toc532483667)

[Scanners 7](#_Toc532483668)

[A7:2017-Cross-Site Scripting (XSS) 8](#_Toc532483669)

[A8:2017-Insecure Deserialization 9](#_Toc532483670)

[A9:2017-Using Components with Known Vulnerabilities 10](#_Toc532483671)

[A10:2017-Insufficient Logging & Monitoring 11](#_Toc532483672)

# Inleiding

Voor het vak security hebben wij de opdracht gekregen om een OWASP top-10 webapplicatie te bouwen. Wij hebben hier ons best voor gedaan en zijn gekomen tot vele verschillende toepassingen om dit project OWASP top-10 valide te krijgen.

Het project is terug te vinden onder de volgende link:

<https://github.com/aaikoosters/blok12-security>

Wij hebben een UCloud gemaakt. Hierin is het mogelijk dat er foto’s ge-upload worden. Deze foto’s zijn voor jou en voor jou alleen. Aan ons de kunst om dit af te handelen en op te lossen.

# A1:2017-Injection

Er zijn twee verschillende punten waar een injectie kan gebeuren: invoervelden (voor het in loggen en registratie) en het uploaden van afbeelding.

## Invoervelden

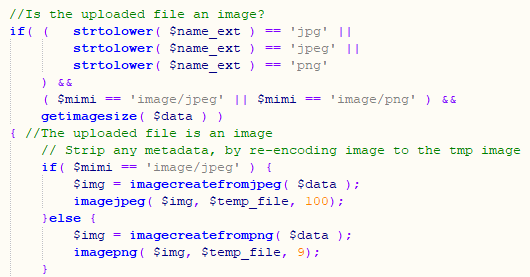
Achter alle invoervelden hebben we gebruik gemaakt van de methoden=’POST’. Daarnaast hebben we bij het uitvoeren van een query bind statements gebruikt. Hieronder is een voorbeeld:



Dat houd in dat alle invoer word gelezen als tekst. Als er een stuk script word geüpload dan word het alsnog als tekst gelezen en zou dat script niks kunnen doen.

## Upload afbeelding

Het uploaden van een file word gedaan door input=’File’. Daardoor kan de gebruiker alleen een file kiezen van zijn of haar eigen computer. De data word ook niet getoond in de inspect mode van de browser. Daar staat dan gewoon: <input type="file" name="image">.

Als een file word geüpload vanaf hier dan gaat het door een aantal validatie stappen heen.

Zoals hierboven staat worden de volgende controles word uitgevoerd:

1. De extensie van de file is JPG, JPEG en PNG.
2. Daarna word er aan de file gevraagd of het een afbeelding is ($mimi = $\_FILES['image']['type'];).
3. Daarna word er aan de file gevraagd wat de afbeelding afmetingen zijn.
4. Als het daardoor komt, geloven we dat het een image is. Alleen gaan we alsnog de file opnieuw maken. Hierdoor word ook de foute code uit de file gehaald.

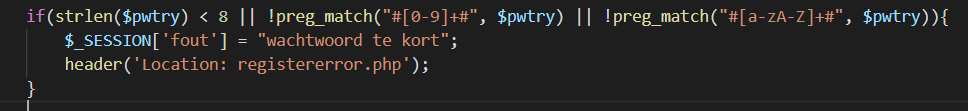
Door deze stappen kunnen we zeker zijn dat er geen verkeerde file word geüpload met malware daarin.

# A2:2017-Broken Authenticatie

Broken authenticatie houd voornamelijk in dat je inlogsysteem beschermd is tegen te zwakke wachtwoorden en brute force attack. Deze twee punten werd dan verwerkt in onze applicatie om te zorgen dat er toch een beveiligd inlogsysteem ontstaat.

## Zwakke wachtwoorden

Om te voorkomen dat ‘users’ te korte wachtwoorden of gemakkelijk wachtwoorden gebruiken werd er gebruik gemaakt van één van de mogelijkheden, een wachtwoord strenght check.

