

# Linux ELF 文件数据完整性保护系统

# Linux ELF 文件数据完整性保护系统

## 作者

张靖棠 (@mrdrivingduck), 南京航空航天大学, 计算机科学与技术学院, 网络空间安全专业

宗华 (@zonghuaxiansheng), 中国科学技术大学, 计算机科学与技术学院, 计算机技术专业

---

## 简介

本解决方案旨在通过密码学技术, 对 (包括但不限于) Intel® x86 平台下基于 Linux 内核的操作系统中的 ELF 文件实现数据完整性保护。方案主要分为两个部分:

1. 用户空间下, 基于信息摘要算法和非对称加密算法的 ELF 文件数字签名程序
2. 内核空间下, 基于内核密钥保留服务的 ELF 文件数据完整性验证模块


上述两部分全部由 GNU C 语言实现, 另有少量用于审计、测试或批处理的 Python / shell 脚本文件。所有代码将在 [MIT License](#) 的许可下开源。

文档的后续内容主要介绍这两个部分的实现原理, 以及使用说明。

## ELF 文件数字签名程序

该程序主要包含三个主要工作:

- 对于 ELF 文件执行时所需要使用的指令、数据等信息, 分别生成固定长度的 **信息摘要**
- 从 X.509 格式的证书中读取公私钥, 根据摘要内容计算 **数字签名**
- 将数字签名附加到原 ELF 文件中, 作为 ELF 完整性的验证信息


 数字签名的附加不能破坏原 ELF 文件的格式, 尤其不能破坏会被操作系统使用的关键执行信息 (指令、数据等)。对于一个不检验 ELF 文件完整性的操作系统, 也能够正常、正确地执行这个 ELF 文件。

代码仓库地址：

- ELF 二进制文件签名程序：<https://github.com/mrdrivingduck/linux-elf-binary-signer>

## 内核 ELF 文件数据完整性验证模块

当操作系统执行一个 ELF 文件时，内核首先将 ELF 文件载入内存，解析出文件中的 **被保护信息** (代码、数据等) 和对应的数字签名。内核计算出被保护信息的摘要，再使用内核信任的内置密钥对数字签名进行解密，比对解密后的签名与信息摘要是否完全一致。如果数据完全一致，说明 ELF 文件的被保护信息未经篡改，内核继续完成 ELF 文件开始执行前的准备工作；如果数据不一致，说明 ELF 文件遭到篡改，为了维护操作系统的安全性，内核拒绝继续执行这个 ELF 文件。

 内核所进行的上述动作对用户来说完全透明。用户在输入执行 ELF 文件的命令后，不需要任何额外的操作。最终的执行结果只可能为以下两种：

1. 内核正常执行 ELF 文件，输出预期结果
2. 内核拒绝执行 ELF 文件，并显示错误原因

代码仓库地址：

- 内核源码树 (基于 Linux kernel 4.15.0 release) :  
<https://github.com/mrdrivingduck/linux-kernel-elf-sig-verify>
- 可动态装载的独立模块：<https://github.com/mrdrivingduck/linux-kernel-elf-sig-verify-module>

# System for protecting integrity of ELF files in Linux

## Author

Jingt. Zhang (@[mrdrivingduck](#)), Nanjing University of Aeronautic and Astronautics (NUAA)

H. Zong (@[zonghuaxiansheng](#)), University of Science and Technology of China (USTC)

---

## Introduction

The solution aims at protecting the integrity of ELF files with modern cryptography techniques, designed for operating systems based on Linux kernel under (but not limited to) Intel® x86 architecture. The solution consists of two parts:

1. An ELF signing program based on message digest and asymmetric encryption algorithm, in user space
2. A kernel module for verifying signature of ELF files based on Linux key retention service, in kernel space

Two parts are both implemented in GNU C, together with some Python / shell scripts for auditing, testing or batch job. All code will be available under the permission of [MIT License](#).


The following parts of this document will introduce the theory and the usage.

## Signing program for ELF files

The three main job for this program:

- Compute the message digest for instructions and data which is necessary for the execution of an ELF file

- Extract the public & private key from certificate in X.509 format, and compute digital signature with message digest
- Attach the digital signature to the original ELF file as verification information


 The attachment of digital signature should not break the format of the original ELF file, especially for the instructions and data which will be used by operating system. For an OS without verifying the integrity of an ELF file, it should also normally and correctly execute the ELF file.

Repository:

- ELF binary signer: <https://github.com/mrdrivingduck/linux-elf-binary-signer>

## Mechanism for verifying integrity of ELF file in kernel

When the OS executes an ELF file, the kernel will firstly load the ELF file into memory, parse and extract the information under protection (instructions, data) and corresponding digital signatures. Then, the kernel will compute the message digest, decrypt the signature with kernel-trusted public key, and compare the decrypted signature to the digest. If they are exactly the same, it means the ELF file is not tampered, the kernel will move on to the preparation for execution; if they are different from each other, it means the ELF file is tampered, the kernel will refuse to run this ELF file, for the sake of security.

 The above-mentioned actions done by the kernel are completely transparent to users. After inputting the command to run an ELF file, the user doesn't need to perform any more actions. The final execution result should only be one of the following:

1. The kernel executes the ELF file normally, and output a result as expected.
2. The kernel refuses to execute an ELF file, and shows the reason of the error.

