Linux平台的ASLR机制

加号减减号 2018-01-30 20:45:43 10175 收藏 10

分类专栏： 技术学习 文章标签： ASLR Linux PIE 开启 关闭

https://blog.csdn.net/Plus\_RE/article/details/79199772

版权

简介

ASLR，全称为 Address Space Layout Randomization，地址空间布局随机化。ASLR 技术在 2005 年的 kernel 2.6.12 中被引入到 Linux 系统，它将进程的某些内存空间地址进行随机化来增大入侵者预测目的地址的难度，从而降低进程被成功入侵的风险。当前 Linux、Windows 等主流操作系统都已经采用该项技术。

具体实现

分级

Linux 平台上 ASLR 分为 0，1，2 三级，用户可以通过一个内核参数 randomize\_va\_space 进行等级控制。它们对应的效果如下：

0：没有随机化。即关闭 ASLR。

1：保留的随机化。共享库、栈、mmap() 以及 VDSO 将被随机化。

2：完全的随机化。在 1 的基础上，通过 brk() 分配的内存空间也将被随机化。

看到这里列出的几项内存空间，很自然有两个地方十分值得注意，第一，代码段以及数据段（data段和bss段）是否被随机化了？第二，堆是否被随机化了？

ASLR 并不负责代码段和数据段的随机化

编写程序进行测试：

#include <stdio.h>

void func();

int uninitialGlobalVar;

int globalVar = 1;

int main(void)

{

int localVar = 1;

printf("Address of func() is %p, in text setment\n", func);

printf("Address of uninitialGlobalVar is %p, in bss segment\n", &uninitialGlobalVar);

printf("Address of globalVar is %p, in data segment\n", &globalVar);

printf("Address of localVar is %p, in stack\n", &localVar);

return 0;

}

void func()

{

;

}

开启完全的ASLR机制，即设置 ASLR 为 2，详细的开启关闭命令后文有介绍。使用如下命令：

sudo bash -c "echo 2 > /proc/sys/kernel/randomize\_va\_space"

1

运行程序，结果如下：

plus@plus:~/Desktop$ ./addr

Address of func() is 0x400595, in text setment

Address of uninitialGlobalVar is 0x601048, in bss segment

Address of globalVar is 0x601040, in data segment

Address of localVar is 0x7ffc353d218c, in stack

plus@plus:~/Desktop$

plus@plus:~/Desktop$ ./addr

Address of func() is 0x400595, in text setment

Address of uninitialGlobalVar is 0x601048, in bss segment

Address of globalVar is 0x601040, in data segment

Address of localVar is 0x7ffc08733eac, in stack

可见栈的地址被随机化了，但是代码段以及数据段的地址均没有发生变化。由此可知，ASLR 并不负责代码段和数据段的随机化工作。

事实上，Linux 平台通过 PIE 机制来负责代码段和数据段的随机化工作，而不是 ASLR。要开启 PIE，在使用 gcc 进行编译链接时添加 -fpie -pie 选项即可：

plus@plus:~/Desktop$ gcc addr.c -fpie -pie -o addr

1

开启 PIE 后，设置 ASLR 为 2，重新运行程序，结果如下：

plus@plus:~/Desktop$ ./addr

Address of func() is 0x564753ff37ee, in text setment

Address of uninitialGlobalVar is 0x5647541f4050, in bss segment

Address of globalVar is 0x5647541f4048, in data segment

Address of localVar is 0x7ffd61090a2c, in stack

plus@plus:~/Desktop$ ./addr

Address of func() is 0x55f9d6af97ee, in text setment

Address of uninitialGlobalVar is 0x55f9d6cfa050, in bss segment

Address of globalVar is 0x55f9d6cfa048, in data segment

Address of localVar is 0x7fffa095687c, in stack

plus@plus:~/Desktop$

可见开启 PIE 之后，代码段以及数据段的地址也被随机化了。

那么还有一个问题，如果没有开启 ASLR，那么 PIE 会生效吗？测试一下便知，开启 PIE 后，设置 ASLR 为 0，运行程序结果如下：

plus@plus:~/Desktop$ ./addr

Address of func() is 0x5555555547ee, in text setment

Address of uninitialGlobalVar is 0x555555755050, in bss segment

Address of globalVar is 0x555555755048, in data segment

Address of localVar is 0x7fffffffde4c, in stack

plus@plus:~/Desktop$ ./addr

Address of func() is 0x5555555547ee, in text setment

Address of uninitialGlobalVar is 0x555555755050, in bss segment

Address of globalVar is 0x555555755048, in data segment

Address of localVar is 0x7fffffffde4c, in stack

plus@plus:~/Desktop$

可以看到，所有地址都没有被随机化。因此得出结论，ASLR 不负责代码段以及数据段的随机化工作，这项工作由 PIE 负责。但是只有在开启 ASLR 之后，PIE 才会生效。

