HTML5 ur ett säkerhetsperspektiv

Magnus Gulin

Kandidatavhandling i datavetenskap

Institutionen för informationsteknologi vid Åbo Akademi

Handledare: Frank Wickström

Våren 2013

Sammandrag

Utvecklingen av Internet, och då särskilt World Wide Web, går framåt i en rasande takt. När även banktjänster och andra personliga uppgifter hanteras på nätet, är det viktigt att de hanteras säkert. När webbläsare uppdateras för att hantera nya protokoll och språk medför det också att nya säkerhetshål introduceras. HTML5 medför både nya funktioner som kan utnyttjas för illasinnade aktiviteter, men också funktioner som skall skydda användaren. Med min kandidatavhandling vill jag undersöka dessa för- och nackdelar med HTML5.

Nyckelord: HTML, HTML5, säkerhet, webbläsare, HTTP

Innehållsförteckning

[1 Inledning 1](#_Toc356406911)

[2 Vad är HTML 2](#_Toc356406912)

[3 Nyheter i HTML5 3](#_Toc356406913)

[3.1 PostMessage 3](#_Toc356406914)

[3.2 Lokalt sparade webbapplikationer 3](#_Toc356406915)

[3.3 Local Storage 4](#_Toc356406916)

[3.4 Web sockets 5](#_Toc356406917)

[4 Säkerhetsbrister i HTML 5](#_Toc356406918)

[4.1 Webbkodsinjektion 6](#_Toc356406919)

[4.1.1 Webbkodsinjektion i HTML5 8](#_Toc356406920)

[4.2 Cross-site request forgery (CSRF) 9](#_Toc356406921)

[4.2.1 CSRF i HTML5 10](#_Toc356406922)

[4.3 Klicknappning 12](#_Toc356406923)

[4.3.1 Klicknappning i HTML5 13](#_Toc356406924)

[4.4 Säkerhetsbrister som har tillkommit i HTML5 14](#_Toc356406925)

[5 Skydd mot säkerhetsbrister 15](#_Toc356406926)

[5.1 Slumpmässigt attribut 15](#_Toc356406927)

[5.2 Undvika webbkodsinjektion 15](#_Toc356406928)

[5.3 HTTPOnly-flagga i kakor 17](#_Toc356406929)

[5.4 Klicknappnings-skydd 17](#_Toc356406930)

[5.5 Skydd från användarens synvinkel 18](#_Toc356406931)

[6 Sammanfattning 19](#_Toc356406932)

[7 Källförteckning 21](#_Toc356406933)

# Inledning

I HTML5 finns många nya funktioner och möjligheter att bygga webbapplikationer på ett sätt som gör att de mera påminner om de program vi vanligtvis använder på en dator. Denna utökade funktionalitet gör också att webbläsaren har svårare att veta a) om information/kod den har mottagit från en källa är pålitlig, och b) om den får skicka denna information vidare till en annan källa. HTML5 innehåller säkerhetsfunktioner som gör att webbläsaren får information om hur den skall hantera dylik information, och därmed stoppar attacker som webbkodsinjektion (eng. *Cross-site Scripting*), CSRF (eng. *Cross-site Request Forgery)* och klicknappning (eng. *clickjacking)*.

På Internet finns mycket information om hur man bygger webbapplikationer, men däremot finns det mindre kunskap om hur man sedan testar och skyddar sin applikation för säkerhetsbrister. De flesta kunskapskällor visar över huvud taget inte vilka saker man vill undvika i sin applikation för att undvika att användarna drabbas av exempelvis stulna lösenord. Detta är enligt min mening en bra motivering för att skriva om just detta.

# Vad är HTML

Hypertext Markup Language (*HTML*) är ett märkspråk, och används för att bygga upp layout och funktionalitet på webbsidor. Första versionen HTML kom ut 1995. Språket har sedan uppdaterats i flera omgångar, och har därmed fått mera funktionalitet [1].

Nu för tiden används ofta HTML för att strukturera material, medan själva layouten byggs upp med hjälp av stilmallar. Standarden dessa använder kallas Cascading Style Sheets(*CSS*).

Förutom att se på strukturerad text och bilder kan man på webbsidor också köra små program, så kallad *webbkod*. Webbkod introducerades först under namnet LiveScript, och döptes senare till JavaScript. Microsoft gjorde sedan sin egen implementation som döptes till JScript. Språket blev standardiserat i juni 1997 under namnet ECMAScript. Alla webbläsare strävar till att vara kompatibla med ECMAScript, men kan också innehålla utökad funktionalitet. ECMAScript är alltså själva språket, medan de olika implementationerna är dialekter av språket. ECMAScript har uppdaterats i omgångar, och på grund av dess rika funktionalitet används det i majoriteten av säkerhetshålen som uppstår i webbapplikationer. För att undvika förvirring mellan de olika implementationerna används hädanefter begreppet webbkod [2].

De flesta webbläsare har dessutom stöd för insticksmoduler. Dessa utökar webbläsarens funktionalitet och kan exempelvis visa proprietära filformat. Vanliga insticksmoduler är Java applets, Silverlight och Flash. Säkerhetsbrister kan också förekomma i dessa, men detta tas inte upp i den här avhandlingen.

