**10-编译工具**

|  |  |  |  |
| --- | --- | --- | --- |
| **名称** | **所有者** | **修改时间** | **创建时间** |
| [11-camke工具](https://cqbk0n8o1h.feishu.cn/wiki/Z3glwlSCMiLW5VkxVs3csTVonbV) | 李2017 | 2024-05-14 09:34 | 2024-05-14 09:24 |
| [12-Linux-API编程](https://cqbk0n8o1h.feishu.cn/wiki/Skn0wxQpWiYfNukYQu1cglh2nEe) | 李2017 | 2024-05-14 09:26 | 2024-05-14 09:24 |

1. **代码统计工具**

1、cloc

命令行 cloc file

2、vscode counter

2. **编译选项**

编译防止逆向

1. [arm-linux-strip工具使用方法\_arm编译 strip-CSDN博客](https://blog.csdn.net/lansemuju/article/details/8274769)

2.1 **arm-linux-strip**

去掉-g，等于程序做了--strip-debug

strip程序，等于程序做了--strip-debug和--strip-symbol

所以strip后程序会更小点

但对于静态库.a之类的文件，只能用--strip-debug

静态编译就不能使用 strip 了

目的：防止被逆向，增加程序编译安全

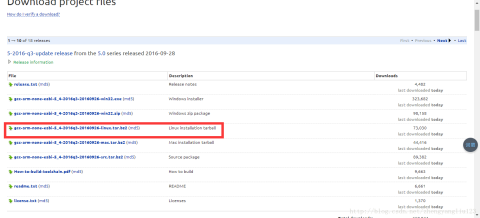
3. **开源工具链**

3.1 **none-eabi**

gcc-arm-none-eabi

https://launchpad.net/gcc-arm-embedded/+download

3.1.1 **手动安装**



/etc/profile文件设置环境变量

|  |
| --- |
| Bash $ sudo gedit /etc/profile |

添加这行在末尾，注意是自己放的路径

|  |
| --- |
| Bash export PATH=$PATH:/home/yangliu/Library/gcc-arm-none-eabi-5\_4-2016q3/bin |

随后点击保存。在终端中输入指令更新系统路径。

|  |
| --- |
| Bash $ source /etc/profile |

或，注意下面这个指令“.”后有空格，其效果与source 指令相同。

|  |
| --- |
| Bash $ . /etc/profile |

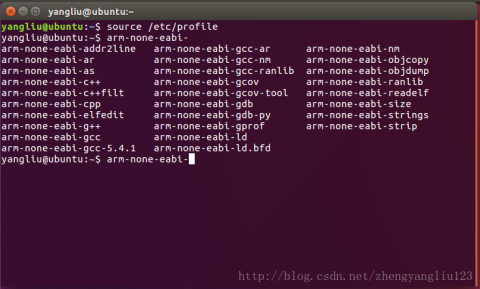
为了查看我们是否已经成功设置路径，在终端中输入：

|  |
| --- |
| Bash $ arm-non |

点击Tab看看系统是否自动补全为：arm-none-eabi-

|  |
| --- |
| Bash $ arm-none-eabi- |

如果补全了，那么说明我们已经设置成功了，我们此时可以直接键入指令来调用工具链的程序。如果没有，说明你上面的环境变量设置步骤有误请重新检查自己的操作。



同时如果你安装的是64位的ubuntu版本，则需要安装lsb-core才能正常使用这个工具链，使用指令：

|  |
| --- |
| Bash $ sudo apt-get install lsb-core |

3.1.2 **直接安装工具**

|  |
| --- |
| Bash sudo apt install make git gcc-arm-none-eabi |

3.2 **linaro工具链**

gcc-arm-linux-gnueabihf

3.2.1 **直接安装工具**

|  |
| --- |
| Bash sudo apt install gcc-arm-linux-gnueabihf |

3.2.2 **示例camke使用**

3.1、cmake使用方法

|  |
| --- |
| Bash #appoint chain SET(CMAKE\_SYSTEM\_NAME Linux) SET(SDKTARGETSYSROOT "/home/toolchain/LINARO/arm-linux-gnueabihf") SET(TOOLCHAIN\_DIR "/home/toolchain/LINARO/bin") set(CMAKE\_CXX\_COMPILER ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf-g++) set(CMAKE\_C\_COMPILER ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf-gcc) set(GNU\_FLAGS "-mfpu=vfp -fPIC") set(CMAKE\_CXX\_FLAGS "${GNU\_FLAGS} ") set(CMAKE\_C\_FLAGS "${GNU\_FLAGS} ") SET(CMAKE\_FIND\_ROOT\_PATH ${TOOLCHAIN\_DIR}  ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf/include  ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf/lib ) <<CMakeLists.txt>> |