Repository:

- The kernel source tree (based on Linux kernel 4.15.0 release):  
<https://github.com/mrdrivingduck/linux-kernel-elf-sig-verify>
- The loadable standalone kernel module: <https://github.com/mrdrivingduck/linux-kernel-elf-sig-verify-module>

# 背景

## ELF 格式

**可执行与可链接格式** (Executable and Linkable Format, ELF) 是一种可执行文件、目标代码、共享库、核心转储文件的通用标准文件格式。ELF 标准最早发布于一个名为 System V Release 4 (SVR4) 的 Unix 操作系统版本的应用二进制接口 (Application Binary Interface, ABI) 标准规范中，并迅速被各大 Unix 版本所接受。1999 年，ELF 格式被选为 Unix 或类 Unix 系统在 x86 处理器平台上的标准二进制文件格式。

在设计上，ELF 格式具有 **灵活、可扩展、跨平台** 的特点。比如，其支持不同的字节顺序 (大小端)、不同的地址空间 (32/64)、不排斥任何特定的 CPU 或指令集体系结构 (Instruction Set Architecture, ISA)。因此，ELF 格式能够在很多不同硬件平台的不同操作系统上运行。

## 数据完整性与机密性

数据完整性 (Data Integrity) 是信息安全的三个基本要素之一，是设计、实现、使用任何系统时都至关重要的方面。数据完整性指对于数据的存储、处理、传输的整个生命周期，维护其 **准确性** 和 **一致性**。数据完整性技术的主要目标是，保证数据确实被所期待地那样记录，并在取回数据之后，确保数据与其被记录之前完全相同。简单来说，数据完整性保护了数据信息免于意外或非意外的修改，如恶意篡改、突发硬件故障、人为错误等。

数据机密性 (Data Confidentiality) 是指数据只能为所限定的使用者查看或理解。一般地，保护数据机密性的有效方式是对数据进行加密。只有合法的使用者能够正确解密数据而理解数据，或者看到数据的原始内容；而非法使用者即使得到了加密数据，也无法轻易理解数据的原始含义。

在网络系统、数据库系统、操作系统的文件系统等场景中，数据机密性与完整性有着广泛的应用。

# 运行环境与依赖

## Linux 发行版

当前解决方案确定可运行于 **Deepin**、**UOS**、**Ubuntu** 等基于 Debian 的 Linux 操作系统发行版。

## Linux 内核版本

当前解决方案基于 Linux kernel 4.15.0 release。由于没有修改任何内核主线代码，因此可以快速扩展到更高版本的 Linux kernel 中。

## 依赖

本解决方案中，运行于用户空间的 ELF 数字签名程序需要解析 ELF 文件格式，还需要应用一些密码学的算法，因此需要一些 Debian 软件包作为依赖。基于 Debian 的 Linux 发行版系列可以使用 **APT** (Advanced Package Tool) 工具轻易安装这些依赖。签名程序依赖的软件包如下：

- `libssl-dev`

解决方案中内核部分的代码不能也不需要依赖任何第三方软件包。



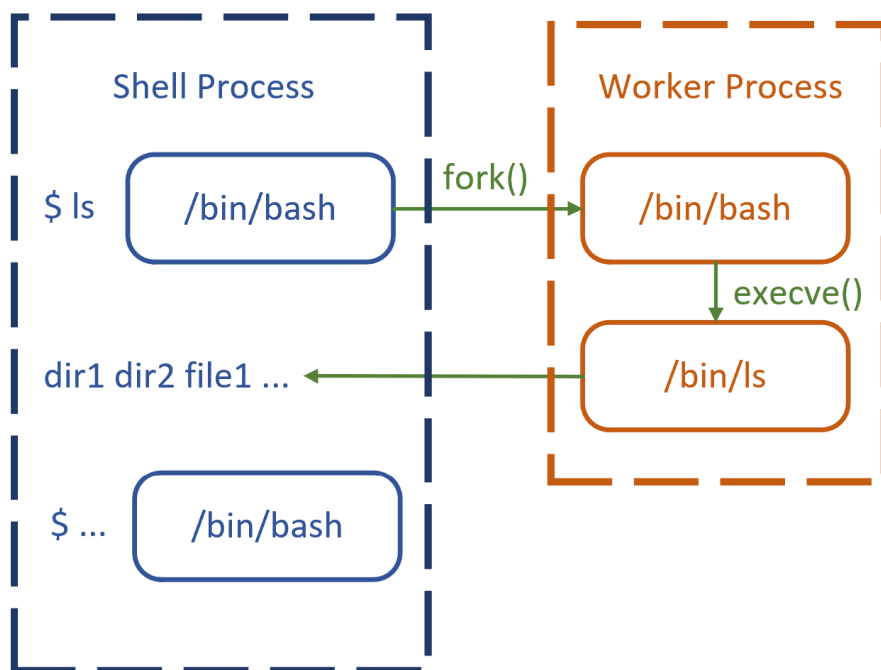
## 第一部分 - 内核 ELF 签名验证模块

# Chapter 1 - 二进制文件执行过程

## 1.1 内核如何执行一个二进制文件

当我们在 shell 中输入命令，试图执行一个二进制文件时，shell 会调用 Linux 内核的 `execve` 系统调用。该系统调用会将二进制文件载入内存，判断其格式是否合法、是否具有执行权限。如果一切正常，则将当前进程的代码段、数据段替换为新程序的段，并设置好命令行参数、环境变量、新的程序入口，完成新程序开始运行前的一切准备工作。

如果 `execve` 的执行一切正常，那么该进程原有程序的运行环境都会被新进程完全替换。对于使用 shell 的用户来说，希望能够在运行完一条命令后再次回到 shell 中继续输入下一条命令。为了避免 shell 进程在调用 `execve` 之后被新程序覆盖，在 shell 的实现中，通常会通过 `fork` 系统调用，先复制出一个当前进程的副本，然后在副本进程中调用 `execve`。执行结束后，回到 shell 的原进程中，使得用户可以继续执行下一个程序。



Shell 执行程序的流程

## 1.2 内核如何识别不同格式的二进制文件

内核支持执行多种不同格式的二进制文件。比如一个 ELF 文件，或是一个 shell script，甚至还可以是一幅图片，或一个 PDF 文档。对于不同的二进制文件格式，内核需要为其准备不同的运行环境：