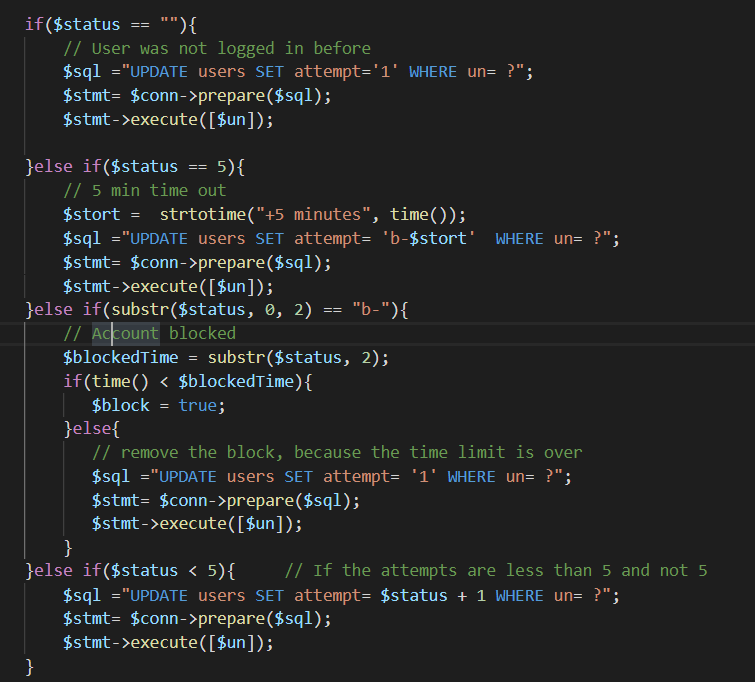


Als het wachtwoord korter is dan acht teken of geen letters bevat of nummers dat krijg de ‘user’ dan een melding dat het wachtwoord niet goed is dat die een ander moet kiezen. Op deze manier wordt er dan voorkom dat te zwakke wachtwoorden worden gebruik.

Een andere manier is ook om de meest voorkomende wachtwoorden te weigeren tijdens het registreren van een ‘user’. Je zoekt dan wat de meest voorkomende wachtwoorden zijn en deze dan ook bij de if-stament plaatsen die dan zorgt dat user een melding krijg dat die een nieuw wachtwoord moet gebruiken. Op deze manier zorg je ook dat de ‘users’ wachtwoord snel gehackt kan worden. Deze laatste manier hebben wij niet geïmplementeerd in onze applicatie maar als dit in productie gaat zullen wij dit zeker implementeren voor meer veiligheid.

## Brute force attacks

Brute force attacks houdt in dat een user verschillende combinaties probeert van gebruikersnamen en wachtwoorden keer op keer totdat die de goede combinatie bereikt. Dit is te voorkomen op twee manieren. De eerste is na een paar mogelijkheden een delay zetten na een aantal keren te hebben geprobeerd. Of de tweede is dan een bijhouden elke keer dat een gebruiksnaam probeert in te loggen en na 5 keren deze ‘user’ dan een time out te geven van ong. 10 minuten. De tweede manier hebben wij dan toegepast in onze applicatie op deze manier:



# A3:2017-Sensitive Data Exposure

In de hele applicatie word gebruikt gemaakt van de methode=’Post’. De data word nergens getoond.

Als het inloggen niet lukt, dan word de username niet getoond.

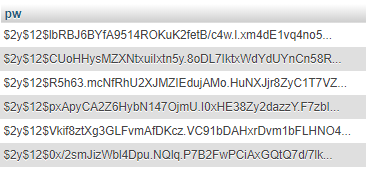
Als iemand inlogt, dan zie je nergens als wie je bent ingelogd.

De authenticatie is gedaan in een aparte file. Die file kan niet worden uitgelezen via een inspect. Die word namelijk veilig verborgen voor de gebruiker. Het programma kan erbij komen maar de gebruiker dus niet.

## De hash

De wachtwoorden worden bij de registratie gehashed. Daarvoor gebruiken we BCrypt. Hier volgt de code die we gebruiken voor het hashen van een wachtwoord: password\_hash("$pwtry", PASSWORD\_BCRYPT, ['cost' => 12]) .

Het resultaat hiervan is:



Alle wachtwoorden zijn versleuteld naar een hash. In het geval dat de data getoond word zal niemand er iets wijzer van word. Zoals alle wachtwoorden hierboven zijn door de hash anders dan elkaar, maar in praktijk zijn ze allemaal 1234.

# A4:2017-XML External Entities (XXE)

Voor dit onderdeel zijn we op het oog valide. We hebben geen XML entities draaien waardoor dit voor nu veilig is. Maar op het moment dat wij wel XML-bestanden hadden gehad dan hadden we het op de volgende manier proberen af te vangen.

Dit kan gecontroleerd worden door verschillende tools. Die kunnen controleren op de boomstructuren binnen de XML lagen. Worden tags naar verwachting ook weer gesloten.

Daarnaast wordt er vanuit PHP een standaard meegeleverd die checkt of dit valide is. Dit is de XMLReader. Deze standaard is waar bekend veilig te gebruiken. We zouden het daarom op die manier toegepast hebben.

Daarnaast wil je XML tijdig valideren omdat het problemen op je websites kan tonen die afgevangen hadden kunnen worden.

# A5:2017-Broken Access Control [Merged]

* Zorg da teen specifiek iemand gegevens kunnen aanpassen aan de database Zorg dat alle toegangscontrole gemonitord en gelogd worden
* Invalideer JWT-tokens wanneer gebruikers uitloggen (toegepast?)

# A6:2017-Security Misconfiguration

Een van de eisen is geen HTTP te gebruiken. Wij wilden dit gaan verhelpen door SSL-certificaten te installeren op onze server. Echter door het gebrek aan een ‘live’ omgeving hebben wij dit proces overgeslagen. Het voegt in deze situatie niks toe. Het voegt niks toe omdat jij als gebruiker en enige bent die toegang heeft tot jouw omgeving.