东风

|  |
| --- |
| Bash #appoint chain SET(CMAKE\_SYSTEM\_NAME Linux) SET(SDKTARGETSYSROOT "/home/toolchain/AG55YIQI/ql-sysroots") SET(TOOLCHAIN\_DIR "/home/toolchain/AG55YIQI/gcc/usr/bin/arm-oe-linux-gnueabi") set(CMAKE\_CXX\_COMPILER ${TOOLCHAIN\_DIR}/arm-oe-linux-gnueabi-g++) set(CMAKE\_C\_COMPILER ${TOOLCHAIN\_DIR}/arm-oe-linux-gnueabi-gcc)   set(GNU\_FLAGS " -rdynamic -mapcs -Wall -pie -fpie -march=armv7-a -mfpu=neon -mfloat-abi=hard -funwind-tables -ffunction-sections --sysroot=${SDKTARGETSYSROOT}") set(CMAKE\_CXX\_FLAGS "${GNU\_FLAGS} ") set(CMAKE\_C\_FLAGS "${GNU\_FLAGS} ") SET(CMAKE\_FIND\_ROOT\_PATH ${SDKTARGETSYSROOT}  ${SDKTARGETSYSROOT}/usr/include  ${SDKTARGETSYSROOT}/lib ) <<CMakeLists.txt>> |

3.2.3 **工具链环境变量设置**

交叉编译工具主要是设置三个环境变量

PATH， ARCH和CROSS\_COMPILE

3.2.3.1 **永久生效**

|  |
| --- |
| Bash vim ~/.bashrc，改写启动配置文件 export ARCH=arm export CROSS\_COMPILE=arm-buildroot-linux-gnueabihf- export PATH=$PATH:/home/book/100ask\_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf\_sdk-buildroot/bin  source ~/.bashrc |

设置完毕后，要执行 source ~/.bashrc 命令使其生效，这条命令是加载这些设置的环境变量。

3.2.3.2 **临时生效**

手动导入环境变量

|  |
| --- |
| Bash export ARCH=arm export CROSS\_COMPILE=arm-buildroot-linux-gnueabihf- export PATH=$PATH:/home/book/100ask\_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf\_sdk-buildroot/bin |

3.2.3.3 **手动指定**

使用make时指定工具链路径

|  |
| --- |
| Bash export PATH=$PATH:/home/book/100ask\_imx6ull-sdk/ToolChain/arm-buildroot-linux-gnueabihf\_sdk-buildroot/bin make ARCH=arm CROSS\_COMPILE=arm-buildroot-linux-gnueabihf- export PATH=$PATH:/home/book/100ask\_imx6ull-sdk/ToolChain/gcc-linaro-6.2.1-2016.11-x86\_64\_arm-linux-gnueabihf/bin make ARCH=arm CROSS\_COMPILE=arm-linux-gnueabihf- |

3.3 **Poky工具链**

3.4 **交叉工具链理解**

交叉工具链使用分二部分，看Cmake示例

其一，工具链gcc指定，这部分是可以在x86下运行并且编译程序使其生成ARM可以运行程序，看其Cmake示例：

CMAKE\_CXX\_COMPILER 、CMAKE\_C\_COMPILER

其二，在编译过程中需要包含的头文件（这些在ARM下面的库和文件，有些可以从板子上直接拷贝），看其Cmake示例：

SDKTARGETSYSROOT

gcc：基本一致，环境目录设置不一

sysroots：有一点各自添加的库，文件。用编译头文件,库文件引用（开源工具链中arm-linux-gnueabihf）

Cmake示例

|  |
| --- |
| Bash SET(CMAKE\_SYSTEM\_NAME Linux) SET(SDKTARGETSYSROOT "/home/toolchain/LINARO/arm-linux-gnueabihf") SET(TOOLCHAIN\_DIR "/home/toolchain/LINARO/bin") set(CMAKE\_CXX\_COMPILER ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf-g++) set(CMAKE\_C\_COMPILER ${TOOLCHAIN\_DIR}/arm-linux-gnueabihf-gcc) |

