Android so加固的简单脱壳

原创

Fly20141201 于 2017-09-24 16:24:21 发布 5065 收藏 6
分类专栏： Android Hook学习 Android系统安全和逆向分析研究 文章标签： android so脱壳 elf修复 so 修复 android so加固 soinfo

本文链接：https://blog.csdn.net/QQ1084283172/article/details/78077603


Android Hook学习 同时被 2 个专栏收录
29 篇文章 3 订阅
订阅专栏


Android系统安全和逆向分析研究
72 篇文章 59 订阅
订阅专栏

本文博客地址：http://blog.csdn.net/qq1084283172/article/details/78077603

Android应用的so库文件的加固一直存在，也比较常见，特地花时间整理了一下Android so库文件加固方面的知识。本篇文章主要是对看雪论坛《简单的so脱壳器》这篇文章的思路和代码的分析，很久之前就阅读过这篇文章但是一直没有时间来详细分析它的代码，最近比较有空来分析一下这篇文章中提到的Android so脱壳器的代码udog，github下载地址为：https://github.com/devilogic/udog，Android so加固的一般手法就是去除掉外壳Android so库文件的 ELF 链接视图 相关的信息，例如区节头表的偏移、区节头表的项数、区节头表名称字符串表的序号等，这样处理以后将Android so加固的外壳so库文件拖到IDA中去分析的时候，直接提示区节头表无效的错误，IDA工具不能对Android so加固的外壳so库文件进行分析，达到抗IDA工具静态分析的目的。

Android so加固中被保护的Android so库文件是由外壳Android so库文件在.init段或者.init_array段的构造函数里自定linker进行内存加载和解密的，被保护的Android so库文件自定义内存加载、映射完成以后将外壳Android so库文件的soinfo*（dlopen函数返回的）修改为被保护Android so库文件的soinfo*，这样被保护的Android so库文件的内存加载就成功了并且就可以被调用了。尽管被加固保护的Android so库文件被加密保护起来了，但是该Android so库文件还是会在内存中进行解密出来，因此我们可以在被加固保护的Android so库文件内存解密时进行内存dump处理，然后对dump出来的Android so库文件进行ELF文件格式的调整和修复以及section区节头表的重建，就可以实现被保护的Android so库文件的脱壳了。

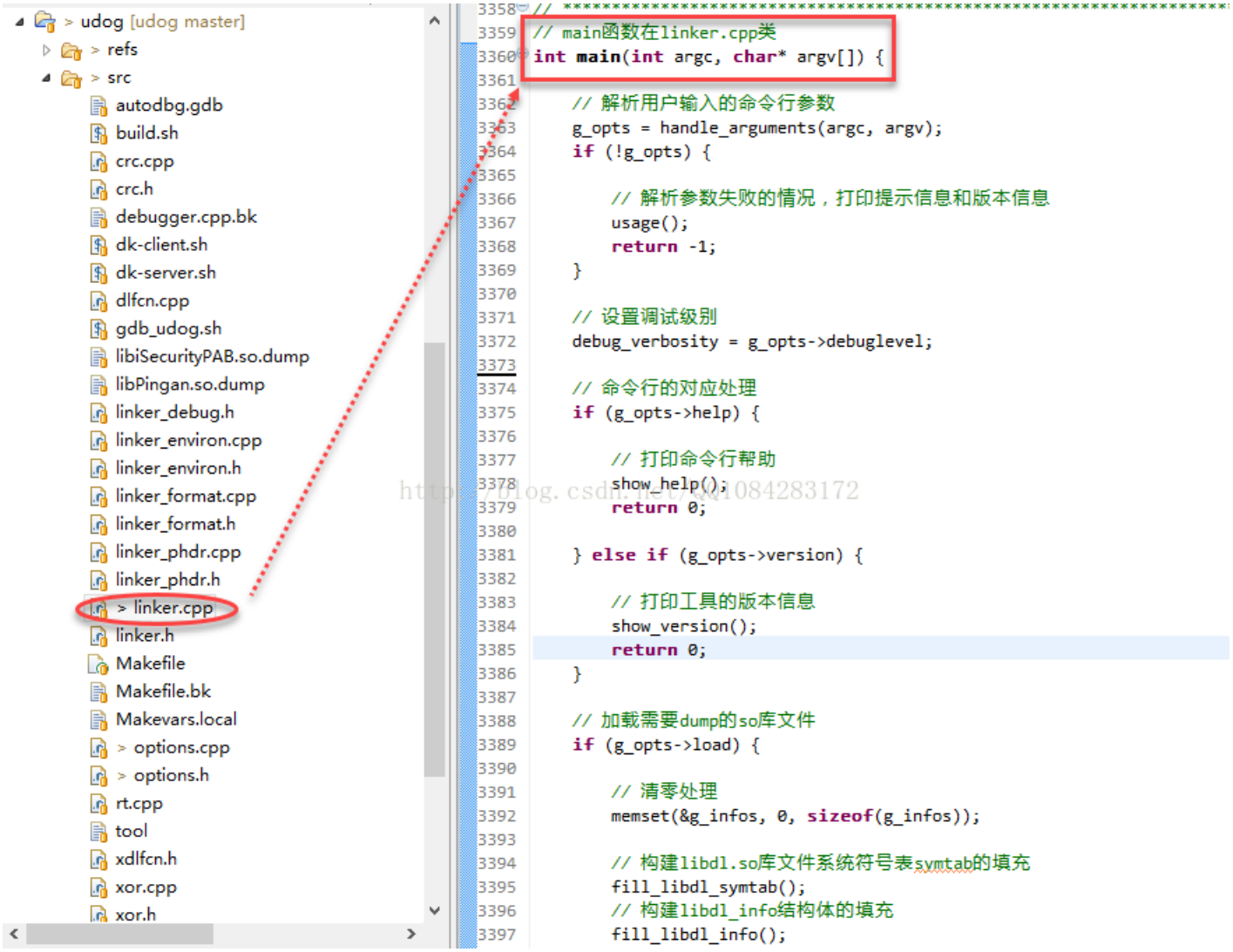
简单so脱壳器这篇文章中提到的so脱壳器udog的代码比较简单，之前以为udog的代码比较复杂，后来整理了一下作者玩命的代码发现很多代码都是废弃的，核心关键的代码部分不是很复杂但是对于学习Android so加固的脱壳很有作用，也是Android so加固脱壳和内存dump后修复的第一步，玩命版主主要实现了被加固Android so库文件的内存dump和ELF文件格式部分参数的修复处理，对于脱壳后ELF文件的section区节头表等的重建并没有实现。

