Kapittel 11

Databasesikkerhet

Læringsmål:

SQL-injection er en alvorlig sikkerhetsmessig trussel i webløsninger i dag. Etter å ha jobbet med dette kapittelet skal du

- forstå hvordan *SQL-injection*-angrep virker
- kjenne til ulike teknikker for å hindre *SQL-injection* generelt
- kunne bruke *prepared statements* for å unngå *SQL-injection* og øvrig hacking
- være mer fortrolig med databaseprogrammering og mulighetene med *mysqli*
- forstå hva *hashing* av passord er, og hvordan du kan bruke *hashing* i praksis

11.1 Hva er SQL-injection?

De fleste forbinder ordet hacker med en person som utfører kriminelle handlinger i digitale sammenhenger. Definisjonen på en hacker implisierer strengt tatt ikke noe kriminelt. En hacker er nemlig en person som setter pris på den intellektuelle utfordringen med å utforske systemer, bryte grenser, jobbe seg rundt begrensninger og gjerne programmere lynraske løsninger når det trengs (kilde: Wikipedia). Siden begrepet er såpass negativ ladet, benevnes ondsinnede handlinger eller uønsket digital aktivitet av ymse slag som *hacking* i denne boka.

Sikkerhet er essensielt i en verden hvor digitale ressurser og sensitive data skal være tilgjengelig fra hvor som helst. Nesten alle større nettsteder har et database-system i bunn. Mange nettsteder har et åpent område som er eksponert for hele verden, og et internt område hvor bare spesielt utvalgte har tilgang. Andre nettsteder er bygget opp rundt den personlige opplevelsen hvor hver enkelt besøkende har sin unike visning av informasjon. Dette ser vi for eksempel i sosiale medier som Twitter, Facebook og YouTube. For å få tilgang må brukeren logge seg inn med brukernavn og passord, og deretter vises informasjon som er skreddersydd til hver enkelt. Det er ofte lagret sensitive data for hver enkelt bruker, som kredittkortinformasjon, personlige opplysninger og preferanser.

Hvor alvorlig er det hvis en hacker klarer å logge seg inn som en bruker uten brukerens samtykke? De som utvikler webløsninger, har et stort ansvar for å gjøre preventive tiltak som gjør det vanskelig for hackere. I dette kapittelet skal vi se på et av de mest utbredte fenomenene for angrep, nemlig SQL injection. Det nye API-et *mysqli* (som erstattet det gamle *mysql*) har støtte for såkalte *prepared statements* (parametriserte spørringer er en mulig norsk oversettelse). Det fins mange kodeeksempler på web som gjør bruk av *mysql*-API-et og ikke har gjort noen spesielle tiltak mot *SQL-injection*. Du må unngå slike. Ved å bruke *prepared statements* (og *mysqli*) kan du unngå *SQL-injection*. I dette kapittelet starter vi med å gi deg en forklaring på hva *SQL-injection* er. Deretter får du lære å kode ved hjelp av *prepared statements* for å unngå problemet. Hvis du synes forklaringene knyttet til hva *SQL-injection* er, blir kryptiske, gå heller rett til selve kodingen. Vi kunne selvsagt hoppet over forklaringen og bare sagt «gjør det slik», men som du vil se utover i teksten, er det viktig å få forståelse for nettopp hvorfor *prepared statements* bør brukes. Det er viktig å forstå hva du sikrer deg mot, når du velger å sikre deg.

11.1.1 En databasedrevet innloggingsside

Vi oppretter nå en tabell som kan lagre brukere og passord, med følgende enkle oppbygning:

```
CREATE TABLE brukere(
   id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
   brukernavn VARCHAR(50) NOT NULL,
   passord VARCHAR(255) NOT NULL
);
```

Nøkkelfeltet i databasetabellen er ikke viktig, men tas med for ordens skyld. I første omgang lagrer vi passordene i klartekst for å øke forståelsen av *SQL-injection*. Vi kommer tilbake til tematikken omkring passordsikring i slutten av kapittelet.

