Sikkerhetsfallgruver og forholdsregler i Web 2.0 med AJAX

Thomas Johan Eggum

Master i datateknikk

Oppgaven levert: Juni 2007
Hovedveileder: Trond Aalberg, IDI
Biveileder(e): Erlend Oftedal, Bekk Consulting AS

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

AJAX-drevne webapplikasjoner introduserer forskjellige sikkerhetsfallgruver. I tillegg til Cross Site Scripting og SQL-injisering, møter slike applikasjoner XML-injisering og lignende angrep. Hvilke type fallgruver og angrep møter disse applikasjonene, og hvilke forholdsregler kan benyttes mot disse? Hvordan håndterer de mest populære rammeverkene disse forholdsreglene?

Oppgaven gitt: 20. januar 2007
Hovedveileder: Trond Aalberg, IDI

SAMMENDRAG

Asynchronous JavaScript and XML (AJAX) er en samling med teknologier, som muliggjør utvikling av webapplikasjoner med interaktive brukergrensesnitt. Sikkerheten i slike webapplikasjoner kan svekkes på grunn av kompleksiteten som AJAX tilfører. Ved å fokusere på funksjonelle krav fremfor sikkerhet, kan det introduseres en rekke fallgruver og angrepstyper under utviklingen.

Det blir i oppgaven undersøkt slike fallgruver og angrepstyper, med eksempler på hvordan de kan utnyttes av ondsinnede. Ved å ta forskjellige forholdsregler, kan det utvikles sikre AJAX-baserte webapplikasjoner, og det er derfor undersøkt et aktuelt utvalg av disse.

Det finnes rammeverk for å systematisere og effektivisere arbeidet til utviklere. Oppgaven tar for seg rammeverkene Direct Web Remoting, Ruby on Rails og Microsoft ASP.NET. Disse er alle gode kandidater til verktøy for å utvikle AJAX-baserte webapplikasjoner, med hensyn til tid og sikkerhet.

Rammeverkene har innebygde mekanismer som hevder å støtte enkelte av forholdsreglene undersøkt. For å teste disse rammeverkene og deres sikkerhetsmekanismer, blir det utviklet en AJAX-basert webapplikasjon i hvert av rammeverkene.

Testingen viser at de fleste mekanismene i rammeverkene må aktiveres manuelt, før de bidrar til økt sikkerhet. Det blir også avslørt en feil i en av sikkerhetsmekanismene til rammeverket Direct Web Remoting. Feilen blir rapportert til utviklerne av rammeverket, sammen med forslag til hvordan denne bør rettes.

FORORD

Denne masteroppgaven er gjennomført av Thomas Johan Eggum vårsemesteret 2007, og markerer det siste ledd i den 5-årige utdanningen i sivilingeniørstudiet, Datateknikk og Informasjonssystemer ved Norges teknisk-naturvitenskapelige universitet (NTNU).

Oppgaven er gitt av Bekk consulting AS, og er utført i samarbeid med NTNU. Hovedveileder har vært førsteamanuensis Trond Aalberg ved NTNU, og biveileder har vært konsulent Erlend Oftedal fra Bekk consulting AS.

Jeg ønsker å takke ovennevnte for meget god veiledning av arbeidet med oppgaven. De har begge gitt konstruktiv kritikk, og har bidratt til at gjennomføringen har vært en lærerik prosess.

Trondheim, 5. Juni 2007

....
Thomas Johan Eggum

INNHOLD

Sammendrag **I**

Forord **III**

Figurer **IX**

Tabeller **XI**

Lister **XIII**

1 Innledning **1**
1.1 Bakgrunn og motivasjon . . . 1
1.2 Oppgavebeskrivelse . . . 2
1.3 Mål . . . 2
1.4 Omfang . . . 2
1.5 Metode . . . 3
1.5.1 Litteraturstudie . . . 3
1.5.2 Eksperiment . . . 3
1.6 Struktur av rapporten . . . 3
1.7 Definisjoner og forkortelser . . . 4
1.7.1 Definisjoner . . . 4
1.7.2 Forkortelser . . . 4

2 Webapplikasjoner **7**
2.1 Infrastruktur . . . 7
2.1.1 Hypertext Transfer Protocol (HTTP) . . . 7
2.1.2 Tilstander ved hjelp av cookies . . . 8
2.2 Web 2.0 med AJAX . . . 9
2.2.1 Byggeklosser i AJAX . . . 9
2.2.2 Navnet AJAX kan forvirre . . . 13
2.3 Sikkerhet i webapplikasjoner . . . 14
2.3.1 Trusselbilde . . . 14

2.3.2 Sikkerhet som prosess . . . 16
2.3.3 Sikkerhetsmål . . . 16
2.4 Oppsummering . . . 17

3 Fallgruver og angrepstyper 19
3.1 AJAX-angrepsflate . . . 20
3.2 Svakhet med HTTP-forespørsler . . . 20
3.2.1 Referer-hodet . . . 20
3.3 Kontrolltegnproblematikk . . . 21
3.3.1 Cross-site Scripting (XSS) . . . 22
3.3.2 Asynkront XSS ved hjelp av AJAX . . . 23
3.3.3 XSS Prototype Hijacking . . . 23
3.3.4 SQL-injisering . . . 24
3.3.5 XML-injisering . . . 24
3.4 Brukerinput . . . 25
3.4.1 Eksempel på manipulering av tjenersidegenerert input . . . 25
3.5 Forretningslogikk i JavaScript . . . 26
3.5.1 Klientsidevalidering med JavaScript . . . 26
3.6 Cross-Domain forespørsler . . . 26
3.6.1 Web Service proxy/bro . . . 26
3.6.2 Dynamisk script-element (Alternativ til XHR-objektet) . . . 27
3.7 Cross Site Request Forgery (XSRF) . . . 29
3.7.1 XSRF ved GET-forespørsel . . . 29
3.7.2 XSRF ved POST-forespørsel . . . 30
3.7.3 XHR-objektet kan ikke benyttes til XSRF . . . 30
3.8 Oppsummering . . . 30

4 Forholdsregler 33
4.1 Validering og filtrering av data på tjenerside . . . 33
4.1.1 Validere input . . . 34
4.1.2 Filtrere kontrolltegn (Output encoding) . . . 35
4.1.3 Hvitlister og applikasjonslogger . . . 35
4.2 Beskyttelse i dybden . . . 35
4.3 Begrense data som sendes til klientside . . . 36
4.3.1 Unngå å plassere applikasjonslogikk i JavaScript . . . 36
4.3.2 Indirekte datatilgang . . . 36
4.4 XSRF-beskyttelse . . . 37
4.5 Cross-domain ved dynamiske script-elementer må basere seg på et tillitsforhold 37
4.6 Oppsummering . . . 37

5 AJAX-rammeverk 39
5.1 Kategorier av rammeverk for AJAX . . . 39
5.1.1 Scriptbibliotek . . . 39
5.1.2 Kodegeneratorer . . . 40
5.1.3 Applikasjonsrammeverk . . . 40
5.2 Direct Web Remoting (DWR) . . . 40
5.2.1 Begrenset funksjonalitet . . . 41
5.2.2 Sikkerhet . . . 41

5.3 Ruby on Rails (RoR) 41
5.3.1 Sikkerhet i Ruby on Rails 42
5.4 Microsoft ASP.NET med AJAX 42
5.4.1 Sikkerhet i ASP.NET AJAX 43
5.5 Oppsummering 43

6 AJAX-webapplikasjon **45**
6.1 Diskusjonsforum som webapplikasjon 45
6.1.1 Use-case Bruker 45
6.1.2 Databasemodell 46
6.1.3 AJAX-basert diskusjonsforum 46
6.2 Diskusjonsforum med DWR og Java 47
6.3 Diskusjonsforum med Ruby on Rails 48
6.4 Diskusjonsforum med ASP.NET 49
6.5 Oppsummering 50

7 Testing **51**
7.1 Sikkerhetsmekanismene i DWR 51
7.1.1 Test av HTML-kontrolltegnfiltrering i DWR 51
7.1.2 Test av XSRF-beskyttelse i DWR 54
7.2 Sikkerhetsmekanismene i RoR 55
7.2.1 Test av inputvalidering i RoR 56
7.2.2 Test av HTML-kontrolltegnfiltrering i RoR 56
7.2.3 Test av SQL-kontrolltegnfiltrering i RoR 57
7.2.4 Test av XSRF-angrep i RoR 57
7.3 Sikkerhetsmekanismene i ASP.NET AJAX 58
7.3.1 Test av inputvalidering i ASP.NET 58
7.3.2 Test av HTML-kontrolltegnfiltrering i ASP.NET 59
7.3.3 Test av SQL-kontrolltegnfiltrering i ASP.NET 60
7.3.4 Test av XSRF-angrep i ASP.NET 60
7.4 Oppsummering 61

8 Vurdering **63**
8.1 Vurdering av DWR 63
8.1.1 Vurdering av HTML-kontrolltegnfiltrering i DWR 63
8.1.2 Vurdering av XSRF-beskyttelse i DWR 64
8.1.3 Begrensninger i DWR 65
8.2 Vurdering av RoR 65
8.2.1 Vurdering av inputvalidering i RoR 65
8.2.2 Vurdering av HTML-kontrolltegnfiltrering i RoR 65
8.2.3 Vurdering av SQL-kontrolltegnfiltrering i RoR 66
8.2.4 Begrensninger i RoR 66
8.3 Vurdering av ASP.NET 66
8.3.1 Vurdering av inputvalidering i ASP.NET 66
8.3.2 Vurdering av HTML-kontrolltegnfiltrering i ASP.NET 67
8.3.3 Filtrering av data access 67
8.3.4 Begrensninger i ASP.NET 67
8.4 Felles vurdering 67

8.4.1 Beskyttelse i dybden 68
8.4.2 Begrense data som sendes til klientside 68
8.4.3 Applikasjonsspesifikk sikkerhet i webapplikasjoner 68
8.4.4 Rammeverkenes støtte for sikkerhet som prosess 68
8.4.5 Sikkerhet påvirker funksjonalitet 69
8.5 Oppsummering 69

9 Konklusjon og egenrefleksjon 71
9.1 Konklusjon 71
9.2 Egenrefleksjon 72
9.2.1 Arbeidet som ble utført 72
9.2.2 Vurdering av metode som ble benyttet 72
9.3 Forslag til videre arbeide 73

Bibliografi 75

A Skjermbilde av diskusjonsforum A

FIGURER

1.1 Fremgangsmåte for å nå hovedmålet til oppgaven 2

2.1 Tradisjonell webapplikasjon i kontrast til webapplikasjon med AJAX [1] 10
2.2 DOM-inspeksjon med Firebug 11
2.3 Overlapp av sikkerhetsfaktorer 17

3.1 Angrepsflaten øker ved plassering av mer logikk på klientsiden 20
3.2 XSS-svakhet i kjent internetttilbyder 22
3.3 Prototype Hijacking [2] 24
3.4 Kontrolltegninjisering av *Web service* gjennom webapplikasjon 27
3.5 Resultat av Yahoo‘s *Web service* 28
3.6 XSS-problem på grunn av upålitelig *Web service* 29

4.1 Sirklene angir plassering for filtrering av kontrolltegn og stjernene for validering av input [3] 34
4.2 Eksempel på indirekte datatilgang ved at koblingen mellom identifikatorene og kontonumrene ligger i sesjonen på tjenersiden 36
4.3 XSRF-beskyttelse ved at en unik parameter sendes i hver forespørsel for å sikre at denne er autentisk 37

5.1 Klientside kommuniserer mot tjenerside ved hjelp av rammeverket DWR [4] . . 40

6.1 Databasemodell diskusjonsforum 46
6.2 Modell av hvordan DWR benyttes i diskusjonsforumet 48
6.3 Modell av hvordan Ruby on Rails benyttes i diskusjonsforumet 49
6.4 Modell av hvordan ASP.NET AJAX benyttes i diskusjonsforumet 50

7.1 Resultat av vellykket injisering av HTML-kontrolltegn i DWR-versjonen av diskusjonsforumet 52
7.2 Resultat ved bruk av metoden *escapeHtml* i DWR 53
7.3 Resultat ved bruk av metoden *replaceXmlCharacters* 53
7.4 Innholdet i en POST-forespørsel for å lagre et innlegg i DWR-versjonen av diskusjonsforumet 55

7.5 Forfalsket forespørsel i DWR-versjonen av diskusjonsforumet 55
7.6 Vellykket injisering av HTML-kontrolltegn i RoR-versjonen av diskusjonsforumet 56
7.7 funksjonen h(), håndterer HTML-kontrolltegn 57
7.8 Forfalsket forespørsel i RoR-versjonen av diskusjonsforumet 58
7.9 Tjenersidefiltrering av input i ASP.NET-versjonen av diskusjonsforumet 60

TABELLER

1.1 Definisjoner på uttrykk som blir benyttet i rapporten 4
1.2 Forkortelser som blir benyttet i rapporten 5

3.1 Fallgruvene påvirker sikkerhetsmål 31

4.1 Hvilke fallgruver og angrep som forholdsreglene kan forhindre 33

6.1 Use-case for bruker i diskusjonsforumet 46

8.1 Hvilke forholdsregler som støttes av mekanismer testet i rammeverkene og hvordan disse støttes 68

LISTER

2.1 HTML DOM-eksempel [5, Kapittel 2] 11
2.2 JavaScript DOM-eksempel [5, Kapittel 2] 11
2.3 XHR-eksempel [5, Kapittel 2] 12
2.4 XHR til å sende forespørsler [5, Kapittel 2] 13
2.5 XHR med tilbakekall [5, Kapittel 2] 13
3.1 Eksempel på kontrolltegn og data 21
3.2 XSS-eksempel 22
3.3 XSS prototype Hijacking 23
3.4 Eksempel på SQL-injisering [6, Kapittel 2] 24
3.5 Eksempel på manipulering av tjenersidegenerert input 25
3.6 Tilgang til Yahoo's *Web service* ved hjelp av dynamiske script-elementer 28
3.7 XSRF-eksempel med POST 30
6.1 *dwr.xml* definerer hvilke klasser som eksponeres for JavaScript i DWR-versjonen av diskusjonsforumet 47
6.2 Rjs benytter partials til å dynamisk endre innholdet i RoR-versjonen av diskusjonsforumet 49
7.1 Forsøk på bruk av metoden *containsXssRiskyCharacters* 53
7.2 Tjenersidevalidering av input i RoR 56
7.3 Tjenersidevalidering i kombinasjon med klientsidevalidering i ASP.NET 58
7.4 Nødvendig test i C# koden på tjenerside for å aktivere tjenersidevalidering i ASP.NET 59
7.5 JavaScript som står for POST-forespørselen for å lagre foruminnlegg i ASP.NET-versjonen av diskusjonsforumet 59
8.1 Feil i DWR's XSS-metode 64
8.2 Feilretting i DWR's XSS-metode 64

KAPITTEL 1
INNLEDNING

1.1 Bakgrunn og motivasjon

Tradisjonelle webapplikasjoner baserer seg på å sende brukerens nettleser en ny webside for hver forespørsel som blir foretatt. Dette medfører ventetid fra brukeren forespør data, til respons kommer fra tjenersiden. Nettleseren til brukeren, vil i en slik situasjon opptre som en "dum" terminal uten tilstandsinformasjon.

Web 2.0 er innført som et begrep på webapplikasjoner med interaktivitet og rask responstid. Slike webapplikasjoner refereres også ofte til som "rike". Disse kan baseres på eksterne komponenter som Flash, Java applets og lignende. Det har lenge vært et ønske om rike webapplikasjoner med allerede tilrettelagt teknologi i eksisterende nettlesere, uten behov for installasjon av tredjeparts programvare. For å oppnå interaktivitet og rask responstid, har det vært et behov for en asynkron kommunikasjonskanal mellom klient og tjener.

Kombinasjon av forskjellige teknologier tilrettelagt i moderne nettlesere, har ført frem til en teknikk kalt AJAX. AJAX ble introdusert av Jesse James Garret februar 2005, og senere har blant annet *Google, MySpace* og *Yahoo* laget webapplikasjoner som baserer seg på denne teknikken. På grunn av stor popularitet, har en rekke AJAX-baserte webapplikasjoner blitt kjøpt for store beløp, og flere konferanser om web 2.0 med AJAX har blitt arrangert verden rundt. Dette har vært med å bidra til at utviklere lager webapplikasjoner basert på AJAX, med bakgrunn i eksempler eller ferdiglagde komponenter. Slike utviklere er ofte ukjente med hvilke sikkerhetsfallgruver som blir introdusert på grunn av manglende fokus på sikkerhet.

Etter registrerte angrep på sider som *Myspace* og *Yahoo*, er det blitt satt spørsmål til hvilke fallgruver og angrepstyper som oppstår i kjølvannet av AJAX. For å utvikle mer sikre AJAX-baserte webapplikasjoner, eksisterer det forskjellige forholdsregler som kan tas. Disse forholdsreglene kan benyttes ved å implementere egne løsninger, eller de kan være innebygde i eksisterende rammeverk for utvikling av AJAX-baserte webapplikasjoner. Funksjonalitet i

rammeverkene som underbygger forholdsregler kan ofte være deaktivert som standard, og vil derfor ikke bidra til sikkerhet før de manuelt aktiveres.

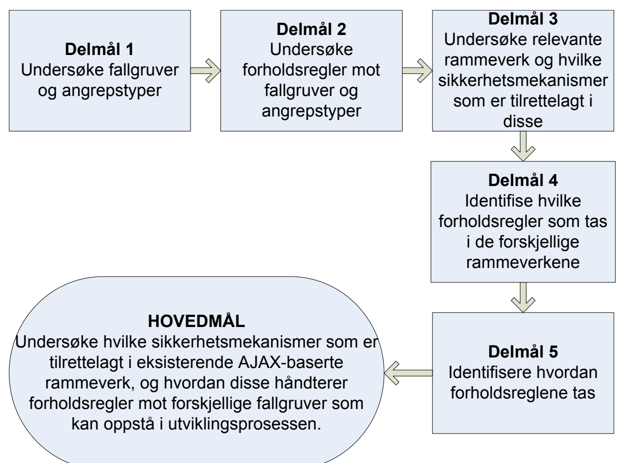
1.2 Oppgavebeskrivelse

AJAX-drevne webapplikasjoner introduserer forskjellige sikkerhetsfallgruver. I tillegg til Cross Site Scripting og SQL-injisering, møter slike applikasjoner XML-injisering og lignende angrep. Hvilke type fallgruver og angrep møter disse applikasjonene, og hvilke forholdsregler kan benyttes mot disse? Hvordan håndterer de mest populære rammeverkene disse forholdsreglene?

1.3 Mål

Hovedmålet med oppgaven er å undersøke hvilke sikkerhetsmekanismer som er tilrettelagt i eksisterende AJAX-baserte rammeverk, og hvordan disse håndterer forholdsregler mot forskjellige fallgruver som kan oppstå i utviklingsprosessen.

Hovedmålet er brutt ned i forskjellige delmål. Figur 1.1 illustrerer de forskjellige delmålene for oppgaven, og hvordan disse bygger på hverandre for å nå frem til hovedmålet.



Figur 1.1: Fremgangsmåte for å nå hovedmålet til oppgaven

1.4 Omfang

Det eksisterer en lang rekke fallgruver og angrepstyper som kan oppstå under utvikling av AJAX-baserte webapplikasjoner. Oppgaven begrenser seg til et utvalg av slike fallgruver og angrepstyper, som alle utgjør en stor trussel. Enkelte av disse vil også være aktuelle i tradisjonelle webapplikasjoner. Det blir ikke gått i detalj på kryptering av protokoller, mekanismer for å autentisere brukere, oppsett av webtjenere og lignende infrastrukturelle aspekter.

Med bakgrunn i fallgruver og angrepstyper blir det undersøkt forskjellige forholdsregler som kan tas for å utvikle mer sikre webapplikasjoner. Det blir sett på tre rammeverk som støtter enkelte av disse forholdsreglene. For å identifisere hvordan disse tas, blir det utviklet en AJAX-basert webapplikasjon. Denne danner grunnlag for testing av de forskjellige sikkerhetsmekanismene i rammeverkene.

1.5 Metode

Den første delen av oppgaven baserer seg på litteraturstudie for å kartlegge teori. Teorien danner bakgrunn for det praktiske arbeidet i forbindelse med eksperimentet.

1.5.1 Litteraturstudie

AJAX baserer seg på forskjellige teknologier. Det er nødvendig å sette seg inn i disse teknologiene for å lære om AJAX. Litteraturstudiet danner grunnlag for alt teoretisk stoff som blir beskrevet i kapittel 2, 3, 4 og 5. Først undersøkes webapplikasjoner og hvordan forskjellige teknologier danner AJAX. Videre blir det sett på hvorfor sikkerhet i webapplikasjoner er viktig. Hoveddelen i litteraturstudiet omhandler fallgruver og angrepstyper med tilhørende forholdsregler. Avslutningsvis sees det på aktuelle rammeverk som støtter AJAX, og hvilke sikkerhetsmekanismer som eksisterer i disse.

1.5.2 Eksperiment

Basert på rammeverkene undersøkt i litteraturstudiet, utvikles det en webapplikasjon. Denne webapplikasjonen er beskrevet i kapittel 6. Hensikten med denne er å teste sikkerhetsmekanismene i rammeverkene, for å identifisere hvilke forholdsregler som støttes, og hvordan de støttes. Testingen presenteres i kapittel 7. Resultatene fra testingen blir vurdert i kapittel 8.

1.6 Struktur av rapporten

Denne rapporten er organisert i ni kapitler. Kapittel 2, 3, 4 og 5 baserer seg på teori tilrettelagt av litteraturstudiet. Kapittel 6, 7 og 8 baserer seg på eksperimentet som ble utført. Kapittel 9 presenterer konklusjon med forslag til videre arbeide.

Denne rapporten er organisert i følgende kapitler:

- Kapittel 1: Innledning med motivasjon, oppgavebeskrivelse, mål, omfang, metode, definisjoner og forkortelser
- Kapittel 2: Webapplikasjoner, presentasjon av AJAX og generell sikkerhet i webapplikasjoner
- Kapittel 3: Fallgruver og angrepstyper i AJAX-baserte webapplikasjoner
- Kapittel 4: Forholdsregler mot fallgruver og angrepstyper presentert i kapittel 3

- Kapittel 5: Rammeverk for å effektivisere utvikling av AJAX-baserte webapplikasjoner med hensyn til tid og sikkerhet
- Kapittel 6: Utvikling av webapplikasjon i rammeverk fra kapittel 5
- Kapittel 7: Testing av sikkerhetsmekanismer i webapplikasjonen utviklet i kapittel 6
- Kapittel 8: Vurdering av hvilke forholdsregler sikkerhetsmekanismene i rammeverkene håndterer, og hvordan de håndteres
- Kapittel 9: Konklusjon og forslag til videre arbeide

1.7 Definisjoner og forkortelser

Engelske ord som ikke benyttes i det norske språk skrives med *kursiv*, for eksempel *Web Service*. Forkortelser skrives med vanlige store bokstaver, og defineres første gang de benyttes med *kursiv*, f.eks *World Wide Web* (WEB). Tekniske navn som JavaScript, Java og Ruby, skrives på vanlig måte uten bruk av kursiv.