堆的随机化

在 Linux 平台上，堆空间的分配是通过 mmap() 以及 brk() 这两个系统调用完成的。在上面分级效果可以看到，当等级为 1 时，通过 mmap() 分配的堆空间将被随机化，但并不包括 brk() 分配的堆空间。在等级为 2 时，通过 brk() 分配的内存空间也将被随机化，即此时堆空间被完全随机化。

那么什么时候申请的堆空间是通过 mmap() 或者 brk() 分配的呢？我查阅资料后发现，glibc 是选择 mmap() 还是 brk() 进行分配，主要的决定因素是申请的堆内存空间的大小。在 glibc 的 malloc.c 中有一个宏定义常量 M\_MMAP\_THRESHOLD，它的默认值为 128 \* 1024，即 128Kb。在早些时候（或许是 2006 年之前），当申请的内存空间的大小小于 M\_MMAP\_THRESHOLD 时，glibc 将通过 brk() 来进行堆空间的分配，而当申请的内存空间大小大于或者等于 M\_MMAP\_THRESHOLD 时，将选择 mmap() 来负责堆空间的分配。

然而，或许是 2006 年之后，事情变得复杂。在 glibc 2.24 的 malloc.c 中有这样一段注释：

Update in 2006:

The above was written in 2001. Since then the world has changed a lot. Memory got bigger. Applications got bigger. The virtual address space layout in 32 bit linux changed.

In the new situation, brk() and mmap space is shared and there are no artificial limits on brk size imposed by the kernel. What is more, applications have started using transient allocations larger than the 128Kb as was imagined in 2001.

The price for mmap is also high now; each time glibc mmaps from the kernel, the kernel is forced to zero out the memory it gives to the application. Zeroing memory is expensive and eats a lot of cache and memory bandwidth. This has nothing to do with the efficiency of the virtual memory system, by doing mmap the kernel just has no choice but to zero.

In 2001, the kernel had a maximum size for brk() which was about 800 megabytes on 32 bit x86, at that point brk() would hit the first mmaped shared libaries and couldn’t expand anymore. With current 2.6 kernels, the VA space layout is different and brk() and mmap both can span the entire heap at will.

Rather than using a static threshold for the brk/mmap tradeoff, we are now using a simple dynamic one. The goal is still to avoid fragmentation. The old goals we kept are

1) try to get the long lived large allocations to use mmap()

2) really large allocations should always use mmap()

and we’re adding now:

3) transient allocations should use brk() to avoid forcing the kernel having to zero memory over and over again

The implementation works with a sliding threshold, which is by default limited to go between 128Kb and 32Mb (64Mb for 64 bitmachines) and starts out at 128Kb as per the 2001 default.

This allows us to satisfy requirement 1) under the assumption that long lived allocations are made early in the process’ lifespan, before it has started doing dynamic allocations of the same size (which will increase the threshold).

The upperbound on the threshold satisfies requirement 2)

The threshold goes up in value when the application frees memory that was allocated with the mmap allocator. The idea is that once the application starts freeing memory of a certain size, it’s highly probable that this is a size the application uses for transient allocations. This estimator is there to satisfy the new third requirement.

可见，当前 M\_MMAP\_THRESHOLD 这个值的大小是会动态改变的。如果系统认为当前申请的空间不是长期使用，而是短暂使用的，那么即使是大空间，也有可能使用 brk() 进行分配。这里就不详细说了，因为我也还没有理解。但无论如何，当申请的内存空间小于 128Kb 时，系统将通过 brk() 来负责堆空间的分配，这是正确的结论。

写程序测试一下：

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

int \*p128, \*p1k, \*p10k, \*p100k, \*p200k, \*p1mb;

p128 = (int\*)malloc(128);

printf("Address of malloc(128) is %p\n", p128);

p1k = (int\*)malloc(1024);

printf("Address of malloc(1024) is %p\n", p1k);

p10k = (int\*)malloc(10\*1024);

printf("Address of malloc(10\*1024) is %p\n", p10k);

p100k = (int\*)malloc(100\*1024);

printf("Address of malloc(100\*1024) is %p\n", p100k);

p200k = (int\*)malloc(200\*1024);

printf("Address of malloc(200\*1024) is %p\n", p200k);

p1mb = (int\*)malloc(1000\*1024);

printf("Address of malloc(1000\*1024) is %p\n", p1mb);

free(p128);

free(p1k);

free(p10k);

free(p100k);

free(p200k);

free(p1mb);

return 0;

}

开启保留的 ASLR，即设置ASLR等级为1，如下：

sudo bash -c "echo 1 > /proc/sys/kernel/randomize\_va\_space"

