# 栈保护（SP）

## 编译参数

-fstack-protector-strong

## 原理

函数调用时会被分配栈空间，通过栈溢出可以被用于攻击，栈保护就是在函数栈尾加上一随机个校验值，当栈被溢出修改时，这个值会改变，当这个值被改时，函数就会调用\_\_stack\_chk\_fail函数。

二进制特征为每个函数开始前都会压栈多一个变量，弹栈时会做一个检查，检查后根据结果跳转至\_\_stack\_chk\_fail。

## 生效条件

1. 编译器支持ssp库。
2. 编译时加入编译选项，编译选项有三级，分别是-fstack-protector，-fstack-protector-strong，-fstack-protector-all，强度不同加入密度也不同。Gcc4.1版本开始支持-fstack-protector和-fstack-protector-all，4.9版本开始支持-fstack-protector-strong。
3. -fstack-protector-all强度最强，只要是存在出口的函数都会添加，如果函数不存在任何出口，比如所有分支都调用exit或者没有任何栈操作，则不会添加栈保护。
4. -fstack-protector-strong与-fstack-protector-all相比，添加条件除了原本-fstack-protector-all的添加条件以外增加了可能出错的栈操作的前提，比如数组操作，指针操作等，存在这些操作的函数会被添加栈保护，没有的话就不会。
5. 而-fstack-protector则是只有在风险较高的栈操作时才会被添加。
6. 当编译时后面添加-fno-stack-protector时，任何栈保护选项都不会生效。

## 例外情况

1．\_\_stack\_chk\_fail是在函数返回之前进行插桩，如果函数没有得到返回，则不会插桩，例如：死循环、return之前有exit等。

2．库由其他库链接而成，SP为编译参数，链接时不生效，为正常结果

3．库如果没有函数，，则不需要添加sp栈保护

4．glibc编译ld.so.new使用自己的编译选项，为内部策略

5．一些简单文件由于没有数组等情况，导致添加stack-protector-strong选项后扫描出问题，可通过修改为stack-protector-all高强度栈保护来扫描确认

## 分析建议

SP选项的例外情况比较多，工具扫描出来的，大部分均为例外情况。实际已经添加了SP选项，具体的分析步骤可如下进行：

1. 使用make V=1，生成详细的编译日志。查看对应文件的编译过程，查看是否有加-fstack-protector-strong，且未加-fno-stack-protector。

注意，此处需要确认的是具体文件的编译情况，而不是编译日志中是否包含-fstack-protector-strong。

例如，对于coreutils-8.31-2.aarch64.rpm\_/coreutils-8.31-2.aarch64.cpio\_/usr/libexec/coreutils/libstdbuf.so文件，其编译日志中包含如下信息：



## 修改示例

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| V2R8\_tar\_binscope/gcc-7.3.0-20190804.h16.eulerosv2r8.aarch64/usr/lib/gcc/aarch64-linux-gnu/7.3.0/libgomp.a(openacc.o) | NO | UNUSED | UNUSED | UNUSED | UNUSED | UNUSED | UNUSED | NO |

上述文件扫描出来，openacc.o的SP选项为NO状态。经分析，openacc.o由源码文件openacc.f90生成，如下图所示。



与openacc.f90对应的Makefile.in文件中的内容如下图所示：





从上图可以看出，设置FCFLAGS，可对相应的\*.f90文件添加SP选项，因此在spec文件中设置FCFLAGS选项。



# 二进制文件地址无关（PIC PIE）

## 编译参数

PIC：

–fPIC（-fpic）

PIE：

-pie –fPIE（-fpie）

## 原理

PIE通过aslr实现函数地址随机化，可执行程序采用地址无关实现，攻击者将很难借助系统中的可执行程序实施攻击。

PIC是通过GOT表来作为引索，从而实现共享库函数地址无关，使得多进程可以共用同一个共享库。

## 生效条件

当前版本的gcc无法在静态链接时生效PIC PIE，高版本可以支持。

## 例外情况

静态链接不生效

## 分析建议

使用binscope扫描工具或ldd命令可直接看到对应文件是否为静态链接。若不为静态链接，则属于未添加的情况，需要分析添加相应的编译选项。

## 修改示例

添加-fPIC –fPIE的分析参考SP选项。

上述选项修改不对会产生一个编译错误(recompile with –fPIC?)。该错误原因是使用了-fpie –fpic，需要使用-fPIC –fPIE。





# GOT表只读（RELRO）

## 编译参数

-Wl,-z,relro,-z,now

## 原理

动态链接共享库时，如果GOT表可以被改写，攻击者可以通过恶意改写攻击。

## 生效条件

当对象为可执行文件时，只有存在链接共享库，且共享库开启地址无关。

## 例外情况

可执行文件只有静态链接时或是链接库无地址无关时，此项选不生效

## 分析建议

该选项除静态链接文件外，大部分扫描出来的异常均为part relro，理论上part relro均可修改为full relro，需要分析修改。

修改示例：

对于gcc包，其libobjc库未加该选项，经分析，可如下修改：

其Makefile.in中的内容为：





修改spec，添加如下：



# BIND\_NOW

## 编译参数

-Wl,-z,now

## 原理

在加载对象时正确解析符号（而不是在使用时，这称为延迟绑定）。

## 生效条件

当对象为可执行文件时，只有存在链接共享库，且共享库开启地址无关。

## 例外情况

可执行文件只有静态链接时或是链接库无地址无关时，此项选不生效。

## 分析建议

该选项与RELRO的要求一致。

# RPATH/RUNPATH

## 编译参数

-Wl,--disable-new-dtags,--rpath [path]

## 原理

rpath选项是链接选项，主要用于防护LD\_LIBRARY\_PATH替换同名动态库的攻击。通过加入此选项可以指定一个运行时动态库搜索的路径，该路径的搜索优先级高于LD\_LIBRARY\_PATH指定的路径。也就是说，可执行文件在运行阶段进行动态库搜索时会首先在--rpath指定的路径查找动态库，然后才会到LD\_LIBRARY\_PATH指定的路径搜索。因此可以有效防御LD\_LIBRARY\_PATH =[attackpath]来替换同名动态库的攻击。

## 生效条件

只要设置了则会有rpath。使用readelf –d xx可查看相关的rpath路径。

## 例外情况

无。

## 分析建议

1. rpath除了使用—rpath显示指定外，libtool工具也会引入。libtool工具引入的rpath直接去除，会导致编译不通过。可以通过chrpath工具去除相应的rpath。
2. 对于/usr/lib、/usr/lib64等LD\_LIBRARY\_PATH也会查找的路径，可直接使用chrpath去除。
3. 对于/usr/lib/systemd这种路径，直接去除会导致程序运行时找不到相应的库。则可以考虑向/etc/ld.so.conf.d中写入xx.conf的配置文件，指明路径。然后去除rpath。

例如如，对于systemd包，其包含的rpath路径均为/usr/lib/systemd。这种路径直接使用chrpath去除，会导致最后找不到相关库的问题。因此，其去除办法如下：







# 附录

安全编译选项扫描工具，由工具部提供。