- 对于一个 ELF 文件，内核需要找到该文件的代码段和数据段，以及程序的首条指令地址，并将这些信息设置在进程控制块中
- 对于一个 shell script，内核需要寻找脚本指定的解释器，并将脚本文件名作为参数，启动脚本解释器解释执行脚本
- 对于一个 PDF 文档或图片，内核也需要寻找对应的解释器，并将文件名作为解释器参数，启动解释器

为了识别二进制文件的格式，内核会将二进制文件的前 128 个字节载入内存中如下的结构体中：

```
include/linux/binfmts.h

1  #define CORENAME_MAX_SIZE 128
2
3  /*
4   * This structure is used to hold the arguments that
5   * are used when loading binaries.
6   */
7  struct linux_binprm {
8      char buf[BINPRM_BUF_SIZE];
9      /* ... */
10 } __randomize_layout;
```

内核根据 magic value (每个格式特有的字节序列) 等信息，识别出二进制文件的格式。比如：


- 根据文件开头是否是 `#!` 来判断是否是一个脚本文件
  - 根据文件开头是否是 `0x7f` `0x45(E)` `0x4c(L)` `0x46(F)` 来判断是否是一个 ELF 文件
  - ...
-

## 1.3 二进制文件格式处理程序 (Binary Format Handler)

在 Linux 4.15.0 中，内核能够执行哪些格式的二进制文件呢？

在该版本的内核源代码中，已经内置了部分二进制文件格式的处理程序 (handler)，位于内核代码的 `fs/` 目录下：

- `binfmt_aout` - a.out 格式
- `binfmt_elf` - ELF (Execute and Link-able Format) 格式
- `binfmt_elf_fdpic`
- `binfmt_em86`
- `binfmt_flat`
- `binfmt_misc` - 可在内核运行期间自行注册解释程序的二进制文件格式
- `binfmt_script` - 脚本文件格式，用于执行 shell、Perl 等格式脚本

 编译内核前，根据目标硬件平台进行编译配置后，有些二进制文件格式的处理模块将不会被编译进内核主映像；有些二进制文件格式的处理模块将会被编译为内核模块。

其中，每一种格式对应的处理程序，在编码上都以模块的形式实现 (虽然编译后并不一定是模块)，并且都用类似面向对象中多态的形式实现了一个统一的接口：

```
include/linux/binfmts.h

1  /*
2   * This structure defines the functions that are used to
3   * load the binary formats that linux accepts.
4   */
5  struct linux_binfmt {
6      struct list_head lh;
7      struct module *module;
8      int (*load_binary)(struct linux_binprm *);
9      int (*load_shlib)(struct file *);
10     int (*core_dump)(struct coredump_params *cprm);
11     unsigned long min_coredump; /* minimal dump size */
12 } __randomize_layout;
```

其中，关注这三个函数指针：

- `*load_binary` 指向装入该格式二进制文件的函数
- `*load_shlib` 指向装入该格式共享库的函数
- `*core_dump` 指向装入该格式核心转储文件的函数

在操作系统初始化阶段，这些二进制格式的处理函数被依次添加到了一个链表上。当执行一个二进制文件时，内核实现将其头 128 字节以及相关信息读入 `struct linux_binprm` 结构体，然后依次遍历链表上的每一个 `struct linux_binfmt` 结构体，以 `struct linux_binprm` 作为参数，依次调用每种二进制格式对应的 `*load_binary` 函数。这一过程在 `fs/exec.c` 的如下函数中实现：

```
fs/exec.c

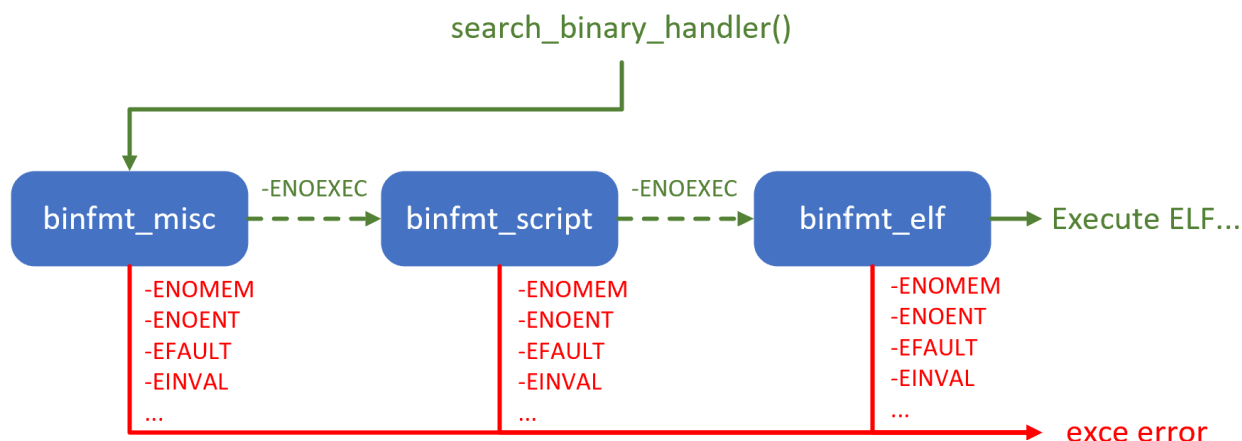
1  int search_binary_handler(struct linux_binprm *bprm)
2  {
3      /* 遍历二进制文件处理链表 */
4      list_for_each_entry(fmt, &formats, lh) {
5          /* 调用每个二进制格式的 load_binary() 函数 */
6          retval = fmt->load_binary(bprm);
7          /* 如果返回 -ENOEXEC，那么继续遍历链表 */
8          /* 如果返回其它错误，则退出 */
9          if (retval != -ENOEXEC || !bprm->file) {
10             read_unlock(&binfmt_lock);
11             return retval;
12         }
13     }
14     /* ... */
15     return retval;
16 }
```

