**Какво е XSS**

Cross Site Scripting (XSS) е атака, която използва уязвимост на приложението и „вмъква“ нежелан код, който се изпълнява в браузъра на крайния потребител. Както методите, така и целите на атаката могат да бъдат най-различни.

Cross Site Scripting атаките са възможни най-вече благодарение на немърливостта на програмиста относно данните идващи от външния свят. За предотвратяването на XSS атаки (а и на други видове атаки) е от първостепенно значение филтрирането на всички данни, които са под влиянието на потребителя. Примерите са много. Това може да бъдат както форми, попълвани от потрбителя, така и връзки включващи GET параметри или други данни, до които потребителя има достъп.

Възможностите пред злодеятеля са не малко. Освен на пръв поглед безобидните атаки, при които се вмъква HTML или CSS код, сериозна опастност представлява и вмъкването на JavaScript. Проблемът е, че изпълнението на зловредния кода на машината на нищо не подозиращия потребител, може осигури на атакуващия пълен достъп до профила на жертвата, а с това и до данните му и самоличността му в интернет пространството.

Грижата за сигуронстта, разбира се, е не само в ръцете на програмистите, създали приложението. Необходимо е потребителят да бъде бдителен и особено внимателен, особено при забелязани промени. Всеизвестно е, че няма 100%-ова зацита от който да е вида атака. За това, за да гарнитира сигурността си в интернет, потребителя е длъжен да спазва минимална „хигиена“.

**Видове атаки**

Можем да разделим видовете XSS атаки на 3 типа, в зависимост от изволзвания подход и типа уязвимост:

Постоянна (persistent)

Това е атака, при която зловредният код се записва на сървъра на уебсайта и се изпълнява в браузъра на всеки посетил страница, показваща това съдържание. Този тип атаки са особено опасни, тъй като потребителят няма как да разбере, че страницата, която ще отвори е компрометирана, а често и след осъществяването на атаката, няма никакви индикации за това.

Обикновенно обект на такиза атаки стават форми за коментари, страници за гости, за изпращане съобщения и като цяло, страници, позволяващи на потребителя да въвежда информация, която в последствие се съхранява на сървъра и се зарежда при всяко зареждане. Тук същинсткият проблем е в липсата на филтрация на данните, както при въвеждането им, така и при представянето им на крайния потребител.

В този случай възможностите са почти неограничени. От безобидно вмъкване на HTML или CSS код, целящо единствено демонстриране на съществуващият проблем, през използването на уязвимостта за вмъкване на реклами или подвеждащи връзки, до кражба на бисквитки, посредством JavaScript код, а с това и достъп до данните и профила на жертвите.

Непостоянни (non-persistent)

Основната разлика с постоянните XSS атаки е, че в този случай кодът, чрез който се извръшва атаката не е съхранен на сървъра на сайта. В най-често срещаният си вариант, непостоянната атака се състои в подвеждането на потребителя да кликне на линк, чиито GET параметри са кампрометирани.

Масова практика в множество уебсайтове е при търсене в сайта или опти да се отвори несъществуваща страница да се покаже съобщение от вида на „Не са намерени резултати съдържащи **\*\*\***“, където \*\*\* е търсеният стринг, например. Проблемът е, че в тази ситуация изключително лесно може да се вмъкне зловреден код посредством GET параметъра на страницата и така потребителят да стане жертва последвайки наглед безобиден линк към иначе добре познат сайт. Все пак в тази ситуация потребителите не са съвсем беззащитни. В тяхна полза е да бъдат внимателни към връзките, на които кликат, особено ако са нови за тях.

DOM-базирани атаки

Това са всички атаки, които по един или друг начин внасят изменения в DOM дървото на уебсайта. Честа практика е уязвимости от този тип да се използват за вмъкването на реклами или провокативни връзки, подтикващи потребителят да кликне на тях да бъде отведен на потенциално опасна страница. Този тип атаки също позволяват вмъкването на JavaScript.

При последните два вида атаки важна роля играе и умението на злосторника да използва подходи от социалната инженерия. Това би му позволило по-лесно да спечели доверието на жертвата си и да я подведе да кликне на „инфектирания“ линк.

Това е само едно възможно разделение на съществуващите методи за Cross site scripting. При едно по-подробно разглеждане на възможните атаки, может да обособим доста повече категории. За по-задълбоченото им изследване ще ги разгледаме под формата на няколко обобщаващи примера.