1

运行结果如下：

plus@plus:~/Desktop$ ./alloc

Address of malloc(128) is 0x602010

Address of malloc(1024) is 0x6020a0

Address of malloc(10\*1024) is 0x6024b0

Address of malloc(100\*1024) is 0x604cc0

Address of malloc(200\*1024) is 0x7f0a58b13010

Address of malloc(1000\*1024) is 0x7f0a58a18010

plus@plus:~/Desktop$ ./alloc

Address of malloc(128) is 0x602010

Address of malloc(1024) is 0x6020a0

Address of malloc(10\*1024) is 0x6024b0

Address of malloc(100\*1024) is 0x604cc0

Address of malloc(200\*1024) is 0x7fc65800f010

Address of malloc(1000\*1024) is 0x7fc657f14010

plus@plus:~/Desktop$

可以看到，当前 ASLR 等级为 1，当申请空间为 200K 以及 1Mb 时，系统通过 mmap() 分配空间，得到的地址是随机的；而当申请空间为 128、1024、1K 以及 100K（小于 128K）时，系统是通过 brk() 来进行分配的，得到的地址是静止的。

然后开启完全的 ASLR，如下：

sudo bash -c "echo 2 > /proc/sys/kernel/randomize\_va\_space"

1

重新运行程序，结果如下：

plus@plus:~/Desktop$ ./alloc

Address of malloc(128) is 0x1183010

Address of malloc(1024) is 0x11830a0

Address of malloc(10\*1024) is 0x11834b0

Address of malloc(100\*1024) is 0x1185cc0

Address of malloc(200\*1024) is 0x7fd845ed6010

Address of malloc(1000\*1024) is 0x7fd845ddb010

plus@plus:~/Desktop$ ./alloc

Address of malloc(128) is 0x1164010

Address of malloc(1024) is 0x11640a0

Address of malloc(10\*1024) is 0x11644b0

Address of malloc(100\*1024) is 0x1166cc0

Address of malloc(200\*1024) is 0x7f98d3ba1010

Address of malloc(1000\*1024) is 0x7f98d3aa6010

plus@plus:~/Desktop$

可见开启完全的 ASLR 后，通过 brk() 分配的较小的堆空间的地址也被随机化了。

查看/开启/关闭 ASLR

ASLR 的等级可以通过一个内核参数 randomize\_va\_space 来进行控制，查看其值即可知道当前系统的 ASLR 的等级，如下：

cat /proc/sys/kernel/randomize\_va\_space

1

自然地，可以通过更改 randomize\_va\_space 这个内核参数的值来开启或者关闭 ASLR，如下：

0：关闭ASLR

sudo bash -c "echo 0 > /proc/sys/kernel/randomize\_va\_space"

1

1：保留的ASLR

sudo bash -c "echo 1 > /proc/sys/kernel/randomize\_va\_space"

1

2：完全的ASLR

sudo bash -c "echo 2 > /proc/sys/kernel/randomize\_va\_space"

1

应当注意，这只是暂时的更改，重启系统将丢失更改。

bash 脚本

每次输入命令行毕竟很麻烦，于是我写了一个 bash 脚本，分享给大家：

#!/bin/bash

if [ $# == 0 ] # $# means the number of parameters

then

echo 'current ASLR level:'

cat /proc/sys/kernel/randomize\_va\_space

echo 'use option "-h" for help.'

elif [ $# == 1 ]

then

if [ $1 == 0 ]

then

sudo bash -c "echo 0 > /proc/sys/kernel/randomize\_va\_space"

echo "change ASLR level to:"

cat /proc/sys/kernel/randomize\_va\_space

elif [ $1 == 1 ]

then

sudo bash -c "echo 1 > /proc/sys/kernel/randomize\_va\_space"

echo "change ASLR level to:"

cat /proc/sys/kernel/randomize\_va\_space

elif [ $1 == 2 ]

then

sudo bash -c "echo 2 > /proc/sys/kernel/randomize\_va\_space"

echo "change ASLR level to:"

cat /proc/sys/kernel/randomize\_va\_space

elif [ $1 == "-h" ]

then

echo ""

echo "### bash ./ASLR"

echo "--> show current ASLR level."

echo ""

echo "### bash ./ASLR -h"

echo "--> show help info."

echo ""

echo "### bash ./ASLR 0"

echo "--> change ASLR level to 0."

echo ""

echo "### bash ./ASLR 1"

echo "--> change ASLR level to 1."

echo ""

echo "### bash ./ASLR 2"

echo "--> change ASLR level to 2."

echo ""

else

echo "syntax error!"

echo 'use option "-h" for help.'

fi

else

echo "syntax error!"

echo 'use option "-h" for help.'

fi

假设将脚本命名为 ASLR，那么使用时如下输入即可：

bash ./ASLR

1

或者：

bash ./ASLR -h

1

链接

参考资料：https://securityetalii.es/2013/02/03/how-effective-is-aslr-on-linux-systems/

脚本github仓库：https://github.com/plu-s/ASLR\_bash.git

————————————————

版权声明：本文为CSDN博主「加号减减号」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/Plus\_RE/article/details/79199772