自己总结了一下壳的三个发展流程。
1.文件本地加密，导入内存解密，壳加载器跑完不再做其他事情。
2.在程序正常运行时，壳可以重新接管控制权。
3.虚拟机保护。

这个简单的脱壳器，就是针对第一阶段而言。对于当前强度不是很高android保护而言。直接从内存中dump就好了。修复其实就是对重定位表以及got表的修复。 这份代码中，没搞那么复杂。我太懒了，有需求的朋友自己修改吧。

这个简单的脱壳器就是linker代码的简单应用。 用法直接看option.cpp就可以了。
下来之后直接 make DEBUG=1 all 即可编译调试版本。 make all 可以编辑发布版本。
配置ndk在最外层的 Makevars.global中配置ndk的路径。
从github下载时，切换到dev版本即可。 代码很乱，随意修改了一下linker而已。
以下是下载地址:
https://github.com/devilogic/udog

udog代码的入口 main函数 在linker.cpp文件中如下图所示：



在linker.cpp文件中，main函数工作流程是: 先对用户输入的命令行参数进行解析处理得到 用户参数解析结果描述结构体options_t，然后根据用户输入的命令参数解析的结果options_t 进行Android so的脱壳相关的操作。udog脱壳器中帮助命令行在文件options.cpp中实现，如下图所示：

linker.cpp | linker.h | options.h | options.cpp

```
// 打印软件版本以及提示信息
void usage() {

    printf("udog [options] file\n");
    printf("http://www.nagapt.com\n");
    show_version();
}

void show_version() {

    printf("V%s\n", UDOG_VERSION_STRING);
}

// 使用帮助的命令行参数
void show_help() {

    printf("\t----------------------------------------\n");
    printf("\t|==== Android Native Lib Cracker ====  |\n");
    printf("\t----------------------------------------\n");
    printf("udog [options] file\n");
    printf("-d, --dump=file                    dump load so to file\n");
    printf("--clear-entry                      clear DT_INIT value\n");
    printf("-c, --check                        print code sign\n");
    printf("--xcto=offset(hex)                 set xct offset\n");
    printf("--xcts=size(hex)                   set xct size\n");
    printf("-h, --help                         show help\n");
    printf("-v, --version                      show version\n");
    printf("--debug=level                      show debug information\n");
    printf("http://www.nagapt.com\n");
    show_version();
    printf("\n");
}
```


用户输入的命令行参数解析结果保存结构体options_t中，如下图代码所示：

```
// 保存用户输入命令行参数的解析结果
struct options_t {

 bool call_dt_init;
 bool call_dt_init_array;
 bool call_dt_finit;
 bool call_dt_finit_array;
 bool load_pre_libs;
 bool load_needed_libs;

 bool load;
 bool not_relocal;              /* 不对重定位表进行修复 */
 bool make_sectabs;             /* 重建elf文件的区节头表 */
 bool dump;
 bool help;
 bool version;
 bool debug;
 bool check;
 bool clear_entry;

 int debuglevel;
 unsigned xct_offset;
 unsigned xct_size;
 char dump_file[128];
 char target_file[128];
};
```