1.7.1 Definisjoner

Ord og uttrykk er definert i tabell 1.1, og ved behov blir disse spesifisert i teksten.

Ord/Uttykk	**Forklaring**
Angriper	Ondsinnet person som utnytter fallgruver i en webapplikasjon
Fallgruve	Sikkerhetshull og sårbarheter som kan oppstå i webapplikasjoner
Forholdsregel	Forsiktighetsregel for å motvirke en fallgruve eller angrepstype
Injisere	Å sende inn
Input	Data som sendes til en webapplikasjon
Klientside	Avgrenset område på brukerens datamaskin
Kode	Kildekode i en webapplikasjon
Klient	Applikasjon som kommuniserer mot tjenerside
Man in the middle attack	En tredjepart har tilgang til å lese datastrøm mellom klient og tjener
Nettleser	Engelsk *web browser*, som er en klient for å benytte en webapplikasjon
Tjenerside	Avgrenset område på tjeneren hvor en webapplikasjon kjører
Webapplikasjon	Programvaresystem som benyttes ved hjelp av en nettleser
Webside	Et elektronisk dokument som kan vises i en nettleser

Tabell 1.1: Definisjoner på uttrykk som blir benyttet i rapporten

1.7.2 Forkortelser

Forkortelser som er brukt i oppgaven er listet ut i tabell 1.2.

Forkortelse	**Forklaring**
AJAX	Asynchronous JavaScript and XML
CSS	Cascading Style Sheet
DHTML	Dynamisk HTML
DOM	Document Object Model
DRY	Dont repeat yourself
DWR	Rammeverket Direct Web Remoting
JSON	JavaScript Object Notation
JSP	JavaServer Pages
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
MVC	Model-View-Controller
RJS	Ruby-generated JavaScript
RHTML	Ruby-generated HTML
RoR	Rammeverket Ruby on Rails
RPC	Remote Procedure Call
TCP	Transmission Control Protocol
URL	Uniform Resource Locator
WEB	World Wide Web
XHR	XMLHttpRequest
XML	Extensible Markup Language
XSRF	Cross Site Request Forgery
XSS	Cross Site Scripting

Tabell 1.2: Forkortelser som blir benyttet i rapporten

KAPITTEL 2
WEBAPPLIKASJONER

Webapplikasjon er en applikasjon som kommuniserer med brukeren ved hjelp av en klient, kalt nettleser. På grunn av manglende logikk på klientside i tradisjonelle webapplikasjoner, sees ofte en nettleser på som en "tynn"[1] klient. Det er i motsetning til vanlige applikasjoner og programvare, hvor det kreves installasjon av programvare, ikke nødvendig med annet enn en nettleser for å benytte en webapplikasjon. [7, Kapittel 1]

2.1 Infrastruktur

En webapplikasjon kan basere seg på plattformer som JAVA, PHP, .NET og Ruby on Rails. Felles for alle er at de benytter *HyperText Markup Language* (HTML)[2], *Cascading Style Sheet* (CSS) og eventuelt JavaScript til å formidle innhold til klientside. Dette innholdet blir levert til brukerens nettleser ved hjelp av protokollen *Hypertext Transfer Protocol* (HTTP) [8]. Det blir sett nærmere på CSS og JavaScript i delkapittel 2.2.1.

2.1.1 Hypertext Transfer Protocol (HTTP)

HTTP er en særdeles enkel forespørsel/respons-protokoll som baserer seg på følgende forløp mellom klient og tjener: [9, Kapittel 14]

- Klienten åpner en HTTP-tilkobling til tjeneren
- Klienten sender en forespørsel til tjenerside
- Tjeneren foretar en operasjon basert på forespørselen
- Tjeneren sender tilbake en respons som inneholder data

[1]En tynn klient, er klientprogramvare som baserer seg på en sentral tjener for prosessering

[2]HTML er et markeringsspråk for å lage websider

- Klienten sender en ny forespørsel, alternativt stenges HTTP-tilkoblingen

For å forsikre at data sendt mellom klient og tjener kommer i riktig rekkefølge og til riktig tid, benyttes HTTP over protokollen *Transmission Control Protocol* (TCP). Denne protokollen benytter flytkontroll, sekvensnummer, kvitteringer og tidtakere for å oppfylle dette [10, Kapittel 3]. En forespørsel fra klienten kan basere seg på åtte forskjellige HTTP-metoder. Disse er HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS og CONNECT [8]. GET og POST er de mest brukte metodene i webapplikasjoner, hvor GET benyttes for å etterspørre data og POST benyttes for å sende data. Eksempel på hoder i en GET-forespørsel:

```
GET /~thomaseg/?verdi1=enverdi&verdi2=enverditil HTTP/1.1
Host: idi.ntnu.no
User-Agent: Mozilla/5.0  Gecko/20061204 Firefox/2.0.0.1
Accept-Language: en-us,en;q=0.5
Referer: http://idi.ntnu.no/~thomaseg/
```

GET-forespørselen sender parameterne (verdi1 og verdi2) som en del av *Uniform Resource Locator* (URL)[3]. Eksempel på hoder i en POST-forespørsel:

```
POST /~thomaseg/test.html HTTP/1.1
Host: idi.ntnu.no
User-Agent: Mozilla/5.0  Gecko/20061204 Firefox/2.0.0.1
Referer: http://idi.ntnu.no/~thomaseg/
Accept-Language: en-us,en;q=0.5
verdi1=enverdi&verdi2=enverditil
```

Her vises en POST-forespørsel som forespør objektet *test.html* på adressen *idi.ntnu.no*. Det blir i motsetning til GET hvor parameterne er en del av URL, sendt parametere som egne felt i POST-forespørselen. De fleste nettlesere er implementert slik at brukere ikke får tillatelse til å klikke tilbake etter en POST-forespørsel uten å sende data på nytt. [6, Kapittel 1]

2.1.2 Tilstander ved hjelp av cookies

Da HTTP er en tilstandsløs protokoll, benyttes tilleggsfunksjonalitet for å oppnå tilstander mellom tjener og klient. Til dette formålet blir det brukt en unik identifikator som identifiserer den aktuelle sesjonen. Cookies er mest brukt, selv om noen velger å sende identifikatoren som en parameter for hver forespørsel mot tjeneren. Cookies går ut på at webapplikasjonen ber klienten huske en spesifikk verdi. Denne verdien er vanligvis en unik identifikator som identifiserer klienten. Informasjonen i cookien blir deretter sendt til tjeneren for hver forespørsel klienten foretar seg, slik at tjeneren vet hvem denne tilhører. [6, Kapittel 1] og [10, Kapittel 2]

[3] URL benyttes til å identifisere en global ressurs på internett

2.2 Web 2.0 med AJAX

En klassisk sidebasert webapplikasjon, baserer seg på at klienten opptrer som en “dum” terminal. Den kan ikke lagre tilstandsinformasjon, og baserer seg på at tjeneren hele tiden sender nye websider som skal vises avhengig av hvilke operasjoner brukeren foretar. Disse operasjonene avhenger av hvilke parametere som blir sendt ved hjelp av POST og GET forespørsler nevnt i delkapittel 2.1.1. Denne metoden medfører en effektiv ventetid for brukeren fra data blir sendt, til en ny webside blir presentert. Ventetiden avhenger i første omgang av tidsforsinkelsen mellom klient og tjener, men vil også være preget av mengde data som blir overført, og prosesseringstid på tjenersiden. [5, Kapittel 1]

En AJAX-basert webapplikasjon flytter noe av applikasjonens logikk over til brukerens nettleser. Websiden som blir levert til brukeren inneholder en stor andel JavaScript. Dette muliggjør at innholdet i websiden oppdateres asynkront, uten behov for å navigere seg frem og tilbake mellom websider. Dette vil bidra til blant annet mindre trafikk mellom tjener og klient, og mulighet til å lagre tilstand på klientsiden som igjen vil resultere i raskere responstid. Med AJAX er det også mulighet for å koble brukervalg til et langt bredere spekter med visuelle hendelser. Dette gjør det mulig å oppnå tilnærmet brukergrensesnitt som en frittstående skrivebordapplikasjon. [5, Kapittel 1]

Figur 2.1 viser en grafisk fremstilling av hvordan AJAX-baserte webapplikasjoner opptrer fremfor tradisjonelle webapplikasjoner. Figuren illustreres en tradisjonell webapplikasjon hvor brukeren må vente til en ny webside lastes, i motsetning til å laste data asynkront ved hjelp av AJAX.

2.2.1 Byggeklosser i AJAX

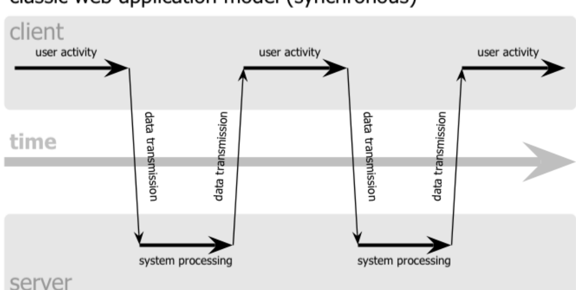
AJAX er ingen teknologi, men heller en kombinasjon av eksisterende teknologier. Disse teknologiene er byggeklossene i AJAX:

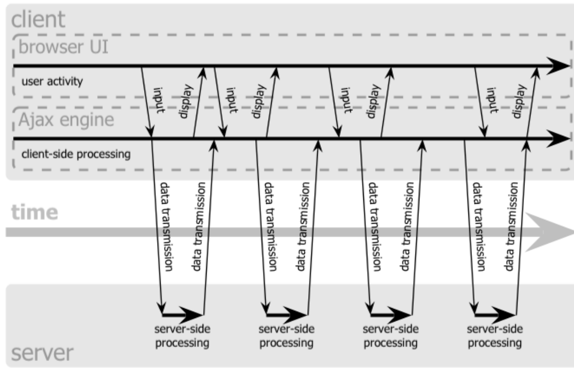
- Cascading Style Sheet (CSS)
- JavaScript
- Document Object Model (DOM)
- XMLHttpRequest-objektet (XHR)

Cascading Style Sheet (CSS)

CSS[4] benyttes til å definere utseende og brukeropplevelsen til komponenter som skal inngå i en webside. CSS er en veletablert del av webdesign, og er derfor like ofte brukt i tradisjonelle webapplikasjoner som AJAX-baserte. Med CSS kan det defineres farger, rammer, bakgrunn, bilder og posisjonering av elementer som skal inngå i en webside. [5, Kapittel 2]

[4] CSS er dokumentert på `http://www.w3.org/Style/CSS/`





Figur 2.1: Tradisjonell webapplikasjon i kontrast til webapplikasjon med AJAX [1]

JavaScript

JavaScript opptrer som limet mellom de forskjellige komponentene i AJAX. Dette gjør det mulig å manipulere innholdet i en webside dynamisk, samtidig som det kan sendes og mottas forespørsler mot tjenersiden asynkront. Nøkkelfunksjonene i JavaScript består av [5, Appendix B]:

- Variabler er løst definert, og er derfor ikke knyttet mot spesifikke datatyper som *integer* og *string*
- Koden blir dynamisk tolket og lagret som klartekst uten behov for prekompilering
- Funksjoner i JavaScript bindes opp til objektet som eies av dem, og kan bli kalt kun fra dette objektet
- I motsetning til andre programmeringsspråk som Java, benytter ikke JavaScript arv eller *interfaces*, selv om det er mulig å simulere dette

Document Object Model (DOM)

DOM[5] eksponerer en webside til JavaScript. DOM inneholder en trestruktur av alle HTML-elementer i en webside, hvor <HTML> er øverste node (foreldre node), denne inneholder ofte

[5] Document Object Model spesifikasjon tilgjengelig på `http://www.w3.org/DOM/`

en node av typen <BODY>, som igjen kan inneholde andre elementer, for eksempel <div> og <table>. Figur 2.2 viser et eksempel på hvordan et DOM-tre kan representeres ved hjelp av *Firebug* for nettleseren Mozilla Firefox[6]:

```
Clear  Inspect  Options▾        Console  Debugger  Inspector
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
  <body onload="dwr.engine.setActiveReverseAjax(true); ForumManager.getThread(1); pageLoad();">
    <div class="topmenuWindow" id="topmenuWindow">
    <div class="chatWindow" id="chatWindow">
    <div class="forumWindow" id="forumWindow">
      <div class="threadOptionsArea" id="threadOptionsArea">
      <div class="answerArea" id="answerArea">You must log in to post in the forum!</div>
      <div class="threadArea" id="threadArea">
        <br/>
        <h2>Forum for web 2.0 security issues with focus o...</h2>
      </div>
    </div>
    <div style="font-size: 80%;" class="dwr-debug" id="dwr-debug"/>
    <div style="font-size: 80%;" class="debug" id="debug"/>
  </body>
</html>
```

Figur 2.2: DOM-inspeksjon med Firebug

AJAX-baserte webapplikasjoner benytter JavaScript til å manipulere innholdet i DOM-treet. Denne manipuleringen kan være blant annet å endre farger og innhold i elementer. DOM-treet ble i tradisjonelle webapplikasjoner bygd opp ved å definere en HTML-fil. Et eksempel på en HTML-fil vises i liste 2.1:

```
<HTML>
 <BODY>
  <DIV id="test">Skal manipulere DOM-treet med JavaScript</DIV>
 </BODY>
</HTML>
```

Liste 2.1: HTML DOM-eksempel [5, Kapittel 2]

JavaScript i liste 2.2 kan benyttes til å manipulere DOM-treet til HTML-filen i liste 2.1. Her vises det hvordan JavaScript benyttes til å legge et barneelement til DIV-elementet med id="test":

```
var testDiv = document.getElementById("test");
var childEl = document.createElement("div");
childEl.id="childOfTest";
childEl.innerHTML="Dette er barneelementet";
testDiv.appendChild(childEl);
```

Liste 2.2: JavaScript DOM-eksempel [5, Kapittel 2]

[6]Firebug er tilgjengelig på `https://addons.mozilla.org/en-US/firefox/addon/1843`

XMLHttpRequest-objektet (XHR)

XHR-objektet kan sees på som hjertet i AJAX-teknikken, da denne benyttes for å kommunisere mot tjenersiden asynkront. XHR[7] er en ustandardisert utvidelse av nettlesernes DOM som støttes av de fleste nettlesere. Microsoft var den første på banen med å implementere XHR som en ActiveX[8] komponent. Senere fulgte de andre nettleserne, og implementerte sine varianter med samme API og funksjonalitet. XHR-objektet har til hensikt å tillate programmatisk generering av GET og POST forespørsler ved hjelp av JavaScript.

Eksemplet presentert i liste 2.3 benytter "Objektgjenkjenning" for å hente ut XHR-objektet. Alternativt til å gjette hvilken nettleserversjon brukeren benytter, blir det forespurt hvilken variant av objektet som er tilgjengelig. En slik metodikk er mer åpen for fremtidige nettlesere, og vil være mindre utsatt for utdatering. [5, Kapittel 2]

```
1 function getXMLHTTPRequest() {
2  var xRequest=null;
3  if (window.XMLHttpRequest) {
4   xRequest=new XMLHttpRequest();
5  }else if (typeof ActiveXObject != "undefined"){
6   xRequest=new ActiveXObject("Microsoft.XMLHTTP");
7  }
8  return xRequest;
9 }
```

Liste 2.3: XHR-eksempel [5, Kapittel 2]

Når XHR-objektet er tilgjengelig, kan det benyttes til å sende forespørsler mot tjenersiden, se liste 2.4.

XHR-objektet tillater funksjonalitet for tilbakekall fra tjenersiden til klient i tillegg til forespørsler. Dette kan sees på som inngangen fra tjenersiden og inn i klientsidelogikken. Dette realiseres ved å deklarere en tilbakekallfunksjon som blir kalt når resultatene fra tjenersiden er klare. Tilbakekallfunksjoner er tilpasset den hendelsesbaserte biten av JavaScript. Dette består blant annet av *onkeypress* og *onmouseover*. For å åpne for tilbakekall fra tjenersiden, benyttes *onload* og *onreadystatechange*. Det tas utgangspunkt i metoden *sendRequest* og legges til en ny linje i denne, og utvider med funksjonen *onReadyStateChange* slik liste 2.5 viser. Merk at den globale variabelen *req* benyttes for å forenkle eksemplet. Denne bør ikke være global hvis det skal benyttes flere samtidige forespørsler mot tjenersiden. [5, Kapittel 2]

XHR benytter HTTP

HTTP nevnt i delkapittel 2.1.1 benyttes som bæreprotokoll for XHR-objektet. Alle XHR-forespørsler sendes derfor i en HTTP-forespørsel med metode definert (GET eller POST). På grunn av dette, resulterer ikke AJAX-baserte forespørsler i problemer med eksisterende brannmurer og lignende. AJAX kan også kombineres med godt implementerte sikkerhetsmekanismer for å oppnå ende til ende kryptering mellom klientside og tjenerside med *Secure HTTP* (HTTPS)[9]. [9, Kapittel 14]

[7] API for XMLHttpRequest tilgjengelig på `http://www.w3.org/TR/XMLHttpRequest/`

[8] ActiveX gjør det mulig å utvikle tillegg for blant annet Microsoft Internet Explorer

[9] HTTPS betegner en sikker HTTP-tilkobling ved bruk av kryptering

```
1  function sendRequest(url,params,HttpMethod){
2   if (!HttpMethod){
3    HttpMethod="GET";
4   }
5   req=getXMLHTTPRequest();
6   if (req){
7    req.open(HttpMethod,url,true);
8    req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
9    req.send(params);
10  }
11 }
```

Liste 2.4: XHR til å sende forespørsler [5, Kapittel 2]

```
1  var READY_STATE_UNINITIALIZED=0;
2  var READY_STATE_LOADING=1;
3  var READY_STATE_LOADED=2;
4  var READY_STATE_INTERACTIVE=3;
5  var READY_STATE_COMPLETE=4;
6  var req;
7  function sendRequest(url,params,HttpMethod){
8   if (!HttpMethod){
9    HttpMethod="GET";
10  }
11  req=getXMLHTTPRequest();
12  if (req){
13   req.onreadystatechange=onReadyStateChange;
14   req.open(HttpMethod,url,true);
15   req.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
16   req.send(params);
17  }
18 }
19
20 function onReadyStateChange(){
21  var ready=req.readyState;
22  var data=null;
23  if (ready==READY_STATE_COMPLETE){
24   data=req.responseText;
25  }else{
26   data="loading...["+ready+"]";
27  }
28  //... Benytt dataene ...
29 }
```

Liste 2.5: XHR med tilbakekall [5, Kapittel 2]

2.2.2 Navnet AJAX kan forvirre

Bokstaven X i AJAX står for *Extensible Markup Language* (XML)[10], noe som kan forvirre enkelte til å tro at AJAX baserer seg på å sende XML-data til og fra tjenersiden. Dette er misvisende, da XHR-objektet kan bære alle former for tekst. Det finnes hovedsaklig tre forskjellige *pattern*[11] for å presentere innhold fra tjener til klient ved hjelp av AJAX. [5, Kapittel 5]

- Content-centric (Klientsiden mottar HTML-innhold)
- Script-centric (Klientsiden mates med kildekode)

[10] XML er et fleksibelt datautvekslingsformat

[11] Et *pattern* er et mønster som følges, da andre har oppnådd gode resultater ved bruk av dette tidligere.

- Data-centric (Klientsiden som applikasjon)

Content-centric

Ved bruk av *content-centric pattern*, baserer tjenersiden seg på å generere HTML-innhold som sendes til klienten. Dette innholdet kan deretter plasseres direkte inn i DOM-treet til websiden ved hjelp av JavaScript. Det er i en slik situasjon svært begrenset hva som må ligge av logikk i klientsidens JavaScript. [5, Kapittel 5]

Script-centric

Ved å benytte *Script-centric pattern*, sendes det i stedet JavaScript til klienten avhengig av hvilke brukervalg som er gjort. Dette åpner for langt mer komplekse situasjoner, men samtidig tillater det at klienten laster HTML og JavaScript uavhengig av hverandre. [5, Kapittel 5]

Data-centric

Data-centric pattern baserer seg på å sende data til klienten, og la JavaScript i nettleseren behandle dataene før de presenteres. For eksempel kan XML-data bli sendt til en JavaScript-funksjon som forvandler disse dataene til HTML-data. Disse kan deretter plasseres direkte inn i DOM-treet til websiden. [5, Kapittel 5]

2.3 Sikkerhet i webapplikasjoner

Sikkerhet i webapplikasjoner avhenger av flere faktorer. Disse faktorene er blant annet sikkerheten i operativsystemet tjeneren kjører på, kommunikasjonskanalen mellom klient og tjener, og hvordan webapplikasjoner er implementert.

Enkelte personer tror sikkerhet i webapplikasjoner begrenser seg til gode brannmurer, kryptert kommunikasjon og *patching* av programvare og lignende infrastrukturrelaterte oppgaver. Resultatet av dette er en sterk økning i antall webapplikasjoner med logiske hull, og som er utsatte for en rekke sårbarheter. Utdanningsinstitusjoner fokuserer ofte på hvordan det skrives programvare, men glemmer gjerne biten som dreier seg om sikkerhet i denne. [6, Kapittel 1]

2.3.1 Trusselbilde

En trussel kan sees på som potensial for tap eller skade av et system og sine brukere [11, Kapittel 2]. Et trusselbilde varierer avhengig av forskjellige faktorer. Disse kan være [11, Kapittel 3]:

- Verdien til den aktuelle ressurs som må være beskyttet
- Kunnskapsnivå til potensielle angripere
- Tidsforbruk for å detektere et angrep

- Hvor godt utstyrt en potensiell angriper er
- Hvor godt kjent en aktuell sårbarhet er

Verdien til den aktuelle ressurs som må være beskyttet

Verdien til en ressurs kan ha stor innflytelse på hvorvidt et angrep mot en webapplikasjon blir utført. Dette kan basere seg på blant annet økonomisk gevinst eller fordi størrelsen og omfanget til webapplikasjonen gjør det til en prestisjeoppgave. Dette kan sees i sammenheng med angrep av typen *Denial of service*[12] som er utført mot webapplikasjoner som Ebay. [11, Kapittel 3]

Kunnskapsnivå til potensielle angripere

På lik linje som en ressurs kan ha en verdi, kan en angriper ha et stort spekter av kompetanse. Det finnes mange som er flinke til å finne sårbarheter som er eksponert, slik at disse kan utnyttes. Ved å kjenne til interne detaljer i en webapplikasjon, kan det kartlegges hvordan denne reagerer på forskjellig input. Større innsyn i en webapplikasjon fører derfor til en større trussel.