Brukernavn og tilhørende passord vil ligge i samme rad. Dette er vist i tabellen nedenfor. Tabeller med brukernavn og passord er vanlig i mange systemer som har innloggingsfunksjonalitet. Et PHP-script må kunne ta seg av innloggingen og foreta passordsjekk opp mot databasetabellen.

Tabell med brukernavn og passord

```
SELECT * FROM brukere

+----+------------+----------+
| id | brukernavn | passord  |
+----+------------+----------+
|  1 | per        | 1234     |
|  2 | line       | shhhj    |
|  3 | sah        | hemmelig |
|  4 | karinor    | j98@KLmK |
|  5 | ola        | ola65    |
+----+------------+----------+
```

11.1.2 Trenger du riktig brukernavn og passord for å logge inn?

Strategien for passordsjekk er enkel. Dersom antall rader er større enn eller lik 1, må det bety at både passordet og brukernavnet ble funnet i tabellen minst én gang og *i samme rad*. Dermed må det være riktig bruker som logger seg inn. I motsatt fall vises en feilmelding. Mange script er laget på nettopp denne måten, og logikken virker (og er) solid.

Scriptet i kodesnutt 11.1 på neste side kan forklares slik: Brukeren får se et skjema med felter for brukernavn og passord som må fylles ut. Hvis innloggingsknappen er trykket, vil *else*-delen av PHP-koden utføres (fordi form action går til samme side). Her kobler vi til databasen, bygger opp en spørring basert på data fra de to tekstfeltene og viser SQL-spørringen på skjermen bare for å se hvordan den ser ut. Deretter utfører vi spørringen, og da kan to ting skje:

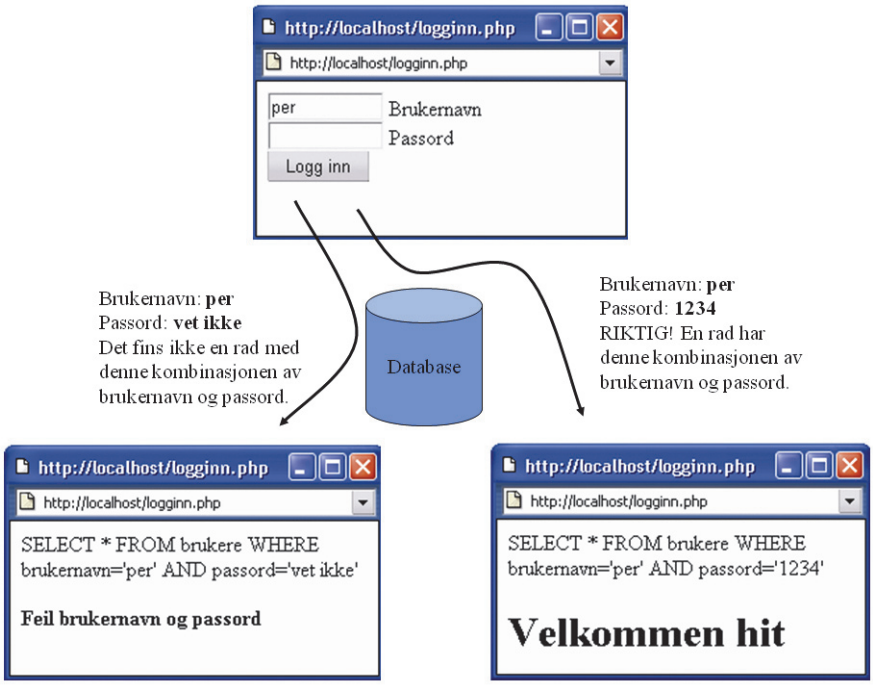
– MySQL finner et resultat, det vil si en rad i databasetabellen «brukere» som oppfyller kriteriet om at både brukernavn *og* passord skal forekomme i raden samtidig. Brukeren får se en velkomsthilsen.
– MySQL finner ikke et resultat, og ingen rader returneres. Det vil si at resultatsettet er tomt og objektvariabelen `num_rows` får tallet 0. Brukeren får en enkel feilmelding.