3.4.1 **目前工具链分类**

目前我们使用的工具链基本上分以下几类

1、高通的AG35、AG55(工具链基本一样，差别少数的ARM库和头文件)

2、联发科的 64位 aarch64位

3、华为的工具链

4、开源的linaro和none 嵌入式工具链

3.4.2 **交叉编译参数**

./configure --build,--host,--target,--prefix='/path'设置

build：执行代码编译的主机，正常的话就是你的主机系统。这个参数一般由config.guess来猜就可以。当然自己指定也可以。

host：编译出来的二进制程序所执行的主机，因为绝大多数是本机编译，本机执行，所以这个值就等于build。只有交叉编译的时候（也就是本机编译，其他系统机器执行）才会build和host不同。用host指定运行主机。

target：这个选项只有在建立交叉编译环境的时候用到，正常编译和交叉编译都不会用到。他用build主机上的编译器，编译一个新的编译器（binutils, gcc,gdb等），这个新的编译器将来编译出来的其他程序将运行在target指定的系统上。

 prefix：这个参数允许用户定制软件的安装目录，方便软件的管理和使用，以及避免对系统默认软件包的干扰。如果没有使用--prefix指定安装目录，大多数软件包会默认安装到/usr/local目录。

4. **相对路径**

4.1 **运行时获取相对路径**

4.1.1 **函数头**

函数1：

|  |
| --- |
| Bash char \*getcwd(char \*buf, size\_t size); |

头文件：unistd.h

说明：获取到可执行程序的绝对路径，存放到buf中，size是指定的buf大小。若size指定的大小比 buf短，会返回NULL。

若不知道buf的长度，可以设置size为0，则getcwd会调用malloc动态给buf分配空间，不过后续要记得调用free释放buf。

不足：使用后发现，该函数返回的是执行可执行程序的绝对路径。

函数2：

|  |
| --- |
| Bash int readlink(const char \* path ,char \* buf,size\_t bufsiz); |

头文件：unistd.h

说明：该将参数path的符号连接内容存到参数buf所指的内存空间，返回的内容不是以NULL作字符串结尾。

返回值即为字段buf的长度。若传入的参数bufsiz小于符号连接的内容长度，过长的内容会被截断。

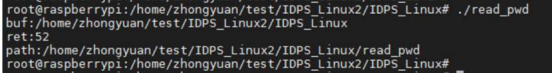
不足：这个函数并不是直接获取程序的绝对路径，是通过获取当前程序的自身信息拿到的路径。

且得到的路径包括了可执行程序名称，需要再处理。

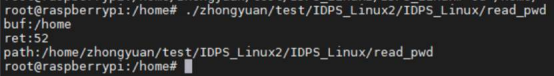
4.1.2 **测试过程**

测试程序源码：

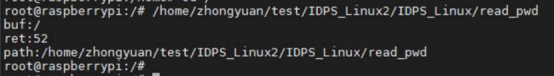
若是在/home/zhongyuan/test/IDPS\_Linux2/IDPS\_Linux路径执行，得到结果



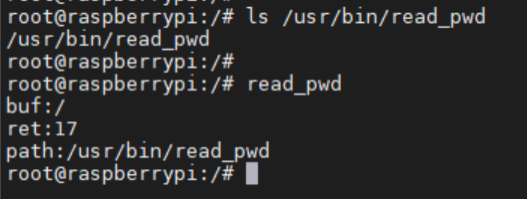
若是在/home路径执行，得到结果



若是在 / 路径执行，得到结果



若将可执行程序放入环境 /usr/bin 目录下



4.1.3 **结论**

在程序中可使用readlink来找到绝对路径，进而得到相对路径。

4.2 **编译时指定动态库的相对路径**

4.2.1 **测试环境**

软件：

代码路径：ssh://git@code.geelib.qihoo.net:10022/IOV\_SecDev/IDPS\_Linux.git

分支：dongfeng\_IDP

4.2.2 **Linux系统下的RPATH**

什么是RPATH

rpath全称是run-time search path。Linux下所有elf格式的文件都包含它，特别是可执行文件。它规定了可执行文件在寻找.so文件时的第一优先位置。