# Nyheter i HTML5

HTML är uppbyggt av taggar. Taggar kan innehålla ett eller flera attribut, och varje attribut kan ha ett värde.

<taggnamn attribut1 attribut2=”100” > </taggnamn>  
<img src=”bild.jpg” >

Översta raden visar ett exempel där vi har en *taggnamn-*tagg med början- och sluttagg, samt två attribut varav det första saknar värde och det andra attributet har värdet 100. Den undre raden visar en *img*-tagg där attributet *src* har värdet *bild.jpg*. Detta är fungerande HTML-kod och skulle visa en bild.

HTML5 introducerar flera nya taggar för multimedia, så som *audio*, *video* och *canvas* [3]. Med hjälp av dessa behöver webbläsaren inte längre insticksmoduler för att kunna hantera multimedia.

## PostMessage

*Postmessage* [4] är en funktion för att kommunicera mellan olika domäner. En webbsida kan innehålla en eller flera ramar. Dessa ramar infogar andra webbsidor inuti sig själva. Genom att använda PostMessage kan man sända och ta emot data både mellan sidorna i ramarna, men också från och till den huvudsida som innehåller själva ramarna. PostMessage erbjuder också filtrering så att den mottagande sidan kan bestämma att data endast får tas emot från en viss värd.

## Lokalt sparade webbapplikationer

Vanligtvis behöver webbläsaren alltid en internetuppkoppling för att användaren skall kunna använda en webbapplikation. Om man då har en ostabil internetuppkoppling blir det svårt att använda den effektivt, eller till och med omöjligt. Offline Web Applications (*OWA*) [5], försöker åtgärda detta.

I HTML5 har *html*-taggen har fått ett nytt attribut, *manifest*. Här fastställs var *manifest*-filen finns. I *manifest*-filen specificeras vilka filer som hör till applikationen och bör sparas i webbläsarens cacheminne, samt vilka filer som hör till servern och därmed inte bör sparas lokalt.

En HTML5-webbsida som implementerar OWA bör exempelvis börja med följande:

<!DOCTYPE HTML>  
<html manifest="cache-manifest">

Filen *cache-manifest*:s innehåll är uppbyggd enligt följande:

CACHE MANIFEST  
index.html  
help.html  
style/default.css  
images/logo.png  
images/backgound.png

NETWORK:  
server.cgi

Filerna som är uppräknade under raden *CACHE MANIFEST* är de filer som bör sparas lokalt, medan filen som finns under *NETWORK* inte bör sparas.

## Local Storage

Traditionellt sett har HTTP-kakor (eng. *http cookies*) använts för att spara data på en användares dator. Dessa har nackdelen att de kan max innehålla 4kB data. De sänds dessutom med i varje förfrågan webbläsaren gör, vilket ger onödig trafik. Därför har det med tiden uppkommit flera lösningar för att spara data lokalt. Internet Explorer implementerade userData, Adobe:s Flash-insticksmodul kunde spara data som Local Shared Objects och Google Gears var ytterligare ett system som kunde användas. Det fanns alltså ingen brist på tekniker att använda, men däremot inget sätt som fungerade enkelt med alla webbläsare utan insticksmoduler. Detta försöker *Local Storage* att åtgärda, och alla moderna webbläsare stöder nu detta. Local Storage är egentligen inte bundet till HTML5-specifikationerna, men eftersom det har uppstått vid ungefär samma tidpunkt som HTML5 brukar det nämnas [6] [7].

## Web sockets

Tvåvägskommunikation i webbapplikationer har traditionellt sätt varit klumpigt att använda. Eftersom en HTTP-server inte kan skicka egna förfrågningar till en webbläsare, måste webbläsaren göra förfrågningar till servern med jämna mellanrum. Detta ger mycket onödig trafik, eftersom HTTP-fält alltid är med i både klientens förfrågan och serverns svar. Det går att kringgå detta genom att använda sig av *HTTP long polling* och *HTTP streaming*, men dessa tekniker följer inte helt HTTP-specifikationerna [8].

Med web sockets får webbläsaren, och därmed webbapplikationer, möjlighet att använda tvåvägs *TCP-sockets* för kommunikation. Protokollet är för tillfället kompatibelt med HTTP, och passerar därmed fritt genom brandväggar och HTTP-proxyservrar. Web sockets är inte heller bundet till endast HTTP-klienter och -servrar, utan kan användas av vilka klienter och servrar som helst [9] [10].

# Säkerhetsbrister i HTML

Säkerhetsbrister i webbapplikationer var tidigare inte ett så stort problem, för användare kunde vanligtvis bara hämta information. Nu för tiden kan användarna av en webbapplikation dela med sig av information även till andra användare, och om detta informationsflöde inte kontrolleras kan skadlig data spridas från en användare till en annan. Det här kapitlet behandlar de vanligaste säkerhetsbristerna som kan finnas i en webbapplikation, och hur de påverkas i övergången till HTML5.