在这个函数中，可能会有以下三种情况出现：

1. 当前处理函数不能识别这个二进制文件的格式，则返回 `-ENOEXEC` 的错误码，继续尝试用下一个二进制格式处理函数识别该文件的格式
2. 当前处理函数成功识别了这个二进制文件的格式，并顺利完成了执行这个二进制文件的准备工作，`execve` 系统调用正确返回
3. 当前处理函数成功识别了这个二进制文件的格式，但在执行该文件准备工作的过程中发生了错误 (比如动态分配内存失败，或该文件的内容损坏/不一致)，则将相应错误码返回

`execve` 系统调用并停止遍历链表。最终 `execve` 系统调用将错误码返回 `bash`，向用户提示运行失败的原因

以内核执行 ELF 文件为例，内核搜索 ELF 处理函数的过程如下图所示：



二进制文件处理函数链表

## 1.4 内核识别二进制文件格式的次序

上面的链表体现了典型的 Intel 架构 64-bit 机器中，内核默认编译配置下，识别一个二进制文件格式的次序。哪些因素决定了内核以不同的优先级依次识别各个二进制文件格式呢？

内核是在 **初始化阶段** 逐步构造二进制格式处理函数链表的。每个格式的处理函数被挂上链表的时机和方式由如下因素决定：

1. 二进制格式处理模块在内核中的初始化等级
2. 二进制格式处理模块被编译的顺序
3. 二进制格式处理模块被挂上链表的方式 (在链表头部插入 / 在链表尾部插入)

### 内核初始化等级

在内核初始化阶段，内核为其各个模块的初始化划分了优先级：

```
include/linux/init.h
```

```

1  /*
2   * A "pure" initcall has no dependencies on anything else, and purely
3   * initializes variables that couldn't be statically initialized.
4   *
5   * This only exists for built-in code, not for modules.
6   * Keep main.c:initcall_level_names[] in sync.
7   */
8  #define pure_initcall(fn)          __define_initcall(fn, 0)
9
10 #define core_initcall(fn)          __define_initcall(fn, 1)
11 #define core_initcall_sync(fn)    __define_initcall(fn, 1s)
12 #define postcore_initcall(fn)     __define_initcall(fn, 2)
13 #define postcore_initcall_sync(fn) __define_initcall(fn, 2s)
14 #define arch_initcall(fn)         __define_initcall(fn, 3)
15 #define arch_initcall_sync(fn)    __define_initcall(fn, 3s)
16 #define subsys_initcall(fn)       __define_initcall(fn, 4)
17 #define subsys_initcall_sync(fn)  __define_initcall(fn, 4s)
18 #define fs_initcall(fn)           __define_initcall(fn, 5)
19 #define fs_initcall_sync(fn)      __define_initcall(fn, 5s)
20 #define rootfs_initcall(fn)       __define_initcall(fn, rootfs)
21 #define device_initcall(fn)       __define_initcall(fn, 6)
22 #define device_initcall_sync(fn)  __define_initcall(fn, 6s)
23 #define late_initcall(fn)         __define_initcall(fn, 7)
24 #define late_initcall_sync(fn)    __define_initcall(fn, 7s)

```

其中，等级越高 (优先级数值越小) 的模块越先被初始化。在每个二进制格式处理函数中，都会声明当前模块在哪个等级上被初始化。以 ELF 格式处理模块 ( `binfmt_elf` ) 为例：

`fs/binfmt_elf.c`

```

1  core_initcall(init_elf_binfmt);
2  module_exit(exit_elf_binfmt);
3  MODULE_LICENSE("GPL");

```

以上代码表示内核会在 `core_initcall` (1 号优先级) 的等级上，通过调用 `init_elf_binfmt()` 函数，将 ELF 格式的处理模块插入到链表中。

## 模块编译的相对顺序

模块编译的相对顺序体现在 Makefile 文件中，位于前面的模块先被编译。以 `fs/Makefile` 为例：

fs/Makefile

```
1 obj-$(CONFIG_BINFMT_AOUT) += binfmt_aout.o
2 obj-$(CONFIG_BINFMT_EM86) += binfmt_em86.o
3 obj-$(CONFIG_BINFMT_MISC) += binfmt_misc.o
4 obj-$(CONFIG_BINFMT_SCRIPT) += binfmt_script.o
5 obj-$(CONFIG_BINFMT_ELF) += binfmt_elf.o
6 obj-$(CONFIG_COMPAT_BINFMT_ELF) += compat_binfmt_elf.o
7 obj-$(CONFIG_BINFMT_ELF_FDPIC) += binfmt_elf_fdpic.o
8 obj-$(CONFIG_BINFMT_FLAT) += binfmt_flat.o
```

在同一个初始化等级上，先被编译的模块会先被初始化。比如，假设上述所有模块都是由 `core_initcall()` 等级进行初始化，那么模块就按从上到下的顺序依次初始化。

## 处理模块在链表上注册的方式

每个二进制格式处理模块中都定义了初始化函数。在该函数中，该模块将其自身注册到内核的二进制文件处理函数链表上。内核提供两种注册二进制文件处理模块的方式：

- `register_binfmt(&xxx_format)`
- `insert_binfmt(&xxx_format)`

两种注册方式的差别是，`register_binfmt` 将处理模块注册在 **链表尾部**，而 `insert_binfmt` 将处理模块注册在 **链表头部**。

---

## 1.5 对 ELF 文件进行签名验证的思路

本解决方案的目标是，在内核为执行一个 ELF 文件进行原有的准备工作之前，先对 ELF 中的签名进行验证。如果验证通过，则继续进行准备工作；如果验证失败，则内核拒绝执行这个 ELF 文件。