对用户输入的命令行参数进行解析处理的操作在函数handle_arguments中实现，如下图代码所示：

```
// main函数在linker.cpp类
int main(int argc, char* argv[]) {

 // 解析用户输入的命令行参数
 g_opts = handle_arguments(argc, argv);
```

```
// 解析用户输入的命令行参数
struct options_t* handle_arguments(int argc, char* argv[]) {

 // 保存命令行参数解析后的结果
 static struct options_t opts;
 // 清零
 memset(&opts, 0, sizeof(opts));
 // 默认参数的设置
 opts.call_dt_init = true;
 opts.call_dt_init_array = true;
 opts.call_dt_finit = true;
 opts.call_dt_finit_array = true;
 opts.load_pre_libs = true;
 opts.load_needed_libs = true;

 int opt;
 int longidx;
 int dump = 0, help = 0, version = 0,
  debug = 0, check = 0, xcto = 0,
  xcts = 0, clear_entry = 0;
```

```
if (argc == 1) {
 return NULL;
}

// 输入参数选项的解析顺序和规则
// 该数据结构包括了所有要定义的短选项，每一个选项都只用单个字母表示。
// 如果该选项需要参数，则其后跟一个冒号
const char* short_opts = ":hvcd:";
// 解析参数的长选项
struct option long_opts[] = {
 // 1--选项需要参数
  { "dump", 1, &dump, 1 },
 // 0--选项无参数
 { "help", 0, &help, 2 },
 { "version", 0, &version, 3 },
 { "debug", 1, &debug, 4 },
 { "check", 0, &check, 5 },
 { "xcto", 1, &xcto, 6 },
 { "xcts", 1, &xcts, 7 },
 { "clear-entry",0, &clear_entry, 8 },
 // 2--选项参数可选
  { 0, 0, 0, 0 }
};

// 对输入的命令行参数进行解析,longidx为解析参数长选项中的序号数
// 参考:https://baike.baidu.com/item/getopt_long/5634851?fr=aladdin
// 参考:http://blog.csdn.net/ast_224/article/details/3861625
while ((opt = getopt_long(argc, argv, short_opts, long_opts, &longidx)) != -1) {

 switch (opt) {
 case 0:
  // 进行so库文件的dump处理
  if (dump == 1) {

   opts.dump = true;
   // 暂时不对dump的so库文件的重定位表进行修复
   opts.not_relocal = false;
   // 对dump的so库文件的区节头表进行重建
   opts.make_sectabs = true;

   // 当处理一个带参数的选项时，全局变量optarg会指向它的参数
   // optarg为目标so库文件dump后的文件保存路径
   strcpy(opts.dump_file, optarg);
   // 加载dump的so库文件
   opts.load = true;
   dump = 0;

  } else if (help == 2) {

   opts.help = true;
   help = 0;

  } else if (version == 3) {

   opts.version = true;
   version = 0;

  } else if (debug == 4) {
```

```
    opts.debug = true;
    opts.debuglevel = atoi(optarg);
    debug = 0;

   } else if (check == 5) {

    opts.check = true;
    check = 0;

   } else if (xcto == 6) {

    opts.xct_offset = strtol(optarg, NULL, 16);
    xcto = 0;

   } else if (xcts == 7) {

    opts.xct_size = strtol(optarg, NULL, 16);
    xcts = 0;

   } else if (clear_entry == 8) {

    opts.clear_entry = true;
    clear_entry = 0;

   } else {

    //printf("unknow options: %c\n", optopt);
    return NULL;
   }
   break;
  case 'c':
   opts.check = true;
   break;
  case 'h':
   opts.help = true;
   break;
  case 'v':
   opts.version = true;
   break;
  case 'd':
   opts.dump = true;
   opts.not_relocal = false;
   opts.make_sectabs = true;
   strcpy(opts.dump_file, optarg);
   opts.load = true;
   break;
  case '?':
   //printf("unknow options: %c\n", optopt);
   return NULL;
   break;
  case ':':
   //printf("option need a option\n");
   return NULL;
   break;
  }/* end switch */
 }/* end while */


 /* 无文件 */
 if (optind == argc) {
```

```
 if (optind == argc) {
  return NULL;
 }

 // 当函数分析完所有参数时，全局变量optind（into argv）会指向第一个‘非选项’的位置
 // 需要被dump的so库文件的文件路径
 strcpy(opts.target_file, argv[optind]);

 // 返回的引用
 return &opts;
}
```