## Webbkodsinjektion

Webbkodsinjektion innebär att en attackerare utnyttjar säkerhetshål i en webbapplikation för att injicera skadliga HTML-taggar som sedan visas åt andra besökare. Eftersom denna kod visas på den attackerade webbsidan kan den ändra på hemsidans struktur och till exempel skicka användarnas formulärdata och autentiseringskakor till attackeraren [11].

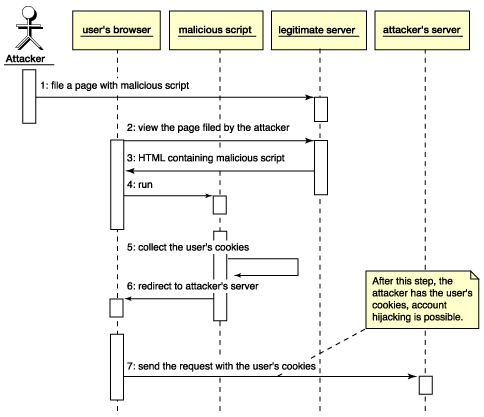
I nedanstående bild visas ett exempel på hur en webbkodsinjektionsattack fungerar. Attackeraren injicerar skadlig webbkod på en legitim sida (1). En inloggad besökare visar denna sida(2), och besökarens laddar ner (3) och kör webbkoden(4). Webbkoden samlar ihop användarens kakor(5) och skickar dessa till attackerarens server (6-7). Efter detta är det möjligt för attackeraren att ta över besökarens konto. 

Bild [12]

Den injicerade koden kan endera omges med *script*-taggar, eller så kan attribut till andra html-taggar användas. Följande kod visar exempel på olika attackvektorer. Koden *alert(document.cookie)* brukar användas för att demonstrera webbkodsinjektion, eftersom den visar sidans kakor i ett dialogfönster. Syftet med att använda just denna kod är att om man kan visa sidans kakor i ett dialogfönster kan en attackerare troligtvis också skicka dem till sin server.

<script>alert(document.cookie)</script>  
<body onload=alert(document.cookie)>  
<b onmouseover=alert(document.cookie)>click me!</b>  
<img src=http://existerar.ej onerror=alert(document.cookie);>

Första raden visar hur webbkodsinjektion fungerar på enklaste sätt, men detta fungerar inte om < och > filtreras bort. Rad 2 visar hur den skadliga koden har lagts till som värde i ett *onload*-attribut. Detta gör att koden körs automatiskt när taggen laddas i webbläsaren. Rad 3 visar motsvarande injektion som rad 2, men här körs koden istället när muspekaren befinner sig ovanför elementet i fråga. Rad 4 visar hur en *img*-tagg automatiskt kan köra kod. Genom att specificera attributet *src* till en ogiltig bildlänk kommer koden i attributet *onerror* att köras. Dessa exempel visar att det inte går att blockera enskilda attribut eller HTML-taggar för att undvika webbkodsinjektion [13].

Webbkodsinjektion kan delas upp i tre kategorier [11]:  
1. Icke-bestående webbkodsinjektion  
Den injicerade koden sparas aldrig på servern, utan kommer till exempel i den besökta sidans länk. Koden når endast de besökare som anländer via just den länken. Ett exempel på detta är sökfunktioner som också visar vilken term man sökt efter.

2. Bestående webbkodsinjektion  
Koden sparas på servern, och man behöver ej klicka på en speciell länk för att se denna. Ofta visas koden också för alla webbsidans besökare. Ett exempel på detta är kommenteringsfunktioner på webbapplikationer så som bloggar.

3. DOM-baserad webbkodsinjektion  
I denna kategori kommer den injicerade koden aldrig att skickas över nätverket, istället så används redan existerande webbkod för att skriva ut den skadliga koden från exempelvis webbadressen.  
Exempel: I webbsidan finns koden

<SCRIPT>  
var pos=document.URL.indexOf("name=")+5; document.write(document.URL.substring(pos,document.URL.length));  
</SCRIPT>

Normalt sett skulle denna kod användas för att välkomna användaren:  
<http://www.vulnerable.site/welcome.html?name=Joe>  
Men skulle också kunna användas för kodinjektion:  
[http://www.vulnerable.site/welcome.html?name=<script>alert(document.cookie)</script](http://www.vulnerable.site/welcome.html?name=%3cscript%3ealert(document.cookie)%3c/script)>

Faktum är att koden som injiceras behöver aldrig skickas till servern [14].

### Webbkodsinjektion i HTML5

I HTML5-standarden har många nya taggar introducerats. En del av dessa används för media, såsom *audio* för att spela upp ljud och *video* för att spela upp videofiler. Andra taggar har en semantisk innebörd, som exempelvis *header* och *article*. Detta gör att webbläsaren har en bättre förståelse för vilket innehåll som är viktigt på en webbsida [15].

Fastän dessa taggar inte direkt påverkar webbkodsinjektion, så har de ändå en betydelse för filter som försöker filtrera bort skadlig webbkod. En webbapplikation som tar emot text från användarna kan exempelvis vilja att användarna skall kunna använda HTML för att strukturera upp sin text, men däremot bör de inte få köra webbkod. Att endast radera *script*-taggar fungerar inte, eftersom det går att köra webbkod också i attribut.