Begrepet *Zero-day-exploit* benyttes til å beskrive en sårbarhet det ennå ikke er utviklet løsning til. Evnen til å finne slike sårbarheter, blir ofte sett på som en prestasjon i "Hackermiljøer". Når en slik sårbarhet blir offentliggjort, deles den med andre usofistikerte angripere, ofte referert til som *script kiddies*. Disse har selv ingen forståelse for hvordan koden som utnytter sårbarheten fungerer, og utgjør derfor bare en trussel så lenge denne ikke er oppdaget og rettet. [11, Kapittel 3]

Tidsforbruk for å detektere et angrep

Tidsforbruk for å detektere et angrep er også en viktig faktor i trusselbildet. Angrep som kan gjøres ubemerket, gir ofte en større profitt. Passordstjeling ved hjelp av brute force[13] er et eksempel som demonstrerer dette. Hvis en angriper har mulighet til å prøve et uendelig stort antall forsøk på å gjette passord, vil det sannsynligvis dukke opp et svakt passord. [11, Kapittel 3]

Hvor godt utstyrt en potensiell angriper er

Et angrep som utføres med tilrettelagte verktøy utgjør en større trussel. Dette kan være blant annet verktøy som programvare for feilretting og testing. [11, Kapittel 3]

[12] Denial of service er et angrep med hensikt å gjøre en applikasjon utilgjengelig, ofte gjort ved å sende flere forespørsler enn applikasjonen takler å ta i mot

[13] Brute force baserer seg på gjette seg frem til en løsning ved å prøve et stort antall kombinasjoner, brukes blant annet til å finne passord

Hvor godt kjent en aktuell sårbarhet er

En sårbarhet som ikke kjennes til utgjør en svært liten trussel. Så fort denne blir oppdaget, avgjør spredningsnivået hvor stor trusselen er. Hvis en sårbarhet blir kjent for mange, utgjør den en svært høy trussel. Det ble funnet en svakhet i databasen Microsoft SQL (ormen Slammer) som beviser dette. Svakheten var kjent lenge, men brukerne reagerte ikke før det ble utviklet en orm som utnyttet denne. [11, Kapittel 3]

2.3.2 Sikkerhet som prosess

For å utvikle sikre webapplikasjoner, er det viktig å tenke som en angriper. Ved å kartlegge hvilke sårbarheter som eksisterer, kan det enklere sees hva en selv er utsatt for. [6, Kapittel 1]

Det er viktig at sikkerhet i webapplikasjoner ikke blir et produkt som legges på når utviklingen er ferdiggjort. Det er i tillegg til å tenke på de funksjonelle kravene for å glede brukerne, viktig å ha fokus på hvordan ikke glede potensielle angripere [6, Kapittel 7]. Det er derfor viktig i ethvert utviklingsprosjekt å innføre regler på hvordan sikkerhet skal tas hånd om. Ansvaret må ligge på utviklerne, og disiplin må opparbeides hos den enkelte utvikler [12, Kapittel 2].

2.3.3 Sikkerhetsmål

Modeller har blitt foreslått for å illustrere mål i en sikkerhetspolicy. En av disse er *CIA triad* eller "CIA triangelen" [11, Kapittel 3].

- Konfidensialitet
- Integritet
- Tilgjengelighet

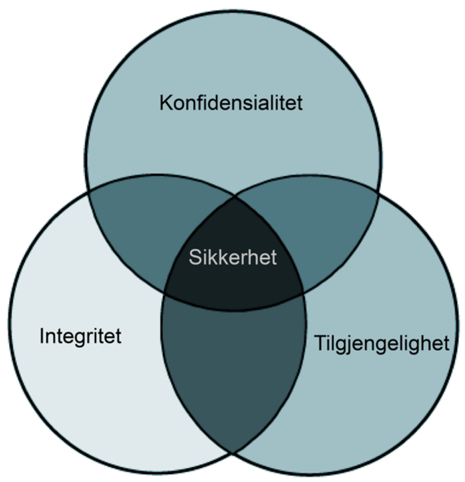
Den amerikanske *National Security Telecommunications and Information Systems Security Comitte* (NSTISSC) påpeker at disse tre er "Kritiske informasjonskarakteristikker". Sikkerhet oppnås når alle disse faktorene er tilstedeværende. Figur 2.3 viser hvordan kombinasjonen av alle tre faktorene samarbeider for sikkerheten, og samtidig hvordan konflikter oppstår når bare to faktorer er tilstede. [11, Kapittel 3]

Konfidensialitet

Konfidensialitet baserer seg på ansvaret for å forhindre tilgang til informasjon fra uautoriserte brukere. Konfidensialitet omfatter alle typer tilgang. En bedrift i sterk konkurranse ønsker eksempelvis ikke å tilgjengeliggjøre konfidensiell informasjon for konkurrenter. [11, Kapittel 3]

Integritet

Integritet kontrollerer hvem som har tilgang til å endre innhold i data. Her er målet å unngå endringer, i motsetning til konfidensialitet hvor målet er å unngå all form for tilgang. Integritet



Figur 2.3: Overlapp av sikkerhetsfaktorer

og konfidensialitet skaper en konflikt. Dette fordi konfidensialitet baserer seg på at informasjon ikke blir sett, i motsetning til integritet som ikke legger begrensninger på dette. Dette kan sees i figur 2.3 som det området hvor konfidensialitet og integritet overlapper hverandre. [11, Kapittel 3]

Tilgjengelighet

Data som ikke er tilgjengelig når en bruker trenger dem, har liten verdi. Tilgjengelighet har til hensikt å påse at data blir tilgjengeliggjort ved en gyldig og autorisert forespørsel. Manglende tilgjengelighet kan oppstå basert på uautoriserte som utfører angrep av typen *Denial of service*. På samme måtte som med integritet oppstår det konflikt. Tilgjengelighet baserer seg på å tilgjengeliggjøre, i motsetning til konfidensialitet som begrenser tilgang. [11, Kapittel 3]

2.4 Oppsummering

Dette kapittelet har tatt for seg en introduksjon til webapplikasjoner ved å se på infrastruktur, byggeklosser i web 2.0 med AJAX og generell sikkerhet i webapplikasjoner. Først ble det nevnt hvordan HTTP-protokollen benyttes for å forespørre data fra tjenerside. Det ble sett at AJAX baserer seg på allerede eksisterende teknologier, med eksempler på hvordan XHR-objektet kan benyttes til å forespørre tjenerside asynkront. Avslutningsvis er det nevnt generelt om sikkerhet i webapplikasjoner, og hvordan konfidensialitet, integritet og tilgjengelighet sammen skaper grunnlag for dette.

KAPITTEL 3
FALLGRUVER OG ANGREPSTYPER

Fallgruve, direkte oversatt fra det engelske ordet *pitfall*, er satt sammen av ordene *pit* og *fall*. Ordet *pit* betyr et hull eller gruve i bakken og *fall* betyr å falle. Ordet fallgruve har et stort bruksområde, og kan derfor benyttes i flere sammenhenger. Ordet blir i oppgaven benyttet til å definere sikkerhetshull og sårbarheter som kan oppstå under utvikling av AJAX-baserte webapplikasjoner.

En fallgruve oppstår i en webapplikasjon når det legges til funksjonalitet som åpner for et eller flere sikkerhetshull. Et slikt sikkerhetshull kan føre til at uønsket innhold sendes mellom brukerens nettleser og en webapplikasjon, og kan resultere i uønskede handlinger både i webapplikasjonen eller i brukerens nettleser. Det finnes også fallgruver som gjør det mulig for en angriper å utføre handlinger på vegne av andre. Dette kan utnyttes hvis en angriper lurer en bruker til å besøke en webside eller klikker på en link, samtidig som brukeren er logget inn i den utsatte webapplikasjonen.

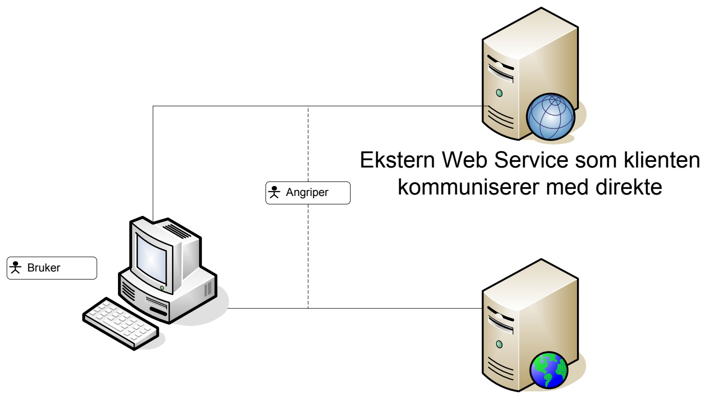
Dette kapittelet ser på forskjellige faktorer som kan resultere i angrep rettet mot slike fallgruver. Flere av disse er også aktuelle i tradisjonelle webapplikasjoner, men AJAX tilfører mer kompleksitet på grunn av sin asynkrone natur [2]. Det blir sett på syv faktorer som alle danner grunnlag for fallgruver og angrepstyper:

- AJAX-angrepsflate
- Svakhet med HTTP-forespørsler
- Kontrolltegnproblematikk
- Brukerinput
- Forretningslogikk i JavaScript
- Cross-Domain forespørsler
- Cross Site Request Forgery

3.1 AJAX-angrepsflate

Antall måter en klient kommuniserer mot webapplikasjonen i kombinasjon med kompleksiteten til disse, benyttes som definisjon på en angrepsflate. Dette kan sees på som alle punkter i webapplikasjonen hvor det tas i mot, eller sendes informasjon. AJAX tilfører ikke flere angrepspunkter, men bidrar til å øke kompleksiteten til de som benyttes. Alle slike punkter i en webapplikasjon er potensielle for angrep, og det kan fort oppstå fallgruver basert på disse. [12, Kapittel 2]

En bidragsgiver for utvidet angrepsflate ved bruk av AJAX er direktekommunikasjon fra klientens JavaScript mot eksterne *Web Service*[1], ofte omtalt som forespørsel av typen *Cross-Domain*. Figur 3.1 illustrerer at to punkter er utsatt for angrep i en slik situasjon. Problemstillinger med *Web service* blir nevnt i delkapittel 3.6. [12, Kapittel 2]



Figur 3.1: Angrepsflaten øker ved plassering av mer logikk på klientsiden

3.2 Svakhet med HTTP-forespørsler

AJAX benytter både POST og GET forespørsler som har opphav fra klienten. Disse forespørslene er fult modifiserbare før de sendes til tjener. Blant annet kan alle HTTP-hodene som ble vist i delkapittel 2.1.1 endres før forespørselen blir sendt. Derfor er det sett fra tjenersiden ingenting som kalles klientsidesikkerhet [6, Kapittel 1]. Ved utvikling av en webapplikasjon, er det derfor svært viktig å holde oversikt over hvilke data som blir overført mellom klient og tjener. [6, Kapittel 3].

3.2.1 Referer-hodet

Et av hodene i HTTP er Referer-hodet[2]. Både forespørsler av typen GET og POST fra delkapittel 2.1.1, viser referer-hodet med verdi satt til URL-adressen fra websiden det ble forespurt. Denne

[1] Web Service er programvare for å støtte interoperabilitet mellom maskiner på Internett

[2] Referer-hodet er spesifisert på `http://www.w3.org/Protocols/HTTP/HTRQ_Headers.html#z14`

vil kunne lekke informasjon til andre sider, og vil derfor være potensiell for misbruk. Når et objekt blir forespurt med enten GET eller POST-forespørsel, vil referer-hodet inneholde URL til forespørselens opphav. For eksempel ved å laste en webside med innhold av et bilde som linker til en annen webside, vil forespørselen for det aktuelle bildet inneholde URL til opphavsiden i referer-hodet.

Et annet problem med referer-hodet er at den genereres på klientsiden. Det er ingenting som forhindrer brukeren å endre denne ved bruk av for eksempel et *proxyverktøy*[3]. Derfor er det liten gevinst i å benytte denne som en sikkerhetssjekk på at brukerens forespørsel kommer fra en gitt webside. [6, Kapittel 1]

3.3 Kontrolltegnproblematikk

En webapplikasjon sender som regel data til et eller flere underliggende system. Et slikt systemet kan være en SQL-database, shell-kommando, XML-fil eller en nettleser. Flere av disse systemene benytter kontrolltegn [13] til å utføre forskjellige operasjoner. For å kommunisere med disse systemene sendes det tekst som inneholder noe kontrolltegn i tillegg til data. Liste 3.1 viser en HTML-fil hvor linje 1,2,4 og 5 inneholder kontrolltegn, og linje 3 er vanlig data.

```
1 <html>
2  <head>
3         Dette er data, mens alt det andre er kontrolltegn.
4  </body>
5 </html>
```

Liste 3.1: Eksempel på kontrolltegn og data

Kontrolltegn representerer i seg selv ingen direkte trussel, men kan opptre som en fallgruve hvis utviklere tror det alltid transporteres rene data. Dette kan bidra til at en angriper kan injisere data med innhold av kontrolltegn for å utføre ønskede operasjoner. Dette kan være kontrolltegn for å utføre SQL-operasjoner, XML-operasjoner eller JavaScript som kan bli presentert for en brukers nettleser. Det er derfor viktig å ta hensyn til hvor data sendes, og hvordan disse tegnene blir tatt hånd om. [6, Kapittel 2]

Det blir i dette delkapittelet sett på fem angrepstyper rettet mot fallgruver som baseres seg på kontrolltegn:

- Cross-site Scripting (XSS)
- Asynkront XSS ved hjelp av AJAX
- XSS Prototype Hijacking
- SQL-injisering
- XML-injisering

[3]Proxyverktøy er programvare for å modifisere HTTP-forespørselen før den sendes til tjenerside

3.3.1 Cross-site Scripting (XSS)

XSS som også er aktuelt i tradisjonelle webapplikasjoner, går ut på å lure en webapplikasjon til å vise ondsinnet HTML-kode med innhold av JavaScript. Ved å presentere dette for en annen bruker, kan en angriper blant annet stjele den aktuelle sesjonsid som står for tilstandshåndteringen, nevnt i delkapittel 2.1.2. Slik er det mulig å opptre som offeret når det kommuniseres mot webapplikasjonen. [6, kapittel 4]

Dette er enkelt å illustrere, og det trengs ikke å lete lenge for å finne problemet i en reel webapplikasjon. En av Norges mest populære internetttilbydere, tilbyr brukerne sine å lese e-post gjennom en webapplikasjon. En test av denne viste at *body*-elementet med attributtet *onload* ikke filtreres. Dette kan benyttes til å kjøre JavaScript, og ved å sende en e-post med følgende emne til en bruker, vil en beskjed vise meldingen “Åpent for XSS”, se figur 3.2[4].

```
<body onload="alert('Apent for XSS')">
```

Figur 3.2: XSS-svakhet i kjent internetttilbyder

En angriper kan utnytte dette til blant annet å få innsyn i en brukers e-post ved å sende JavaScript som redirigerer til en ekstern webside med *document.cookie*[5] som parameter. Deretter kan angriper plassere den stjålne cookie til sesjonen i sin nettleser, for så å kapre sesjonen som pågår. Programkoden i liste 3.2 viser et eksempel på JavaScript som kan benyttes for å stjele den aktuelle cookie til tilstandshåndtering, og dermed kapre den pågående sesjonen. [6, Kapittel 4]

```
1  <body onload="document.location.replace(
2 "http://badguy.com/stjeleapp.jsp?cookievalue="+ document.cookie)">
```

Liste 3.2: XSS-eksempel

[4] Svakheten ble rapportert 28.mai 2007

[5] *document.cookie* benyttes i JavaScript for å hente ut den aktuelle cookie som benyttes

3.3.2 Asynkront XSS ved hjelp av AJAX

AJAX-baserte webapplikasjoner åpner for mer avanserte XSS-angrep. AJAX tilfører mulighet til å skjule angrepene ved å benytte asynkrone forespørsler. Hvis en angriper lykkes i å presentere et JavaScript for en annen bruker, kan dette utnyttes til å sende forespørsler med den aktuelle brukerens sesjon uten å forfriske websiden. Med tradisjonell XSS kan det kun stjeles informasjon fra websiden brukeren ser på, samt informasjon i eventuelle cookies som benyttes av denne. Da AJAX baserer seg på at JavaScript bygger innhold i dokumentet, vil JavaScriptet som blir presentert på vegne av angriper vedvare helt til brukeren velger å se en annen webside. [14]

AJAX-baserte webapplikasjoner kan medføre at JavaScript injisert ved hjelp av XSS fungerer som virus. JavaScriptet som blir injisert, kan asynkront injisere seg i flere deler av webapplikasjonen, uten å forfriske websiden som presenteres for brukeren. Et eksempel på dette ble demonstrert i oktober 2005, da et medlem i *MySpace.com* lagde en profil som inkluderte en selvforplantende XSS-orm, ved navnet *Samy worm*. Når profilen til medlemmet ble vist for en bruker, la ormen automatisk brukeren til i *Samy's* venneliste. Ormen kopierte seg også inn i brukerens profil, som igjen førte til at den spredte seg videre ved presentasjon for andre. [14]

3.3.3 XSS Prototype Hijacking

XSS Prototype Hijacking er et angrep som kan føre til total kontroll over en AJAX-basert webapplikasjon. Grunnlaget for et slikt angrep er at JavaScript er et prototypebasert programmeringsspråk. Det vil si at objekter lages ikke av klasser, men opprettes ved å klone det originale objektet som eksisterer. Nye funksjoner til et slikt objekt kan enkelt defineres ved å legge de til det originale objektet. [2]

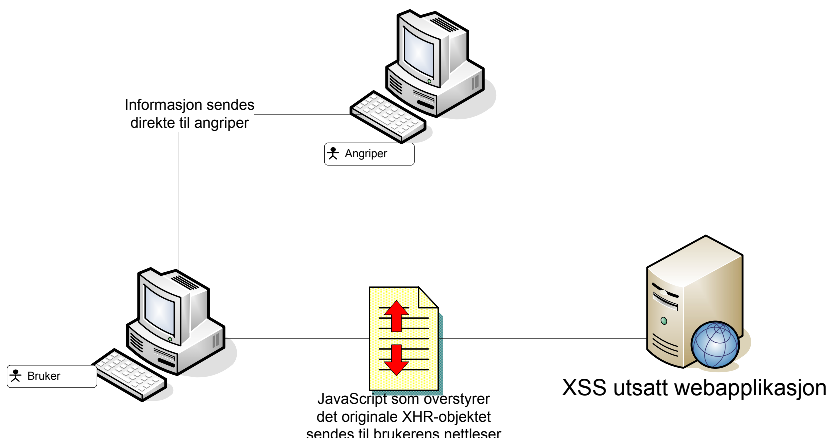
XHR-objektet er en av byggeklossene i AJAX, og står for den asynkrone kommunikasjonen sett i delkapittel 2.2.1. Liste 3.3 linje 1,2 og 3 vises hvordan en ny funksjon kan legges til det originale XHR-objektet. På linje 4 opprettes deretter en klone av dette objektet. Denne kopien og alle andre kopier av XHR-objektet vil heretter ha funksjonen *newFunction* knyttet til seg. [2]

```
1 XMLHttpRequest.newFunction=function(){
2  return ``A value'';
3 }
4 var xmlhttp = new XMLHttpRequest();
```

Liste 3.3: XSS prototype Hijacking

Selv om slik funksjonalitet kan være ekstremt kraftig ved utvikling, medfører den også angrepsmuligheter da det originale objektet kan overskrives med selvgenererte funksjoner. Dette kan utnyttes til å lage en *wrapper* som pakker inn det originale XHR-objektet. Hvis webapplikasjonen er svak for XXS, nevnt i delkapittel 3.3.1 og 3.3.2, kan en angriper endre alle funksjoner tilknyttet XHR-objektet. [2]

Prototype Hijacking kan blant annet benyttes til angrep av typen *man in the middle*, ved at JavaScript sender informasjon til angriper slik figur 3.3 illustrerer. En annen fordel sett fra angripers side, er muligheten til å forbigå alle former for kryptering, engangs passord og lignende da JavaScriptet kjører i bakgrunn i den autentiserte brukerens sesjon. [2]



Figur 3.3: Prototype Hijacking [2]

3.3.4 SQL-injisering

SQL-injisering er et kontrolltegnproblem som også kan opptre i tradisjonelle webapplikasjoner. Situasjonen vises i liste 3.4, der det benyttes *JavaServer Pages* (JSP) til å hente ut to parametere, for deretter å bygge en SQL-spørring basert på disse. Denne danner bakgrunn for innlogging i en webapplikasjon. [6, Kapittel 2]

```
1 String userName = request.getParameter("user");
2 String passWord = request.getParameter("pass");
3 String sqlQuery = "SELECT * FROM Usr "
4         + "WHERE userName="+userName+" "
5         + "AND PassWord="+passWord;
```

Liste 3.4: Eksempel på SQL-injisering [6, Kapittel 2]

Programkode i liste 3.4 er mottagelig for SQL-injisering, da det ikke legges noen begrensninger på hva variablene userName og passWord inneholder. Hvis en angriper skriver brukernavnet john' - - og lar passordfeltet være tomt, vil innloggingen fullføres så fremt en bruker med brukernavn = "john" eksisterer i databasen. Tegnkombinasjonen "- -" definerer start på kommentering i SQL-setninger, og vil i denne sammenheng medføre at alt fra "- -" i SQL-spørringen ikke blir tolket og kjørt. [6, Kapittel 2]

3.3.5 XML-injisering

XML-injisering er på lik linje med SQL-injisering et kontrolltegnproblem [13]. På samme måte som ved SQL-injisering utføres et slikt angrep ved at angriper benytter kontrolltegn til å manipulere og hente ut informasjon fra tjenerside.