Вмъкване на HTML и CSS

Това е може би първото което можем да опитаме, за да проверим дали избраният от нас сайт прави някаква филтрация на входните данни. Успешното вмъкване на HTML код във форми за коментари или полета за тръсене и вмъкването му в DOM дървото като валиден код е достатъчно показателно, че уебсайтът е уязвим за XSS атаки и можем да продължим с по-дръзки опити.

Но преди това нека разгледаме един пример.

[Пример 1]

Следният php код е идеалният пример за вмъкване на нефилтрирани потребителски данни на целевата страница. Достатъчно е да заредим адресът index.php?search=<b>XSS атака</b>, за да видим резултата.

В зависимост от целите на атаката, дори тази наглед безобиден проблем може да има сериозни последствия. Да си представим ситуация в, която потребителят въвежда URL адреса на личния си сайт при писане на коментар и този този адрес се показва под формата на връзка до коментара му. Директното постаявена на информацията в сайта през PHP кода, без филтриране, дава възможност на атакуващия да постави произволен HTML код, в това число и *<script>* таг, съдържащ JavaScript код. С това неусетно преминахме към най-обширната тема, а именно, множеството възможности за вмъкване и изпълнение на JavaScript код.

Вмъкване на JavaScript

Има много възможности за това. Освен вече споменатия *<script>* таг, проблем може да бъде, дори и уязвимост, позволяваща манипулация на *style* параметъра на който и да било елемент от DOM дървото. Посредтвом *eval()* също може да се изпълни JavaScript код.

JavaScript-а дава много възможности на атакуващия. Безспорно най-сериозна е опастността от кражба на бисквитки. Не по-малко неприятно би било и директното пренасочване на посетителя към друг URL адрес.

Добавянето на event handler-и също може да е сериозен проблем. Чрез такъв можем например да подменим адреса, към който води коя да е връзка от страницата и да пренасочим нищо не подозиращият потребител към наш адрес. Дори можем да стигнем по-далеч в този пример, ако приемем, че на нашият адрес ние сме създали „огледално“ копие на атакувания сайт и потребителят се окаже жертва на фишинг атака, без дори да предполага за това.

Но да се върнем на примера с коментарите и URL адреса. Тук можем да подходим както при SQL инжекциите, за да вмъкнем нашия „паразитен“ код в сайта „гостоприемник“.

$this\_come\_from\_db['name'] = 'Ваньо';

$this\_come\_from\_db['url'] = 'http://evilhacker.com">Visit the evilhacker\'s home page</a><script>alert(\'You have been attacked\');</script><a style="display: none;" href="#';

echo '<a href="'.$this\_come\_from\_db['url'].'">'.$this\_come\_from\_db['name'].'</a>';

Разбира се, далеч не е необходимо да бъдем така директни и да се набиваме на очи. Подобна уязвимост може да бъде използвана съвсем дискретно от някое SEO, например, за набавянето на dofollow връзки в битката за предни позиции в Google.

Съобразно целите ни обаче може да подходим и доста по- грубо, като най-безцеремонно пренаочим жертвата на нашия „дяволски“ сайт посредством *document.location*:

$this\_come\_from\_db['url'] = '"></a><script>document.location=\'http://evilhacker.com/\';</script><a href="#';

По-горният пример ни дава почти неограничени възможности за промени по външния вид и структурата на сайта, както и поведението му. В JavaScript-а можем да вмъкнем и event handler-и, които да изпълняват код при регистрирано определено действие на потребителя.

Атаки от този тип са възможни най-вече благодарение на безпрепятственото въвеждане на двойни и единични кавички от потребителя. Филтрирането на входните данни за тези и някои други специални символи би свело възможностите за XSS атака до минимум. Кавичките и знаците за по-голямо и по-малко ни дават силата да затваряме тагове по-рано от очакваното и след тях да вмъкнем нашия код, както направихме в двата примера по-рано. Подобен е и проблемът с уязвимостите, които позволяват SQL инжекции. Проверката за наличие на кавички, особено там където не очакваме такива, е жизнено важна за приложението и може да предотврати голяма част от възможните атаки, и то не само XSS.

За да онагледим по-добре какви биха могли да бъдат последствията при директното визуализиране на нефилтрирано потребителско съдържание, нека разгледаме един малко по-обширен пример.