## Cross-site request forgery (CSRF)

Med CSRF menas förfalskandet av förfrågningar mellan webbapplikationer. Det är frågan om webbkod som skickar en oönskad förfrågan från en sida till en annan. Som exempel kan vi ta en legitim webbapplikation, där användaren är inloggad. Applikationen har ett formulär för att skapa nya användare, och skickar data från formuläret till servern. En attackerare kan skapa en färdigt ifylld kopia av formuläret på sin webbsida, som dessutom kommer att skickas automatiskt. Om användaren luras att besöka sidan med det falska formuläret kommer dennes webbläsare att skicka in formuläret till webbapplikationen, och därmed skapa en användare [16].

Exempelkod för detta skulle se ut ungefär så här på den legitima sidan:

<form action="/url/profile.php" method="post">  
<input type="text" name="username"/>  
<input type="text" name="password"/>  
<input type="submit" name="submit" value="Update"/>  
</form>

Koden definierar ett textfält vardera för användarnamn och lösenord, samt en knapp för att skicka in formuläret.

Motsvarande CSRF-kod på attackerarens sida skulle kunna se ut såhär:

<body onload="javascript:form.submit();">  
<form action="http://victim.example.com/profile.php" id="form" method="post">  
<input type="hidden" name="username" value="hacker">  
<input type="hidden" name="password" value="hacker123">  
</form>

Eftersom alla *input*-taggarna har attributet *type* specifierat till *hidden*, kommer ingenting att synas för användaren. I *body*-taggens attribut *onload* sätts koden som skickar formuläret automatiskt. I detta fall skulle användaren märka detta, eftersom han eller hon blir vidarebefordrad till den legitima sidan. Förfrågningen går också att köra i bakgrunden, exempelvis genom att visa koden i en ram [16].

Denna attack går också att kombinera med webbkodsinjektion. Då kan funktionen *XMLHttpRequest* användas, vilken är speciellt gjord för att köra förfrågningar i bakgrunden. Dock kan man endast skicka förfrågningar till samma domän som själva koden finns på, tack vare *same origin policy* [17].

### CSRF i HTML5

CSRF fungerar på samma sätt även i HTML5. Ett par funktioner har dock tillkommit, vilket gör att webbläsarna kan utnyttjas på nya sätt.

Om webbkod på domän A tidigare försökt skicka en förfrågan till domän B har detta stoppats på grund av *same origin policy*, vilket innebär att förfrågningar endast skall kunna skickas till samma domän som webbkoden finns.

I HTML5 finns en ny tillämpning vid namn CORS (eng. *Cross Origin Resource Sharing*) [18] som ändrar på detta. Denna möjliggör att webbkod kan skicka förfrågningar till andra domäner, förutsatt att de tillåter detta. Det finns två sorts förfrågningar, enkla förfrågningar (eng. *simple request*) och egentliga förfrågningar (eng. *actual request*). Till enkla förfrågningar hör alla förfrågningar som man kan få webbläsaren att utföra även utan CORS, såsom GET, POST och HEAD. Övriga mera avancerade förfrågningar hör till kategorin egentliga förfrågningar.

Om webbkod på domän A försöker utföra en egentlig förfrågan mot domän B skickar webbläsaren först en *preflight*-förfrågan till domän B. Denna förfrågan skickar med HTTP-fälten(eng. *HTTP-headers*) *Origin*, *Access-Control-Request-Method* och *Access-Control-Request-Headers* (*ACRM* och *ACRH*). Med ACRM specificeras förfrågningsmetoden, det vill säga HEAD, GET eller POST. Med ACRH specificeras eventuella egna HTTP-fält man vill använda. Origin är domänen förfrågningen kommer från, i detta fall ”A”. Servern svarar med tre HTTP-fält; *Access-Control-Allow-Origin*, *Access-Control-Allow-Methods* och *Access-Control-Allow-Headers*. Dessa är motsvarande de i förfrågningen, men berättar istället vilka värden domän B accepterar. Om de specificerade värdena motsvarar de tillåtna värdena skickar webbläsaren sedan den riktiga förfrågningen.

Exempel på *pre-flight*-förfrågning:

Origin: <http://A.com>  
Access-Control-Request-Method: POST  
Access-Control-Request-Headers: X-special

Samt serverns svar.

Access-Control-Allow-Origin: <http://A.com>  
Access-Control-Allow-Methods: GET, POST  
Access-Control-Allow-Headers: X-special

Detta exempel visar alltså vad som skulle skickas före en egentlig förfrågan. En enkel förfrågan skickar ingen pre-flight-förfrågan, utan skickar riktiga förfrågningen på en gång, och ser först i efterhand om den lyckades [18].