解决方案的核心思想是，以 **二进制格式处理程序** 的形式，实现一个 ELF 签名验证模块 (`binfmt_elf_signature_verification`)，并将该模块注册在链表中 ELF 处理模块 (`binfmt_elf`) 之前。在该模块中，实现签名验证的逻辑。如果签名验证通过，则返回

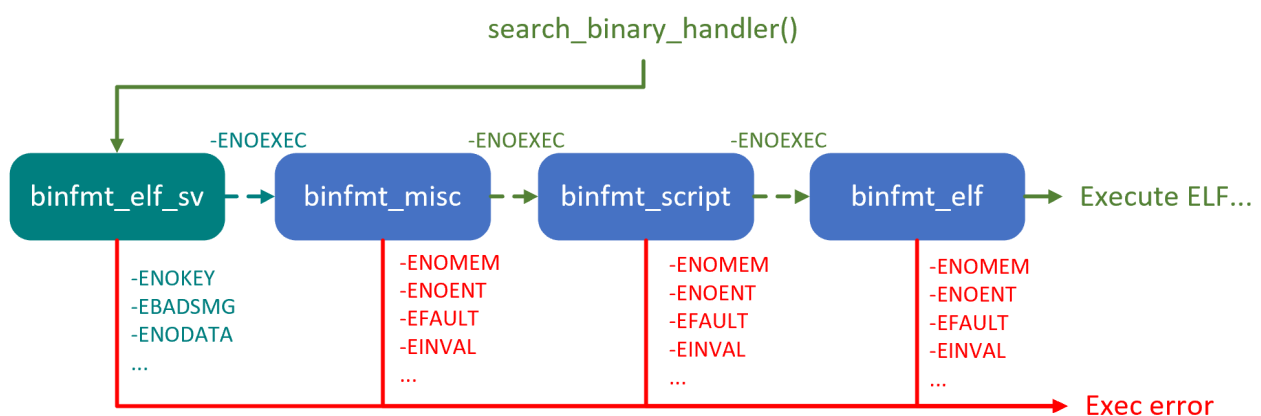


`-ENOEXEC` 错误码，使得内核继续遍历链表，最终能够调用真正的 ELF 处理模块；如果签名验证不通过，则直接返回其它错误码，使得内核不再继续调用真正的 ELF 处理模块。

在这个 ELF 签名验证模块中，需要进行的工作主要有两点：

1. 格式检查 - 判断该二进制文件是否符合 ELF 格式，跳过所有非 ELF 格式的文件
2. 签名验证 - 如果文件符合 ELF 格式，则取出其中附带的数字签名，并进行签名验证

用户可以在内核启动后，通过 `insmod` 命令动态加载这个模块，通过 `rmmod` 命令动态卸载这个模块。当用户准备挂载该模块时，内核中的二进制处理模块链表已经初始化完成了。由于模块被插入链表的方式只有从链表头插入和从链表尾插入，而插入到链表尾部使得模块无法在系统内置的 `binfmt_elf` 模块之前被执行，因此只能在链表的头部插入这个处理模块：



ELF 签名验证模块被内核装载

ELF 签名验证模块可能返回的错误原因：

- 内核中缺少用于签名验证的密钥
- ELF 文件中没有附带数字签名
- ELF 文件中附带了数字签名，但其中的内容无法通过签名验证
- ELF 格式正确但内容损坏
- 内核运行时的一些突发错误 (如动态分配内存失败)
- ...

非 ELF 格式的二进制文件 (如 shell 脚本) 将无法通过

`binfmt_elf_signature_verification` 模块的 ELF 格式检查，从而返回 `-ENOEXEC`，由之

后其它的二进制文件处理模块进行处理。

---

## 1.6 参考资料

[LWN.net - How programs get run](#)

[Stackoverflow - How does kernel get an executable binary file running under linux?](#)

[Linux Journal - Playing with Binary Formats](#)

[The Linux Kernel - Kernel Support for miscellaneous Binary Formats \(binfmt\\_misc\)](#)

# Chapter 2 - ELF 文件格式分析

## 2.1 关于 ELF 格式

**可执行与可链接格式** (Executable and Linkable Format, ELF) 是一种可执行文件、目标代码、共享库、核心转储文件的通用标准文件格式。ELF 标准最早发布于一个名为 System V Release 4 (SVR4) 的 Unix 操作系统版本的应用二进制接口 (Application Binary Interface, ABI) 标准规范中，并迅速被各大 Unix 版本所接受。1999 年，ELF 格式被选为 Unix 或类 Unix 系统在 x86 处理器平台上的标准二进制文件格式。

在设计上，ELF 格式具有 **灵活、可扩展、跨平台** 的特点。比如，其支持不同的字节顺序 (大小端)、不同的地址空间 (32/64)、不排斥任何特定的 CPU 或指令集体系结构 (Instruction Set Architecture, ISA)。因此，ELF 格式能够在很多不同硬件平台的不同操作系统上运行。