另外，elf文件中还规定了runpath。它与rpath相同，只是优先级低一些。

搜索.so的优先级顺序

RPATH： 写在elf文件中

LD\_LIBRARY\_PATH： 环境变量

RUNPATH： 写在elf文件中

ldconfig的缓存： 配置/etc/ld.conf\*可改变

默认的/lib, /usr/lib

可以看到，RPATH与RUNPATH中间隔着LD\_LIBRARY\_PATH。为了让用户可以通过修改LD\_LIBRARY\_PATH来指定.so文件，大多数编译器都将输出的RPATH留空，并用RUNPATH代替RPATH。

**查看RPATH**

对于任意的elf文件，可以使用$ readelf -d xxx | grep 'R\*PATH'来查看。

结果有两类，一个是RPATH，另一个是RUNPATH。前文也说了，一般情况下，RPATH为空，而RUNPATH不为空。

RPATH中有个特殊的标识符$ORIGIN。这个标识符代表elf文件自身所在的目录。当希望使用相对位置寻找.so文件，就需要利用$ORIGIN设置RPATH。多个路径之间使用冒号:隔开。

**设置RPATH**

1、在gcc中，设置RPATH的办法很简单，就是设置linker的rpath选项：

|  |
| --- |
| Bash gcc -Wl,-rpath,/your/rpath/ test.cpp |

如果需要设置$ORIGIN：

|  |
| --- |
| Bash gcc -Wl,-rpath,'$ORIGIN/lib' test.cpp |

注意，虽然选项里写着RPATH，但它设置的还是RUNPATH。原因在前文有交代。

2、在Makefile中的写法如下：

|  |
| --- |
| Bash CPPFLAGS= '-Wl,-rpath,$$ORIGIN:$$ORIGIN/../lib' |

3、在CMake中，事情则有些不同。由于CMake需要包揽软件安装的事宜，因此CMake使用两个变量来控制RPATH：INSTALL\_RPATH和BUILD\_RPATH。

设置的办法是：

|  |
| --- |
| Bash SET\_TARGET\_PROPERTIES(target PROPERTIES INSTALL\_RPATH "$ORIGIN") |

注意，在CMake中，多个RPATH使用分号隔开，而不是冒号。这是估计是因为冒号在CMake语法中有其他用途.

4、在QMake中，也可以设置RPATH：

|  |
| --- |
| Bash unix:!mac{ QMAKE\_LFLAGS += "-Wl,-rpath,\'\$$ORIGIN/\'" } |

**修改RPATH**

|  |
| --- |
| Bash chrpath -r "/lib64" xxx.so chrpath -r "\$ORIGIN" xxx.so |

4.2.3 **cmake中添加引用动态链接**

编译运行的程序需要链接到程序所在路径下的某些个动态库，为方便移植，必须设置链接库的相对路径，比如./lib等等。

默认在Linux系统下动态库的搜寻路径如下： 使用选项 -Wl,-rpath 在编译时指定；

通过配置LD\_LIBRARY\_PATH 来指定；

在/lib 和 /usr/lib 中查找；

其中第一个在gcc编译选项中添加：-Wl,rpath=xxx 会将rpath路径写入到程序中保存起来。

以下两种方式都可以用来配置rpath路径：

1、使用gcc编译选项： add\_definitions(-std=c++11)

|  |
| --- |
| Bash SET(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb -Wl,-rpath=./:./lib") #-Wl,-rpath=./ SET(CMAKE\_CXX\_FLAGS\_RELEASE "$ENV{CXXFLAGS} -O3 -Wl,-rpath=./:./lib") #-Wall |