En server som inte är konfigurerad för CORS, och därmed inte returnerar de rätta HTTP-fälten, borde i teorin inte påverkas säkerhetsmässigt av den här nya tekniken. På samma sätt borde servrar som är rätt konfigurerade vara skyddade från CSRF, eftersom servern endast accepterar på förhand bestämda domäner. Faktum är att när domän A skapar en CORS-förfrågan till domän B och webbläsaren skickar en förfrågan så går den verkligen fram, det är endast svaret på förfrågan som inte domän A har rätt att se. För att använda exemplet från förra kapitlet; domän A kan exempelvis skicka en POST-förfrågan som skapar en ny användare till domän B, och användaren skapas. A kan ändå inte läsa svaret, och faktiskt veta om det lyckades. Detta är självklart en säkerhetsrisk, men det är ändå ingenting som man inte kunnat göra tidigare (kapitel 4.2) [19].

## Klicknappning

Klicknappning (eng. *clickjacking*) är ett förhållandevis nytt hot, som bygger på CSS. CSS ger möjlighet att bygga HTML-sidor i 3 dimensioner med hjälp av *x-*, *y-* och *z-index*. På skärmen syns sedan endast två dimensioner, och objekt överlappar varandra beroende på vilket z-index de har. Attackeraren gör en webbsida som har något oskyldigt i det understa lagret, och är således det som syns åt användaren. När användaren försöker klicka på det här understa lagret så klickar denne egentligen på det osynliga översta lagret. Detta kan exempelvis vara en ram med en annan webbsida [20].

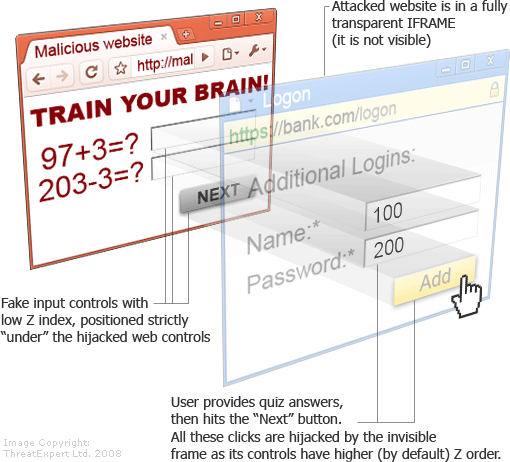


Bild [21]

I ovanstående bild ser vi ett exempel på hur en klicknappningsattack fungerar. Det vänstra fönstrets kontroller ligger under det högra fönstrets kontroller, och således hamnar texten och knapptryckningen i det högra fönstret, fastän allt användaren ser är det vänstra fönstret.

### Klicknappning i HTML5

Klicknappning kan vara lättare att utföra med HTML5. Ett sätt att skydda sig mot detta är att ha webbkod på sin webbsida som gör att sidan bryter sig ut ur ramar. I HTML5 får *iframe*-taggen ett nytt attribut, *sandbox*. Detta attribut bestämmer huruvida en ram skall tillåtas att köra webbkod [22]. I detta fall är det negativt, eftersom en attackerare kan använda detta för att motarbeta utbrytningskoden. Detta behandlas närmare i kapitel 5.4.

Traditionellt sätt har klicknappning närmast använts för att få en användare att klicka på saker denne egentligen inte vill klicka på. Det här fungerar bra för exempelvis dialogfönster och säkerhetsinställningar, men det fungerar mindre bra när användaren måste mata in text. Det är egentligen inte en teknisk fråga, utan en ”social”. Det är enkelt att få en användare att klicka på något, men det är svårt att få en användare att skriva in känslig information.

I HTML5 finns stöd för dra-och-släpp-funktionalitet (eng. *drag-and-drop*), det vill säga att en användare kan dra ett element från en plats till en annan. Detta kan vara till exempel text som man drar till en textinmatningsruta. Dra-och-släpp-funktionaliteten i samband med klicknappning kan också utnyttjas för att få användaren att ofrivilligt dela med sig av privat information från en webbsida till en annan. Detta genom att i en gömd ram visa en sida med privat information, exempelvis Facebook, och i en annan gömd ram visa exempelvis ett formulär för att beställa en produkt. Användaren kan då luras att dra och släppa sitt namn, adress och dylik information från den ena ramen till den andra, för att slutligen beställa hem en produkt åt sig, utan att en enda gång behövt skriva in någon privat information [19].

## Säkerhetsbrister som har tillkommit i HTML5

Eftersom HTML5 bjuder på många nya funktioner ökar också risken för att dessa skall missbrukas. Exempelvis måste filter för webbkodsinjektion uppdateras för att filtrera bort nya html-taggar och -attribut.

HTML5 har också möjlighet att påverka webbläsarens sidhistorik, det vill säga vilken sida man kommer till om man använder tillbakaknappen. Det går inte att lägga till nya domännamn, endast nya länkar till samma domän som webkoden körs på. Det här kan ändå utnyttjas exempelvis för att skapa en sida man inte kan backa tillbaka från. Följande kod [19] demonstrerar hur de senaste 20 besökta sidorna byts till ”NoEscapeFromThisPage.html”. Om även denna sida innehåller koden så adderas ytterligare 20 sidor till historiken.