En variant av XML-injisering er XPath-injisering. Xpath er et spørrespråk for å hente ut data fra et XML-dokument på lignende måte SQL benyttes til å hente ut data fra en database [9, Kapittel 7]. XPath-spørringer baserer seg på å søke og navigere i noder som representerer elementer i et

XML-dokument. En angriper kan på lik linje med SQL-injisering, sende inn kontrolltegn som kan resultere i uheldige hendelser. [15, Kapittel 8]

3.4 Brukerinput

AJAX-baserte webapplikasjoner sender og mottar data asynkront i bakgrunn. Disse dataene forteller hvilke operasjoner som skal foretas på tjenersiden. Å godta feil data fra klienten, kan få webapplikasjoner til å foreta feil valg. Input til en webapplikasjon kan blant annet være:

- URL-parametere
- Data brukeren skriver inn selv (brukergenerert)
- Data tjenerside genererer (for eksempel radioknapper og checkbokser)
- Cookies og annen informasjon i en HTTP-forespørsel
- Eksterne filer eller databaser benyttet av webapplikasjonen

Den "Usynlige sikkerhetsbarrieren" er et begrep som har til hensikt å beskrive vanskeligheten med å vite når data opptrer i webapplikasjonen eller blir sendt ut til brukerens klient. Det er viktig å være klar over at alt som sendes til nettleseren er modifiserbart ved hjelp av blant annet proxyverktøy, DOM-manipuleringsverktøy som *Firebug* for nettleseren Mozilla Firefox, eller å lagre den aktuelle HTML-filen og modifisere den direkte. [6, Kapittel 3]

3.4.1 Eksempel på manipulering av tjenersidegenerert input

Tjenersidegenerert input, kan være data som blir sendt fra tjenersiden til klientsiden for at brukeren skal gjøre et valg. Liste 3.5 viser et eksempel hvor kontonummer er tjenersidegenerert input. Kontonumrene som blir presentert, tillater brukeren å velge hvor det skal overføres penger fra. Det vil for en angriper være fristende å endre disse kontonumrene før det blir foretatt en forespørsel mot webapplikasjonen, i håp om å overføre penger fra en annen bruker.

```
1  <select name="fromAccount">
2   <option value="1234.53.1231">1234.53.1231</option>
3   <option value="1234.23.3241">1234.23.3241</option>
4  </select><br>
5  <input name="toAccount" type="text"/><br>
6  <img src="pay.png" onclick="transfer()"/>
7  <script type="text/javascript">
8   function transfer(){
9    //foreta kall av tjenersidefunksjonalitet for å overføre penger mellom kontoer
10  }
11 </script>
```

Liste 3.5: Eksempel på manipulering av tjenersidegenerert input

3.5 Forretningslogikk i JavaScript

Interaktive AJAX-baserte webapplikasjoner vil kreve klientsidelogikk. Ofte kan utviklere falle for fristelsen å plassere validering, kalkuleringer, autentisering og andre viktige funksjoner i JavaScript. Dette vil bidra til ytterligere angrepspunkter. [12, kapittel 2]

Det er viktig å vite at JavaScript kjører i brukerens nettleser, og er tilgjengelig for lesing og modifisering på klientsiden. Det er derfor liten (ingen) gevinst i å bruke tid på å gjemme bort funksjonalitet i denne. Alle nettlesere har mulighet til å vise alt JavaScript til den aktuelle websiden som sees, i tillegg til alle inkluderte .js (JavaScript-filer). [12, kapittel 2]

3.5.1 Klientsidevalidering med JavaScript

En farlig og falsk sikkerhet i en webapplikasjon, kan basere seg på klientsidevalidering av data som sendes fra klient til tjener for å filtrere forskjellige typer input. Klientsidevalidering må aldri benyttes til dette formålet, men kan spare en ekstra tur til tjenersiden og gi en rask tilbakemelding til brukeren hvis input ikke er gyldig. [12, Kapittel 2] og [6, Kapittel 3]

3.6 Cross-Domain forespørsler

En av de senere bruksområdene til AJAX baserer seg på å samle innhold fra forskjellige kilder også kalt *Web Service*. For eksempel kan det være ønskelig å samle informasjon fra kartsystemet Google Maps[6] mot et system for salg av eiendommer. Et slikt system vil kunne levere en interaktiv navigasjonstjeneste for boligkjøpere som ønsker å se hvor det er boliger til salgs.

For å ivareta sikkerhet og opphavsretter, tillater ikke moderne nettlesere at JavaScript benytter XHR-objektet til å kommuniserer mot andre domener enn der det har opphav. Heller ikke tillates det at JavaScript i en inkludert *IFrame*[7] fra et eksternt domene interpreterer med den aktuelle websidens JavaScript. Denne sikkerhetsmekanismen refereres ofte til som *Cross-domain restriction* [12, kapittel 2] eller *Server of origin policy* [5, Kapittel 7].

For å få tilgang til eksterne *Web Services*, blir det derfor benyttet alternative metoder for å unngå *Server of origin policy*. De mest brukte metodene baserer seg på:

- Web Service proxy/bro
- Dynamisk script-element

3.6.1 Web Service proxy/bro

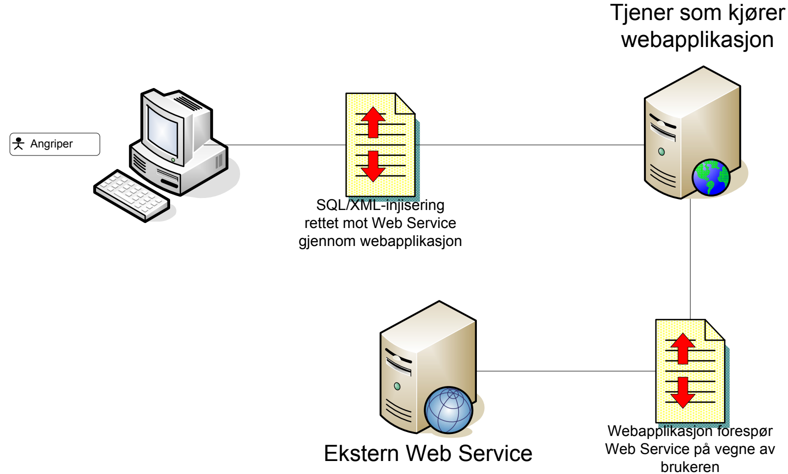
Ved å legge opp til at alle forespørsler mot eksterne *Web services* skjer fra tjenersiden, unngås problemet med usikre kanaler for kommunikasjon direkte fra klientsiden. Slik funksjonalitet defineres som en *proxy* eller *Web service*-bro. Tjeneren vil ha alt ansvar for å hente informasjon fra en ekstern *Web service* på forespørsel fra klienten, for deretter å levere denne informasjonen

[6] Google Maps er en interaktiv karttjeneste tilgjengelig på `http://maps.google.com/`

[7] En *Iframe* benyttes for å inkludere eksternt HTML-innhold i en webside

tilbake. En slik løsning vil ikke bryte med nettlesernes *Server of origin policy*. Samtidig vil det være kontroll på tjeneren over data sendt til den aktuelle *Web service*, og hvilke data returnert til nettleseren. [5, Kapittel 7]

Det eksisterer også ulemper med en slik løsning. Først og fremst vil en slik tjeneste fremstå som en flaskehals. Dette kan føre til mindre interaktivitet hos klienten på grunn av ventetid [5, Kapittel 7.2]. En slik løsning kan i tillegg være utsatt for injiseringsangrep ved at en angriper benytter broen til å XML/SQL-injisere den aktuelle *Web service* gjennom *proxyen* uten at angriperen synliggjør seg selv. Figur 3.4 illustrerer situasjonen. [14]



Figur 3.4: Kontrolltegninjisering av *Web service* gjennom webapplikasjon

3.6.2 Dynamisk script-element (Alternativ til XHR-objektet)

En alternativ metode til å oppnå *cross domain*-forespørsler, baserer seg på bruk av dynamiske script-elementer i kombinasjon med utveksling av JavaScript-objekt. Som JavaScript-objekt er *JavaScript Object Notation* (JSON) [16] mest brukt. JSON er et datautvekslingsformat som er enkelt for mennesker å lese og skrive. Samtidig er det enkelt for maskiner å tolke og generere. Det kan også benyttes andre former for objekter, men JSON blir oftest benyttet på grunn av sin fleksibilitet. Metoden gjør det mulig å utveksle informasjon med den ønskede *Web service* sømløst på kryss av domener, uten å være bundet av nettlesernes *cross domain restriction*. Dette forenkler problemstillingen ved bruk av XHR-objektet til *Cross domain*-forespørsler.

Metoden baseres på hvordan script-elementer kan referere til eksterne filer for JavaScript. En ekstern JavaScript-fil (js) kan deklareres på følgende måte:

```
<script src='eksternFil.js'/>
```

Dette gjør det mulig å forespørre den ønskede *Web service* om eksternt JavaScript som blir benyttet direkte i brukerens nettleser. Metoden kan demonstreres ved følgende forløp basert på eksempelet i liste 3.6, hvor det benyttes Yahoo‘s *Web service* for å hente ut antall søketreff på et bilde. Resultatet av eksempelet vises i figur 3.5 [12, Kapittel 2]

1. Linje 4. oppretter funksjon getImageResults()

2. Linje 5-7. Definerer hvilken URL Yahoo‘s *Web service* svarer på. Parameterne er spesifikke for Yahoo‘s *Web service*. parameteren “callback” definerer tilbakekallfunksjonen som skal kjøres når responsen kommer.

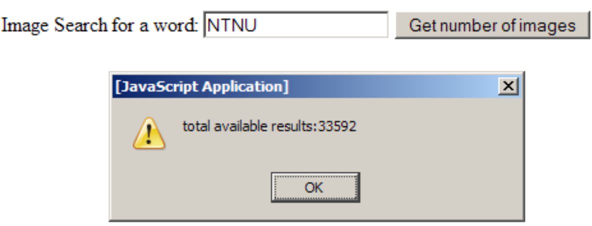
3. Linje 8. Henter ut referanse til <head> elementet til den aktuelle websiden

4. Linje 9 - 11. Lager script-element med attributt src som inneholder url definert i linje 5-6. Deretter legges script-elementet til som barneelement til <head>-elementet. Yahoo formidler responsen til src attributtet i form av et JSON JavaScript-objekt. Resultatet av dette er at JavaScriptet som returneres fra Yahoo blir plassert inne i script-elementet.

5. Linje 13. Funksjon som tar seg av å formidle innholdet i responsen, i dette tilfellet gjøres det ved hjelp av alert()

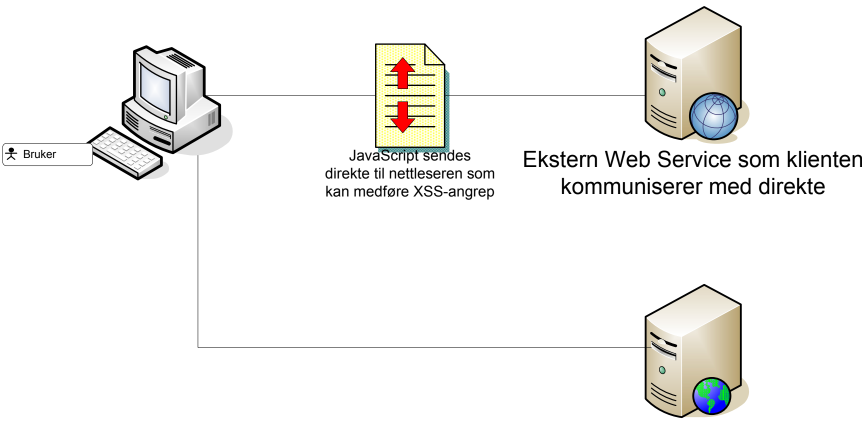
```
1  <html>
2   <head>
3    <script>
4     function getImages(){
5      var st = document.getElementById("image").value;
6      var url = "http://api.search.yahoo.com/ImageSearchService/V1/imageSearch?"+
7       "appid=YahooDemo&query="+st+"&results=2&output=json&callback=handleResults";
8      var headElement = document.getElementsByTagName("head").item(0);
9      var scriptElement = document.createElement("script");
10     scriptElement.src = url;
11     headElement.appendChild( scriptElement );
12    }
13    function handleResults(handleJsonResult){
14     alert( "total available results:"+handleJsonResult.ResultSet.totalResultsAvailable
          )
15    }
16   </script>
17  </head>
18  <body>
19    Image Search for a word: <input type="text" id="image" name="image"/>
20   <input type="button" value="Get number of images" onclick="getImages()" />
21  </body>
22 </html>
```

Liste 3.6: Tilgang til Yahoo‘s *Web service* ved hjelp av dynamiske script-elementer



Figur 3.5: Resultat av Yahoo‘s *Web service*

Det er viktig å legge merke til fallgruven i forbindelse med bruk av en slik løsning. Responsen fra den benyttede *Web service* kan inneholde ondsinnede JavaScript som blir lastet ned i nettleseren til brukeren. Disse vil kjøre med samme rettigheter som alt annet JavaScript i websiden, og kan bidra til et potensielt XSS-angrep, nevnt i delkapittel 3.3.1. Figur 3.6 illustrerer situasjonen. Det er derfor viktig å være kritisk til webapplikasjoner som skal basere seg på denne teknikken og aktørene bak den aktuelle *Web service*. [17]



Figur 3.6: XSS-problem på grunn av upålitelig *Web service*

3.7 Cross Site Request Forgery (XSRF)

XSRF også kjent som *Session Riding* beskriver en ny klasse fallgruver og angrep rettet mot brukere av webapplikasjoner. XSRF er også aktuelle i tradisjonelle webapplikasjoner, men på lik linje med XSS, tilfører AJAX mulighet for asynkrone angrep uten at brukeren er bevist på problemet. XSRF er mindre kjent for utviklere enn fallgruver med kontrolltegn som SQL-injisering og XSS. [18]

XSRF utnytter at webapplikasjoner ikke klarer å skille mellom autentiske forespørsler, og forespørsler som ble initiert fordi brukeren ble lurt. Dette baserer seg på at brukeren har initiert en sesjon med den aktuelle webapplikasjonen. Så lenge brukeren er autentisert, vil et enkelt klikk eller å besøke en webside være nok til å utføre et angrep. Mange utviklere av webapplikasjoner er ikke klar over dette. [18]

For å demonstrere XSRF, kan det tas utgangspunkt i en enkel forespørsel for å overføre penger fra et kontonummer i en webapplikasjon til en bank. [18]

3.7.1 XSRF ved GET-forespørsel

En bank som er utsatt for XSRF, kan basere seg på ta i mot en GET-forespørsel på følgende form for å overføre penger mellom kontoer:

```
GET /transfer.php?ammount=100000&to=2321.23.1234
```

En angriper kan få en bruker som er autentisert mot nettbanken til å klikke på en link med følgende innhold:

```
<a href='bank.com/transfer.php?ammount=100&to=2321.23.1234'>Click</a>
```

Denne linken kan distribueres ved å sende denne på e-post, presentere den på webside eller lignende. [18]

Ofte vil brukere være skeptiske til å klikke på knapper når de ser den aktuelle URL som blir forespurt. Det finnes alternative metoder for å utsette brukere for et slikt angrep. Blant annet kan GET-forespørselen ligge gjemt i et image-element til en webside med src-attributt som genererer forespørselen:

```
<img src='bank.com/transfer.php?ammount=100&to=2321.23.1234'/>
```

Det er i motsetning til situasjonen hvor angriper belager seg på at bruker klikker på en link, her nok at brukeren laster den aktuelle websiden med image-elementet.

3.7.2 XSRF ved POST-forespørsel

Et ofte benyttet råd for å minske risikoen for XSRF, er å benytte forespørsel av typen POST fremfor GET. Dette er misvisende, og det må ikke sees på som sikkert å basere forespørsler på POST for å unngå denne fallgruven. XSRF er like aktuelt for POST-forespørsler som for GET-forespørsler. Liste 3.7 viser innhold i en webside som vil kjøre POST-forespørsel automatisk når den lastes. [18]

```
1 <form action="bank.com/transfer" method="post"/>
2  <input type="hidden" name="to" value="2321.23.1234"/>
3  <input type="hidden" name="amount" value="100000"/>
4  <input type="submit"/>
5 </form>
6 <script type="text/javascript">
7  document.forms[0].submit();
8 </script>
```

Liste 3.7: XSRF-eksempel med POST

3.7.3 XHR-objektet kan ikke benyttes til XSRF

På grunn av restriksjon *Cross-domain restriction* i moderne nettlesere, nevnt i delkapittel 3.6, må en angriper basere seg på å generere POST og GET-forespørsler uten å benytte XHR-objektet. Angripere vil derfor være avhengig av å benytte dynamiske script-elementer, image-elementer eller *iframes* til å foreta GET-forespørsler. Det vil også være nødvendig å benytte metoden vist i liste 3.7 for å foreta POST-forespørsler på en brukers vegne.

3.8 Oppsummering

Dette kapittelet har beskrevet forskjellige fallgruver med eksempler på praktisk bruk for å utføre angrep rettet mot disse. Basert på trusselbildet i delkapittel 2.3 vil disse fallgruvene utgjør en risiko for sikkerhetsmålene som ble nevnt i delkapittel 2.3.

Tabell 3.1 viser at alle de tre sikkerhetsmålene fra delkapittel 2.3 blir svekket av de forskjellige fallgruvene og angrepstypene.

Fallgruver/sikkerhetsmål	Konfidensialitet	Integritet	Tilgjengelighet
Kontrolltegnproblematikk	X	X	X
Brukerinput	X	X	X
Forretningslogikk i JavaScript	X	X	X
Cross-Domain forespørsler	X	X	X
XSRF	X	X	X

Tabell 3.1: Fallgruvene påvirker sikkerhetsmål

KAPITTEL 4
FORHOLDSREGLER

For å forhindre fallgruvene og angrepstypene som vil svekke sikkerheten i AJAX-baserte webapplikasjoner, kan det tas forskjellige forholdsregler. Dette kapittelet ser på fire av dem:

1. Validering og filtrering av data på tjenerside
2. Beskyttelse i dybden
3. Begrense data som sendes til klientside
4. XSRF-beskyttelse

Tabell 4.1 viser hvilke av disse forholdsreglene som bør tas mot de forskjellige faktorene for fallgruver presentert i kapittel 3.

Forholdsregler mot fallgruver og angrepstyper	1	2	3	4
Svakheter med HTTP-forespørsler	X	X	X	-
Kontrolltegnproblematikk	X	X	-	-
Brukerinput	X	X	X	-
Forretningslogikk i JavaScript	-	-	X	-
Cross-Domain forespørsler med proxy	X	X	X	-
Cross-Domain forespørsler med dynamisk script-element	-	-	-	-
XSRF	-	-	-	X

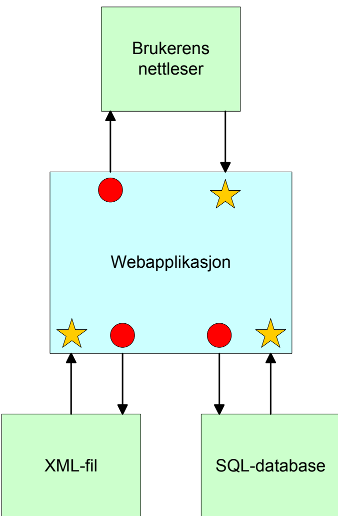
Tabell 4.1: Hvilke fallgruver og angrep som forholdsreglene kan forhindre

4.1 Validering og filtrering av data på tjenerside

En av de mest effektive metodene for å sikre en webapplikasjon, er å tjenersidevalidere og tjenersidefiltrere alle data som blir sendt til og fra en webapplikasjon. Det ble i kapittel 3

sett hvordan svakheter med HTTP-forespørsler, kontrolltegnproblematikk, brukerinput og *cross-domain*-forespørsler kan utgjøre fallgruver og kan bidra til angrep.

Utviklere har ofte problemer med å skille mellom et inputproblem og kontrolltegnproblem, og derfor blir disse to faktorene sett på under ett. Dette gjør det vanskelig for utviklere å lage regler over hvilke kontrolltegn som skal miste mening, når det på samme sted i webapplikasjonen tas hensyn til kontrolltegn for flere subsystemer. Derfor bør det skilles mellom data som ankommer webapplikasjonen og som blir sendt ut fra webapplikasjonen til et subsystem (for eksempel nettleser, database eller XML-fil). Figur 4.1 viser på hvilke steder en webapplikasjon bør plassere håndtering av inputproblemet og kontrolltegnproblemet. [3]



Figur 4.1: Sirklene angir plassering for filtrering av kontrolltegn og stjernene for validering av input [3]

4.1.1 Validere input

Validering av input benyttes for å sjekke at parametere som sendes til webapplikasjonen er gyldig. Dette kan være å kontrollere at data er på riktig form (eksempel bokstaver fra a-z), eller å holde kontroll på at brukere ikke får tilgang til å utføre operasjoner basert på andre parametere enn de som er knyttet mot brukeren. Delkapittel 3.4 nevnte forskjellige typer input i en webapplikasjon, og det ble illustrert et eksempel i delkapittel 3.4.1 hvor kontonummer blir benyttes som tjenergenerert input. Validering av kontonummeret som sendes til webapplikasjonen i forespørselen vil være avgjørende for å unngå uautoriserte pengeoverførsler.

Det er åpenbart at alt brukergenerert input må valideres, men det er viktig å være klar over at alle andre kilder også må valideres. Dette fordi alt som sendes til klienten enkelt kan manipuleres og endres av brukeren. Det kan i tillegg være vanskelig å vite når data opptrer på tjenersiden, og når de når klientsiden. Derfor er det viktig å identifisere alle kilder til input for en webapplikasjon, og alltid validere disse. Målet vil være å sikre at webapplikasjonen behandler data som har forventet form når de ankommer. [6, Kapittel 3]

Mekanismer for validering av input bør plasseres på det punktet hvor data ankommer webapplikasjonen slik som illustrert i figur 4.1 markert med stjerner. Feil i brukergenerert input (for eksempel bruk av bokstaver i kontonummer) bør forkastes med en feilmelding til brukeren. Det bør i tillegg forkastes forespørsler hvor det er modifisert andre typer input med beskjed til brukeren at situasjonen er logget. En slik melding kan fungere preventivt da en angriper ofte avslutter forsøket på å angripe en webapplikasjon, når det gis meldinger om at slike hendelser blir logget. [6, Kapittel 3]

4.1.2 Filtrere kontrolltegn (Output encoding)

Filtrering av kontrolltegn benyttes for å forhindre uønskede operasjoner i en webapplikasjon. Dette vil bidra til å forhindre angrep som XSS og SQL/XML-injisering nevnt i delkapittel 3.3. Disse tegnene kan enten endres slik at de mister mening, eller de kan forkastes. Det er viktig å kartlegge hvilke subsystemer som benyttes, og ta i bruk mekanismer for å håndtere data sendt til hvert av disse. [6, Kapittel 2]

Mekanismer for å forhindre uønskede kontrolltegn bør plasseres rett før data sendes til det aktuelle subsystemet, slik sirklene i figur 4.1 illustrerer. Enkelte velger å behandle dette når data ankommer webapplikasjonen, fordi det blir sett på som et inputproblem. Tre argumenter for å ta seg av filtrering av data før de når sitt subsystem er:

- Ikke bare brukergenerert input må filtreres, det kan hende systemet leser data fra filer og lignende.
- Data kan ha forskjellige subsystem som destinasjon, og disse må derfor individuelt filtreres.
- Data som skal lagres til database, bør ikke filtreres for annet enn SQL-kontrolltegn. Dette for å unngå uheldige lengder på dataene som skal lagres i forholdt til begrensninger i databasefelt.

