Cross Site Scripting - XSS

Tartalom

Bevezető

XSS avagy Cross Site Scripting, CSS rövidítés lenne ésszerű de így össze lehetne keverni a Cascading Style Sheets-el. X-el a megkülönböztetés kedvéért írjuk, és mert angolban a „kereszt” „cross”-ként is hangzik. Rengeteg XSS-el foglalkozó leírást, hosszabb lélegzetvételű white papert találhatunk a neten erről a témáról, de mint minden használatban lévő dolog, ez is évről évre, és napról napra változik. Vannak témák amiket kivesznek belőle, átneveznek és külön esetként kezelik, így pl. a CSRF, amit a következő paperjeim egyikében fogok várható részletezni, és vannak amikkel bővül.
Maga a Cross Site Scripting név nem teljesen fedi le a mögötte lévő tartalmat, mivel a jelentése is mutatja, hogy kereszt-oldal scriptelés, vagyis a használatához, több oldalra, website-ra van szükség. Természetesen ez nem igaz, de látni is fogjuk. A módszer egy szimpla injection fajta, vagyis egy meglévő rendszerbe, egy malicous (kártékony) kódot fecskendezünk be, majd ezt az ártalmas scriptet fogja futtatni a megtámadott fél.

Típusai

Manapság kétféleképpen jellemzi az XSS-esekt a szakma. Vannak a perzisztens és non-perzisztens XSS-ek, vagyis szebben szólva az állandóan látható és a nem állandóan látható XSS-esek. Természetesen száz felé lehet ezenkívül csoportosítani a hibákat és kihasználásuk módját, de egyenlőre maradjunk ezeknél. Mint már gondolhattuk, a perzisztens XSS-esek sokkal veszélyesebbek, mivel az injektálás után a rendszerben maradnak, általában adatbázisba íródnak, és minden oldalletöltéskor megjelennek, még ezzel ellentétben a non-perzisztens XSS-esek általában URL-ekben, vagy nem állandó formátumú kérésekben lapulnak meg, így csak alkalom adtán, rávezetéssel lehet csapdába csalni az áldozatot.

Kihasználhatóság

Az XSS-ek egyszerűségükhöz képest rengeteg veszélyt fenyegetnek. Főleg cookie lopásra használják őket, de phisingtől kezdve minden elképzelhető. Web 2.0-ás portáloknál, szociális hálózatoknál, webmaileknél és hasonló gyakran látogatott websiteoknál cookieban vagy éppen url-ekben található sessionid-khez juthatunk hozzá. CSRF-et használhatunk ki vele és még a lehetőségek száma lassan korlátlan.

Non-Perzisztens XSS

Ezek alá soroljuk mint írtam, azokat a hibákat, amik nem állandóak, így pl. URL-ben közlekednek. Vegyünk egy példát, a dinamizmusnak hála az emberek ellustultak, így mindent próbálnak minél kényelmesebben, gyorsabban megírni, minél több funkcióval. Mindenki látott már olyan oldalakat, ahol ha hiba keletkezik, akkor az embert átdobja a weblap a hiba-oldalra ahol az url-ben megvan adva a hiba, majd azt szebb formában nagyobb betűkkel közli a felhasználónak pl.

http://server.tld/error.php?hiba=Nem%20talalhato%20az%oldal

És a következő szöveggel tér vissza az oldal:

A következő hiba lépett fel:
Nem található az oldal

Gyors szemrevételezés után feltehető, hogy az url-ben megadott szöveget megváltoztatva, a weblapon megjelent tartalom is változik.
Mi történik akkor, ha megváltoztatom erre:

http://server.tld/error.php?hiba=<script>alert(1);</script>

Hoppá, hát igen, ez egy warning típusú ablakban egy 1-est dobott fel, vagyis javascriptet tudtunk injektálni az oldal tartalmába.