<script>  
for(int i=0;i<=20;i++){  
history.pushState({}, ””, ”/NoEscapeFromThisPage.html”);  
}  
</script>

HTML5 innehåller också Web notifications, som tillåter att webbsidor skapar ”popups” utanför webbläsaren, precis som vanliga program. Dessa kan också utformas med html-kod. Detta är ingen direkt säkerhetsrisk, men kan underlätta för en attackerare att härma systemprogram och lura användare till att t.ex. klicka på länkar som leder till illvilliga webbsidor [19].

# Skydd mot säkerhetsbrister

I det här kapitlet demonstreras hur man som programmerare av en webbapplikation bör skydda sig för att undvika säkerhetshål som påverkar användarna. Endast säkerhetshål som gör att legitima användares webbläsare missköter sig tas upp, däremot behandlas inte säkerhetshål som endast finns på serversidan.

## Slumpmässigt attribut

För att skydda sig mot CSRF finns ett flertal alternativ. Det rekommenderade förfaringssättet är att i varje formulär ha ett gömt attribut (eng. *CSRF token*). Servern skapar detta attribut som en slumpmässig textsträng, och kontrollerar att detta inte har ändrats när formuläret skickas in. Attributet kan genereras för varje session, det vill säga att varje gång en användare loggar in så fungerar samma attribut tills dess att användaren loggar ut igen. Man kan också låta generera ett nytt attribut för varje formulär. Detta gör att om man använder webbläsarens bakåtknapp och försöker skicka in ett tidigare skapat formulär kommer det inte att lyckas [23].

## Undvika webbkodsinjektion

Om man måste inkludera data från en användare i sin webbapplikation måste man se till att inte introducera webbkod. Detta kan göras på flertalet sätt, men det viktigaste är att man endast tillåter de saker man vet man vill tillåta, så kallad vitlistning (eng. *whitelist*). The Open Web Application Security Project (*OWASP*) har gjort en lista med regler på hur man framgångsrikt skyddar sig från webbkodsinjektion [24]. De viktigaste reglerna presenteras nedan. Notera att dessa principer gäller för både bestående och icke-bestående webbkodsinjektion, men inte för DOM-baserad webbkodsinjektion.

Regel 0: Användardata bör endast införas på de ställen skaderisken är minst. Platser man speciellt skall undvika är exempelvis dessa:

1. Inuti <script>-taggar
2. Inuti HTML-kommentarer
3. Som attribut
4. Som en ny HTML-tagg
5. Som CSS-kod.

Regel 1: Om användardata läggs inuti ”normala” HTML-element räcker det att HTML-koda den. Grundregeln är att det räcker med att koda följande tecken som sina hexadecimala motsvarigheter.

& 🡪 &amp;  
< 🡪 &lt;  
> 🡪 &gt;  
" 🡪 &quot;  
' 🡪 &#x27;  
/ 🡪 &#x2F;

För att följa vitlistningsmetoden bör man ändå koda om alla icke-alfanumeriska tecken användaren skickar in. I de flesta programmeringsspråk finns färdiga bibliotek för detta.

Regel 2: Om man måste använda användardata som värde till en HTML-taggs attribut gäller en något bredare regel. Om man har följt rådet i förra regeln att koda om alla icke-alfanumeriska tecken är man här också säker. Denna regel gäller endast säkra attribut. Till säkra attribut räknas följande:

align alink alt bgcolor border cellpadding cellspacing class color cols colspan coords dir face height hspace ismap lang marginheight marginwidth multiple nohref noresize noshade nowrap ref rel rev rows rowspan scrolling shape span summary tabindex title usemap valign value vlink vspace width

Övriga attribut som exempelvis href, src, style och onclick kräver strängare filtrering, eftersom de är mera komplicerade.

Detta är i stort sett de regler man behöver för att undvika webbkodsinjektion. Om man har ett behov av placera användardata inuti *script*- eller *style*-taggar finns det många fallgropar som kräver mera regler, men dessa behov kan räknas som undantagsfall.

## HTTPOnly-flagga i kakor

Ett enkelt sätt att skydda sig mot viss webbkodsinjektion är att sätta flaggan *HTTPOnly* i webbapplikationens kakor. Denna gör att webbkod inte får tillgång till webbsidans kakor via variabeln *document.cookie*. Kakorna skickas ändå i HTTP-förfrågningarna, så att servern kan hantera dem som vanligt. Detta är ändå inte någon magisk lösning för alla problem, utan man måste ändå se till att webbkod inte kan introduceras av användarna [25].

## Klicknappnings-skydd

För att skydda en webbsida mot klicknappning måste man undvika att den kan visas i en ram. För webbläsare som har implementerat HTTP-fältet *X-Frame-Options* är detta enkelt, eftersom man med detta fält specifierar just huruvida en sida får visas i en ram eller inte. Detta fält kan sättas till ”deny” för att förbjuda att sidan visas i en ram, eller ”sameorigin” för att endast tillåta att sidan visas i en ram på samma domän [26].