4.1.3 Hvitlister og applikasjonslogger

Hvitlister baserer seg på en liste med alt lovlig som kjennes til [6, Kapittel 3]. En slik liste kan benyttes til å sjekke hvorvidt en operasjon tillates eller ikke. Dette kan benyttes både til filtrering av kontrolltegn og validere input.

Svartelisting er det motsatte av hvitlisting, og vil være mindre omstillingsdyktig [6, Kapittel 3]. Svartelisting definerer i motsetning til hvitlisting hva som er ulovlig, og vil være mindre motstandsdyktig når en “ny” fallgruve blir kjent.

4.2 Beskyttelse i dybden

Beskyttelse i dybden (engelsk *Defense in depth*) baserer seg på flere sikkerhetsmekanismer i parallell. Dette kan gjøres ved å benytte flere mekanismer i webapplikasjonen som utvikles, eller ved å benytte infrastrukturelle verktøy. Dette kan blant annet være strenge tilgangsrettigheter på webtjeneren og databasen som benyttes. [6, Kapittel 2]

4.3 Begrense data som sendes til klientside

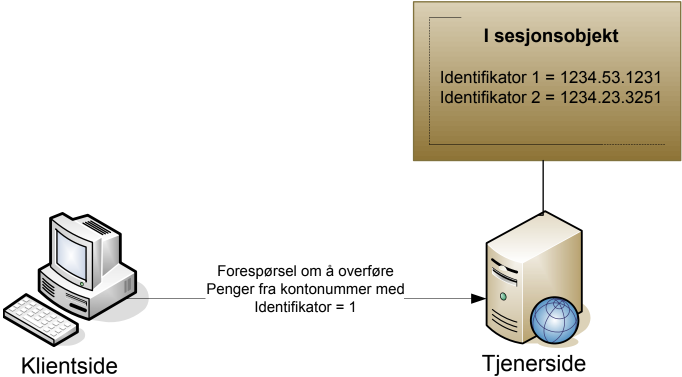
En webapplikasjon baserer seg på kommunikasjon mellom klient og tjener. Ofte vil en angriper forsøke å modifisere JavaScript på klientsiden, eller endre på parametere som sendes til webapplikasjonen. Det er derfor svært viktig å ta hensyn til data sendt til klienten, og data som bør begrense seg til tjenersiden. Alt på tjeneren kan sees på som trygt, i motsetning til nettleseren hvor det er umulig å gjemme hverken logikk eller parametere slik det ble nevnt i delkapittel 3.4 og 3.5.

4.3.1 Unngå å plassere applikasjonslogikk i JavaScript

Det ble nevnt i delkapittel 3.5 at økende klientsidelogikk ved hjelp av JavaScript kan bidra til svekket sikkerhet. Det er derfor viktig at utviklere har et klart skille mellom JavaScript på klientsiden og programkode på tjenersiden.

4.3.2 Indirekte datatilgang

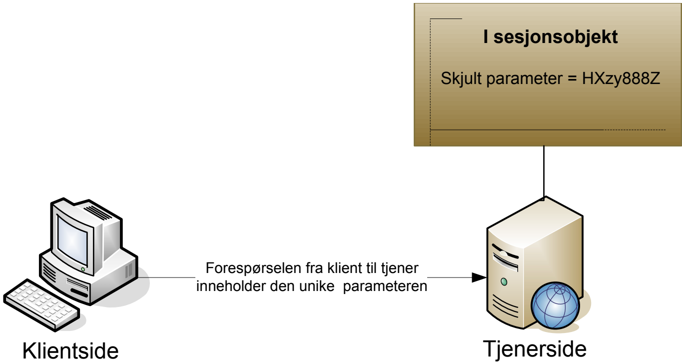
Indirekte datatilgang baserer seg på å ikke eksponere applikasjonskritiske parametere til klientside. Fra eksempelet i delkapittel 3.4.1, kan dette gjøres ved å knytte kontonumrene opp mot identifikatorer som sendes til klienten, slik det er illustrert i figur 4.2. For å angi hvilket kontonummer det skal overføres penger fra, kan det alternativt benyttes tallene 1 og 2 som parametere. Koblingen mellom disse tallene ligger kun på tjenersiden. Et forsøk på å endre denne parameteren, vil i verste fall føre til at en angriper overfører penger fra en av sine andre kontoer. [6, Kapittel 3]



Figur 4.2: Eksempel på indirekte datatilgang ved at koblingen mellom identifikatorene og kontonumrene ligger i sesjonen på tjenersiden

4.4 XSRF-beskyttelse

XSRF er et angrep som skiller seg ut fra kontrolltegnangrep. Dette fordi et angrep baserer seg på å få en bruker til å foreta forespørsler på vegne av angriper. En måte å sikre seg mot dette, er å benytte en ny og unik identifikator i brukerens sesjon for hver forespørsel. Når brukeren foretar en ny forespørsel mot tjenersiden, sendes den forrige unike identifikatoren med slik figur 4.3 illustrerer. Tjenersiden sjekker deretter om denne er lik den som ligger i brukerens sesjon. Hvis disse to er identiske, er det høy sannsynlighet for at forespørselen er autentisk. [12, Kapittel 3] og [18]



Figur 4.3: XSRF-beskyttelse ved at en unik parameter sendes i hver forespørsel for å sikre at denne er autentisk

4.5 Cross-domain ved dynamiske script-elementer må basere seg på et tillitsforhold

Det er ikke definert forholdsregler som sikrer mot fallgruven ved dynamisk script-element, for å oppnå forespørsler av typen cross-domain, nevnt i delkapittel 3.6.2. Ved bruk av denne type løsning, vil sikkerheten basere seg på et tillitsforhold til den aktuelle *Web service* som benyttes. Det er derfor svært viktig å gjøre nøye vurderinger av hvilke eksterne tjenester det ønskes å kombinere i en AJAX-basert webapplikasjon. Det svake leddet i sikkerheten kan være den aktuelle *Web service* som blir benyttet.

4.6 Oppsummering

Dette kapittelet har presentert fire forholdsregler som kan tas for å forminske risikoen for fallgruver og angrepstyper nevnt i kapittel 3. Enkelte av disse forholdsreglene kan tas ved å implementere egne mekanismer, eller de kan være støttet i rammeverk. Neste kapittel ser på tre forskjellige AJAX-baserte rammeverk.

KAPITTEL 5
AJAX-RAMMEVERK

Rammeverk er blitt utviklet for at det skal være enklere å benytte AJAX i webapplikasjoner. Det er ofte tidkrevende å skrive JavaScript for å kommunisere mot tjenersiden ved hjelp av XHR-objektet. XHR-objektet er en del av byggeklossene i AJAX, og ble beskrevet i delkapittel 2.2.1. Slik det ble sett i kapittel 3, finnes det fallgruver som utviklere kan støte på. På grunn av blant annet dette, er det laget AJAX-rammeverk som har til hensikt å effektivisere arbeidet til utviklere, både med hensyn til tid og sikkerhet.

5.1 Kategorier av rammeverk for AJAX

Det finnes i hovedsak tre hovedkategorier av rammeverk, disse er:[12, kapittel 1]

- Scriptbibliotek
- Kodegeneratorer
- Applikasjonsrammeverk

5.1.1 Scriptbibliotek

Et scriptbibliotek består av en samling ferdiglagde JavaScript for manipulering av DOM og Asynkron kommunikasjon mot tjenersiden. Eksempler på slike rammeverk er Prototype JavaScript [19] og Doojo toolkit [20]. Disse rammeverkene hadde tidligere hovedmål for at brukerterskelen for Dynamisk HTML (DHTML)[1] skulle være lavere. [12, Kapittel 1]

[1] Dynamisk HTML beskriver interaktive og animerte websider

5.1.2 Kodegeneratorer

Kodegeneratorer er rammeverk som tillater utviklere å lage interaktive AJAX-baserte webapplikasjoner i sitt eget programmeringsspråk, for deretter å oversette koden for denne til HTML med JavaScript. Ved bruk av slike rammeverk, unngås det å bruke tid på å lære seg JavaScript, og hvordan dette opptrer forskjellig på kryss av nettlesere. Rammeverket fra Google kalt *Google Web Toolkit* (GWT) [21], er en kodegenerator som tillater utvikleren å skrive klientsidekoden i Java, for deretter å konvertere denne til klientside-HTML og JavaScript. GWT tilbyr i tillegg støtte for et bredt utvalg ferdiglagde komponenter. [12, Kapittel 1]

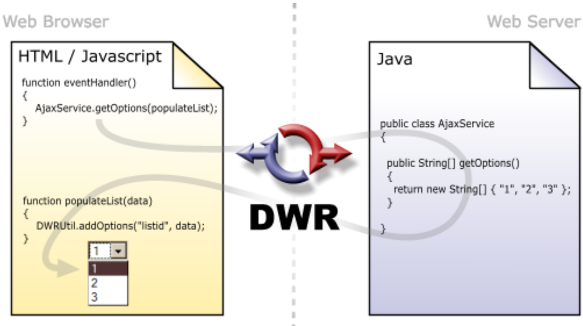
5.1.3 Applikasjonsrammeverk

Et applikasjonsrammeverk er et komplett verktøy for å lage interaktive AJAX-baserte webapplikasjoner. Disse har funksjonalitet tilrettelagt for å utvikle både på klientsiden og tjenersiden, noe som gjør de til sterke kandidater til å støtte forholdsreglene i kapittel 4. Det blir derfor sett nærmere på tre slike rammeverk, og hva de har å tilby.

- Direct Web Remoting 2.0 (DWR) [4]
- Ruby on Rails [22]
- Microsoft ASP.NET med AJAX, tidligere kjent som Microsoft Atlas [23]

5.2 Direct Web Remoting (DWR)

DWR er et *open source* rammeverk som tillater JavaScript i nettleseren å kalle java-metoder på tjenersiden. Dette oppnås ved at JavaScript genereres dynamisk basert på Java-klassene på tjenersiden. Slik funksjonalitet kan minne om *Remote procedure call* (RPC)[2], uten behov for bruk av tredjepart programvare. Figur 5.1 viser hvordan en nettleser kommuniserer mot en Java-klasse på tjenersiden ved hjelp av DWR. [4]



Figur 5.1: Klientside kommuniserer mot tjenerside ved hjelp av rammeverket DWR [4]

DWR versjon 2.0 beta, har funksjonalitet for revers AJAX-kall. Det vil si mulighet for tjenersiden å kalle funksjoner definert i JavaScript på klientsiden. Dette oppnås ved hjelp

[2] RPC tillater å kalle funksjoner og prosedyrer på et annet system over et nettverk

av enten *Comet* (holde XHR-objektets HTTP-tilkobling oppe over tid), *Polling* (JavaScript på klienten etterspør data kontinuerlig) eller *Piggyback* (forespørselen skjer samtidig når tjenersiden blir etterspurt fra klienten). Dette er funksjonalitet som kan være aktuelle for å lage interaktive progresslinjer, eller andre dynamiske effekter generert fra tjenersiden. [4]

5.2.1 Begrenset funksjonalitet

DWR begrenser seg til funksjonalitet mellom nettleserens JavaScript og Java-klasser på tjenersiden. Det støttes ikke funksjonalitet for blant annet databasehåndtering. Derfor må DWR benyttes i kombinasjon med andre rammeverk for å håndtere dette. Hibernate[3] er et eksempel på et slikt rammeverk. Dette rammeverket gjør det mulig å å koble javaobjekter mot databasetabeller.

5.2.2 Sikkerhet

DWR inneholder blant annet klassen *org.directwebremoting.Security*, som tilbyr forskjellige sikkerhetsmetoder. Disse metodene er ganske enkle, og det oppfordres å bruke dem i kombinasjon med andre sikkerhetsmekanismer. Siden DWR kun begrenser seg til JavaScript og DWR-klasser, støttes det ikke funksjonalitet for filtrering av data beregnet til databaser og lignende. Det uttrykkes følgende i JavaDoc[4] for DWR [4]:

"This class represents some simple filters which may protect from simple attacks in low risk environments. There is no replacement for a full security review which assesses the risks that you face."

Det hevdes at DWR har mekanismer for:

- Filtrering av HTML-kontrolltegn
- XSRF-beskyttelse

5.3 Ruby on Rails (RoR)

RoR er et rammeverk som har til hensikt å gjøre det enklere å utvikle, benytte og vedlikeholde webapplikasjoner. RoR bygger på *Model-View-Controller* (MVC)[5] for å strukturere webapplikasjoner mer ryddig. Ved å ta i bruk programmeringsspråket Ruby, tillates det å utvikle webapplikasjoner som er enkle å skrive og lese i ettertid. RoR bygger på to konsepter:

- *Dont repeat yourself* (DRY)
- Konvensjon fremfor konfigurasjon

DRY baserer seg på at all kunnskap i et system skal defineres kun et sted. Konvensjon fremfor konfigurasjon betyr at Rails har ferdiggenererte standarder for å koble sammen en

[3] Hibernate er tilgjengelig på `http://www.hibernate.org`
[4] JavaDoc benyttes for å dokumentere Java-klasser
[5] MVC er et arkitekturmønser som ofte benyttet for å lagdele og separere funksjonalitet i en applikasjon

webapplikasjon. Ved å følge disse konvensjonene, kan det utvikles webapplikasjoner med langt mindre kildekode. [22, Kapittel 1]

RoR støtter utstrakt bruk av AJAX ved hjelp av JavaScript-biblioteket Prototype [19], som har ferdiglagde komponenter for asynkron tjenersidekommunikasjon og visuelle effekter. Modulen *JavascriptHelper*, gjør det mulig for utviklere å skrive all kildekode (både tjenerside og klientside) i Ruby uten å måtte arbeide med langt mer avansert JavaScript. [22, Kapittel 18] og [24, Kapittel 3,4]

RoR støtter også funksjonalitet, kalt *Partials*, som lar utvikleren plassere deler av visningslogikken i separate filer. Dette gjør det svært enkelt hvis samme visningslogikk benyttes flere steder i grensesnittet [24, Kapittel 2]

5.3.1 Sikkerhet i Ruby on Rails

RoR støtter en rekke sikkerhetsmekanismer for å filtrere og validere på tjenerside. Blant annet skal det være støtte for:

- Validering av input
- Filtrering av HTML-kontrolltegn
- Filtrering av SQL-kontrolltegn

Ved at RoR tillater utviklere å skrive all kildekode i Ruby, elimineres behovet for å ta hensyn til å begrense JavaScript som sendes til klienten slik det ble nevnt i delkapittel 4.3.1. Derfor er denne forholdsregelen automatisk støttet.

5.4 Microsoft ASP.NET med AJAX

Microsoft ASP.NET AJAX er en utvidelse til Microsofts .NET-rammeverket, som har til hensikt å tillate rask utvikling av AJAX-baserte webapplikasjoner. Dette oppnås ved å benytte klientside JavaScript-bibliotek som er nettleseruavhengig, og integrere dette med ASP.NET 2.0 baserte tjenersideplattformer. [23]

AJAX-utvidelsen til ASP.NET har lenge gått under kodenavnet Atlas. Denne består av to hoveddeler:

- ASP.NET AJAX-tjenersidekontroll som tillater utviklere å skrive tradisjonell ASP.NET-kode for å unngå kompleksiteten JavaScript påfører utviklere
- Microsoft AJAX-bibliotek som er et JavaScript-bibliotek for å forenkler bruken av visuelle effekter og kommunikasjon mot tjenerside

AJAX-biblioteket kan benyttes plattformuavhengig, og kan lastes ned som en *stand alone* pakke. Tjenersidekontrollen er tett knyttet mot ASP.NET, og benytter AJAX-biblioteket for klientsideprogrammering og kontroll. Utviklere står fritt til å velge om de ønsker å kombinere disse to delene.

5.4.1 Sikkerhet i ASP.NET AJAX

ASP.NET tilbyr validatorer som hevder å validere både på tjenerside og klientside. Blant annet skal det være støtte for:

- Validering av input for å sjekke at data har riktig form
- Filtrering av data sendt til brukerens nettleser
- Innebygd forhindring av å sende kontrolltegn gjennom ASP.NET *data access*, for å forhindre kontrolltegn sendt til SQL-databasen som benyttes

På samme måte som ved RoR, vil rammeverket ASP.NET generere JavaScript selv. Derfor elimineres behovet for å begrense hva som sendes av JavaScript til klienten slik det ble nevnt i delkapittel 4.3.1.

5.5 Oppsummering

Dette kapittelet har presentert rammeverkene Direct Web Remoting, Ruby on Rails og Microsoft ASP.NET, som alle har til hensikt å forenkle arbeidet med å utvikle AJAX-baserte webapplikasjoner, både med tanke på sikkerhet og effektivitet. Rammeverkene blir brukt i neste kapittel 6, hvor det blir utviklet en webapplikasjon i hvert av disse. Denne webapplikasjonen danner grunnlaget for testing av hvilke forholdsregler rammeverkene støtter og hvordan disse støttes.

KAPITTEL 6
AJAX-WEBAPPLIKASJON

Denne oppgaven omhandler fallgruver og angrep mot AJAX-baserte webapplikasjoner og hvordan rammeverkene støtter forholdsregler mot disse. Det ble i kapittel 5 sett på tre rammeverk som hver bygger på tre forskjellige programmeringsspråk. For å undersøke hvordan rammeverkene fungerer og hvordan disse håndterer forholdsregler nevnt i kapittel 4, ble det valgt å utvikle en AJAX-basert webapplikasjon.

Dette kapittelet beskriver valg av webapplikasjon, og hvordan de forskjellige versjonene i DWR, RoR og ASP.NET ble utviklet. Det blir i de neste kapitlene 7 og 8, sett hvilke resultat de forskjellige sikkerhetsmekanismene i rammeverkene gir, hvilke forholdsregler de støtter og hvorvidt disse må aktiveres manuelt.

6.1 Diskusjonsforum som webapplikasjon

Hensikten med webapplikasjonen var å teste flest mulig sikkerhetsmekanismer i rammeverkene nevnt i kapittel 5. Det ble valgt å utvikle et diskusjonsforum basert på AJAX. Et diskusjonsforum tillater brukere å legge inn tekst som blir presentert for andre brukere, og vil derfor være potensielt for flere av fallgruvene i kapittel 3.

6.1.1 Use-case Bruker

Det ble valgt å holde kompleksiteten til webapplikasjonen nede for å unngå at fokus ble flyttet fra å teste sikkerhetsmekanismer, til å utvikle et avansert diskusjonsforum. Funksjonaliteten begrenser seg derfor til å logge inn/ut, navigere i forumtråder og besvare disse. *Use-case*[1] for en bruker i diskusjonsforumet kan sees i tabell 6.1

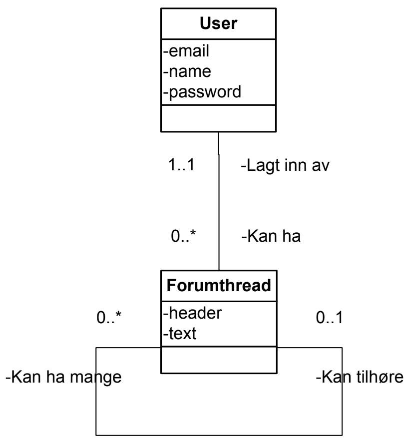
[1] Use-case er en teknikk for å fange de funksjonelle krav i et system

Besvare innlegg:
Aktør: Bruker Forutsetninger: Ingen
Hendelsesforløp:
1. Logg inn 2. Navigere i forumtreet 3. Besvare et innlegg 4. Logg ut

Tabell 6.1: Use-case for bruker i diskusjonsforumet

6.1.2 Databasemodell

Et diskusjonsforum må i tillegg til å lagre informasjon om brukere, lagre foruminnlegg. Det er flere måter å gjøre dette på, men det ble valgt en enkel måte for å holde kompleksiteten nede. Det ble derfor bestemt at databasen skal kun inneholde to tabeller, *User* og *ForumThread*. *User* representerer informasjon relatert til en bruker. *ForumThread* representerer et innlegg i forumet, og baserer seg på et forelder/barn-prinsipp. En *ForumThread* kan ha ingen eller flere barne-*ForumThread*. En *User* kan ha ingen eller mange *ForumThread*. Figur 6.1 viser databasemodell.



Figur 6.1: Databasemodell diskusjonsforum

Navnene til tabellene avviker i Ruby on Rails implementeringen, hvor disse heter *Users* og *Forumthreads*. DWR-versjonen og RoR-versjonen benytter MySql-databasetjener, i motsetning til .NET-versjonen hvor Microsoft SQL benyttes.

6.1.3 AJAX-basert diskusjonsforum

Det er viktig å belyse behovet for å benytte AJAX i en webapplikasjon, og hva dette bidrar til. AJAX i diskusjonsforumet vil muliggjøre navigering i forumtråder, og besvare disse uten å

måtte forfriske den aktuelle websiden. Dette fordi diskusjonsforumet baserer seg på asynkron kommunikasjon mellom klient og tjener. JavaScript vil bygge og endre innholdet i websiden ved å manipulere DOM-treet nevnt i delkapittel 2.2.1 avhengig av brukervalg. Hvis en bruker klikker på et innlegg, vil dette dynamisk presenteres med eventuelle barneelementer og innhold i det aktuelle innlegget. Et svarfelt bygges også dynamisk for å svare på dette. Uavhengig av hvilket rammeverk som benyttes, vil XHR-objektet nevnt i delkapittel 2.2.1 ligge i bunn og stå for den asynkrone kommunikasjonen.

Design

Det ble skrevet en CSS-fil nevnt i delkapittel 2.2.1, som definerer en enkel visuell stil for forumet. Deretter ble det skrevet en standard HTML-fil som benytter elementene i CSS-filen. Grensesnittet for diskusjonsforumet kan sees i vedlegg A, hvor fargen blå er valgt for DWR-versjonen, rød for RoR-versjonen og gul for ASP.NET-versjonen.