根据对用户输入命令行参数的解析结果options_t，进行Android so加固脱壳相关的操作，这里主要关注的是被加固的Android so库文件的内存dump相关的处理部分，如下图所示：

```
// 加载需要dump的so库文件
if (g_opts->load) {

 // 清零处理
 memset(&g_infos, 0, sizeof(g_infos));

 // 构建libdl.so库文件系统符号表symtab的填充
 fill_libdl_symtab();
 // 构建libdl_info结构体的填充
 fill_libdl_info();

 // unsigned ret = __linker_init((unsigned **)(argv-1));
 // if (ret == 0) return ret;

 // 获取到需要被dump的so库文件的文件路径
 char* fname = g_opts->target_file;
 // 动态加载需要被dump的so库文件返回信息描述结构体soinfo指针
 soinfo* lib = (soinfo*)dlopen(fname, 0);
 if (lib == NULL) {

  // 动态库加载失败的情况
  return -1;
 }

 //void* handle = dlsym(lib, "prepare_key");
 //if (handle) {
 // printf("%x\n", *(unsigned*)handle);
 //}

 // 从加载后的外壳so库文件中dump出解密后的被保护的so库文件
 if (g_opts->dump) {

  // dump出被保护的so库文件
  if (dump_file(lib) != 0) {
   return -1;
  }

  // 对dump出来被保护的so库文件进行区节头表的重建(玩命版主没有处理)
  // if (g_opts->make_sectabs) {
  //
  //  if (make_sectables(g_opts->dump_file) != 0) {
  //   return -1;
  //  }
```

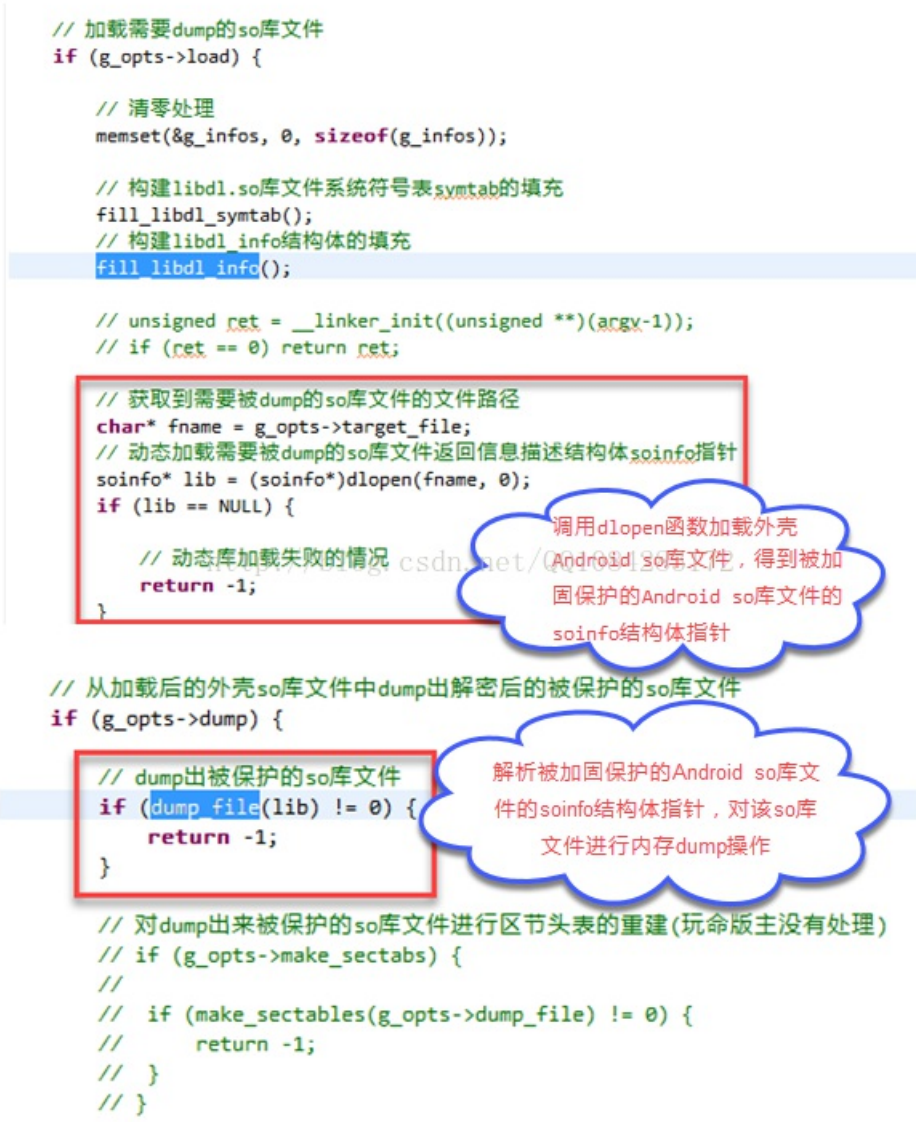
```
  // }

 } else {

  /* 打印代码CRC */
  if (g_opts->check) {
   checkcode_by_x((unsigned char*)(lib->base),
         "code text crc32",
         g_opts->xct_offset,
         g_opts->xct_size);
  }
 }

 // 卸载外壳so库文件的加载
 dlclose(lib);

// 不加载外壳so库文件的处理，crc32的校验
 } else {

 /* 未加载的功能  */
 if (g_opts->check) {

  checkcode(g_opts->target_file, "code text crc32",
      g_opts->xct_offset,
      g_opts->xct_size);
 }

}
```