För äldre webbläsare fungerar detta inte. Därför bör man också inkludera webbkod på sin sida, som kontrollerar om sidan visas i en ram (eng. *frame-breaking code*). Om sidan visas i en ram bör den på något sätt försöka stoppa visningen. Det finns många olika exempel på dylik kod, men faktum är att många av dem går att komma runt. Den kod som rekommenderas av OWASP för tillfället är följande:

<style id="antiClickjack">body{display:none !important;}</style>  
<script type="text/javascript">  
if (self === top) {  
var antiClickjack = document.getElementById("antiClickjack");  
antiClickjack.parentNode.removeChild(antiClickjack);  
} else {  
top.location = self.location;  
}  
</script>

Koden skall läggas till innanför *head*-taggarna. Koden börjar med att definiera sidans innehåll som osynligt. Sedan kontrolleras huruvida sidan visas i en ram eller ett eget fönster. Om sidan visas i ett eget fönster tas den första definitionen bort, det vill säga sidans innehåll blir igen synligt. Om sidan visas i en ram försöker den bryta sig ut ur ramen och visas i ett eget fönster. Skulle detta inte lyckas förblir sidan osynlig [26].

## Skydd från användarens synvinkel

Som användare behöver man inte nödvändigtvis förlita sig på att en legitim webbsida verkligen är säker. Istället finns det flertalet insticksmoduler till webbläsare, som försöker detektera skadlig webbkod och stoppa den. Exempel på detta är Noscript [27] till Firefox, NotScripts [28] till Google Chrome och Browser guard [29] till Internet Explorer.

# Sammanfattning

I och med HTML5 förändras sättet vi kan använda webbsidor och webbapplikationer. Från att man tidigare har varit tvungen att använda insticksmoduler så som Adobe Flash klarar man sig nu med endast webbläsarens förmåga att förstå HTML-kod. Dessutom finns det nu fler möjligheter att dela med sig av data mellan olika webbsidor, vilket är bra från en webbutvecklares synvinkel.

De attacker avhandlingen tar upp förändras inte speciellt mycket av införandet av HTML5. De påverkas alla av de nya funktionerna som tillkommer, men eftersom de nya funktionerna har planerats från en säkerhetssynvinkel så försämras inte det allmänna läget. Exempel på detta är CORS (kapitel 4.2.1) som ger nya anfallsvektorer för CSRF, men som ändå inte tillåter någonting man inte kunnat åstadkomma tidigare.

Klicknappning är en speciellt intressant attack eftersom den är relativt oteknisk. Den utnyttjar det faktum att utvecklingen av nya layoutfunktioner har gått snabbare än säkerhetstestandet. Tyvärr kommer vi antagligen att se mera av motsvarande attacker, eftersom efterfrågan på nya avancerade funktioner är större än efterfrågan på bättre säkerhet.

Det har tillkommit nya funktioner för att skydda sig från vissa attacker. HTTPOnly-flaggan i kakor och HTTP-fältet *X-Frame-Options* är bra exempel på detta. Förhoppningsvis kommer framtidens webbläsare att själv förstå vilken sorts kod som är skadlig och vilken som är ofarlig. I kapitel 5.5 nämns flera insticksmoduler som försöker göra just detta, så troligtvis kommer utvecklingen på det här området att gå snabbt framåt.

**Definitioner:**

**Användare:** Användare är en person eller flera personer som använder en webbläsare för att besöka (legitima) webbsidor och använda webbapplikationer.

**Attackerare:** Attackerare är en eller flera personer som försöker utnyttja säkerhetshål för att få användarnas personliga information eller motsvarande.

**Webbsida:** En sida skriven i HTML, eventuellt med utformning gjord i CSS. Kan också innehålla webbkod och övrig media.

**Webbapplikation:** Med webbapplikation menas en portal gjorda av flera webbsidor. Webbapplikationer har också funktionalitet på serversidan, exempelvis databaser med användarnas information.

**Domän:** I betydelsen domännamn. Används istället för engelskans domain. Det finns flera funktioner som endast fungerar på en och samma domän. Det kan också nämnas att *www.exempel.com* och *admin.exempel.com* är olika domäner.

**Webbadress:** Används istället för förkortningen URL (eng. *Uniform Resource Locator*). En webbadress består av protokollet, domänen samt filnamnet. Exempel: http://exempel.com/sida.html, https://admin.exempel2.com/bild.jpg.