Maar wat als we wel online waren gegaan? Dan hadden we dit uiteraard geïmplementeerd. Echter is het ding van SSL-certificaten zelf signed zijn. Dit zorgt voor schijnveiligheid. Het zorgt voor vertrouwen bij de client. Echter is SSL meer het idee van een hek op het land. Waar ja langs kan rijden zonder extra moeite

## Scanners

We hebben een scanner gebruikt. Namelijk de OWASP ZAP. De uitkomst was:

|  |  |
| --- | --- |
| **Risk Level** | **Number of Alerts** |
| [High](file:///C:\Users\Jordi\Documents\dingetjedingetjeaandemuurJeMoeder.html#high) | 1 |
| [Medium](file:///C:\Users\Jordi\Documents\dingetjedingetjeaandemuurJeMoeder.html#medium) | 3 |
| [Low](file:///C:\Users\Jordi\Documents\dingetjedingetjeaandemuurJeMoeder.html#low) | 4 |
| [Informational](file:///C:\Users\Jordi\Documents\dingetjedingetjeaandemuurJeMoeder.html#info) | 0 |

Er was maar 1 high risk. En dat was dat ze dachten dat er met ‘ or ‘1’=’1 konden inloggen. Dat is niet mogelijk. Voor meer informatie hierover zie A1. Dit is namelijk een injection.

Daarnaast waren de medium risks’ dat je via de browser naar de folder structuur kan gaan. De images zijn namelijk opgeslagen in dezelfde map als de code (of het zit in een map die in dezelfde map als de code staan). Omdat het lokaal is hebben we het zo gedaan. Maar als de applicatie live ging, dan hadden we de afbeeldingen op een andere server opgeslagen.

# A7:2017-Cross-Site Scripting (XSS)

* Een framework die beschermt tegen XSS
* Input valideren door middel van antiforgerytokens
* Niet vertrouwde http requests niet afhandelen

Voor het controleren van de Cross Site Scripting

* Check extensie
* Aan bestand wat voor type
* PHP-methode om formaat photo te vragen
* Als gevalideerd dat foto gemaakt is -> foto bouwen om alle code eruit te filteren.

# A8:2017-Insecure Deserialization

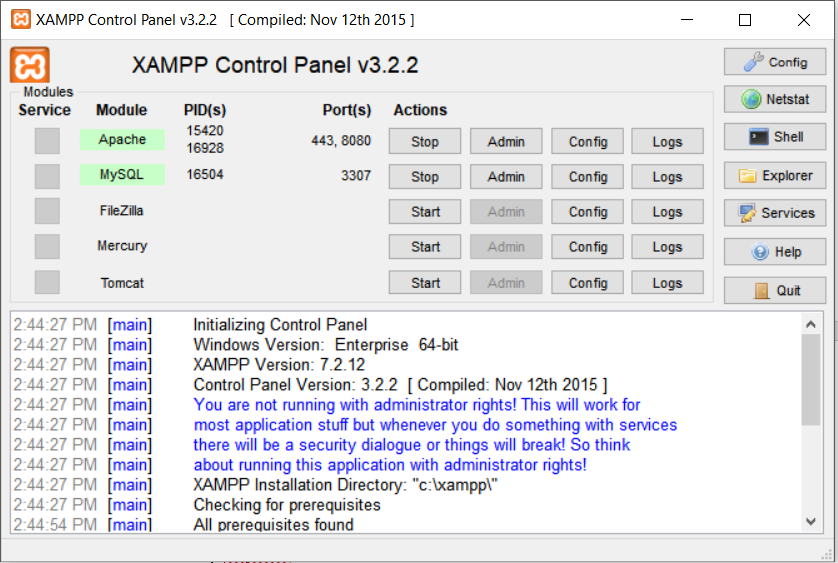
* Logging toevoegen zodat je kan traceren wat er is gebeurt
* Gaat jeroen doen en uitleggen

# A9:2017-Using Components with Known Vulnerabilities

Om dit punt toe te passen hebben wij gezorgd dat het programma die we gebruiken van de officiële site komt. We hebben ook ervoor gezorgd dat we gebruik makten van de laatste versies van dit programma omdat dit dan minder kwetsbaarheden bevatten.

We hadden gebruik gemaakt van:

* XAMPP Version: 7.2.12



En de code is geschreven in:

* PHP

# A10:2017-Insufficient Logging & Monitoring

* Zorg dat logs gegenereerd worden zodat deze logs door een gecentraliseerd logbeheersysteem gebruikt kan worden.
* Het systeem zo monitoren dat er een alarm af gaat wanneer verdacht gedrag wordt gedecreteerd
* Zorg dat er een plan van herstel beschikbaar is indien iets fout is