在进行被加固Android so库文件的内存dump处理之前，还需要了解一下Android so库文件内存加载相关的知识。一般情况下，调用 dlopen函数 实现对Android so库文件的内存加载，调用dlopen函数成功以后返回Android so库文件内存加载后的模块句柄，其实该句柄就是 soinfo* (soinfo结构体的指针)，Android so库文件内存加载成功的内存镜像就是由结构体soinfo来描述的，soinfo结构体在进程内存中比较完整的描述了ELF文件的执行视图相关的信息。

```
/* so信息结构 */
struct soinfo
{
    char name[SOINFO_NAME_LEN];       /* so库文件的文件路径 */
    const Elf32_Phdr *phdr;           /* 指向程序段头表 */
    int phnum;         /* 程序段头表的数量 */
    unsigned entry;       /* so库文件的代码执行入口地址 */
    unsigned base;        /* so库文件内存加载后的基地址 */
    unsigned size;                    /* so库文件所有可加载段的长度 */

    int unused;  // DO NOT USE, maintained for compatibility.

    unsigned *dynamic;       /* .dynamic段描述结构体所在的起始地址*/

    unsigned unused2; // DO NOT USE, maintained for compatibility
    unsigned unused3; // DO NOT USE, maintained for compatibility

    soinfo *next;
    unsigned flags;

    const char *strtab;               /* .strtab段所在的内存地址 */
    Elf32_Sym *symtab;       /* .symtab段所在的内存地址 */

    unsigned nbucket;
    unsigned nchain;
```

```
    unsigned nchain;
    unsigned *bucket;
    unsigned *chain;

    unsigned *plt_got;

    Elf32_Rel *plt_rel;
    unsigned plt_rel_count;

    Elf32_Rel *rel;
    unsigned rel_count;

    unsigned *preinit_array;
    unsigned preinit_array_count;

    unsigned *init_array;
    unsigned init_array_count;

    unsigned *fini_array;
    unsigned fini_array_count;

    void (*init_func)(void);
    void (*fini_func)(void);

#if defined(ANDROID_ARM_LINKER)
    /* ARM EABI section used for stack unwinding. */
    unsigned *ARM_exidx;
    unsigned ARM_exidx_count;
#elif defined(ANDROID_MIPS_LINKER)
#if 0
     /* not yet */
     unsigned *mips_pltgot
#endif
     unsigned mips_symtabno;
     unsigned mips_local_gotno;
     unsigned mips_gotsym;
#endif /* ANDROID_*_LINKER */

    unsigned refcount;
    struct link_map linkmap;

    int constructors_called;                    /* 构造函数已经被调用 */

    /* When you read a virtual address from the ELF file, add this
     * value to get the corresponding address in the process' address space */
    Elf32_Addr load_bias;
    int has_text_relocations;

 /* 表明是否是从主程序中调用 */
 //int loader_is_main;
};
```

被加固Android so库文件脱壳操作的流程: 调用dlopen函数加载外壳Android so库文件，dlopen函数成功返回（即被加固保护的Android so库文件在内存中解密、自定义加载成功得到soinfo*，替换掉外壳Android so库文件返回的soinfo结构体指针 soinfo* 为被加固保护的Android so库文件加载成功后得到的 soinfo*，实现被加固保护的Android so与外壳Android so的无缝衔接）得到被加固保护的Android so库文件的soinfo*（被加固保护Android so库文件内存镜像的描述结构体），然后对被加固保护的Android so库文件的内存soinfo结构体指针进行解析，获取到被加固保护的Android so库文件ELF文件格式执行视图的描述信息，并根据获取到的这些信息对被加固Android so库文件进行内存dump处理。