# Källförteckning

|  |  |
| --- | --- |
| [1] | T. Berners-Lee, "RFC 1866 - The text/html Media type," 10 1995. [Elektronisk]. Tillgänglig från: http://tools.ietf.org/html/rfc1866. [Hämtad 22 04 2013]. |
| [2] | F. W. Zammetti, Practical JavaScript, DOM Scripting and Ajax Projects, Apress, 2007. |
| [3] | R. Clark, O. Studholme, C. Murphy and D. Manian, Beginning HTML5 and CSS3, Apress, 2012. |
| [4] | Web Hypertext Application Technology Working Group (WHATWG), "HTML Standard," [Elektronisk]. Tillgänglig från: http://www.whatwg.org/specs/web-apps/current-work/. [Hämtad 01 04 2013]. |
| [5] | W3C, "Offline web applications," 30 05 2008. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2008/NOTE-offline-webapps-20080530/. [Hämtad 04 04 2013]. |
| [6] | M. Pilgrim, "Local Storage - Dive into HTML5," [Elektronisk]. Tillgänglig från: http://diveintohtml5.info/storage.html. [Hämtad 04 04 2013]. |
| [7] | W3C, "Web Storage," 08 12 2011. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2011/CR-webstorage-20111208/. [Hämtad 04 04 2013]. |
| [8] | Internet Engineering Task Force (IETF), "RFC 6202 - Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," 04 2011. [Elektronisk]. Tillgänglig från: http://tools.ietf.org/html/rfc6202. [Hämtad 04 04 2013]. |
| [9] | I. Fette, "RFC-6455 - The WebSocket Protocol," 12 2011. [Elektronisk]. Tillgänglig från: http://tools.ietf.org/html/rfc6455. [Hämtad 01 04 2013]. |
| [10] | W3C, "The WebSocket API," 20 09 2012. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2012/CR-websockets-20120920/. [Hämtad 04 04 2013]. |
| [11] | CWE, "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')," [Elektronisk]. Tillgänglig från: http://cwe.mitre.org/data/definitions/79.html. [Hämtad 22 3 2013]. |
| [12] | P. Lee, "Cross-site-scripting," [Elektronisk]. Tillgänglig från: http://www.ibm.com/developerworks/tivoli/library/s-csscript/. [Hämtad 21 04 2013]. |
| [13] | OWASP, "Cross-site Scripting (XSS) - OWASP," [Elektronisk]. Tillgänglig från: https://www.owasp.org/index.php/XSS. [Hämtad 15 05 2013]. |
| [14] | A. Klein, "DOM Based Cross Site Scripting or XSS of the Third Kind," 07 2005. [Elektronisk]. Tillgänglig från: http://www.webappsec.org/projects/articles/071105.html. [Hämtad 22 03 2013]. |
| [15] | W3C, "HTML5 differences from HTML4," 25 10 2012. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2012/WD-html5-diff-20121025/. [Hämtad 04 07 2013]. |
| [16] | CWE, "CWE-352: Cross-Site Request Forgery (CSRF)," [Elektronisk]. Tillgänglig från: http://cwe.mitre.org/data/definitions/352.html. [Hämtad 22 03 2013]. |
| [17] | W3C, "XMLHttpRequest," 06 12 2012. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2012/WD-XMLHttpRequest-20121206/. [Hämtad 15 05 2013]. |
| [18] | W3C, "Cross-Origin Resource Sharing," 29 01 2013. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2013/CR-cors-20130129/. [Hämtad 15 05 2013]. |
| [19] | R. McArdle, "HTML5 OVERVIEW: A LOOK AT HTML5 ATTACK SCENARIOS," TrendLabs, 2011. |
| [20] | CAPEC, "CAPEC-103: Clickjacking," [Elektronisk]. Tillgänglig från: http://capec.mitre.org/data/definitions/103.html. [Hämtad 03 22 2013]. |
| [21] | S. Shevchenko, "Threatexpert Blog: Fun with Click and Jack," 06 10 2008. [Elektronisk]. Tillgänglig från: http://blog.threatexpert.com/2008/10/fun-with-click-and-jack.html. [Hämtad 07 04 2013]. |
| [22] | W3C, "HTML5 specifications," 17 12 2012. [Elektronisk]. Tillgänglig från: http://www.w3.org/TR/2012/CR-html5-20121217/. [Hämtad 26 02 2013]. |
| [23] | OWASP, "Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet - OWASP," 21 10 2012. [Elektronisk]. Tillgänglig från: https://www.owasp.org/index.php/Cross-Site\_Request\_Forgery\_%28CSRF%29\_Prevention\_Cheat\_Sheet. [Hämtad 04 08 2013]. |
| [24] | OWASP, "XSS (Cross Site Scripting) Prevention Cheat Sheet - OWASP," 15 3 2013. [Elektronisk]. Tillgänglig från: https://www.owasp.org/index.php/XSS\_%28Cross\_Site\_Scripting%29\_Prevention\_Cheat\_Sheet. [Hämtad 04 08 2013]. |
| [25] | Microsoft, "Mitigating Cross-site Scripting With HTTP-only Cookies," [Elektronisk]. Tillgänglig från: http://msdn.microsoft.com/en-us/library/ms533046.aspx. [Hämtad 04 08 2013]. |
| [26] | OWASP, "Clickjacking Defense Cheat Sheet - OWASP," 03 04 2013. [Elektronisk]. Tillgänglig från: https://www.owasp.org/index.php/Clickjacking\_Defense\_Cheat\_Sheet. [Hämtad 08 04 2013]. |
| [27] | "NoScript," InformAction, [Elektronisk]. Tillgänglig från: http://noscript.net/. [Hämtad 04 08 2013]. |
| [28] | E. Wong, "Chrome web store - NotScripts," [Elektronisk]. Tillgänglig från: https://chrome.google.com/webstore/detail/notscripts/odjhifogjcknibkahlpidmdajjpkkcfn. [Hämtad 22 04 2013]. |
| [29] | Trend Micro USA, "Browser Guard 2011 - Trend Micro USA - Trend Micro USA," [Elektronisk]. Tillgänglig från: http://free.antivirus.com/us/browser-guard/index.html. [Hämtad 22 04 2013]. |