Kodesnutt 11.1 Skjema for innlogging og passordsjekk opp mot databasen

```
<?php
if ( !isset($_POST['knapp']) ) { //vis skjema
?>
  <form action='' method='post'>
  <input type='text' name='navn' size='35' /> Brukernavn<br />
  <input type='text' name='pass' size='35' /> Passord<br />
  <input type='submit' name='knapp' value='Logg inn' />
  </form>
<?php
}//if
else { //prosesser skjemaet
  //koble til databasen med mysqli
  $db = new mysqli("localhost", "bruker", "pass", "database");
  //lag en spørring basert på utfylt brukernavn og passord
  $sqlsetning = "SELECT * FROM brukere ";
  $sqlsetning .= "WHERE brukernavn='" . $_POST['navn'] . "' ";
  $sqlsetning .= "AND passord='" . $_POST['pass'] . "' ";

//skriv ut setningen bare for testformål
  echo $sqlsetning . "<p>";

  //utfør spørringen
  $resultat = $db->query($sqlsetning);

  //hvis 1 (eller flere) rader, så er brukeren logget inn
  $antall = $resultat->num_rows;
  if ($antall >= 1){
    echo "<h1>Velkommen hit</h1>";
  }
  else{
    echo "<strong>Feil brukernavn og passord</strong>";
  }

  $db->close();
}//ferdig med prosessering, dvs ytre else
?>
```

Scriptet innebærer likevel store muligheter for en hacker. Hvorfor, tror du? Innloggingsskjemaet er vist øverst i figur 11.1.

- Når Per skriver inn sitt brukernavn og feil passord, som vist til venstre, vil Per få beskjed om dette.
- Kun ved riktig kombinasjon av brukernavn og passord blir en rad funnet, med vellykket innlogging som resultat. Dette er vist til høyre.
- SQL-spørringen er skrevet ut for å poengtere at innskrevet brukernavn og passord er utgangspunkt for hvilken spørring som blir utført.



Figur 11.1 Spørringen er basert på innskrevet brukernavn og passord, og PHP-scriptet gir beskjed om feil eller riktig innlogging basert på resultatet.

Sannsynligheten er stor for at Ola Nordmann har ett av disse brukernavnene: «ola», «olanor», «onordmann», «olan» og så videre. Tabellen `brukere` har både brukeren «ola» og brukeren «per». Hvor mange bedrifter har vel ikke en person som heter «Per», ansatt? Kombinert med vanlige navn og kontaktinformasjon oppgitt på hjemmesidene er det i de fleste systemer med brukerdatabaser lett å gjette seg til et eksisterende brukernavn. På nettsteder som Twitter er brukernavnet offentlig. Andre nettsteder kan bruke e-postadressen til innlogging.

Passordet er likevel ukjent, og det er passordet som sikkerheten i slike systemer baserer seg på. Er det nok?

11.1.3 Injisering av SQL-kode

Svaret er «nei». Det er ikke nok å basere seg på at passordet er ukjent. Det er nemlig ikke nødvendig for en hacker å vite passordet for å kunne logge seg inn i det systemet vi har gjennomgått på de foregående sidene, med den koden vi har for å behandle innloggingen. Forutsatt at en klarer å gjette seg til riktig brukernavn, kan en kreativ hacker med litt kjennskap til SQL enkelt formulere et brukernavn, som vist i figur 11.2.

Figur 11.2 Hackeren gjetter at det fins en bruker som heter «ola» («per» hadde også fungert like bra), og legger inn gyldig SQL-kode i tekstfeltet for å lure systemet.

Figuren viser at nøyaktig følgende tegn ble oppført som brukernavn: først en «o», så en «l», en «a», en fnutt, et mellomrom, to bindestreker og et siste mellomrom. Dersom du glemmer det siste mellomrommet, fungerer ikke hack-koden (prøv gjerne lokalt på egen maskin). Hva som skrives i passordfeltet har ingen betydning.