Tegyük fel, egy forgalmas oldalon találatunk egy hasonló XSS hibát, amit ki akarunk használni. Úgy írjuk meg, hogy a megnyitása után, a document.cookie javascript változó értékét küldje el a saját weboldalunknak, és a mi scriptünk pedig azt az értéket eltárolja, egy általunk is elérhető helyen. Ezt az url-t elküldjük a rendszeradminisztrátornak, vagy éppen egy jóbarátnak. Az áldozatot megkérjük, hogy lépjen be az oldalra, majd nézze meg az áldalunk összeállított ártalmas kódot. Ha ezt a két lépést megtette, akkor megkaparintottuk a cookieját, amit megadva saját böngészőnknek, máris az ő azonosítójával vagyunk a rendszerben.
Sok helyen a kényelem szempontjából a rendszer megjegyzi a belépés előtt megnyitott címet és eltárolja, vagyis ha valaki egy XSS-t „szeretne” megnézni, bejelentkezés nélkül, akkor a rendszer felajánlja a beléptetést, és beléptetés után rögtön az alapból megnyitott oldalra irányít, mint intelligens és felhasználóbarát weboldal. Ez azt vonja maga után, hogy nincs szükség 2db lépésre az XSS megnyitásánál, hanem elég egy.

Javasolni szokták a támadóknak, hogy használjanak urlencode() és hasonszőrű függvényeket, amivel tovább bonyolíthatják az XSS-ük képet az URL-be, így kevésbé lesz feltűnő az ártalmas kód.

Bonyolultabb non-perzisztens XSS-esek.

XSS veszély sajnos minden olyan fellép, ahol a weboldal a látogató adataival machinál.

Oldalon belül gondoljunk a keresőkre. Bármit beírunk, rákeres és kiírja, hogy talált-e a megadott bementre találatot, vagy sem. A lényeg az, hogy itt is megjeleníti a form-ba írt inputot, így ismét kódot (html+javascript) fecskendezhetünk az oldalba. Ezt már nehezebb eljuttatni az áldozatnak, viszont nem lehetetlen. Erre valók a CSRF támadások többek között.

Hasonlóan XSS-t lehet kiváltani a következők szerkesztgetésével:

- Referer
- User-agent
- Host
- Accept/Accept-Langauge

és így tovább, minden amit a szerver oldali alkalmazás megjelenít.

Perszisztens XSS

Ez a típus sokkal inkább veszélyesebb. Ezek az ártalmas kódok valamely módon eltárolódnak a szerveren, így könnyebb bárkit is rávenni, hogy megnézze.

Két jó példa:
- fórum xss-re hibás, így bárki aki kommentet ír, egy ártalmas kódot is beírhat, ami minden egyes olvasónál megjelenik.
- webmailben xss található, ismerjük az áldozatuk rendszerét, küldünk neki egy HTML levelet XSS tartalommal, amit ha megnéz lefut a kód.

Webmaileknél eszméletlen nagy problémákat okoz ez. SessionId-ket (munkameneteket), jelszavakat, leveleket lehet vele lopni. Jelszó emlékeztetőt, levél aláírásokat, továbbítási szabályokat, szűrő szabályokat megváltoztatni, stb. Ennek is csak a képzelet és a tudás szab határt.

Böngésző-orientált XSS

Sajnos minden böngésző máshogy működik, miden fejlesztő máshogy látja a dolgokat, másban tér el a szabványtól, mást szeretne leegyszerűsíteni és mást lát jónak illetve rossznak. Ebből kifolyólag akadnak problémák és hibák is a böngésző fejlesztés során. Mind a Firefox, mind az Internet Explorer (csak hogy a leggyakrabban használtakat emlegessem, természetesen a többire is igaz) tartalmaz olyan rutinokat, amik olyan helyeken engedik javascript futtatását, ahol átlagos körülmények között sosem gondolnák.