[Пример 2]

Тук са илюстрирани част от проблемите, за които вече стана въпрос. Както се вижда в изходния файл, сме премахнали *X-XSS-Protection* header-а, който по подразбиране Google Chrome прилага в защита на потребителите от XSS атаки. Този header „казва“ на браузъра да използва XSS Auditor-а, чиято цел е да следи за наличие в изходния код на страницата на JavaScript, който присъства и в отговора на сървъра. Това е почти сигурен признак за Cross-site scripting атака, поради което XSS Auditor-а не позволява изпълнението на тази код. В php кода задаваме стойност на *X-XSS-Protection* header-а, за да можем да илюстрираме примера. Но в други браузъри, като например Mozilla Firefox (версия 25.0.1) това не е необходимо.

Още нещо, на коеото е хубаво да обърнем внимание в този пример е полето за email. Както сами можете да се уверите, през него не можем да вмъкнем HTML или JavaScript, който да афектира по някакъв начин на сайта. Причината е във валидацията и филтрирането на входящите данни. Добра практика е винаги първо да проверяваме данните, върху които нямаме пълен контрол дали са от желания от нас вид и в очаквания формат и, второ, да ги филтрираме за нежелани символи като знаците за по-голямо и по-малко, кавички и др. В конкретния пример, прилагането на функцията *htmlspecialchars()* няма особен смисл след проверката за валидност на имейла, тъй като *filter\_var()* не би допуснала подобни символи. По-полезна би била за филтриране на самия коментар, където не можем да забраним използването на кавички или знаци за сравниние.

Що се отнася до бисквитката, която твърдим, че сме откраднали...в примера реално не правим нищо лошо. Реалната кражба става, когато стойността на тази бисквитка достигне до нас и бъде съхранена. И това е по-лесно от колкото изглежда. Достатъчно е само да прехвърлим жертвата към наша страница, последством *document.location*, като подадем за GET параметър и самата бисквитка. От там нататък записването ѝ не предсталява никакъв проблем. За да не изпадаме в такива подробности тук, ето и един адрес, където нагледно е описано как извършим кражбата и съхраним всичката нужна информация:

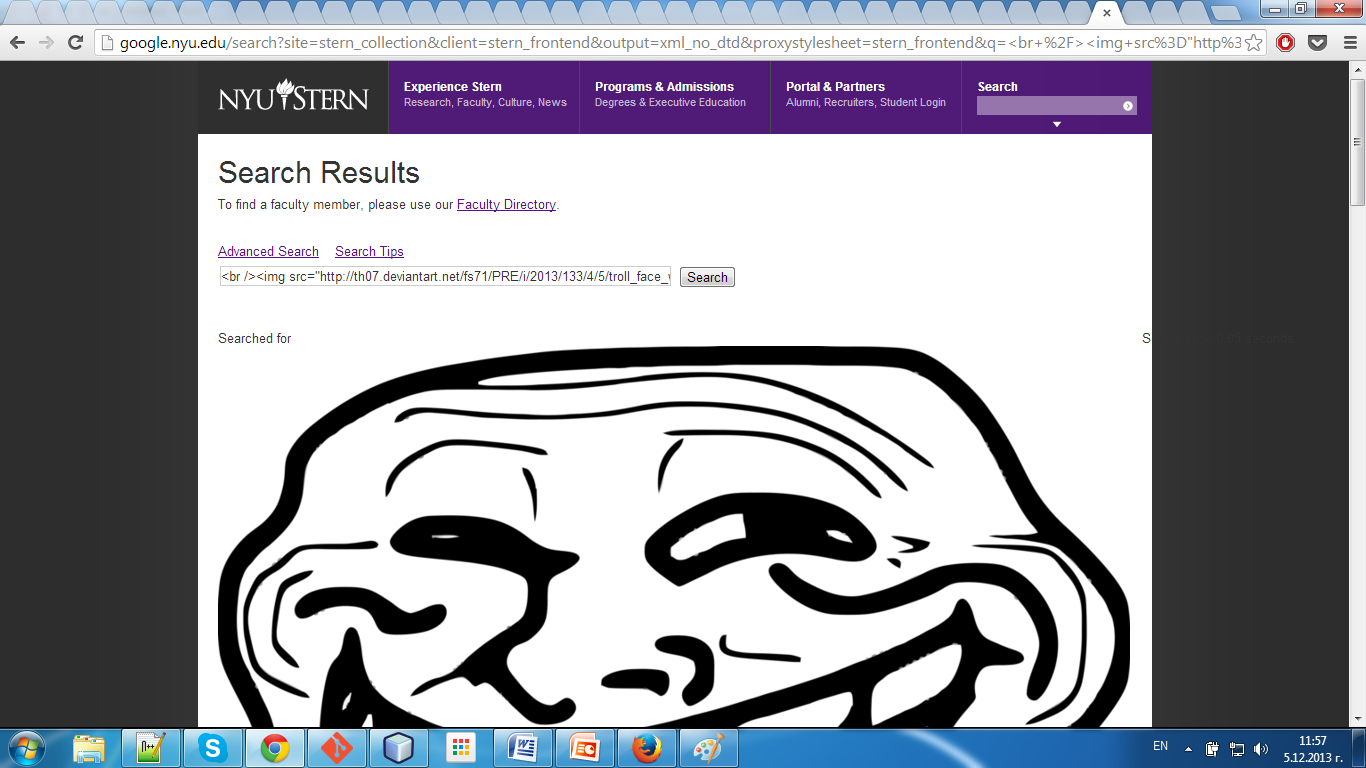
<http://www.go4expert.com/articles/stealing-cookie-xss-t17066/>