---

## 2.2 ELF 文件的视角与格式

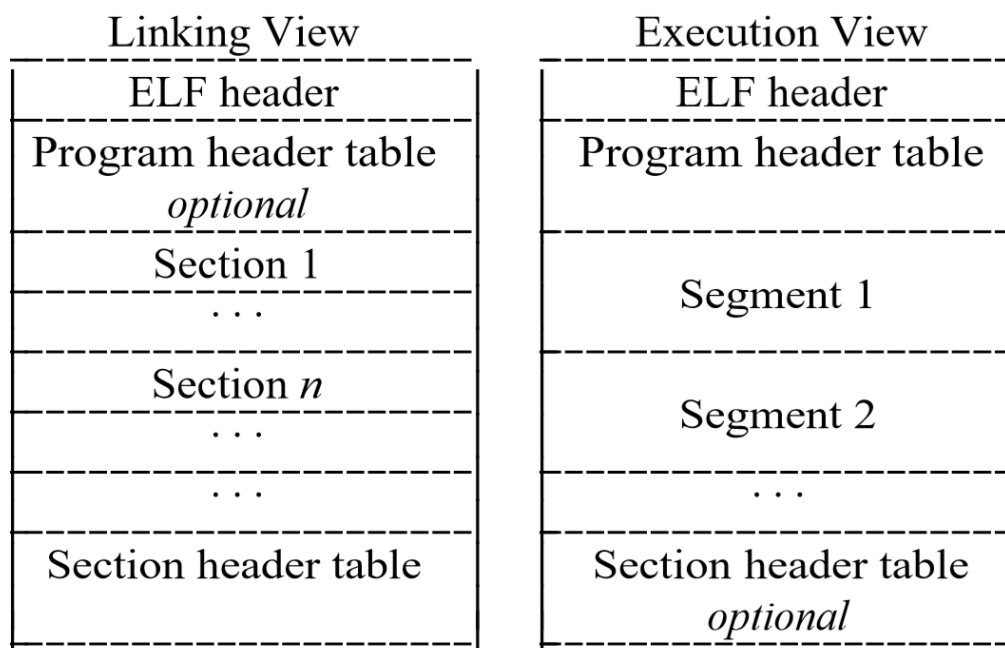
ELF 文件具有三种主要的目标类型：

- 可重定位文件 - 可与其它目标文件静态链接在一起的代码和数据
- 可执行文件 - 可被操作系统装载到进程地址空间中并执行
- 共享对象文件 - 与其它可重定位文件或共享库文件链接为另一个目标；或与可执行文件和其它共享库文件一起形成进程映像

根据 ELF 文件被使用的方式，文件内容可以有两种视角：

- 链接视角 - 链接器将 ELF 文件链接在一起时使用的视角
- 执行视角 - 操作系统将 ELF 文件装载到进程地址空间时使用的视角

两种视角如下图所示：



ELF 视角

Section 主要用于链接器对代码的重定位，如汇编程序中的 `.text` `.data`。而当文件载入内存执行时，目标代码中的 section 会被链接器组织到可执行文件的各个 segment 中。

如上图所示，ELF 文件除去各 section/segment 以外，主要包含三个重要部分：

1. ELF header
2. Program header table
3. Section header table

ELF header 指明了 ELF 文件的整体信息，如 ELF 文件的 magic value、类型、版本、目标机器等。另外，ELF 还指明了 program header table 与 section header table 两个表在文件中的偏移位置、条目个数、条目大小。这两个表的位置和长度随着 section/segment 的个数而变化，而 ELF header 总是位于文件最开头，且长度固定。显然，如果想要访问 program header table 和 section header table 中的信息，必须通过 ELF header 来找到它们在文件中的确切位置。

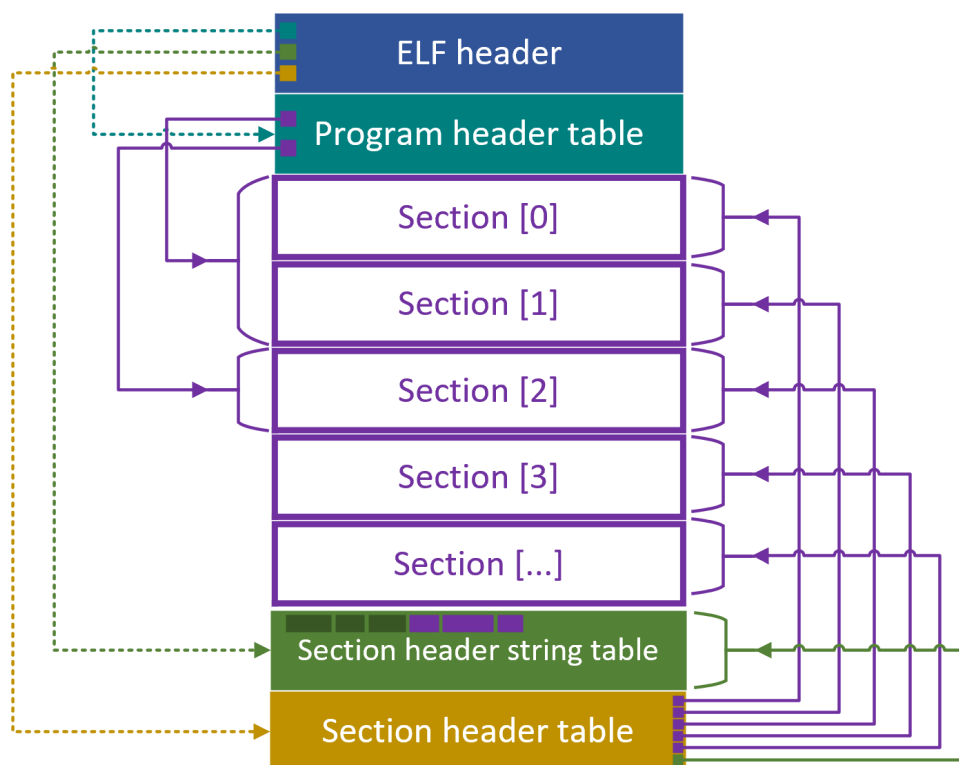
Program header table 主要描述了将哪一个或哪几个 section 组织为一个 segment，以及各个 segment 的描述信息。Section header table 描述了 ELF 文件中所有的 section，以及每个 section 的类型、长度等描述信息。

**i** Section header table 中并不存储每个 section 的名称。所有 section 的名称全部存储在一个名为 section header string table 的 section 中，名称之间用 `\0` 分隔。在 ELF header 中，记录了该 section 在 section header table 中的索引。