Így CSS (Cascading Style Sheets) tulajdonságokon belül sem várná el sok ember, hogy javascriptet tudjon értelmeztetni a böngészővel, vagy éppen XML-be tároljon egy-két javascript utasítást. A dolog érdekessége, hogy néha böngésző hiba miatt, néha célirányos tervezés eredményeként sikerül ezeket a dolgokat a böngészőbe integrálniuk a készítőknek.
Egy biztonságos webmail a HTML levelek megjelenítésekor figyelembe veszi a benne található tag-eket, az ártalmasokat megszünteti, az ártalmatlanok tulajdonságait és eventjeit átírja/semlegesíti.
Pl. a BODY tag-eket kiszedik, így a SCRIPT, STYLE és még sorolhatnám, hogy miket, és a nélkülözhetetleneket megtartja. A linkelés vagyis az „A”-t is, viszont annak és remélhetőleg a többi megmaradnak is átírja az eventjeit, pl az onclick, onload stb-ket, így egy esetleges nagyobb fejfájástól kíméli meg a felhasználókat.
Sok helyen elfelejtik a pontban írott hibákat és nem számolnak velük. Alapokkal, mint az előbb felsoroltakkal számolnak, de az ínyencségeket nem törlik ki.

A következő például mennyire kártékony lehetne, ha működne:

szép javascript háttér

Rengeteg hasonló dologgal rendelkeznek még manapság is ezek a böngészők, lehet az egyikben működik és a másikban pedig nem.

Karakterkódolásos XSS

Elég érdekes és új keletű volt ez a probléma pár éve, de még ma is megvannak a problémák vele és természetesen lesznek is. A karakterkódolások minden embernek fejfájást okoznak, mert rengeteg létezik belőlük. Legismertebb karakterkódolási probléma ebben a régióban (Magyarország) a latin1-latin2-utf8 kódolások közben megjelenő „?”-kérdőjelek. Természetesen nem csak efajta gondot okozhatnak a karaktertáblák, illetve azoknak a helyes lekezelései.
Na de a lényeg. Változó hosszúságú karaktertáblákban általában létezik egy (vagy több) módosító karakter, amiről tudja az értelmező, hogy az nem egy, hanem több byte-hosszúságban kell értelmezni, ezek után a karakterek után bármit írhatunk, a megjelenítésben és értelmezésben egy karakternek fognak látszódni.

<input type=”text” value=”USERINPUT” /><input type=”submit” />

Ha a USERINPUT helyére mi írhatunk tetszőleges, viszont a szimpla ” szűrve van, akkor mi a teedő? Megnézzük az oldal karakterkódolási követelményét, majd ha változó hosszúságú akkor a karaktertáblából kiválasztjuk a benne lévő 2byte hosszúságú ”-nek megfelelő hexa kódot és megadjuk az oldalnak.
Az oldal a szimpla ” karaktereket szűrni fogja ascii kód szerint, viszont a karakterkódolásban lévő 2 vagy több byteos ugyanezen értelmű karaktert már nem fogja,

mivel az ascii kódja más, illetve hosszabb (2 vagy több ascii kódja van felsorolva).

Transzformációs XSS

Ez egy speciálisabb eset, főleg fejletlen fórumokban, blogokon lehet használni, ahogy saját leírónyelvet akarnak használni a biztonságosság kedvéért a fejlesztők.
Ilyen pl. a BBCode. Bár a jól implementált Bbcode-ban nem találunk ilyen hibákat, viszont a rosszul implementáltakban, vagy a hasonlóan alkalmazott leírónyelvekben esetlegesen találhatunk.
Első probléma, amikor a tag-en belül kell megadni egy értéke, pl. a betű színét, és a ”-ket nem szűrik, ekkor bármit meg lehet tenni a rendszerrel. Eventeket adni a tag-nak ami transzformáció ([] karakterek <>-re cserélése, eredeti tartalom meghagyása) után, esetlegesen új tag-eket létrehozni benne stb.
Egy másik problémát a karakterkódolásos XSS-ekre lehet visszavezetni. Ha egy tag-en belülre megint csak mi adhatunk tetszőleges szöveget, akkor az inputunk utolsó karakterének az oldal karakterkódolásának megfelelő módosító karaktert írjuk, így a bezáró macskaköröm elveszik, majd szabadon garázdálkodhatunk.