2、使用cmake配置 set(CMAKE\_SKIP\_BUILD\_RPATH FALSE)

|  |
| --- |
| Bash set(CMAKE\_BUILD\_WITH\_INSTALL\_RPATH TRUE) set(CMAKE\_INSTALL\_RPATH\_USE\_LINK\_PATH TRUE) set(CMAKE\_INSTALL\_RPATH "./lib") |

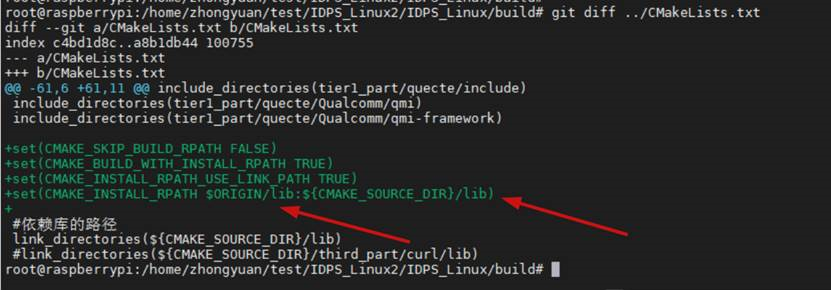
其中RPATH可以使用"./lib"或"./"配置，有可以使用"$ORIGIN/lib"或“\${ORIGIN}/lib”，这里必须加上\符号，否则无法识别。