## 2.3 内核从 ELF 中取得数字签名的步骤

内核在对 **二进制文件处理函数链表** 进行遍历时，已经读取了该文件的 **头 128 字节**。如果该二进制文件是一个 ELF 文件，那么已读取的内容中已经包含了 ELF 文件的 ELF header。由此，首先通过 ELF header 中的 magic value 检验二进制文件是否是一个 ELF 文件；判断 ELF 文件类型是否为 `ET_EXEC`（可执行文件）或 `ET_DYN`（动态链接文件）。

其次，根据 ELF header 中指示的 section header table 的位置、条目个数、每个条目的大小，可以将 section header table 装载到内存；根据 ELF header 中指示的 section header string table 的索引，以及已经装入内存的 section header table，可以将 section header string table 装载到内存。



同时遍历 section header table (每个 section 的描述信息) 和 section header string table (每个 section 的名称)，可以定位到与签名程序约定好的签名信息 section 与被签名 section，如：

- 被签名 section `.text` 与签名信息 section `.text_sig`
- ...

在找到这两个相互对应的 section 之后，再根据 section header table 中指示的这两个 section 在文件中的偏移与长度，将这两个 section 的具体数据装入内存。

最终，基于每对匹配的 section 数据进行签名验证。如果所有的签名验证都正确，那么 **ELF 签名验证模块** 会返回 `-ENOEXEC` 错误码，使内核随后调用真正的 ELF 处理模块完成相应的工作；如果签名验证错误，那么模块返回其它错误码，内核将无法继续执行这个 ELF 文件。

---

## 2.4 参考资料

Executable and Linking Format (ELF) Specification

# Chapter 3 - 内核密钥保留服务

## 3.1 Linux 密钥保留服务

**Linux 密钥保留服务 (Linux key retention service)** 是在 Linux 2.6 中引入的，它的主要意图是在 Linux 内核中缓存身份验证数据。远程文件系统和其他内核服务可以使用这个服务来管理密码学、身份验证标记、跨域用户映射和其他安全问题。它还使 Linux 内核能够快速访问所需的密钥，并可以用来将密钥操作（比如添加、更新和删除）委托给用户空间。

Root 用户可以用过 `proc` 文件系统查看内核中的密钥：

```
1 $ cat /proc/keys
2 125ce30c I----- 1 perm 1f030000 0 0 keyring .dns_resolver:
3 155ea96d I----- 1 perm 1f030000 0 0 keyring .id_resolver: e
4 178570f1 I----- 1 perm 1f0b0000 0 0 keyring .builtin_truste
5 1ab86c77 I----- 1 perm 1f030000 0 0 asymmetri sforshee: 00b28
6 1abf10d6 I--Q--- 1 perm 1f3f0000 0 65534 keyring _uid_ses.0: 1
7 390c087a I----- 1 perm 1f0b0000 0 0 keyring .builtin_regdb_
8 3e099031 I--Q--- 2 perm 1f3f0000 0 65534 keyring _uid.0: empty
9 3e22b50c I----- 1 perm 1f030000 0 0 asymmetri WatchDog: ELF v
```

上述信息显示了公钥的序号 (Serial number)、类型 (key-ring/asymmetry/...)、状态、过期时间、描述等信息。

在编译内核时通过配置 `CONFIG_SYSTEM_TRUSTED_KEYS` 选项，引用 **PEM** 格式的证书文件，就能够在内核的系统密钥环 (`.system_keyring`) 上添加额外的 **X.509** 公钥证书。关于该编译选项的说明如下：

`certs/Kconfig`

```
1 config SYSTEM_TRUSTED_KEYS
2     string "Additional X.509 keys for default system keyring"
3     depends on SYSTEM_TRUSTED_KEYRING
4     help
5         If set, this option should be the filename of a PEM-formatted file
6         containing trusted X.509 certificates to be included in the default
7         system keyring. Any certificate used for module signing is implicitly
8         also trusted.
```

```
9
10 NOTE: If you previously provided keys for the system keyring in the
11 form of DER-encoded *.x509 files in the top-level build directory,
12 those are no longer used. You will need to set this option instead.
```

在内核启动之后，也可以通过 `keyctl` 系统调用向内核中动态添加新的公钥。但内核只允许 X.509 封装信息已经被 `.system_keyring` 中已有密钥进行合法签名后的新密钥加入系统密钥环。

我们为 ELF 的签名与验证生成了一对 **RSA** 公私钥，将公私钥以符合 X.509 标准的方式导入到一个 PEM 编码的证书中。并通过上述机制，将证书编译进内核的系统密钥环上。这样，在 **ELF 签名验证模块** 中，可以通过使用系统密钥环中的公钥，对 ELF 文件中的签名信息进行验证。

## 3.2 访问系统内置密钥进行签名验证

首先，我们对 Linux 内核中已有的 **内核模块签名** 验证机制的代码进行了分析。在内核源代码目录 `certs/system_keyring.c` 中，定义了内核内置的受信密钥：

```
certs/system_keyring.c
1 static struct key *builtin_trusted_keys;
```

但由于这个变量没有被声明为 `extern`，因此无法在其它内核代码中直接引用这个变量。但是在这个源文件中，开放了 `verify_pkcs7_signature()` 函数，使得其它内核代码能够通过这个函数，间接使用内置密钥环的签名验证功能：

```
certs/system_keyring.c
1 /**
2  * verify_pkcs7_signature - Verify a PKCS#7-based signature on system data
3  * @data: The data to be verified (NULL if expecting internal data).
4  * @len: Size of @data.
5  * @raw_pkcs7: The PKCS#7 message that is the signature.
```



```

6  * @pkcs7_len: The size of @raw_pkcs7.
7  * @trusted_keys: Trusted keys to use (NULL for builtin trusted keys only,
8  *                (void *)1UL for all trusted keys).
9  * @usage: The use to which the key is being put.
10 * @view_content: Callback to gain access to content.
11 * @ctx: Context for callback.
12 */
13 int verify_pkcs7_signature(const void *data, size_t len,
14                            const void *raw_pkcs7, size_t pkcs7_len,
15                            struct key *trusted_keys,
16                            enum key_being_used_for usage,
17                            int (*view_content)(void *ctx,
18                                                  const void *data, size_t len,
19                                                  size_t asn1hdrlen),
20                            void *ctx)
21 {
22 ...