[font color=”INPUT”]
INPUT = #FFF/

Legyen / egy módosító karakter, ami a változó hosszúságú karakterkódolásnál jelenti, hogy nem egy byteként kell értelmezni a következő karaktert, hanem 2byteként avagy többként. Ha 2byteként értelmezi, akkor a / módosító karakterrel megy a macskaköröm is.

Védekezés

Ha kellőképpen átláttuk a problémát, akkor saját magunknak is megtudunk határozni egy policy-t a rendszerünkre amivel szűrjük ezeket az injectionokat. Olyan helyeken, ahol nem számolunk azzal, hogy a felhasználó HTML-t akar megjeleníteni, ott elég szimplán a <>”' karaktereket lecserélni. Ehhez a legjobb eszköz php-ban a következő függvény:
htmlentities – http://php.net/htmlentities
Bemenetnek megadjuk az ellenőrizendő stringet, a konverzió típusát amit szeretnénk (ENT_NOQUOTES ajánlott) és végül a karakterkódolás típusát. Ez a függvény extra hasznos változó byte-hosszúságú karakterkódolásoknál, mint pl. az UTF8.

Azokon a weblapokon, ahol számolni kell azzal az eshetőséggel, hogy a felhasználók nem csupán text formátumban szeretnének adatokat feldolgozni, a rendszerprogramozónak kötelessége lekezelni a speciális eseteket. Példának vehetjük manapság a webmaileket,

ahol egy böngészőben nem csak szimpla szöveget, hanem formázott szöveget (HTML és társai) szeretnénk megjeleníteni. Ebben az esetben, nem írhatjuk át a tag-eket és egyéb entitásokat, mert a felhasználó csak egy ömlesztett szöveget látna előtt. Viszont ellenőrizetlenül nem hagyhatjuk az adathalmazokat, mivel a fentebb ismertetett módszereket így ki lehet használni. Megoldás erre, hogy a rendszer az adatokat leellenőrzi, kiveszi a felesleges illetve esetlegesen veszélyes tageket, a maradék tag-eknek a jellemzőit lecseréli, vagy megszünteti.
Erre a strip_tags függvény tökéletesen alkalmas: http://php.net/striptags
A megadott függvényeket meghagyja, a többit eltávolítja. Egy pontos whitelist-et írva tökéletesen meghatározhatóak a meghagyható tagek. Ne feledkezzünk el, mint már írtam az eseményekről és a tulajdonságokról. Ezeket érdemes szűrni, illetve az eventeket elcsúnyítani.
Régi hibaforrás, az hogy a támadó olyan HTML levelet küld, amiben a tagek egymásba vannak építve, és egyszeri szűrés nem segít a problémán. Tegyük fel a rendszer kiszűri a BODY tag-eket, viszont ezt csak egyszer teszi meg.

<BODY onload=”alert(1)”>Hello world!</BODY>

Egyrészt egy közel helyesen működő webmail „onload”-ból „xonload”-ot vagy hasonlót készít, vagyis a böngésző számára értelmezhetetlenné teszi, másrészt törli a BODY tag-et, így semmi nem marad a megadott szövegből, csak hogy „Hello world!”. Akkor van a probléma, ha a rendszer a következő levelet kapja:

<BO<BODY></BODY>DY onl<BODY></BODY>oad=”alert(1)”>Hello world!
</BO<BODY></BODY>DY>

Ellenőrzéskor 3db BODY-/BODY tag-et kiszór, viszont megmarad az előbb említett adat, vagyis betöltődés után lefut a javascript és feldob egy 1-est a warning típusú ablakban.

Ez ellen úgy tehetünk, hogy egy while ciklust írunk, ami addig fut, amíg tiszta nem lesz az adatunk, mindenféle káros tagektől.

Sajnos manapság is majdnem minden webmail allergiás erre a fajta inputra, vagyis szinte mind sebezhető.

Bucsay Balázs – http://www.rycon.hu – earthquake[at]rycon[dot]hu