6.2 Diskusjonsforum med DWR og Java

Implementeringen i DWR og Java er delt opp i følgende komponenter:

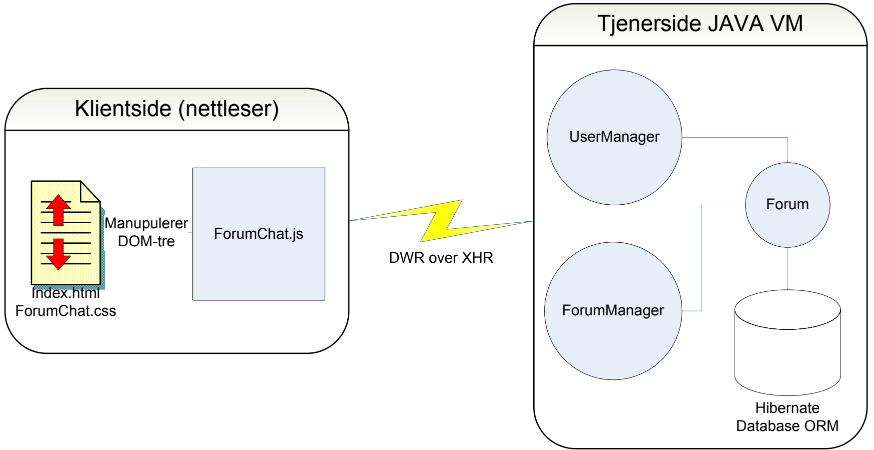
- HTML, JavaScript og CSS
- DWR-transparente javaobjektene ForumManager.java og UserManager.java
- Forum.java som kommuniserer mot databasen ved hjelp av rammeverket Hibernate, nevnt i delkapittel 5.2.1

ForumManager.java og *UserManager.java* inneholder metoder som kalles fra JavaScript på klientsiden. Metodene i disse klassene kaller funksjoner i JavaScript på klientsiden ved hjelp av reverse AJAX-kall. JavaScriptet er definert i filen *ForumChat.js* som befinner seg på klientsiden. For å tilgjengeliggjøre objektene av *ForumManager.java* og *UserManager.java* i JavaScript på klientside, måtte disse defineres i en XML-fil som vises i liste 6.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE dwr PUBLIC
3      "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
4      "http://www.getahead.ltd.uk/dwr/dwr20.dtd">
5  <dwr>
6    <allow>
7      <create creator="new" javascript="ForumManager" scope="application">
8        <param name="class" value="org.web2security.forum.controller.ForumManager"/>
9      </create>
10     <create creator="new" javascript="UserManager" scope="application">
11       <param name="class" value="org.web2security.forum.controller.UserManager"/>
12     </create>
13   </allow>
14 </dwr>
```

Liste 6.1: *dwr.xml* definerer hvilke klasser som eksponeres for JavaScript i DWR-versjonen av diskusjonsforumet

Ved hjelp av DWR, benyttes JavaScript til å manipulere DOM-treet definert av HTML-dokumentet. HTML-dokumentet baserer utseendet på de forskjellige komponentene definert i den tilhørende CSS-filen. Dette danner grunnlag for å oppnå den dynamiske effekten som ble beskrevet i delkapittel 6.1.3. *ForumManager* og *UserManager* benytter et objekt av klassen *Forum.java* for å formidle databaserelaterte oppgaver. Dette gjøres ved å benytte rammeverket Hibernate. Figur 6.2 viser en modell av de forskjellige komponentene som bygger diskusjonsforumet.



Figur 6.2: Modell av hvordan DWR benyttes i diskusjonsforumet

6.3 Diskusjonsforum med Ruby on Rails

RoR-versjonen av diskusjonsforumet skiller seg totalt ut fra utgaven som bygger på DWR og Java. Hovedsaklig på grunn av at RoR bygger på Ruby som er et annet programmeringsspråk, men også fordi RoR baserer seg på konvensjon fremfor konfigurasjon. Det ble brukt litt tid på å sette seg inn i RoR ved å se på forskjellige eksempler, og ved å lese boka *Agile Web Development with Rails* [22]. Deretter ble det utviklet en versjon av diskusjonsforumet som ikke benytter AJAX, før AJAX ble tilført. Hovedsaklig er diskusjonsforumet i RoR delt opp i tre deler:

- *View* inneholder *Ruby-generated JavaScript* (RJS), *Ruby-generated HTML* (Rhtml) og CSS
- *Controller* tilbyr funksjoner som *View* benytter seg av
- *Model Users.rb* og *ForumThreads.rb* som gjenspeiler databasetabellene

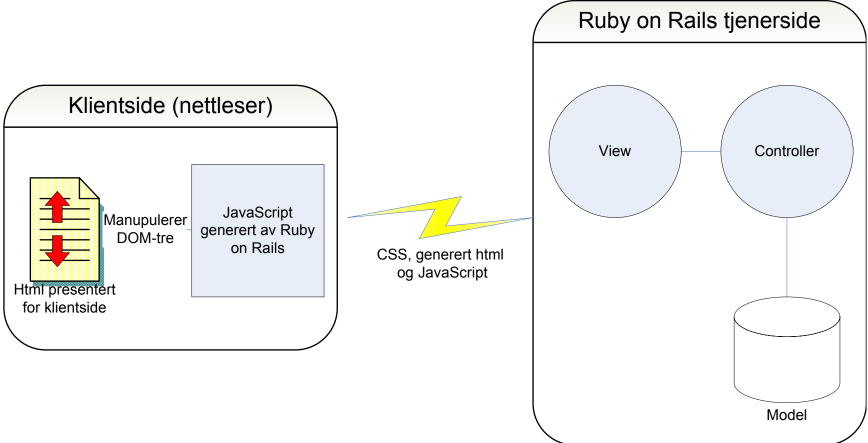
View består av rhtml-filer som kan minne om JSP og ASP-filer, men har en langt høyere fleksibilitet. *View* består i tillegg av RJS-filer med Ruby-kode som genererer JavaScript. Utseendet til *view* er definert av den samme CSS-filen som ble benyttet i DWR-versjonen. *Controller* bistår *view* med en rekke funksjoner, en for hver *view*-fil. Diskusjonsforumet benytter RJS-filer til å hente informasjon fra *controller* dynamisk. Deretter benyttes *Partials* nevnt i delkapittel 5.3 direkte fra RJS-filene til å manipulere DOM-treet, for å oppnå den dynamiske

effekten beskrevet i delkapittel 6.1.3. Liste 6.2 viser hvordan *partials* benyttes for å endre innhold på websiden:

```
1  page[:parent].replace_html :partial => 'forumthread', :object => @forumthread_parent
2  page[:current].replace_html :partial => 'forumthread', :object => @forumthread
3  page[:child].replace_html :partial => 'forumthread', :collection => @forumthread_childs
4  page[:answerArea].replace_html :partial => 'forumthread_form', :object => @forumthread
5  page[:threadArea].replace_html :partial => 'forumthread_show', :object => @forumthread
```

Liste 6.2: Rjs benytter partials til å dynamisk endre innholdet i RoR-versjonen av diskusjonsforumet

På linje 1 endrer RoR DOM-element med id=parent og setter innholdet i denne basert på en *partial* med navn *forumthread* som lager presentasjon av objektet med navn *forumthread_parent*. Linje 3 gjør akkurat det samme, men her benyttes en *collection* med flere *forumthreads*. Figur 6.3 viser hvordan komponentene i diskusjonsforumet henger sammen.



Figur 6.3: Modell av hvordan Ruby on Rails benyttes i diskusjonsforumet

6.4 Diskusjonsforum med ASP.NET

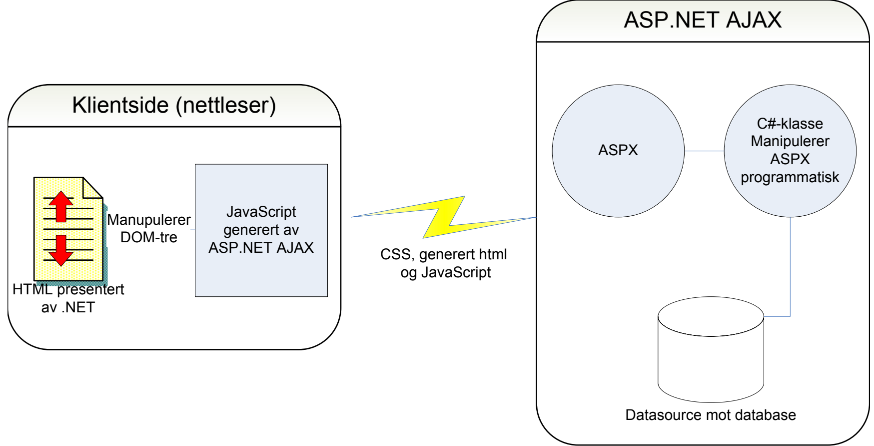
På samme måte som Ruby on Rails versjonen, skiller også .NET versjonen seg totalt fra sine søsken. For å lære mer om .NET, ble det sett på forskjellige eksempler tilgjengelig på ASP.NET sidene til Microsoft [23].

Det ble valgt å benytte Microsofts‘s AJAX-tjenersidekontroll fra delkapittel 5.4 for å få klienten til å sende asynkrone kall mot tjenersiden. Diskusjonsforumet omfatter tre deler.

- ASPX-fil for visning og presentasjon med tilhørende CSS-fil
- C# (*Code behind*) for å unngå å blande logikk og presentasjon
- *Datasource* som definerer spørringer mot databasen

Det ble benyttet en ASPX-fil *index.aspx* som inneholder alle div-elementer stylet av den tilhørende CSS-filen. For å skru på AJAX-funksjonaliteten, ble det lagt til et ASP-element kalt *ScriptManager*. Deretter ble alt innhold som skulle bli dynamisk oppdatert lagt inn i en *UpdatePanel*. For å få til den dynamiske navigeringen i innleggene i forumet, ble det benyttet en C#-klasse for å dynamisk manipulere ASPX-filen. Denne får tilgang til alle ASP-komponenter i *index.aspx* ved å bytte ut div-elementer med ASP-paneler. Disse blir oversatt til div-elementer når HTML genereres og sendes til klienten. Funksjonene i C#-klassen henter informasjon fra *DataSource*-objektet, hvor det på forhånd er definert spørringer for blant annet å hente ut brukere, foruminnlegg og lagre disse.

Figur 6.4 viser hvordan komponentene henger sammen i ASP-versjonen av diskusjonsforumet.



Figur 6.4: Modell av hvordan ASP.NET AJAX benyttes i diskusjonsforumet

6.5 Oppsummering

Dette kapittelet har beskrevet hvordan et AJAX-basert diskusjonsforum er utviklet i rammeverkene DWR, RoR og ASP.NET. Diskusjonsforumet danner grunnlaget for å teste sikkerhetsmekanismer til disse rammeverkene i neste kapittel 7.

KAPITTEL 7
TESTING

Dette kapittelet viser systematisk hvordan testingen av de tre webapplikasjonene ble gjort, ved å utføre forskjellige angrep som mekanismene har til hensikt å forhindre. Målet med testingen er å se hvilke forholdsregler fra kapittel 4, de forskjellige mekanismene støtter, og hvordan disse støttes. Ved å teste disse mekanismene i de AJAX-baserte webapplikasjonene som er utviklet, kan det sees hvilke begrensninger og mangler disse har.

Det blir i kapittel 8 vurdert de forskjellige rammeverkene, og deres håndtering av forholdsreglene. Her nevnes det også hvordan det bør tas hensyn til de som ikke støttes.

7.1 Sikkerhetsmekanismene i DWR

Sikkerhetsmekanismene nevnt i delkapittel 5.2 baserer seg på klassen *Security* som inneholder følgende metoder:

- escapeHtml
- replaceXmlCharacters
- containsXssRiskyCharacters

DWR-hevder i tillegg å støtte XSRF-beskyttelse. Alle disse mekanismene ble testet ut, for deretter å bli vurdert i delkapittel 8.1.

7.1.1 Test av HTML-kontrolltegnfiltrering i DWR

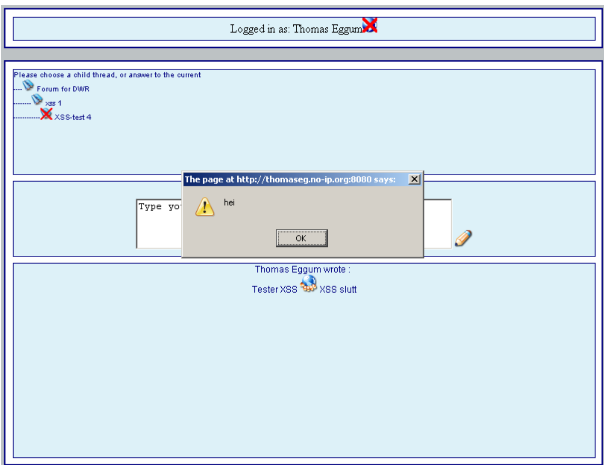
For å teste hvorvidt DWR filtrerer HTML-kontrolltegn automatisk, ble det i DWR-versjonen av diskusjonsforumet forsøkt å poste et innlegg for å fremtvinge et XSS-angrep nevnt i delkapittel 3.3.1. Dette ble gjort ved å skrive et innlegg med følgende JavaScript-innhold:

```
<script type='text/javascript'>alert('hei');</script>
```

Når det ble gjort forsøk på å se det aktuelle innlegget, var resultatet at JavaScriptet ble sendt til nettleseren, men ble ikke automatisk kjørt. Det ble derfor forsøkt å lagre et alternativt innlegg med følgende innhold:

```
Tester XSS
<img src="image.png" onclick="alert('hei')"/>
XSS slutt
```

Resultatet av det aktuelle innlegget vises i figur 7.1.



Figur 7.1: Resultat av vellykket injisering av HTML-kontrolltegn i DWR-versjonen av diskusjonsforumet

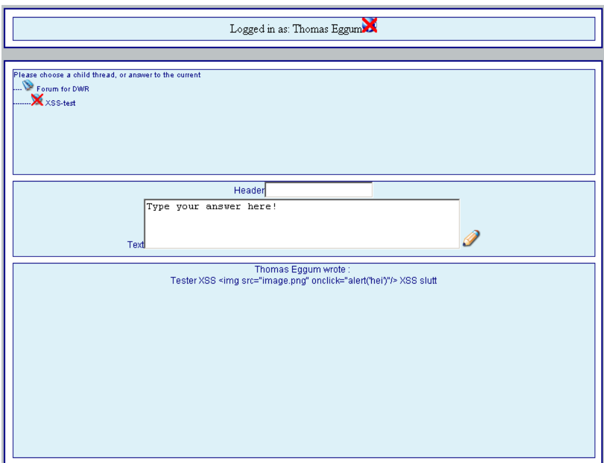
Etter å ha verifisert at det var mulig å injisere JavaScript i diskusjonsforumet, ble det gjort forsøk på å aktivere de forskjellige metodene i klassen *Security* før data ble sendt fra tjenerside til klient. Deretter ble det samme innlegget forsøkt vist på nytt, for å se hvordan HTML-kontrolltegn ble håndtert ved bruk av hver av dem.

Test av metoden escapeHtml

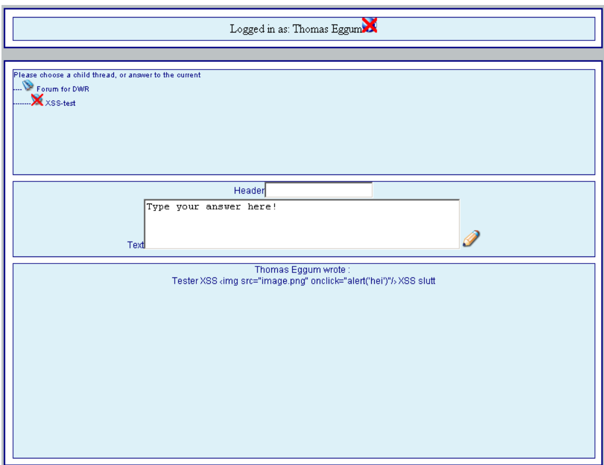
Ved bruk av *escapeHtml*, ble alle HTML-kontrolltegn endret slik at disse mistet mening. Dette resulterte i at tegnene ble behandlet som vanlig tekst slik figur 7.2 viser.

Test av metoden replaceXmlCharacters

Etter å ha sett resultatet av metoden *escapeHtml*, ble denne byttet ut med metoden *replaceXml-Characters* før data ble presentert for nettleseren. Resultatet ved bruk av denne metoden, var at alle kontrolltegn som ble lagret i innlegget ble byttet til lignende “ufarlige” tegn slik figur 7.3 viser.



Figur 7.2: Resultat ved bruk av metoden *escapeHtml* i DWR



Figur 7.3: Resultat ved bruk av metoden *replaceXmlCharacters*

Test av metoden containsXssRiskyCharacters i DWR

Denne metoden virket også aktuell å bruke for å unngå HTML-kontrolltegn som kan resultere i XSS-angrep. Det hevdes i JavaDoc til klassen, at denne metoden skal returnere *true* hvis dataene inneholder et av HTML-kontrolltegnene:

`&, <, >, ' eller ''`

Metoden ble aktivert slik liste 7.1 viser.

```
1  if(containsXssRiskyCharacters(data)){
2          //Dataene inneholder kontrolltegn
3  }else{
4          //Dataene inneholder ikke kontrolltegn
5  }
```

Liste 7.1: Forsøk på bruk av metoden *containsXssRiskyCharacters*

Deretter ble det gjort forsøk på å se det aktuelle innlegget, for å kontrollere at de injiserte HTML-kontrolltegnene ville bli fanget opp.

Det viste seg at denne metoden feilet, og hindret derfor ikke JavaScriptet i å bli sendt ut til nettleseren. Det ble gjort forsøk på å lagre forskjellige varianter av innlegg i forumet, som viste at metoden kun ga resultat hvis et innlegg inneholdt alle de nevnte HTML-kontrolltegnene. Dette strider i mot funksjonaliteten som ble påstått. Denne feilen vurderes nærmere i delkapittel 8.1.1.

7.1.2 Test av XSRF-beskyttelse i DWR

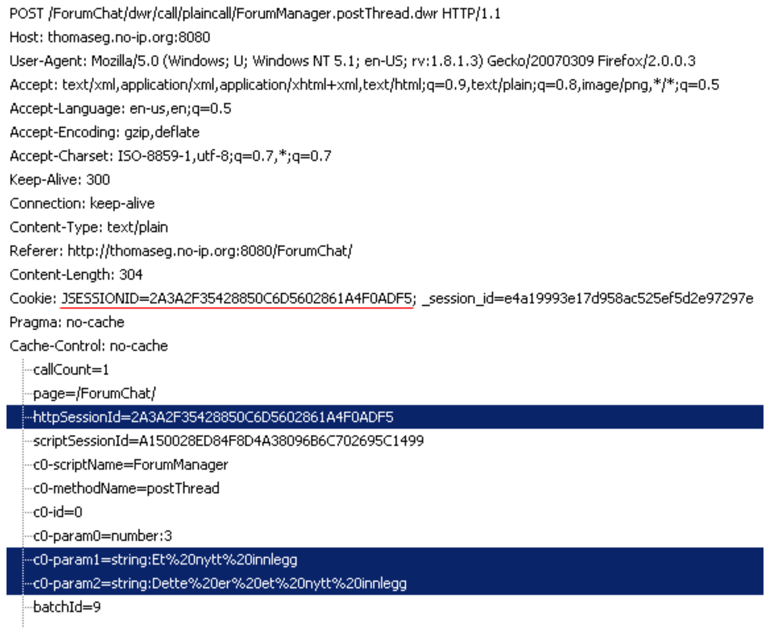
Slik det ble nevnt i delkapittel 5.2.2, hevdes det at DWR har innebygd og automatisk XSRF-beskyttelse. Denne mekanismen var det interessant å se nærmere på. Derfor ble utviklerne av rammeverket DWR kontaktet, og det ble diskutert hvordan forskjellige problemstillinger med XSRF kan oppstå i webapplikasjoner. Det ble i delkapittel 4.4 beskrevet en forholdsregel mot XSRF. Alternativt til å benytte en ny og unik nøkkel som en skjult parameter for hver forespørsel, velger utviklerne av DWR å bruke *JSESSIONID* til dette. Denne verdien er en del av den benyttede cookie, og står for tilstandshåndteringen. Denne er kun tilgjengelig fra JavaScript i den aktuelle websiden brukeren står på. Sikkerheten her baseres på at JavaScript ikke har tilgang til *JSESSIONID* fra andre domener enn der det har opphav. En angriper kan derfor ikke lage en webside *badguy.com* som plukker ut *JSESSIONID* fra *goodguy.com*, og benytter denne som parameter for å forfalske en forespørsel mot tjenersiden når brukeren besøker *badguy.com*. Hvis en angriper får tak i den aktuelle *JSESSIONID*, har man i tillegg et XSS-problem nevnt i delkapittel 3.3.1 hvor angriper selv kan lage forespørslene istedet for å få en bruker til å utføre disse.

For å teste ut dette, ble det sett nærmere på innholdet i POST-forespørselen som blir sendt fra brukerens nettleser til tjenersiden. Til dette ble det benyttet et tillegg til nettleseren Mozilla Firefox kalt *Live HTTP headers*. Dette tillegget viser innholdet i forespørsel og respons som blir sendt til og fra tjenersiden. Innholdet i en POST-forespørsel for å lagre et nytt innlegg i DWR-versjonen av diskusjonsforumet kan sees i figur 7.4.

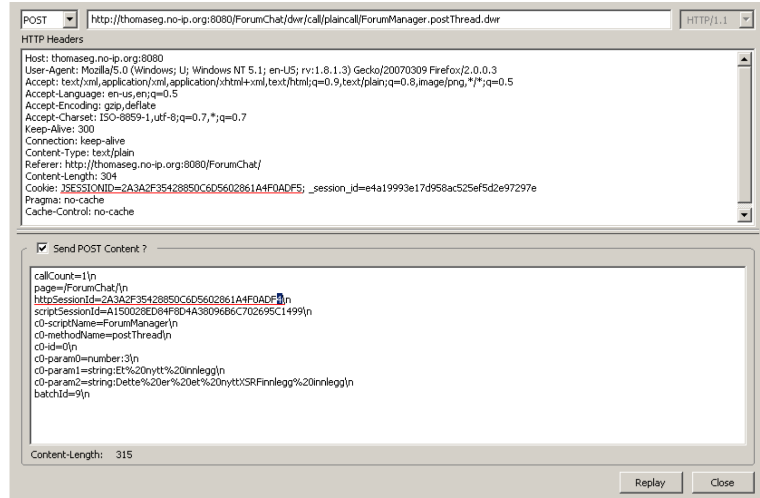
Linjene merket i figur 7.4, viser hvordan *JSESSIONID* sendes som en parameter med navn *httpSessionID* og at overskrift og tekstinnholdet sendes som parametere i forespørselen. Det observeres også at *JSESSIONID* som er første del i *Cookie:*, er helt identisk med parameteren *httpSessionId*. Dette har til hensikt å oppnå XSRF-beskyttelse, ved at disse sammenlignes på tjenerside.

Etter å ha sett innholdet i denne POST-forespørselen, ble det forsøkt å forfalske en forespørsel ved å bruke *Replay* funksjonen i *Live HTTP headers*. Denne gjør det mulig å kopiere en forespørsel, modifisere denne, for deretter å kjøre en ny. Det ble gjort et forsøk på å endre det siste tegnet i *httpSessionID* slik figur 7.5 viser. Resultatet av denne forfalskede forespørselen ble forkastet på tjenersiden.

Slik det ble sett i delkapittel 3.7 om XSRF, lager angriper en link eller en webside som får brukeren til å lage denne forfalskede forespørsel fra sin nettleser. Alternativt til å lage en slik webside, har vi i dette eksempelet benyttet verktøyet *Live HTTP headers* til å forfalske en forespørsel, og som det ble sett forkaster DWR forespørsler på tjenersiden hvis parameter *httpSessionId* ikke er satt til riktig verdi.



Figur 7.4: Innholdet i en POST-forespørsel for å lagre et innlegg i DWR-versjonen av diskusjonsforumet



Figur 7.5: Forfalsket forespørsel i DWR-versjonen av diskusjonsforumet

7.2 Sikkerhetsmekanismene i RoR

Det ble nevnt i delkapittel 5.3 at RoR støtter forholdsregler for å validere input i tillegg til å tjenersidefiltrere både HTML og SQL-kontrolltegn. Funksjonen *validates_format_of* benyttes for å validere input mot definerte lovlige tegn, *h()* benyttes for filtrering av HTML-kontrolltegn, og *quote()* for filtrering av SQL-kontrolltegn.

7.2.1 Test av inputvalidering i RoR

Inputvalidering har til hensikt å påse at data har riktig format når de ankommer webapplikasjonen slik det ble nevnt i delkapittel 4.1.1. Det ble forsøkt å aktivere validering av innhold i nye foruminnlegg ved å begrense lovlige tegn til mellomrom, store og små bokstaver, og tall. Dette ble gjort ved å benytte koden i liste 7.2 i *model*-filen *forumthread.rb*.