还可以同时定义多个RPATH，比如：“$ORIGIN:$ORIGIN/lib”，中间使用:分割。

原文链接：https://blog.csdn.net/weixin\_39710041/article/details/111964990

4.2.4 **测试过程**

1、修改CMakeLists.txt

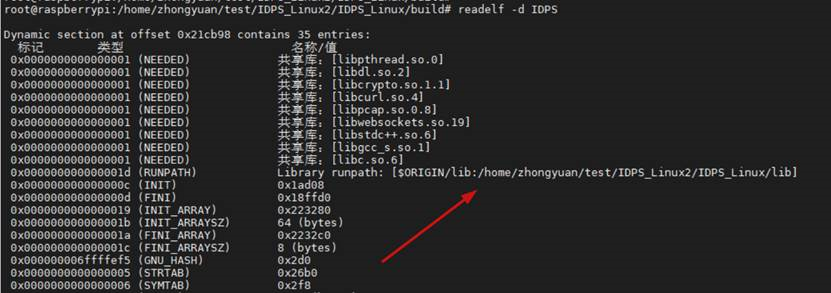


$ORIGIN/lib $ORIGIN这个标识符代表可执行程序自身所在的目录。根据可执行程序的位置变化而变化。

${CMAKE\_SOURCE\_DIR}/lib ${CMAKE\_SOURCE\_DIR}代表cmake的编译目录，固定不变。

2、使用 readelf 查看可执行程序

|  |
| --- |
| Bash readelf -d IDPS |

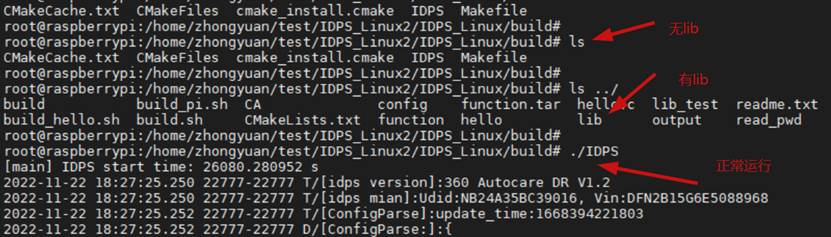


3、删除 保留

删除 $ORIGIN/lib 目录；

保留 ${CMAKE\_SOURCE\_DIR}/lib 目录；

IDPS正常运行。

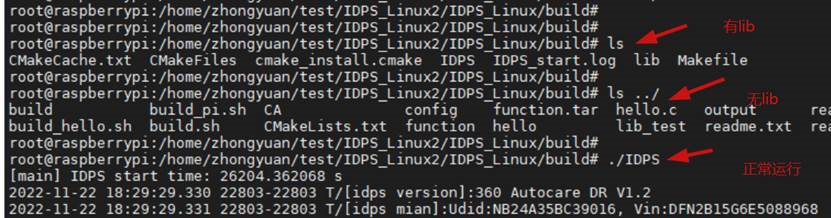


4、保留 删除

保留 $ORIGIN/lib 目录；

删除 ${CMAKE\_SOURCE\_DIR}/lib 目录；

IDPS正常运行。

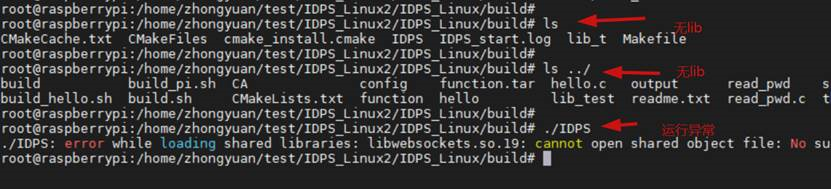


5、删除 删除

删除 $ORIGIN/lib 目录；

删除 ${CMAKE\_SOURCE\_DIR}/lib 目录；

IDPS运行异常。

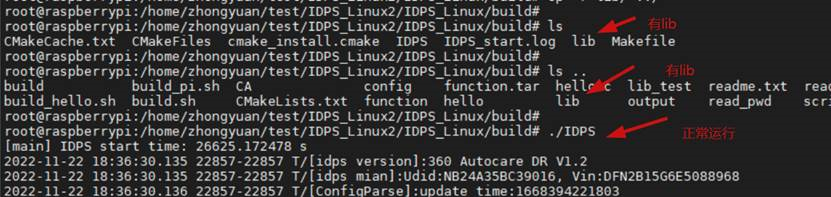


6、保留 保留

保留 $ORIGIN/lib 目录；

保留 ${CMAKE\_SOURCE\_DIR}/lib 目录；

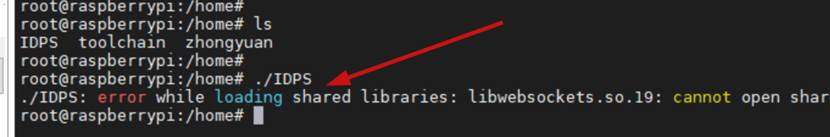
IDPS运行异常。



2.3.6绝对路径执行

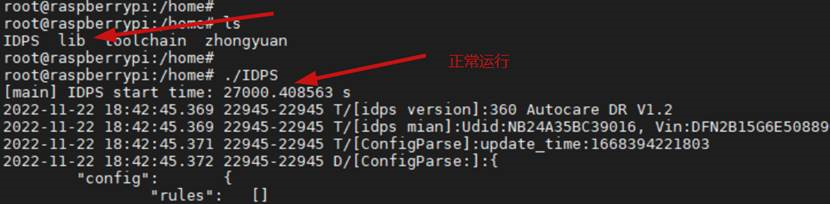
将$ORIGIN/lib 和 ${CMAKE\_SOURCE\_DIR}/lib 都删除。IDPS移动到/home目录下。