Да се върнем малко назад към атаките насочени към DOM дървото на сайта. Както вече споменахме, XSS може да ни послужи не само за кражба на бисквитки и неправомерен достъп до данни. Следващият пример е за това как можем да използваме уязвимост, за да вмъкнем свое съдържание почти незабележимо и без да застрашаваме сигурността на потребителя.

[Пример 3]

Първото, което е необходимо да направим, след като открием уязвимост в даден сайт, е да разгледаме структурата на HTML кода около мястото, където ще се намира нашият код. Целта е да спазим тази структура, като затворим и в последствие отворим всички необходими тагове, като просто направим желаните промени. В примера вмъкнахме един не много дискретен линк. На негово място обаче можем да сложим например наша AdSense реклама, кликовете на която ще носият приходи не на жертвата, а на нас. И, ако сайтът „гостоприемник“ има много посетители, толкова по-добре за нас. Един SEO оптимизатор например би съзрял в това чудестна мъзможност за link building, като просто добави своите връзки. Дори и те да останалт незабелязани от потребителите, googlebot-а няма да ги пропусне.

И, за да не бъда голословен, ще дам един пример от живия живот. Нашата жертва ще бъде сайта на нюйоркския университет *Leonard N. Stern School of Business*.



Тук можете да се поупражнявате в реални условия, но не забравяйте, че всичко е с учебна цел!

**Защита**

И понеже целта на този реферат не е да се превръща в учебник за XSS атаки, а само да илюстрира нагледно как те действат, време е да обърнем внимание и на добрите практики, който ще ни помогнат да не ставаме жертва на Cross-site scripting нито като програмисти, нито като потребители. И все пак, разглежданите примери са жизнено важни за разбирането на необходимостта от спазване на следващите правила. Факт е, че за да можем да се противопоставим на дадна атака, първо трябва добре да разбираме как тя работи. Умението да осъществяваме такива атаки ще ни позволи да погленем на своето приложение и през погледа на неетичния хакер и да бъдем крачка пред него. Нека първо разгледаме от какво се нуждае един уебсайт, за да защити себе си и посетителите си от подобен род атаки.

Както не веднъж споменахме, основната грижа на програмиста е да провери всички данни, които по един или друг начин идват от външния свят и въру които нашето приложение няма контрол. Това са всички GET и POST заявки. Не бива да разчитаме и на информация от id, data или други атрибути на HTML таговете, особено за работа с базата данни. Тази информация, макара и на пръв поглед да се задава от самото приложение, е уязвима и може да бъде лесно манипулирана. За това винаги трябва да имаме предвид какъв точно формат на данните очакваме и да проверяваме дали, това което е идва, не съдържа неподходящи данни.

Нека си припомним втория пример, където успяхме, както да направим структурни промени по DOM дървота на страницата, така и да вмъкнем и изпълним JavaScript код. Основните уязвимости вече ги разгледахме. Следващият пример се бази на същият код, но с вече приложени филтри и проверка на данните.