```
1  validates_format_of :text,
2  :with => %r{^[-a-z A-Z0-9]+$},
3  :message => "Contains unacceptable characters"
```

Liste 7.2: Tjenersidevalidering av input i RoR

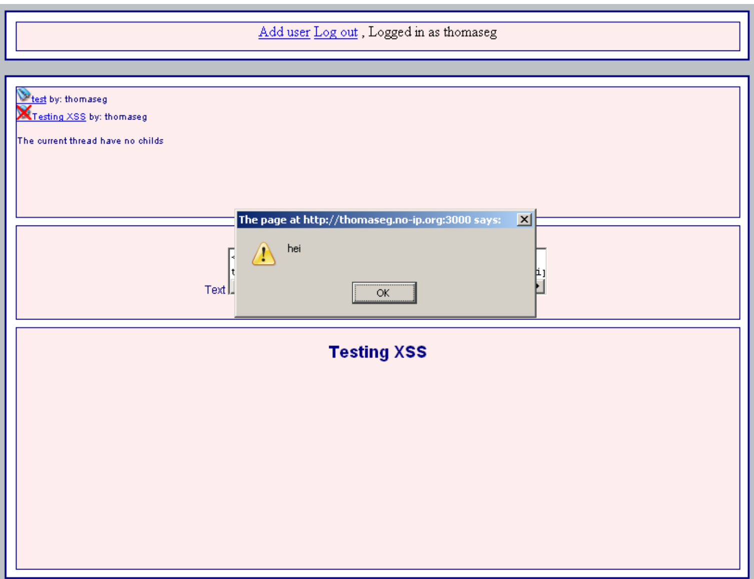
Dette forhindret å lagre innlegg i diskusjonsforumet med andre tegn enn de som var spesifisert.

7.2.2 Test av HTML-kontrolltegnfiltrering i RoR

Det ble på samme måte som ved DWR, gjort forsøk på å lagre et innlegg i diskusjonsforumet med følgende JavaScript-innhold:

```
<script type='text/javascript'>alert('hei');</script>
```

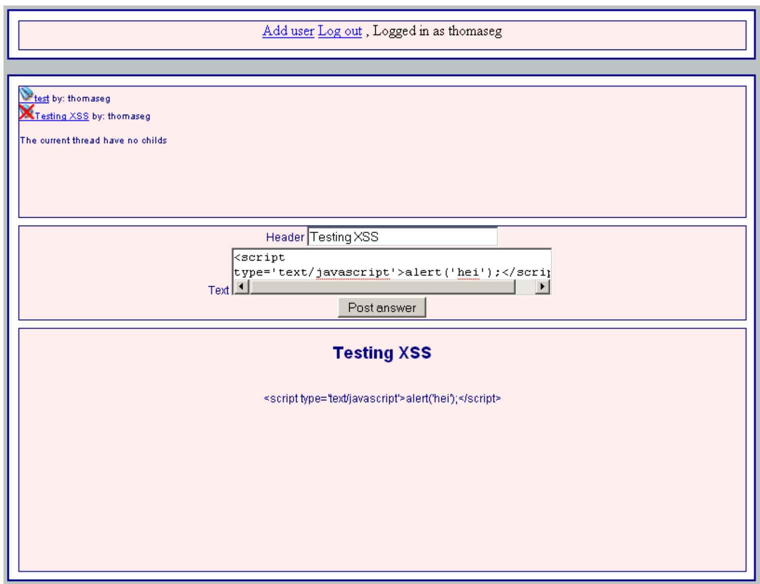
Dette har til hensikt å presentere den alarmerende meldingen "hei" til alle som ser det aktuelle innlegget. Etter å ha lagret innlegget, ble det forsøkt vist. Resultatet av dette sees i figur 7.6 som viser at JavaScriptet ble sendt ut til nettleseren, og at RoR-versjonen av diskusjonsforumet var XSS-utsatt.



Figur 7.6: Vellykket injisering av HTML-kontrolltegn i RoR-versjonen av diskusjonsforumet

Etter å ha verifisert fallgruven i diskusjonsforumet, ble det gjort forsøk på å benytte funksjonen *h()*, før data sendes til nettleseren. Denne funksjonen resulterte i at alle HTML-kontrolltegn

mistet mening. Dette medførte at JavaScriptet ikke ble injisert, men ble presentert som vanlig tekst slik figur 7.7 viser.



Figur 7.7: funksjonen h(), håndterer HTML-kontrolltegn

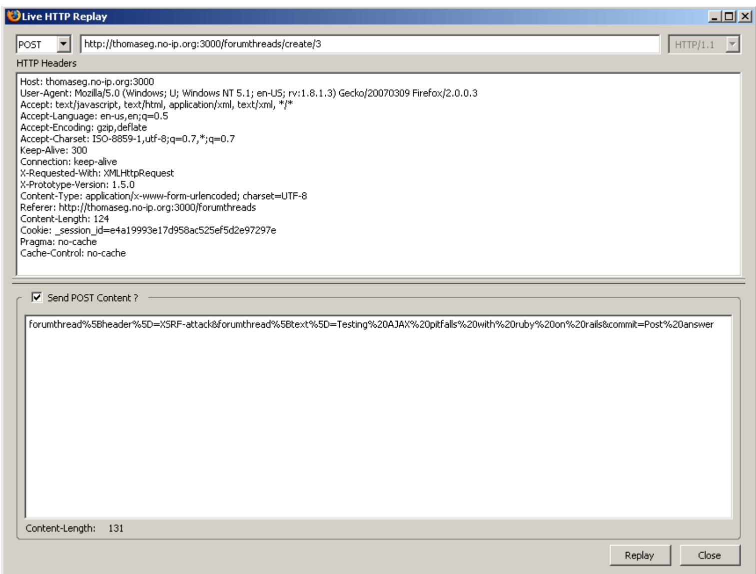
7.2.3 Test av SQL-kontrolltegnfiltrering i RoR

Det ble i diskusjonsforumet benyttet predefinerte funksjoner som *save()* og *find()*. Disse fjerner SQL-kontrolltegn og eliminerer behovet for bruk av *quote()* funksjonen. Hvis vi imidlertid hadde brukt egendefinerte SQL-spørringer, ville det vært viktig å benytte denne funksjonen. Det ble gjort forsøk på SQL-injisering ved å sende inn brukernavn "thomaseg' - - " , slik det ble vist i eksemplet i delkapittel 3.3.4 med hensikt å logge inn uten et passord. Dette resulterte i en mislykket innlogging.

7.2.4 Test av XSRF-angrep i RoR

Det ble testet å utføre et XSRF-angrep på diskusjonsforumet ved å "dikte opp" en POST-forespørsel på bakgrunn av en tidligere utført POST-forespørsel. Det ble benyttet samme metodikk som ved DWR-rammeverket i delkapittel 7.1.2 med tillegget *Live HTTP Header* for nettleseren Mozilla Firefox. Figur 7.8 viser hvordan en forespørsel ble forfalsket.

Den forfalskede forespørselen resulterte i at et innlegg med overskrift "XSRF-attack" ble lagret. Dette viser at RoR-versjonen av diskusjonsforumet var utsatt for XSRF-angrep.



Figur 7.8: Forfalsket forespørsel i RoR-versjonen av diskusjonsforumet

7.3 Sikkerhetsmekanismene i ASP.NET AJAX

Testing av sikkerhetsmekanismer i ASP.NET kunne ikke utføres før en feil var rettet. Ved å skrive ugyldige tegn i tekstfelter, ble det produsert en JavaScript feilmelding i klienten. Det viste seg at dette var et kjent problem, og kunne unngås ved å benytte *ValidateRequest='false'* i sidedeklareringen for ASPX-filen.

7.3.1 Test av inputvalidering i ASP.NET

For å legge føringer på hva slags innhold de forskjellige feltene i diskusjonsforumet kunne inneholde, måtte det manuelt aktiveres mekanismer som tar seg av dette. Valideringsmekanismen i ASP.NET ble derfor benyttet for å oppnå klientsidevalidering i kombinasjon med tjenersidevalidering. Klientsidevalideringen vil spare tjenersiden for unødvendig arbeide hvis dataene ikke har riktig form. Tjenersidevalideringen fungerer som en sikkerhetsbarriere for å unngå at data på uønsket format slipper gjennom til webapplikasjonen.

For å teste denne funksjonaliteten, ble det lagt til et asp-element av typen *RegularExpressionValidator* slik det vises i liste 7.3. Denne har til hensikt å forby alt annet enn bokstaver, tall og oppholdsrom.

```
1 <asp:RegularExpressionValidator ID="validTxtText"
2     ControlToValidate="txtText"
3     runat="server"
4     ValidationExpression="^[a-z A-Z0-9]*$"
5     ErrorMessage="You may only use alphanumeric characters" />
```

Liste 7.3: Tjenersidevalidering i kombinasjon med klientsidevalidering i ASP.NET

For å aktivere tjenersidevalideringen, var det i tillegg nødvendig å legge til en test på hvorvidt websiden validerer eller ikke, før det gjennomføres lagring av foruminnlegg. Dette ble gjort slik liste 7.4 viser. Ved å utelate dette, ville kun klientsidevalideringen vært aktivert.

```
1 Page.Validate();
2 if (Page.IsValid)
3 {
4         //Websiden validerte, kjør videre
5 }
6 else
7 {
8         //Websiden validerte ikke, avvis forespørsel
9 }
```

Liste 7.4: Nødvendig test i C# koden på tjenerside for å aktivere tjenersidevalidering i ASP.NET

Etter å aktivere mekanismen for validering av input to steder, ble det forsøkt å skrive inn ulovlige tegn i diskusjonsforumet. Klientsidevalideringen ga riktig resultat, men siden denne ikke bidrar til annet enn å avlaste tjeneren med unødvendige forespørsler slik det ble nevnt i delkapittel 3.5.1, var det ønskelig å slå av denne. Til dette ble tillegget *Firebug* for nettleseren Mozilla Firefox benyttet. Dette tillegget tilater en bruker å redigere DOM-treet for en webside. Submit-knappen for å lagre et innlegg, inneholdt koden vist i liste 7.5. For å slå av klientsidevalideringen ble argumentene for *WebForm_PostBackOptions* endret til:

```
WebForm_PostBackOptions("imgSaveText", "", false, "", "", false, false)
```

Det ble deretter gjort et nytt forsøk på å poste ulovlige tegn til diskusjonsforumet.

```
1 <input type="image" style="border-width: 0px;" onclick="javascript:
    WebForm_DoPostBackWithOptions(new WebForm_PostBackOptions("imgSaveText", "", true,
     "", "", false, false))" src="images/wi0062-24.png" id="imgSaveText" name="
    imgSaveText"/>
```

Liste 7.5: JavaScript som står for POST-forespørselen for å lagre foruminnlegg i ASP.NET-versjonen av diskusjonsforumet

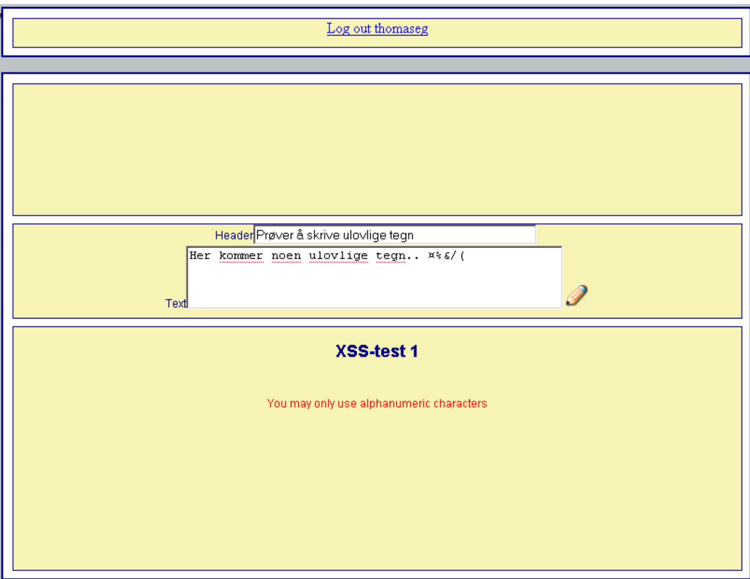
Resultatet av tjenersidefiltrering kan sees i figur 7.9, hvor forespørselen ble forkastet.

7.3.2 Test av HTML-kontrolltegnfiltrering i ASP.NET

Etter å ha testet validering av input, ble denne slått av slik at det lå til rette for å teste mekanismen for filtrering av HTML-kontrolltegn. Poenget med å slå av filtrering av input, var å lykkes med å lagre innlegg med andre tegn enn denne tillot. Det ble forsøkt å lagre et innlegg med et JavaScript innhold med hensikt å vise en knapp med en *onload-event* som viser en melding med verdien "1":

```
<img onclick="alert(1)" src="wi0054-16.png"/>
```

Deretter ble det verifisert at dette innholdet faktisk ble lagret til databasen. Det ble også gjort forsøk på å se det aktuelle innlegget, som resulterte i at JavaScriptet ble sendt til klienten. Dette viser at ASP.NET versjonen av diskusjonsforumet også var XSS-utsatt.



Figur 7.9: Tjenersidefiltrering av input i ASP.NET-versjonen av diskusjonsforumet

Etter å verifisere svakheten, ble det gjort forsøk på å benytte funksjonen *Server.HtmlEncode* som står for filtrering av HTML-kontrolltegn. Resultatet av dette, var at alle kontrolltegn ble endret slik at dataene som ble levert til klienten ble:

```
&lt;img src="" onclick="alert(1)"/&gt;
```

7.3.3 Test av SQL-kontrolltegnfiltrering i ASP.NET

Det ble gjort forsøk på SQL-injisering for å teste ut effekten av den innebygde kontrolltegnhåndteringen. Det ble gjort forsøk på å logge inn med blant annet brukernavn "thomaseg' - - " som vist i eksempelet fra delkapittel 3.3.4 uten hell.

7.3.4 Test av XSRF-angrep i ASP.NET

På lik linje med diskusjonsforumet utviklet i DWR og RoR, var det nødvendig å se hvorvidt ASP.NET-versjonen var utsatt for XSRF. For å teste dette ble det først logget inn i diskusjonsforumet for deretter å lagre et innlegg. Den resulterende POST-forespørselen ble fanget opp ved hjelp av *Live HTTP headers*. Deretter ble alle cookies i nettleseren slettet, webtjeneren restartet, og det ble logget inn i diskusjonsforumet på nytt. Det ble verifisert at en ny cookie-verdi ble satt i den nye sesjonen.

For å forfalske en POST-forespørsel for å lagre et innlegg i forumet, ble *replay* funksjonen i *Live HTTP headers* benyttet. POST-forespørselen som ble fanget opp ved første innlogging, ble endret til å sende cookie-verdien til den nye sesjonen. Resultatet av dette var at den forfalskede forespørselen ble godtatt på tjenersiden, og at et nytt innlegg ble lagret i diskusjonsforumet. Dette viser at ASP.NET versjonen av diskusjonsforumet også var utsatt for angrep av typen XSRF.

7.4 Oppsummering

Dette kapittelet har vist hvordan de forskjellige sikkerhetsmekanismene i DWR, RoR og ASP.NET ble testet, og hvilke resultater disse ga. Testresultatene danner grunnlag for vurderingen i neste kapittel 8.

KAPITTEL 8
VURDERING

Dette kapittelet gir vurdering av resultatene fra testingen som ble utført i kapittel 7. Det ble sett at nesten alle mekanismene måtte aktiveres manuelt for å bidra til forholdsregler mot fallgruver og angrep. Dette kan være med å bidra til flere webapplikasjoner utsatt for en rekke fallgruver og angrep, da det ofte fokuseres mer på krav til funksjonalitet enn sikkerhet. Det ble i tillegg sett at det hovedsakelig var støtte for forholdsregelen validering og filtrering av data på tjenerside nevnt i delkapittel 4.1.

Testingen som ble utført, avdekket feil i en av sikkerhetsmekanismene til DWR. Dette viser hvor viktig det er å teste ut mekanismer som velges å benyttes, for å se resultatet av disse. Det må ikke stoles blindt på dokumentasjon, eksempler på Internett og lignende, da disse kan gi misvisende informasjon. Sikkerhet er en svært viktig faktor ved utvikling av webapplikasjoner, og derfor må det ikke være tvil om hvilket resultat som oppnås ved bruk av ferdiglagde mekanismer.

8.1 Vurdering av DWR

DWR krevde at tjenersidefiltrering av HTML-kontrolltegn måtte aktiveres manuelt. Det ble sett at en av mekanismene ikke fungerte slik det er påstått, og at rammeverket støttet en alternativ XSRF-beskyttelse for å hindre forfalskede forespørsler.

8.1.1 Vurdering av HTML-kontrolltegnfiltrering i DWR

Både metoden escapeHtml og replaceXmlCharacters støtter opp mot tjenersidefiltrering av HTML-kontrolltegn nevnt i delkapittel 4.1. Dette fordi alle kontrolltegn i innlegget ble endret slik det er påstått.

Det er viktig å være klar over at disse metodene ikke tillater noe som helst HTML-kontrolltegn. Da det i enkelte tilfeller kan være ønskelig å tillate brukere å benytte noen av disse, bør det

benyttes hvitlister fra delkapittel 4.1.3 for å lage regelsett over hva som skal tillates.

Feil i metoden containsXssRiskyCharacters

Slik det ble vist i delkapittel 7.1.1, ga denne metoden feil resultat i forhold til hva som ble påstått. Det ble derfor sett nærmere på kildekoden for den aktuelle metoden, som kan sees i liste 8.1. Denne metoden skulle returnert *true* hvis strengen original inneholder en av tegnene:

`&, <, >, ' eller ''`

Feilen var at metoden returnerte *true* hvis og bare hvis alle disse tegnene er representert i tekststrengen som sendes inn. Ved å gjøre modifikasjon slik liste 8.2 viser, ga metoden den ønskede funksjonaliteten.