运行异常

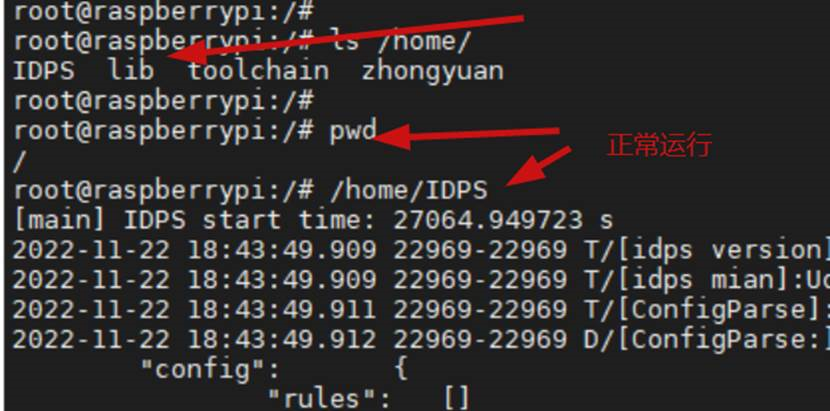


在/home目录下，增加lib库。

相对路径执行。



绝对路径执行。



4.2.5 **结论**

可以根据可执行程序自身所在的目录，在编译时就指定动态库的路径。而不受系统变量的影响。

5. **安全编译**

5.1 **编译选项**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 缓解措施 | GCC编译选项 | NAD  Linux | SOC  Linux | MCU  AutosarCP |
| 地址空间布局随机化 | -pie  --fPIE | ⚪ | ⚪ | ✘ |
| 可执行空间保护 | -z noexecstack | ⚪ | ⚪ | ✘ |
| GOT表写保护 | -Wl,-z,relro,-z,now | ⚪ | ⚪ | ✘ |
| 堆栈保护 | -fstack-protector-all | ⚪ | ⚪ | ⚪ |
| 编译时缓冲区溢出检查 | -D\_FORIFY\_SOURCE=2 | ⚪ | ⚪ | ⚪  （-O level） |

备注：

SOC编译器为aarch64-poky-linux-gcc (GCC) 7.3.0；

NAD编译器为gcc820-v1；

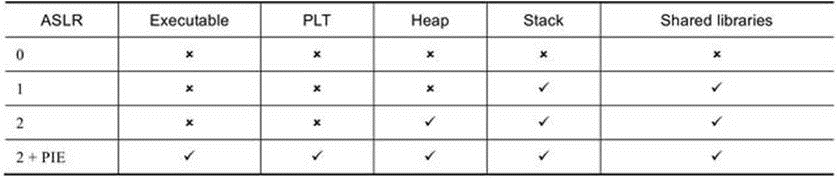
MCU编译器为BiSheng\_Arm32leM\_2.7.0.1\_win64。

5.2 **漏洞缓解措施**

1.地址空间布局随机化

ASLR技术：Address Space Layout Randomization，是一种操作系统用来抵御缓冲区溢出攻击的内存保护机制。，使程序每次运行的地址（堆地址、栈地址、引用库的地址等）都是不同的，增加攻击者预测程序地址空间的难度。需要配置/proc/sys/kernel/randomize\_va\_space，含三种情况，0表示关闭，1表示部分开启，2表示完全开启。

PIE技术：position-independent executable，是指可在主存储器中任意位置正确地运行，而不受其绝对地址影响的一种机器码。



1.可执行空间保护

如果操作系统可以将内存中的部分或所有可写区域标记为不可执行，则可能会阻止堆栈和堆内存区域成为可执行的。这有助于防止某些缓冲区溢出攻击成功，尤其是那些注入和执行代码的攻击，如Sasser和Blaster蠕虫。这些攻击依赖于内存的某些部分，通常是堆栈，既可写又可执行；否则，攻击就会失败。操作系统应支持使用可执行空间保护。

3.GOT表写保护

4.在ELF文件（可执行与可链接文件格式）的空间中，GOT（Global Offset Table）表存放着所有全局变量指针的全局偏移表，能够把位置无关的地址定位到绝对地址。GOT表在链接全过程中是可写的，在第一次加载程序时解析所有GOT条目，并使GOT为只读。如果GOT表内容被篡改，则函数和变量的引用也将跳转到未知的地址上去。

5.堆栈保护

堆栈保护，有时称为“堆栈粉碎保护”或“堆栈保护器”，是一种编译器技术，允许程序在运行时探测何时发生特定类型的缓冲区溢出。在缓冲区边界上插入探测变量，当缓冲区被溢出时，探测值首先被覆盖，通过检查探测值是否被修改，来判断是否发生了溢出攻击。它们在编译时被添加到二进制文件中。