被加固保护Android so库文件的内存dump操作在 函数dump_file 中实现，代码如下图所示：

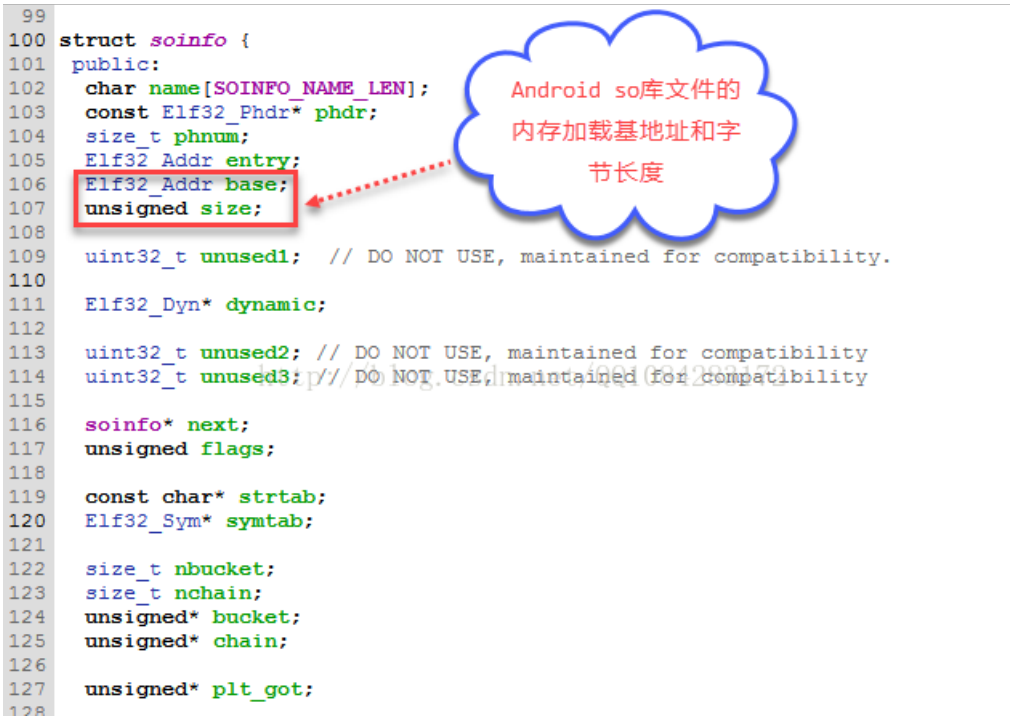
```
// 从外壳so动态加载返回的soinfo中dump出被加固so库文件
// 参考的源码文件: /bionic/linker/linker.h
int dump_file(soinfo* lib) {

 // 创建新文件，用以保存dump出来的so库文件
 FILE* fp = fopen(g_opts->dump_file, "w");
 if (NULL ==fp) {

  printf("create new file: %s error !", g_opts->dump_file);
  return -1;
 }

 // 修改外壳so库文件的整个内存加载区域为可读可写可执行
 int ret = mprotect((void*)lib->base, lib->size, 7 /**全部权限打开**/);

 // 打印外壳so库文件内存加载返回的soinfo中的信息
 printf("-----------------------------------------------------\n");
 // so库文件的内存加载地址
```

```
printf("base = 0x%x\n", lib->base);
// so库文件的内存加载映射大小
printf("size = 0x%x\n", lib->size);
// so库文件的代码指令的入口地址
printf("entry = 0x%x\n", lib->entry);
// so库文件的程序段头表的数量
printf("program header count = %d\n", lib->phnum);
printf("-------------------------------------------------\n");

// dump出来的so库文件的大小
unsigned dump_size = lib->size;
unsigned buf_size = dump_size + 0x10;

// 申请内存空间
unsigned char* buf = new unsigned char [buf_size];
if (NULL == buf) {

 printf("size = alloc memery err !\n");
 return -1;
}
// 将soinfo描述的so库文件的内存数据拷贝到申请的内存空间中
memcpy(buf, (void*)lib->base, lib->size);

// 定位到soinfo描述的so库文件(ELF)的文件头Elf32_Ehdr
Elf32_Ehdr* elfhdr = (Elf32_Ehdr*)(void*)buf;

// 修改区节头表的数量为0
elfhdr->e_shnum = 0;
// 修改该elf文件的区节数据的文件偏移为0
elfhdr->e_shoff = 0;
// 修改该elf文件区节表名称字符串所在的区节头表的序号为0
elfhdr->e_shstrndx = 0;

// 获取到该elf文件的程序段头表的文件偏移
unsigned phoff = elfhdr->e_phoff;
// 定位到该elf文件的程序段头表的位置
Elf32_Phdr* phdr = (Elf32_Phdr*)(void*)(buf + phoff);

// 遍历该elf文件的程序段头表
for (int i = 0; i < lib->phnum; i++, phdr++) {

 // 获取该程序段头描述的程序段所在的相对虚拟内存地址
 unsigned v = phdr->p_vaddr;
 // 修正该程序段的文件偏移地址为虚拟内存地址
 phdr->p_offset = v;

 // 获取该程序段头描述的程序段的内存对齐后的数据长度大小
 unsigned s = phdr->p_memsz;
 // 修正该程序段的文件数据长度大小为内存对齐后的数据长度大小
 phdr->p_filesz = s;
}

/* 是否清除DT_INIT入口点 */
if (g_opts->clear_entry)
 fix_entry(buf, lib);

// 将该soinfo描述的so库文件修正后的内存数据写入到新创建的g_opts->dump_file文件中
ret = fwrite((void*)buf, 1, dump_size, fp);

// 刷新文件流
```

```
 fflush(fp);
 // 资源的清理
 if (buf) delete [] buf;
 // 关闭文件
 fclose(fp);

 printf("Dump so Successful\n");
 return 0;
}
```