```
1 public static boolean containsXssRiskyCharacters(String original)
2 {
3         return (original.indexOf('&') != -1
4             && original.indexOf('<') != -1
5             && original.indexOf('>') != -1
6             && original.indexOf('\'') != -1
7             && original.indexOf('\"') != -1);
8 }
```

Liste 8.1: Feil i DWR's XSS-metode

```
1 public static boolean containsXssRiskyCharacters(String original)
2 {
3         return (original.indexOf('&') != -1
4             || original.indexOf('<') != -1
5             || original.indexOf('>') != -1
6             || original.indexOf('\'') != -1
7             || original.indexOf('\"') != -1);
8 }
```

Liste 8.2: Feilretting i DWR's XSS-metode

Årsaken til feil i denne metoden kan sees i sammenheng med at DWR-versjonen som ble testet var en betaversjon av 2.0. Utviklere av webapplikasjoner kan likevel basere filtrering av kontrolltegn på metodene i klassen *Security*. Derfor ble det bestemt at feilen skulle rapporteres til utviklerne av DWR, og tilbakemeldingen var at denne ville bli rettet og være inkludert i neste versjon av rammeverket.

Feilen ble rettet 17. april 2007 (ikke mange dagene etter feilen ble rapportert). Innsyn i CVS[1] for klassen *org.directwebremoting.Security* kan sees på adressen `http://fisheye5.cenqua.com/browse/dwr/java/org/directwebremoting/Security.java`.

8.1.2 Vurdering av XSRF-beskyttelse i DWR

Det ble i delkapittel 7.1.2 sett at XSRF-beskyttelsen i DWR baserer seg på å sende *JSESSIONID* som en ekstra parameter i hver forespørsel. Det ble også sett at DWR forkaster forespørsler

[1]CVS - Concurrent Version System er et verktøy for versjonskontroll av filer

hvis den aktuelle parameteren ikke er lik *JSESSIONID*. Mekanismen bidrar til beskyttelse mot XSRF, fordi nettlesernes *Cross Domain restriction* ikke tillater at websider fra andre domener enn opphavet kan hente ut denne verdien.

Et alternativ til *JSESSIONID*, kan benytte en ny identifikator som en nøkkel for hver forespørsel slik det ble nevnt i delkapittel 4.4. Dette vil gjøre det svært vanskelig for en angriper å stjele denne, for deretter å forfalske en forespørsel før brukeren selv lager en ny. En slik nøkkel kan basere seg på å kryptere kombinasjonen av *JSESSIONID* og et datostempel (nåværende dato). En negativ side med denne løsningen er at krypteringen vil kreve ressurser på tjenersiden, og ved en stor brukermasse kan dette medføre skaleringsproblemer.

8.1.3 Begrensninger i DWR

Sikkerhetsmekanismene i DWR begrenser seg til tjener mot klient, og omfatter ikke filtrering av data sendt til andre systemer, som blant annet SQL-databaser. Utviklere er derfor avhengig av å implementere egne metoder for dette, eventuelt benytte DWR i kombinasjon med andre rammeverk som støtter dette. Diskusjonsforumet som ble utviklet, baserer seg på bruk av DWR i kombinasjon med Hibernate for databasehåndtering. DWR mangler også mekanismer for å validere input.

DWR bør kombineres med forholdsregelen sikkerhet i dybden fra delkapittel 4.2, ved å kombinere flere sikkerhetsmekanismer i parallell. Dette kan gjøres ved å unngå å kjøre webtjeneren med administratorrettigheter, og sette tilgangsrettigheter på databasenivå.

Fra testingen ble det sett at alle mekanismene i DWR med unntak av XSRF-beskyttelse måtte aktiveres manuelt. Det er viktig å være klar over dette, for å unngå fallgruver og angrepstyper basert på HTML-kontrolltegn.

8.2 Vurdering av RoR

Det ble sett fra testingen at RoR støtter validering av input og filtrering av HTML-kontrolltegn, selv om disse manuelt måtte aktiveres. Det ble også sett at de innebygde funksjonene for å hente ut verdier fra databasen forhindrer SQL-injisering.

8.2.1 Vurdering av inputvalidering i RoR

Testingen viser at RoR støtter enkel tjenersidevalidering av input. Det gjør det mulig å lage enkle og avanserte regelsett for å definere hvilke input som skal tillates. Det ble i diskusjonsforumet definert at kun store og små bokstaver, oppholdsrom og tall skulle være lovlig å benytte.

8.2.2 Vurdering av HTML-kontrolltegnfiltrering i RoR

Det ble i delkapittel 7.2.2 sett at *h()*-funksjonen automatisk filtrerer HTML-kontrolltegn på data som blir sendt til brukerens nettleser. På grunn av dette, bør denne benyttes på alt av data som skal sendes til brukerens nettleser.

På lik linje med mekanismene for filtrering av HTML-kontrolltegn i DWR, vil det i enkelte tilfeller være aktuelt å tillate noen kontrolltegn. Til dette kan det utvikles egne løsninger med bruk av hvitlister fra delkapittel 4.1.3, for å lage egendefinerte regelsett for hva som skal tillates.

8.2.3 Vurdering av SQL-kontrolltegnfiltrering i RoR

Resultatet fra delkapittel 7.2.3 viser at predefinerte databasefunksjoner ikke trenger å ta hensyn til filtrering av input. Det bør i situasjoner hvor det defineres egne SQL-spørringer, benyttes funksjonen quote() konsekvent for å forhindre injisering av SQL-kontrolltegn.

8.2.4 Begrensninger i RoR

Det ble vist ved testing at diskusjonsforumet i RoR var utsatt for XSRF-angrep. Med bakgrunn i dette, kan en angriper lage en webside med JavaScript, og lokke en bruker til å besøke denne etter å ha logget inn i diskusjonsforumet. Denne websiden kan lage et stort antall POST-forespørsler som lagrer innlegg i diskusjonsforumet. Det bør derfor benyttes en mekanisme som underbygger forholdsregelen nevnt i delkapittel 4.4 for å sikre mot dette.

Beskyttelse i dybden kan oppnås på samme måte som ved DWR, ved å sette tilgangsrettigheter på databasetabeller og kjøre den benyttede webtjeneren med begrensede rettigheter.

På lik linje med DWR, måtte funksjonen *h()* aktiveres manuelt før HTML-kontrolltegn ble filtrert. Det er viktig å være klar over dette, slik at det ikke utvikles XSS-utsatte webapplikasjoner.

8.3 Vurdering av ASP.NET

ASP.NET med AJAX ga et komplisert bilde av hvordan og hvor filtrering av kontrolltegn foregikk, og hvor det ble tatt hensyn til validering av input. Etter å undersøke JavaScript-koden som blir sendt ut til nettleseren, ble det prøvd og feilet for å deaktivere klientsidevalideringen slik det ble vist i delkapittel 7.3.1. Det virket i tillegg tungvint å måtte aktivere validering av input manuelt to steder.

8.3.1 Vurdering av inputvalidering i ASP.NET

Mekanismen støtter opp mot forholdsregelen tjenersidevalidering av data som ble nevnt i delkapittel 4.1.1. Dette gir kontroll på at data har ønsket form når de ankommer tjenersiden. Alternativt til å bare forkaste forespørselen slik det ble gjort i testingen, bør det benyttes en feilmelding til brukeren i tillegg til logging av hendelsen. Liste 7.4 i delkapittel 7.3.1 kan i motsetning til å avvise forespørselen, kjøre kode for å generere en feilmelding til brukeren. En slik feilmelding må ikke inneholde tjenersidespesifikk informasjon, da dette kan benyttes som et springbrett for andre typer angrep.

8.3.2 Vurdering av HTML-kontrolltegnfiltrering i ASP.NET

Slik det ble vist i delkapittel 7.3.2 ble alle HTML-kontrolltegn endret ved at de ble gjort om fra kontrolltegn til vanlige data. Dette støtter opp mot forholdsregelen for tjenersidefiltrering av kontrolltegn i delkapittel 4.1.2.

Det vil i enkelte tilfeller være aktuelt å tillate noe HTML-kontrolltegn i data sendt ut til brukerens nettleser. På samme måte som det er foreslått i DWR og RoR, kan hvitlister fra delkapittel 4.1.3 benyttes til å definere egne regelsett.

8.3.3 Filtrering av data access

Den innebygde mekanismen i ASP.NET *data access* forhindrer SQL-injisering og understøtter derfor også forholdsregelen om tjenersidefiltrering av kontrolltegn. Kontrolltegnene har i denne situasjonen SQL-databasen som destinasjon, og det vil være viktig å påse at webapplikasjonen ikke er utsatt for SQL-injisering.

8.3.4 Begrensninger i ASP.NET

Det ble sett i delkapittel 7.3.4 at ASP.NET-versjonen av diksusjonsforumet var utsatt for XSRF-angrep. På lik linje med RoR-versjonen av diskusjonsforumet, kan dette utnyttes ved at en angriper får en innlogget bruker til å besøke en webside som lager oppdiktede forespørsler mot tjenersiden. Derfor bør det vurderes mekanismer som underbygger forholdsregelen mot XSRF, beskrevet i delkapittel 4.4.

Fra testingen ble det også sett at tjenersidevalidering var en funksjonalitet som måtte aktiveres i tillegg til klientsidevalidering. Det samme gjaldt filtrering av HTML-kontrolltegn. Ved bruk av ASP.NET, er det viktig å håndtere dette for å unngå fallgruver.

8.4 Felles vurdering

Det ble i kapittel 4 presentert fire forholdsregler som kan tas for å unngå fallgruver fra kapittel 3. Tabell 4.1 i kapittel 4 listet ut disse forholdsreglene, og hvilke fallgruver de kan forhindre.

Fra testingen ble det sett at ikke alle de fire forholdsreglene støttes i rammeverkene, og utviklere må derfor benytte andre løsninger eller lage egne. Tabell 8.1 viser hvilke forholdsregler de forskjellige rammeverkene støtter og hvordan de støttes.

- \- angir ikke påvist støtte
- M angir støtte som manuelt må aktiveres
- A angir automatisk støtte
- D angir delvis støtte

Hvordan forholdsregler støttes i rammeverkene	DWR	RoR	ASP.NET
1. Validering og filtrering av data på **tjenerside**			
Validering av input	-	M	M
Filtrering av kontrolltegn			
HTML-kontrolltegn	M (feil i en metode)	M	M
SQL-kontrolltegn	-	A	A
2. Beskyttelse i dybden	-	D	D
3. Begrense data som sendes til klientside	-	D	D
4. XSRF-beskyttelse	A	-	-

Tabell 8.1: Hvilke forholdsregler som støttes av mekanismer testet i rammeverkene og hvordan disse støttes

8.4.1 Beskyttelse i dybden

Denne forholdsregelen baserer seg på å kombinere flere tiltak i parallell. Rammeverket ASP.NET og RoR støtter denne forholdsregelen til en viss grad, ved å kombinere validering av input og HTML-kontrolltegnfiltrering. Som det ble nevnt i delkapittel 4.2 kan det også benyttes eksterne tiltak for å støtte opp mot dette.

8.4.2 Begrense data som sendes til klientside

Ved bruk av rammeverkene RoR og ASP.NET, var det ikke nødvendig å skrive JavaScript. Dette bidrar til å unngå applikasjonslogikk i JavaScript, da rammeverkene håndterer alt som sendes til nettleseren automatisk.

Utviklere av AJAX-baserte webapplikasjoner må begrense hva som tilgjengeliggjøres for en potensiell angriper. Før en operasjon gjøres basert på parametere i en forespørsel, må mekanismer påse at disse er autentiske, og resulterer i gyldige operasjoner. Bruk av indirekte datatilgang fra delkapittel 4.3.2, er en god måte å begrense data som sendes til klienten.

8.4.3 Applikasjonsspesifikk sikkerhet i webapplikasjoner

Det ble innledningsvis i rapporten, i delkapittel 2.3 nevnt hvordan enkelte personer har et forvrengt syn på dette med sikkerhet i webapplikasjoner. Det er ikke nok å ha administratorer som setter opp gode brannmurer og lignende for å sikre en webapplikasjon. En webapplikasjon baserer seg på å tillate forespørsler til og fra klient ved hjelp av protokollen HTTP fra delkapittel 2.1.1. Enkelte brannmurer opererer på applikasjonsnivå, og kan blant annet se på innholdet i de forskjellige hodene i en HTTP-forespørsel. Det en brannmur ikke ser, er hva de forskjellige parameterne resulterer i på den aktuelle webapplikasjonen. Derfor er det viktig å innføre rutiner for å benytte forholdsregler tidlig i utviklingsprosessen.

8.4.4 Rammeverkenes støtte for sikkerhet som prosess

Vi har sett fra testingen av sikkerhetsmekanismene i kapittel 7, at flere mekanismer må manuelt aktiveres før de gir resultat. Alle de tre versjonene av webapplikasjonen som ble utviklet, var

sårbare for forskjellige fallgruver. Hvis sikkerhetsmekanismer hadde vært standard aktivert, ville dette forhindret fallgruvene som webapplikasjonene var utsatt for. Hvis mekanismene reduserer funksjonalitet eller ikke gir ønsket resultat, kan de eventuelt deaktiveres eller byttes ut med egenlagde.

Utviklingen på de forskjellige diskusjonsforumene ble gjort i rekkefølgen DWR, RoR og ASP.NET. Etter at ASP.NET-versjonen var utviklet, måtte det benyttes mer tid på å gå gjennom kildekoden for å teste sikkerhet i DWR-versjonen. Dette gir et inntrykk av at det mest effektive er å ta hensyn til forholdsregler underveis i utviklingsprosessen. Selv om kompleksiteten er holdt på et lavt nivå i diskusjonsforumet, kan det sees en tendens til hvordan en kombinasjon av tid og funksjonalitet vanskeliggjør kartlegging av hvilke forholdsregler som bør tas, og hvor mekanismer for disse bør plasseres.

Selv om rammeverkene har mekanismer som støtter forholdsregler mot enkelte fallgruver, er det viktig å ta hensyn til de andre forholdsreglene det ikke finnes mekanismer for. Selv om dette kan stjele tid i utviklingen, vil det være et bedre alternativ enn å tenke på sikkerhet når de funksjonelle kravene er på plass. Ved utvikling av diskusjonsforumet ville det vært bedre å tatt hensyn til forholdsreglene fra start, istedet for å bruke tid på forholdsregler til slutt.

8.4.5 Sikkerhet påvirker funksjonalitet

En webapplikasjon som er utilgjengelig gir heller ingen funksjonalitet. Derfor bør krav til sikkerhet likestilles med krav til funksjonalitet. Det hjelper lite å ha et system som tilbyr godt brukergrensesnitt med spennende og interaktiv funksjonalitet tilført av AJAX, når det er blitt utilgjengelig på grunn av fallgruver og angrepstyper.

8.5 Oppsummering

Dette kapittelet har vurdert de forskjellige mekanismene som ble testet i kapittel 7. Resultatene fra vurderingen danner grunnlag for konklusjonen som blir presentert i neste kapittel 9.

KAPITTEL 9

KONKLUSJON OG EGENREFLEKSJON

Med bakgrunn i målsetningen med oppgaven, trekkes det konklusjoner i forbindelse med rammeverkenes støtte for forholdsregler mot fallgruver og angrepstyper. Det reflekteres også over metodene som ble valgt å benytte.

Målsetningen med oppgaven var å undersøke forskjellige sikkerhetsmekanismer i eksisterende AJAX-baserte rammeverk. Hensikten var å se hvordan disse støtter forholdsregler mot fallgruver og angrepstyper som kan oppstå ved utvikling av AJAX-baserte webapplikasjoner.

9.1 Konklusjon

Vi konkluderer med at AJAX-baserte webapplikasjoner ikke nødvendigvis medfører en større angrepsflate, men bidrar til en økt kompleksitet. Dette fordi AJAX-baserte webapplikasjoner benytter XHR-objektet til å kommunisere mot tjenersiden asynkront, og det kan være vanskelig å kartlegge alle situasjoner som resulterer i en forespørsel mot tjenersiden.

Cross-domain-restriksjonen med XHR-objektet har til hensikt å hindre forespørsler mot andre domener enn der webapplikasjonen har opphav. Dette bidrar til en falsk sikkerhet, fordi den ikke minsker faren for angrep som XSS. Det finnes alternative metoder som tillater forespørsler mot andre domener, ved blant annet å benytte dynamiske script-elementer. Denne restriksjonen bør derfor ikke sees på som en forholdsregel for å forhindre fallgruver og angrep.

Det eksisterer forskjellige rammeverk som har til hensikt å forenkle utvikling av AJAX-baserte webapplikasjoner både med hensyn til tid og sikkerhet. Eksperimentet viser hvordan mekanismer i de tre rammeverkene Direct Web Remoting, Ruby on Rails og ASP.NET, hovedsaklig støtter forholdsregler mot validering og filtrering av data sendt til og fra en webapplikasjon. Enkelte mekanismene måtte aktiveres manuelt. Dette kan være en fallgruve for utviklere som baserer sitt arbeide på eksempler og informasjon på Internett. Utviklere med

liten erfaring, vet lite om hvilke forholdsregler som bør tas. Disse vil heller ikke ha oversikt over hvilke mekanismer som bør aktiveres i det benyttede rammeverket.

Mekanismer som støtter forholdsregler mot fallgruver i rammeverkene bør være standard aktivert. Dette vil bidra til mindre AJAX-baserte webapplikasjoner som er utsatt for fallgruver og angrepstyper. På denne måten vil sikkerhet være en del av utviklingsprosessen, istedet for å bli et produkt som legges til senere. Dessuten vil kompleksiteten tilført av AJAX vanskeliggjøre kartleggingen av hvor mekanismene bør benyttes senere i utviklingsfasen.

Fra testingen ble det sett at bare et av rammeverkene hadde støtte for forholdsregelen mot XSRF. XSRF er en fallgruve som skiller seg ut fra andre fallgruver. Derfor er det viktig at AJAX-baserte rammeverk har mekanismer for å underbygge forholdsregelen mot XSRF. Alle rammeverkene støttet imidlertid mekanismer for å filtrere HTML-kontrolltegn, for å underbygge forholdsregelen om tjenersidefiltrering.

En av mekanismene som ble testet i rammeverket Direct Web Remoting, resultere i uventet og feil resultat. Feilen ble rapportert til utviklerne av rammeverket, og ble rettet raskt. Dette viser at mekanismene bør testes, slik at det kan bekreftes at disse støtter opp mot forholdsreglene som hevdes.

Selv om det eksisterer mekanismer i rammeverkene som støtter enkelte forholdsregler for å unngå fallgruver og angrep, er det viktig å ikke tro at disse vil være nok for å lage en 100% sikker webapplikasjon. Vi har sett på en rekke aktuelle fallgruver og angrepstyper i AJAX-baserte webapplikasjoner, og hvilke forholdsregler som støtter opp mot disse. Det vil eksistere andre aktuelle fallgruver som ikke er nevnt i denne oppgaven, med egne forholdsregler mot disse.

9.2 Egenrefleksjon

Etter gjennomføringen av oppgaven, reflekteres det over valg av metoder og arbeidet som ble utført basert på disse.

9.2.1 Arbeidet som ble utført

AJAX har blitt et “hot” tema det siste året på grunn av muligheten til å lage interaktive webapplikasjoner, uten behov for bruk av tredjepart programvare som flash. Dette har resultert i en rekke rammeverk tilrettelagt for å gjøre det enklere å utvikle AJAX-baserte webapplikasjoner.

Ved å teste de utvalgte rammeverkene, var det mulig å få bekreftet hvilke forholdsregler som støttes og hvordan de støttes.

9.2.2 Vurdering av metode som ble benyttet

Oppgaven baserer seg på stoff tilegnet gjennom litteraturstudiet og det praktiske arbeidet utført for å utvikle en webapplikasjon i rammeverkene DWR, RoR og ASP.NET.

Litteraturstudie

Litteraturstudiet bidro til å undersøke bakgrunn for AJAX-baserte webapplikasjoner og fallgruver og angrepstyper rettet mot disse. Det ble benyttet forskjellige praktiske eksempler og illustrasjoner for å beskrive teorien bedre.

Eksperiment

Eksperimentet var avgjørende for å teste ut de forskjellige mekanismene. Av oppgaven har vi lært at en god måte å teste sikkerhetsmekanismer i rammeverk, er å teste disse i en reell webapplikasjon.

9.3 Forslag til videre arbeide

Selv om det finnes en rekke mekanismer i rammeverk tilrettelagt for utvikling av webapplikasjoner med AJAX, er det viktig å ikke basere all sikkerhet kun på disse. Det finnes mange andre aspekter vedrørende sikkerhet som ikke er blitt omtalt i denne oppgaven, herav menneskelige feil og infrastrukturelle problemer.

Videre arbeide med sikkerhetsfallgruver og forholdsregler i Web 2.0 med AJAX bør være en kontinuerlig prosess. Denne prosessen bør basere seg på å lære av feil som en selv eller andre har gjort tidligere. Slik kan det kartlegges flere fallgruver og hvilke forholdsregler som kan tas mot disse. Mekanismer i rammeverkene kan deretter utvikles som understøtter disse. På denne måten kan man oppnå sikrere webapplikasjoner.

Det ble sett fra testingen at mekanismene måtte aktiveres og benyttes forskjellig i hvert rammeverk. Videre arbeide kan være å implementere egne mekanismer i rammeverkene som aktiveres på samme måte. Det kan benyttes konfigurasjonsfiler til å definere hvilke regler de forskjellige mekanismene skal basere seg på. Dette kan være mekanismer for validering av input, filtrering av kontrolltegn eller XSRF-beskyttelse.

BIBLIOGRAFI

[1] Jesse James Garret. A new approach to web application, 2005.
Webside hentet 01.02.2007 fra:
http://adaptivepath.com/publications/essays/archives/000385.php.

[2] Stefano Di Paola and Giorgio Fedon. Subverting ajax, 2006. Tilgjengelig på:
http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf.

[3] Erlend Oftedal. Why input validation is not the solution for avoiding sql injection and xss, 2006.
Webside hentet 01.04.2007 fra:
http://erlend.oftedal.no/blog/?blogid=16.

[4] Getahead. Direct web remoting, 2007.
Webside hentet 10.02.2007 fra:
http://getahead.org.

[5] Dave Crane, Eric Pascarello, and Darren James. *Ajax in action*. Manning, 2005.

[6] Sverre H. Huseby. *Innocent Code*. Wiley, 2004.

[7] Jose Annunziato and Stephanie Fesler Kaminaris. *javaServer Pages*. Sams, 2001.

[8] The Internet Society. Hypertext transfer protocol, 1999.
Webside hentet 01.02.2007 fra:
http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[9] David Hunter, Andrew Watt, Jeff Rafter, Jon Duckett, Danny Ayers, Nicholas Chase, Joe Fawcett, Tom Gaven, and Bill Patterson. *Beginning XML, 3rd Edition*. Wrox, 2004.

[10] James F. Kurose and Keith W.Ross. *A top-down approach featuring the Internet*. Addisin Wesley, 2005.

[11] Rickland Hollar and Richard Murphy. *Enterprise Web Services Security*. Charles River Media, 2006.

[12] Jason Schmitt. *Secure ASP.NET AJAX Development*. Pearson Education, Inc, 2007.

[13] Sverre Huseby. Common security problems in the code of dynamic web applications, 2005. Tilgjengelig på:
http://www.webappsec.org/projects/articles/062105.pdf.

[14] Billy Hoffman. Ajax security dangers, 2006. Tilgjengelig på:
http://www.spidynamics.com/assets/documents/AJAXdangers.pdf.

[15] Joel Scambray, Mike Shema, and Caleb Sima. *Hacking Exposed, Web applications*. McGraw-Hill, 2006.

[16] JSON. Introducing json, 2005.
Webside hentet 01.03.2007 fra:
http://www.json.org.

[17] Jason Levitt. Json and the dynamic script tag, 2005.
Webside hentet 01.03.2007 fra:
http://www.xml.com/pub/a/2005/12/21/json-dynamic-script-tag.html?page=2.

[18] Nenad Jovanovic, Engin Kirda, and Christophher Kruegel. Preventing cross site request forgery attacks, 2005. Tilgjengelig på:
http://www.seclab.tuwien.ac.at/papers/noforge.pdf.

[19] Sam Stephenson. Prototype javascript framework, 2007.
Webside hentet 15.03.2007 fra:
http://www.prototypejs.org/.

[20] Dojo Foundation. Dojo toolkit, the javascript toolkit, 2007.
Webside hentet 12.03.2007 fra:
http://dojotoolkit.org/.

[21] Google. Google web toolkit, 2007.
Webside hentet 13.03.2007 fra:
http://code.google.com/webtoolkit.

[22] Dave Thomas and David Heinemeier. *Agile Web Development with Rails*. The Pragmatic Programmers, 2005.

[23] Microsoft Corporation. Asp.net ajax, 2007.
Webside hentet 16.03.2007 fra:
http://ajax.asp.net.

[24] Scott Raymond. *Ajax on Rails*. O‘Reilly, 2006.

VEDLEGG A

SKJERMBILDE AV DISKUSJONSFORUM