11.1.4 Forklaring til hva som skjer

Her er forklaringen på hvorfor dette fungerer. Spørringen som skal utføres, blir lagret midlertidig i en tekststreng og inneholder bare vanlig tekst. Du husker sikkert at to skråstreker i starten på en PHP-setning angir en kommentar. Hva skjer med kommentarer? De ignoreres når koden kjører. Også i SQL kan vi bruke kommentarer, og det er nettopp det bindestrekene `--` gjør. De angir starten på en SQL-kommentar.

Alt som følger etter kommentaren, ignoreres av databasetjeneren. Fnutten (apostrofen) gir i utgangspunktet hackeren et problem, siden det gir feil dersom tre fnutter brukes i en spørring, men dette løses elegant ved å sette inn en kommentar rett etter. Hva som settes inn av PHP-scriptet etterpå, har ingen betydning lenger.

Passordsjekken kommenteres ut

Når brukeren skriver inn formularet som vist i figuren over, vil SQL-spørringen se slik ut:

```
SELECT * FROM brukere
WHERE brukernavn='ola' -- ' AND passord='hva som helst'
```

Denne spørringen og spørringen under er ekvivalente. Kun ved å gjette riktig brukernavn blir innloggingen vellykket.

```
SELECT * FROM brukere
WHERE brukernavn='ola'
```

Hackeren har med foregående eksempel lykkes med å kommentere ut passordsjekken. Siden det er en bruker som heter «ola» i databasen, vil `SELECT * FROM brukere WHERE brukernavn='ola'` selvsagt gi suksess, med følgende resultat:

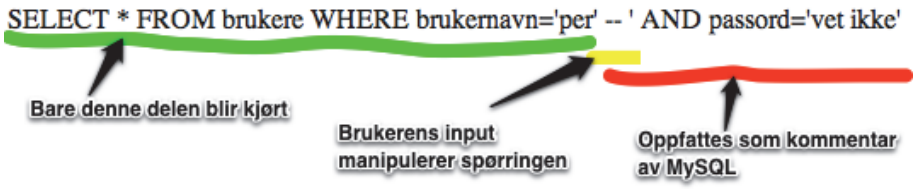
```
+----+------------+---------+
| id | brukernavn | passord |
+----+------------+---------+
|  5 | ola        | ola65   |
+----+------------+---------+
```

PHP-scriptet vil finne ut at resultatsettet består av minst én rad. Hackeren kan logge inn på grunn av programsetningene

```
$antall = $resultat->num_rows;
if ($antall >= 1) echo "<h1>Velkommen hit</h1>";
else echo "<strong>Feil brukernavn og passord</strong>";
```

fra kodesnutt 11.1.

La oss dele opp SQL-spørringen mer visuelt for å tydeliggjøre hva som skjer.



Figur 11.3 Oppdeling av SQL-spørringen viser hvorfor det går galt.

Denne formen for angrep kalles *SQL-injection* (injection betyr injeksjon eller innsprøyting) og er nokså vanlig. Som webutvikler er det derfor særdeles viktig å ta forholdsregler for å unngå slike angrep. Mulige løsninger skal vi snart komme tilbake til, men først skal skadepotensialet utdypes ytterligere. Vi gjentar at du gjerne

må teste dette i praksis på *din egen maskin* (localhost), men vit at det er ulovlig å ødelegge for andre. Målet med gjennomgangen er å motivere til å forebygge, ikke til å volde skade.

Merk: Om du tester dette ut på din maskin og ikke får til å «hacke» deg inn ved å skrive inn `per' --`, så les avsnittet om *magic_quotes* i avsnitt 11.1.5.

11.1.5 En liten, men småviktig notis om magic_quotes

De som står bak PHP som programmeringsspråk, har i mange år prøvd å hjelpe «nybegynnere» til å unngå sikkerhetshull. I forbindelse med *SQL-injection* er et mulig tiltak å «escape» fnutter, som vi skal se senere. Hvis brukeren skriver inn en `'` i et tekstfelt som sendes over via POST eller GET, vil PHP sette inn en `\` automatisk i tillegg til fnutten. I SQL vil nemlig `\` bety at neste tegn ikke skal tolkes (escapes), og resultatet blir da `\'`, hvor fnutten er ufarliggjort.