根据被加固保护Android so库文件内存镜像描述结构体 soinfo* 从进程内存中dump出so库文件的初步操作流程梳理如下：

1. 根据 传入参数soinfo* 获取到被加固Android so库文件的内存加载基地址和内存所有段的长度并修改该so库文件所在内存区域的内存属性为可读可写可执行。




```
struct soinfo {
 public:
  char name[SOINFO_NAME_LEN];
  const Elf32_Phdr* phdr;
  size_t phnum;
  Elf32_Addr entry;
  Elf32_Addr base;
  unsigned size;

  uint32_t unused1;  // DO NOT USE, maintained for compatibility.

  Elf32_Dyn* dynamic;

  uint32_t unused2; // DO NOT USE, maintained for compatibility
  uint32_t unused3; // DO NOT USE, maintained for compatibility

  soinfo* next;
  unsigned flags;

  const char* strtab;
  Elf32_Sym* symtab;

  size_t nbucket;
  size_t nchain;
  unsigned* bucket;
  unsigned* chain;

  unsigned* plt_got;
```

```
// 从外壳so动态加载返回的soinfo中dump出被加固so库文件
// 参考的源码文件：/bionic/linker/linker.h
int dump_file(soinfo* lib) {

    // 创建新文件，用以保存dump出来的so库文件
    FILE* fp = fopen(g_opts->dump_file, "w");
    if (NULL ==fp) {

        printf("create new file: %s error !", g_opts->dump_file);
        return -1;
    }

    // 修改外壳so库文件的整个内存加载区域为可读可写可执行
    int ret = mprotect((void*)lib->base, lib->size, 7 /**全部权限打开**/);

    // 打印外壳so库文件内存加载返回的soinfo中的信息
    printf("-----------------------------------------------\n");
    // so库文件的内存加载地址
    printf("base = 0x%x\n", lib->base);
    // so库文件的内存加载映射大小
    printf("size = 0x%x\n", lib->size);
    // so库文件的代码指令的入口地址
    printf("entry = 0x%x\n", lib->entry);
    // so库文件的程序段头表的数量
    printf("program header count = %d\n", lib->phnum);
    printf("-----------------------------------------------\n");

    // dump出来的so库文件的大小
    unsigned dump_size = lib->size;
    unsigned buf_size = dump_size + 0x10;

    // 申请内存空间
    unsigned char* buf = new unsigned char [buf_size];
    if (NULL == buf) {

        printf("size = alloc memery err !\n");
        return -1;
    }
    // 将soinfo描述的so库文件的内存数据拷贝到申请的内存空间中
    memcpy(buf, (void*)lib->base, lib->size);
```

2. 定位到soinfo描述的so库文件(ELF)的文件头Elf32_Ehdr，由于在Android so库文件内存加载时是基于ELF文件的可执行视图，因此该so库文件的链接视图的信息都会被去掉。为了避免so库文件内存dump后被IDA分析出错一般会将Android so库文件中seciton区节头表相关描述信息的参数设置为0，**注意**：Android 7.0版本的linker在进行Android so库文件的加载时会进行seciton区节头表相关描述信息参数的检查。

```
    // 定位到soinfo描述的so库文件(ELF)的文件头Elf32_Ehdr
    Elf32_Ehdr* elfhdr = (Elf32_Ehdr*)(void*)buf;

    // 修改区节头表的数量为0
    elfhdr->e_shnum = 0;
    // 修改该elf文件的区节数据的文件偏移为0
    elfhdr->e_shoff = 0;
    // 修改该elf文件区节表名称字符串所在的区节头表的序号为0
    elfhdr->e_shstrndx = 0;
```