6.编译时缓冲区溢出检查

操作系统的编译器都支持对某些内存和字符串函数的轻量级缓冲区溢出保护。这种漏洞缓解技术可以防止某些简单的编码错误成为可利用的漏洞。

5.3 **敏感信息检查**

检查范围：

1.日志

2.网络外发的数据，或接收端处于可控范围内除外。

检查内容：

ICC中需要进行安全存储的敏感数据信息包括以下内容但不限于：

⚪ 系统配置文件；

⚪ PKI体系证书、秘钥：CA根证书、临时通信证书、通信证书、加密秘钥（Package Key）、用于申请通信证书的公私钥对儿；

⚪ OTA流程中使用的证书；

⚪ 联网应用中使用的密钥信息；

⚪ VIN和SN

⚪ GNSS位置信息

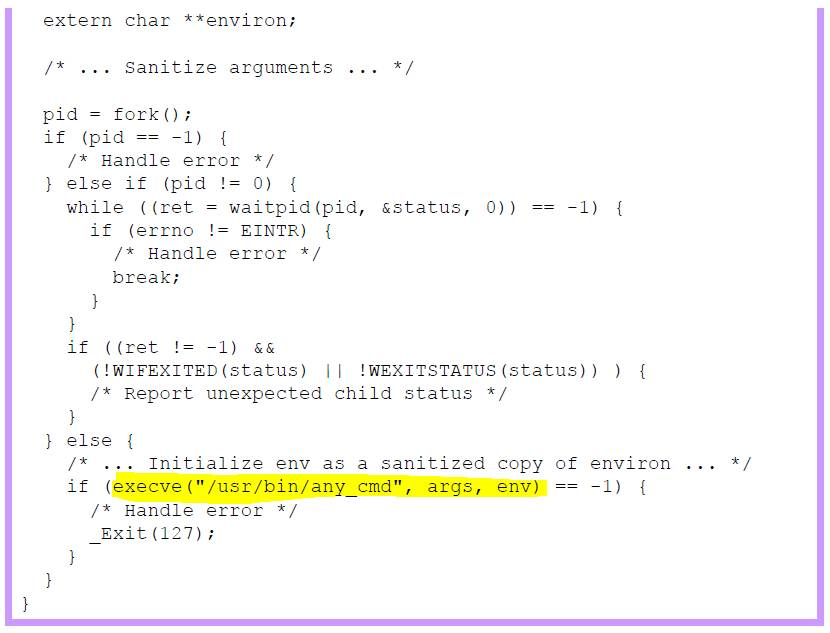
⚪ 账户信息

5.4 **禁用的函数**

3.1) strcpy(),strcat(),strncat(),sprint(),memcpy()

3.2) system(),popen(),exec()函数族

system()和popen()函数最终都调用了shell，由shell启动进程，容易造成命令注入攻击。使用execve()函数替换。



标黄部分的执行文件路径不能有拼接。

3.3) fopen()

使用fopen打开文件时，对文件内容进行校验。

5.5 **删除调试信息**

gcc编译时取消-g开关，或使用strip命令清楚调试信息。使用（gdb）bt/list/file进行自查，

5.6 **密码**

涉及到密码的，统一使用密码工具生成32byte长度随机密码。

工具使用openssl或gpg命令生成。gpg命令如下：

|  |
| --- |
| Bash gpg --gen-random --armor 2 32 |

密码不能硬编码到程序中，或明文放入存储器中。目前方案是写入ICC的配置数据库。涉及账户名也要写入数据库。ModuleName和ItemName模块自己定义即可。

6. **逆向工具**

6.1 **命令工具**

6.1.1 **strace命令**

使用strace或dtrace工具（Linux/Unix）：这些工具可以追踪一个程序的所有系统调用和信号。在Linux中，strace可以显示程序和Linux内核之间的交互，包括每个系统调用、其参数和返回值。

|  |
| --- |
| Python strace -o output.txt program\_name |

6.1.2 **readelf命令**

readelf常用命令

语法：readelf (选项)(参数:文件),除了-v和-H之外，其它的选项必须有一个被指定参数

选项 -h(elf header)，显示elf文件开始的文件头信息。

选项 -A，arch-specific 显示CPU构架信息。

选项 -a，all 显示全部信息,等价于 -h -l -S -s -r -d -V -A -I。

选项 -v，version 显示readelf的版本信息。

选项 -H，help 显示readelf所支持的命令行选项。

7. **第三方库编译**

7.1 **开源库编译**

7.1.1 **libusb**

https://github.com/libusb/libusb

依赖工具

|  |
| --- |
| C++ sudo apt-get update sudo apt-get install build-essential autoconf libtool |

编译

|  |
| --- |
| C++ ./bootstrap.sh ./configure make |

7.1.2 **hidapi**

https://github.com/signal11/hidapi

编译hidapi需要libusb

|  |
| --- |
| C++ ./bootstrap ./configure make |