Hvis brukeren forsøker seg på å skrive inn SQL-kode for å logge seg inn slik:

Figur 11.4 Forsøk på hacking.

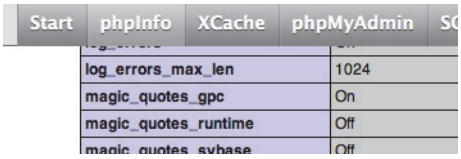
så vil resultatet for noen PHP-tjenere kunne bli slik:

SELECT * FROM brukere WHERE brukernavn='per \'--'
AND passord='hva som helst her'

Feil brukernavn og passord

Figur 11.5 Forsøket på hacking nøytraliseres automagisk, altså uten at programmereren har gjort noen tiltak.

At resultatet blir slik, skyldes at direktivet `magic_quotes_gpc` er slått på i konfigurasjonsfilen til PHP («php.ini»). Dermed setter PHP automatisk inn escaping for å hindre at fnutten tolkes.



Figur 11.6 Skjermbilde fra php.ini-filen i en installasjon av PHP 5.3 (del av pakken MAMP), hvor magic_quotes_gpc er slått på.

Som det står i både PHP-manualen og en rekke steder på Internett, viste det seg at dette sikkerhetstiltaket ikke var særlig lurt. PHP-teamet har derfor bestemt at fra og med PHP 5.4 er direktivet ikke bare slått av, men også fjernet helt.

Dersom du har en eldre versjon enn 5.4.0, står kanskje `magic_quotes_gpc` påslått. Om du bruker pakkeløsninger som WAMP eller MAMP, så er det også en viss sannsynlighet for at det står på. Mange brukere som installerer slike pakker, glemmer å oppdatere dem, og i tillegg henger de som lager pakkene, litt etter med å inkludere siste versjon av PHP. De søker nemlig heller stabilitet enn å ta med det siste nye av PHP-distribusjoner. Det beste er å slå direktivet av eller oppgradere til MAMP/WAMP med nyeste versjon (som forhåpentligvis har det avslått som standard). Det er mye bedre å kode tiltak mot *SQL-injection* selv enn å stole på automatisk håndtering fra PHPs side. Det vil i praksis bety å bruke *prepared statements* og `$db->real_escape_string()`. Vi kommer tilbake til begge disse tingene senere.

11.2 Hvorfor er SQL-injection uheldig?

Eksempelet med passordsystemet som ble hacket, illustrerer hvordan en kreativ utfylling av et tekstfelt kan gi tilgang til et lukket system. Det gir en uautorisert person muligheter til å logge seg inn, men *SQL-injection* kan dessverre brukes til langt mer destruktiv virksomhet.

11.2.1 Tap av data

Det er fint å få nyhetsbrev om siste nytt fra et nettsted, for eksempel IT-nyheter. For å automatisere avmelding av dem som abonnerer på nyhetsbrevet, er det vanlig å la abonnentene melde seg av ved å klikke på en lenke som kan se slik ut:

http://www.nettstedMedNyhetsbrev/slett.php?id=2543

Når siden besøkes, er det feltet `id` som sier nøyaktig hvem som skal slettes fra databasen. Scriptet ***slett.php*** kan nå enkelt og greit foreta sletting av riktig person:

Kodesnutt 11.2 Scriptet som sletter, består trolig av en slik spørring

```
$sql = "DELETE FROM abonnent
        WHERE personid ='" . $_GET['id'] . "'";
$tilkobling->query($sql);
echo "Du vil ikke motta flere nyhetsbrev fra oss";
```

De som har laget denne løsningen, har tenkt på sikkerhet og unngår å avsløre feltnavn fra databasen for omverdenen. Personen som skal slettes, identifiseres ut fra `$_GET['id']`, mens selve feltet i databasetabellen heter `personid`.