# Welke veiligheidsproblemen komen vaak voor en hoe zijn deze te voorkomen in mijn applicatie?

Luc van den Acker,

Semester 3 software

Een website beveiligen kan aardig uitdagend zijn. Waar moet je op letten? Wanneer is een website nou ‘veilig’? De OWASP(Open Web Application Security Project) foundation brengt jaarlijks een lijst uit van de tien meest voorkomende security issues van het jaar. Wat zijn, volgens de OWASP foundation, de meest voorkomende risico’s, en hoe kan ik deze oplossen in mijn project?

## Hoe is Broken Acces Control te voorkomen?

Met een Acces Control beleid zorg je ervoor dat gebruikers geen beschikking hebben over functionaliteiten of data waar ze geen toegang tot horen te hebben. Wanneer een gebruiker toch toegang kan krijgen tot functionaliteiten of data die niet toegankelijk horen te zijn, spreken we van Broken Acces Control. Denk bijvoorbeeld aan een gebruiker die, door bijvoorbeeld de URL te bewerken, toegang krijgt tot gegevens van andere gebruikers.

Er zijn een aantal stappen te nemen om Broken Acces Control te voorkomen. De eerste stap is deny by default, behalve voor public resources. Dit houdt in dat alle functionaliteiten en data die niet voor iedereen toegankelijk zijn standaard niet bereikbaar zijn. Wanneer een pagina überhaupt niet bereikbaar is zonder de juiste authenticatie, kunnen er ook minder snel problemen ontstaan. Voor mijn applicatie wil ik niet dat standaard gebruikers nieuwe clubs/competities aan kunnen maken. Alleen een admin mag deze aanmaken. Om die reden zijn de functionaliteiten om nieuwe entiteiten aan te maken alleen beschikbaar vanaf de admin-frontend. Een normale gebruiker heeft geen toegang tot deze applicatie, en kan dus ook de create-en update functionaliteiten niet bereiken zonder toegang te krijgen tot de admin-applicatie.

Een andere optie is om het gebruik van Cross-Origin Resource Sharing te minimaliseren. Via een CORS-policy kun je, door extra HTTP header informatie, aangeven welke resources wel of niet beschikbaar zijn vanaf andere adressen. Door het aantal adressen met toegang tot API-resources te beperken, kun je voorkomen dat data opgevraagd wordt van buiten je netwerk. Binnen mijn applicatie horen standaard gebruikers geen toegang te hebben tot create en update functionaliteiten. Om ervoor te zorgen dat deze geen API-calls kunnen sturen naar methodes die deze afhandelen, is het alleen mogelijk om deze calls te maken vanaf het adres waarop de admin-applicatie draait. Wanneer je niet op het adres van de admin zit, is het dus niet mogelijk om data te bewerken.

Ook is het aan te raden om bij te houden wat voor API-requests je krijgt, en vanaf welk adres deze requests binnenkomen. Wanneer er bijvoorbeeld veel gefaalde POST-requests binnenkomen vanaf een adres zonder toegang tot deze functionaliteiten, wil je dit natuurlijk weten. Als we weten wat we binnenkrijgen, krijgen we een beeld van welke risico’s en problemen er zijn. Zo is het duidelijk wat er moet gebeuren om je applicatie veiliger te maken.

Wanneer een gebruiker uitlogt, wil je alle stateful session identifiers ongeldig maken. Ook horen stateless JWT tokens maar kortstondig geldig te zijn. Door deze gegevens een beperkte levensduur te geven, geef je hackers minder tijd en ruimte om er misbruik van te maken. Aangezien JWTs en session identifiers vaak gevoelige of belangrijke informatie bevatten, wil je deze zodra het niet meer nodig is zo snel mogelijk verwijderen.

## Welke stappen zijn er om Injection te voorkomen?

Vrijwel alle functionaliteiten in mijn applicatie zijn afhankelijk van input, ofwel injection. Dit kan automatisch gebeuren, maar vaak komt de input van een gebruiker af. Aangezien een gebruiker zelf door de applicatie kan navigeren, kiest de gebruiker welke data weergegeven moet worden. Als een gebruiker bijvoorbeeld vraagt om de details van een club, worden de details van die specifieke club weergegeven. Helaas zijn er aardig wat risico’s verbonden aan het gebruik van injection. Wat kan er foutgaan en hoe zijn deze fouten op te lossen?

Het is natuurlijk niet de bedoeling dat een gebruiker alle, inclusief gevoelige data kan opvragen. Vandaar valideren en filteren we data die door de gebruiker ingevoerd is. Als dit niet gebeurt, en een gebruiker dus vrije toegang heeft tot de database, krijgt iedereen die wil toegang tot alle data in een database. Aangezien we dit niet willen, valideren we requests vooraf, en wordt data gefilterd voordat deze verstuurd wordt naar de frontend. Wanneer we requests valideren, is het niet zomaar mogelijk om gevoelige data op te vragen, en worden ongeldige requests dus ook afgewezen. Omdat bepaalde data in een object gevoelig kan zijn, bijvoorbeeld een user-id of een wachtwoord, wordt data gecontroleerd en gefilterd voordat deze opgestuurd wordt. Als er geen gevoelige data vanuit de API opgestuurd kan worden, kan deze ook niet in de response body af te lezen zijn. Tenzij het absoluut noodzakelijk is, hoort een gebruiker nooit toegang te hebben tot gevoelige data.