[Пример 4]

Първото, което виждаме при попълване на формата е филтриране на данните. Най-напред минаваме всички данни през *trim()* фукцията, която премахва символите за празно място в началото и края на стринга. Това няма особено отношение към защита от XSS атаки, но е добра практика за безпроблемното боравене с данните в последствие. Важна в случая е функцията *htmlspecialchars()*. Тя заменя най-опасните за cross-site scripting символи с техните еквивалентни според [ISO-8859-1](http://www.w3schools.com/tags/ref_entities.asp). По този начин символите ще бъдат изобразени от браузъра, но като plain текст и няма бъдат разглеждани като HTML. Става въпрос за следните: **&**, **“**, **‘**, **<** и **>**.

Един често срещан случай, която не е залегнал в примера е, когато очакване число. Най-често това е id на някакъв елемент в базата данни, което получаваме като GET параметър. Може да бъде и някаква целочислена стойност от форма, като година, възраст, брой артикули и т.н. Общото е, че очакваме цяло число. В такъв случай не е необходимо да филтрираме данните по описаната вече схема. Достатъчно е да ги преобразуваме към тип *int*, което автоматично ще премахне всички сивмоли, освен числата. Това става по следния начин:

$id = (int)'$\_GET['id'];

Втората стъпка е валидирането на данните. На този етап трябва да проверим дали информацията е в този формат, в който я очакваме. Най-често срещаният пример е с валидирането на имейл. HTML5 вече ни помага в това отношение, защото когато зададем тип *email* на *input* полето, браузърът автоматично прави проверка за наличието на някои от основните елементи на имейла - @, точка, имена на потребител и хост и др. Все пак е добре да не разчитаме на 100% на това и също да правим проверка. За тази и други често срещани проверки можем да се възползваме от функцията *filter\_var()*. За по-специфични случаи е добре да използваме регулярни изрази.

Изпращането на данни посредством форма трябва да става чрез POST заявка. Използването на GET заявка ще позволи лесното изпращане на данни от трети лица, без дори за отварят страницата с формата, което в никакъв случай не бива да се случва.

Освен, ако по някаква причини не се нуждаем от достъп на JavaScript-а до бисквитките на потребителя, е добре да използваме HttpOnly бисквитки. За целта е достатъчно да добавим следния код в края на бисквитката:

; HttpOnly

По този начин, браузърите поддържажи HttpOnly ще игнорират опитите на JavaScript кода в клиентската част на приложението да получи достъп до бисквитките.

Освен да следваме тези съвети при писане, е добре да хвърляме по едно око и на готовите решения, които използваме, като форуми, книги за гости, системи за коментари и др. Една от масово разпространените грешки в подобни системи е директното показване на въведената от потребителя информация при търсене, което не е върнало резултат или на страници за грешки, като 404 например.

Когато става въпрос за cross-site scripting, на щрек трябва да са както програмисти, така и потребители. Дори и един уебсайт да е вече компрометиран, това не означава, че всеки негов посетител автоматично се превръща в жертва. Някои от най-разпространените атаки, които се извършват чрез отваряне на специфична страница, в която е бил вмъкнат зловредния код, могат да бъдът избегнати, ако потребителите бъдат бдителни и проверяват къде води една връзка преди да я последват. Съмнителни непознати адреси, особено разминаващи се със съпътстващият ги текст, са почти със сигурност „инфектирани“. Често злодеятеля използва прийоми от социалната инженерия, за да подведе жертвата си. Единственият метод за защита в този случай е критичният поглед на потребителя към всяка необичайна връзка, имейл и съобщение, дори и то да е получено на пръв поглед от познат.

Same origin policy е концепция, която работи в полза на крайния потребител и сериозна стъпка в защита от XSS атаки. Тя следва политиката на изпълнение на даден сайт само на JavaScript код произхождащ от същият сайт, проверявайки за целта комбинацията от използваните протокол, име на хост и порт. Така например, ако заредим даден сайт в *iframe* на нашата страница, която е на различен домейн, няма да имаме достъп чрез JavaScript до DOM дървото на сайта в *iframe*-а. Последствие от Same origin policy е и HttpOnly, за което стана въпрос по-рано.

Има и други начини да се защитим от XSS атаки. Забраната на JavaScript и ActiveX от браузъра би ни дала почни 100%-ова защата от по-серизните опасности на cross-site scripting-а, но е практически неприложима в днешно време, когато почти няма сайт, който да не използва JavaScript.