3. 由于ELF文件加载内存时需要进行内存对齐的处理，因此内存中的Android so库的程序段的文件偏移和文件数据长度的大小需要进行修正处理。（从Android so库文件比较完整修复的角度来考虑，这一步不是必须甚至是应该去掉的，参考《ELF section修复的一些思考》。）

```
 // elf32文件的程序段的描述头结构
typedef struct elf32_phdr{
  Elf32_Word    p_type;  // 程序段的属性值
  Elf32_Off  p_offset; // 程序段的文件偏移
  Elf32_Addr    p_vaddr; // 程序段的相对虚拟地址RVA
  Elf32_Addr    p_paddr; // 程序段的物理地址
  Elf32_Word    p_filesz; // 程序段的文件数据长度大小
  Elf32_Word    p_memsz; // 程序段的文件数据内存对齐处理后的长度大小
  Elf32_Word    p_flags; // 程序段加载到内存后的可读可写可执行等内存属性值
  Elf32_Word    p_align; // 程序需要内存对齐的数值
} Elf32_Phdr;
```

```
// 获取到该elf文件的程序段头表的文件偏移
unsigned phoff = elfhdr->e_phoff;
// 定位到该elf文件的程序段头表的位置
Elf32_Phdr* phdr = (Elf32_Phdr*)(void*)(buf + phoff);

// 遍历该elf文件的程序段头表
for (int i = 0; i < lib->phnum; i++, phdr++) {

    // 获取该程序段头描述的程序段所在的相对虚拟内存地址
    unsigned v = phdr->p_vaddr;
    // 修正该程序段的文件偏移地址为虚拟内存地址
    phdr->p_offset = v;

    // 获取该程序段头描述的程序段的内存对齐后的数据长度大小
    unsigned s = phdr->p_memsz;
    // 修正该程序段的文件数据长度大小为内存对齐后的数据长度大小
    phdr->p_filesz = s;
}
```


4. Android so库文件的 .init段构造函数地址是否清除 的处理。

```
/* 是否清除DT_INIT入口点 */
if (g_opts->clear_entry)
    fix_entry(buf, lib);

// 将该soinfo描述的so库文件修正后的内存数据写入到新创建的g_opts->dump_file文件中
ret = fwrite((void*)buf, 1, dump_size, fp);

// 刷新文件流
fflush(fp);
// 资源的清理
if (buf) delete [] buf;
// 关闭文件
fclose(fp);

printf("Dump so Successful\n");
return 0;
}
```

```
void fix_entry(unsigned char* buf, soinfo* lib) {

 // 获取.dynamic段所在的内存地址
 unsigned* d = lib->dynamic;
 // 遍历.dynamic段的描述结构体
 while (*d) {

  // 获取到so库文件的初始化代码地址的描述结构体
  if (*d == DT_INIT) {

   // 获取到so库文件的初始化代码地址所在的文件偏移
   unsigned offset = (unsigned)(d+1) - lib->base;
   // 设置so库文件的初始化代码地址的相对虚拟地址为0
   *(unsigned*)(void*)(buf + offset) = 0;
   break;
  }
  d += 2;
 }
}
```

5. Android so库文件脱壳的修复还缺少的步骤：对比较简单的Android so脱壳和修复来说，上面的这些脱壳步骤已经可以了，IDA已经能够比较正常的分析了，但是基于Android so脱壳和修复进一步处理而言，上面的步骤3应该去掉。udog作者进行了Android so库文件的内存dump和初步的ELF文件修复处理，关于ELF文件的section区节头表相关信息的重建还没有完成，留给读者自己来完成，但是作者玩命已经给出了大致的修复思路。关于Android so库文件内存dump后修复的详细处理思路可以参考文章《ELF section修复的一些思考》、《从零打造简单的SODUMP工具》、《基于init_array加密的SO的脱壳》、《安卓so文件脱壳思路（java版）附源代码》、《ELF文件格式学习，section修复》，后面有时间我也会对这几篇文章进行学习和分析。

```
// 对dump出来被保护的so库文件进行区节头表的重建
// fname为g_opts->dump_file即dump出来的so库文件的文件路径
int make_sectables(char* fname) {

    //unsigned size = 0;
    //Elf32_Shdr shdr;
    FILE* fp = fopen(fname, "wr");
    if (NULL == fp) {
        return -1;
    }

    // 哈哈,玩命没有处理
    fseek(fp, 0, SEEK_END);

    /* .dynamic节 */

    /* 节名表节 */
    /* 符号节 */
    /* 字符串节 */

    fflush(fp);
    fclose(fp);
    return 0;
}
```


6. Android so库文件内存dump处理操作的说明：自我感觉文章中，对于 dump_file函数 处理的Android so库文件的描述不是很准确（这里提到的被内存dump处理的Android so库文件不一定是被加固保护的Android so库文件的，因为 dlopen函数返回的soinfo* 也可能是外壳so加载器的Android so库文件的，具体以实际操作得到的结果为准，与Android so加固的对抗思路有一定的关系，不好准确描述。）