De hierboven beschreven controles en filters vangen veel ongewenste injection af. Wanneer vervalste data meegegeven wordt in een API-call, ontstaan er natuurlijk ook problemen. Binnen de toegepaste ORM search parameters willen we kunnen controleren of de data in een request wel valide is. Zo niet, hoort de request afgewezen te worden. Door speciale escape characters te specificeren, is het mogelijk om onderscheid te maken tussen juiste en onjuiste data. De interpreter van het toegepaste ORM-framework accepteert op die manier alleen data die voldoet aan de eisen, en dus het gespecificeerde escape character bevat.

## Hoe zijn Identification and Authentication Failures te voorkomen?

Aangezien verschillende gebruikers verschillende data kunnen aanroepen, gebruiken we identificatie en authenticatie. Door authenticatie toe te passen, kunnen we onderscheid maken tussen verschillende gebruikers, waardoor functionaliteiten of data wel of niet toegankelijk worden. Wat kan er fout gaan tijdens identificatie en authenticatie?

Door multi-factor authenticatie toe te passen, wordt de kans op ongewenst gedrag kleiner. Één slot openbreken is natuurlijk makkelijker dan meerdere sloten kraken. Wanneer je bijvoorbeeld, naast een user-id, ook een authenticatietoken nodig hebt om gevoelige data van een gebruiker op te vragen, is het moeilijker om met geautomatiseerde credential stuffing, of door brute force door de beveiliging heen te komen. Hoe meer authenticatiestappen er nodig zijn, hoe veiliger de applicatie.

Aangezien een admin of manager meer rechten en mogelijkheden heeft binnen een applicatie, willen we deze accounts zo veilig mogelijk houden. Zo is het bijvoorbeeld niet verstandig om default credentials voor deze accounts te te shippen of deployen. Wanneer deze op ieder moment actief zijn, vergroten we de kans op problemen. Een authenticatiecode voor een admin hoort natuurlijk niet gebruikt te kunnen worden als de admin niet ingelogd is.

Simpele wachtwoorden zijn makkelijker te achterhalen. Denk hierbij aan wachtwoorden als “Wachtwoord” of “test1234”. Om te voorkomen dat gebruikers matige wachtwoorden instellen, vragen we om een bepaalde complexiteit en variatie in wachtwoorden. Denk hierbij aan verschillende tekens, onderscheid tussen hoofdletters en kleine letters, cijfers en een minimaal aantal tekens. Wanneer het verplicht is om minstens één van alle bovenstaande tekens in een wachtwoord te verwerken, is het onmogelijk om voor een matig wachtwoord te kiezen. Daarnaast wordt van een gebruiker gevraagd om één keer per jaar een nieuw wachtwoord in te stellen. Door complexe wachtwoorden, die jaarlijks vervangen worden te verplichten, is er minder kans dat wachtwoorden geraden worden. Uiteindelijk is ieder wachtwoord natuurlijk te achterhalen, maar hoe meer variatie, hoe lastiger het is.

Het is natuurlijk niet de bedoeling dat een vijandige gebruiker oneindig veel login-pogingen krijgt. Om dit te voorkomen, zijn er een paar stappen mogelijk. Zo is het aan te raden om een limiet te stellen aan het aantal pogingen. Wanneer deze limiet bereikt wordt, is het tijdelijk onmogelijk om in te loggen op het desbetreffende account, en krijgt de eigenaar van het account een melding van de mislukte pogingen, in combinatie met het advies om een nieuw wachtwoord in te stellen. Een account helemaal blokkeren is niet aan te raden, omdat een hacker zo de mogelijkheid krijgt om het account van een onschuldige gebruiker compleet te blokkeren. Dit noemen we een DDoS-aanval, oftewel Distributed Denial of Service. Een DDoS-aanval is poging om een server gedeeltelijk of compleet te overbelasten met een gigantische stroom aan internetverkeer. Om DDoS-aanvallen tegen te gaan, wordt een toenemende wachttijd na mislukte inlogpogingen geïmplementeerd. Bijvoorbeeld wanneer gebruiker A tot vijf keer toe het verkeerde wachtwoord invoert, moet hij 30 seconde wachten tot hij weer mag proberen in te loggen. Falen er vervolgens nog vijf pogingen, moet hij een minuut wachten, daarna twee minuten, enzovoort. Op deze manier wordt het aantal requests beperkt, en dus kan de server niet overbelast raken. Hiernaast beperkt deze wachttijd het aantal pogingen om de inloggegevens van een gebruiker te raden, waardoor gevoelige data van een gebruiker iets veiliger is.

## Conclusie

Cyber security is een heel breed onderwerp, met heel veel potentiële risico’s. Het is niet mogelijk om een applicatie 100% ‘veilig’ te maken. De grootste risico’s binnen mijn applicatie zijn, naar mijn mening, Broken Acces Control, Injection en Identification and Authentication Failures. Deze problemen, zoals beschreven in de OWASP top 10, komen vaak voor en brengen aardig wat risico’s met zich mee. Het valt op dat twee van de drie punten vooral ontstaan door het gebruik van distributed sofware. Omdat de verschillende onderdelen van het project op verschillende adressen draaien, moet er gecommuniceerd worden via API-calls. Deze calls zijn redelijk eenvoudig af te vangen en na te bootsen, waardoor er natuurlijk misbruik van gemaakt kan worden. Door de stappen uit te voeren, zoals beschreven in dit onderzoek, worden een hoop risico’s verkleind, of zelfs helemaal verholpen.