```

在内核代码中，通过 `#include <linux/verification.h>` 使用该函数时，输入 **签名数据** 与 **被签名数据** 的 **缓冲区内存地址** 和 **缓冲区长度**，就能够使用内置密钥完成签名认证。因此，**ELF 签名验证模块** 只要能够从 ELF 文件中正确提取 **PKCS #7** 格式的签名数据，以及签名保护的目标数据，就可以通过这个函数验证数字签名是否正确。

### 3.3 参考资料

[IBM Developer - Linux 密钥保留服务入门](#)

[The Linux Kernel - Kernel module signing facility](#)

[Signature verification of kernel module and kexec](#)

## 第二部分 - ELF 文件签名程序

# Chapter 4 - 信息摘要与 PKCS #7 签名

## 4.1 信息摘要

信息摘要能够将任意长度的消息变成固定长度的短消息。采用单向 hash 函数，将需要加密的明文摘要成一串 **固定长度** 的密文，也被称为数字指纹。不同的明文摘要成密文，其结果总是不同的；而对于同样的明文，摘要必定一致。一个 hash 函数的好坏是由发生碰撞的概率决定的。如果攻击者能够轻易地构造出两个具有相同 hash 值的消息，那么这样的 hash 函数是很危险的。

一些常见的摘要算法有：

- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512
- ...

---

## 4.2 数字签名

数字签名 (又称公钥数字签名) 是只有信息的发送者才能产生的，别人无法伪造的一段数字串。这段数字串同时也是对信息发送者发送信息真实性的有效证明。它是一种类似写在纸上的普通的物理签名，但是使用了公钥密码体制中的技术实现，用于鉴别数字信息。数字签名通常定义两种互补的运算：

- 签名
- 验证

数字签名是 **非对称密钥加密技术** 与 **信息摘要技术** 的应用。发送报文时，发送方用一个 hash 函数从报文文本中生成摘要，然后用发送方的私钥对这个摘要进行加密，这个加密后的摘要将作为报文的数字签名和报文一起发送给接收方；接收方使用与发送方相同的 hash 函数从接收到的原始报文中计算出报文摘要，再使用发送方的公钥对报文附加的数字签名进行解密。如果这两个摘要相同，那么接收方就能确认原始报文未经篡改。

---

## 4.3 PKCS #7

PKCS (Public Key Cryptography Standards) #7 被命名为 CMS (Cryptographic Message Syntax Standard), 是 RSA 公司 提出的最著名的标准。这一标准也是 S/MIME (Secure/Multipurpose Internet Mail Extensions) 的基础。PKCS #7 提供了一种用途广泛的创建数字签名的语法和格式, 其中包含的信息有:

- 生成摘要的 hash 算法
- Detached / Attached 格式
- 签名公钥证书
- 签名者信息
- 证书签发者信息
- 认证属性 (签名时间、消息摘要等)
- 签名加解密算法
- 签名数据

通过 OpenSSL 工具可以查看一段数字签名的具体信息 (以一个 ELF 文件的 .text section 的签名为例):

```
1  $ openssl cms -inform der -in .text_sig \  
2    -noout -cmsout -print  
3  CMS_ContentInfo:  
4    contentType: pkcs7-signedData (1.2.840.113549.1.7.2)  
5    d.signedData:  
6      version: 1  
7      digestAlgorithms:  
8        algorithm: sha256 (2.16.840.1.101.3.4.2.1)  
9        parameter: <ABSENT>  
10     encapContentInfo:  
11       eContentType: pkcs7-data (1.2.840.113549.1.7.1)  
12       eContent: <ABSENT>  
13     certificates:  
14       <ABSENT>  
15     crls:  
16       <ABSENT>  
17     signerInfos:  
18       version: 1  
19       d.issuerAndSerialNumber:  
20         issuer: O=WatchDog, CN=ELF verification/emailAddress=mrdrivingdu  
21         serialNumber: 0x4879F322F1671DF368169F7F3ED0F84150A2CC86  
22       digestAlgorithm:  
23         algorithm: sha256 (2.16.840.1.101.3.4.2.1)
```

```

24     parameter: <ABSENT>
25     signedAttrs:
26         <ABSENT>
27     signatureAlgorithm:
28         algorithm: rsaEncryption (1.2.840.113549.1.1.1)
29         parameter: NULL
30     signature:
31         0000 - 37 87 f2 4a 6b 84 ad 7e-e0 38 76 ec 3a fe d5 7..Jk..~.8
32         000f - 05 33 45 4c 4d 41 ed c6-66 78 83 6d ad 60 e3 .3ELMA..fx
33         001e - 43 62 fd 6e d8 65 f7 ea-05 0a 50 ff 5c 8f 1c Cb.n.e....
34         002d - b1 e2 ec 64 86 c0 80 49-e7 88 87 86 e3 8f 7c ...d...I..
35         003c - ae f2 96 b8 80 03 c4 d0-36 94 d9 45 5d 44 c5 .....6.
36         004b - 7c de 6c fa 02 2f a9 88-fd e4 fe 6a cc d1 a6 |.l../....
37         005a - f5 cd c6 af a2 b3 cb c2-1c 4b 21 06 33 14 de .....K
38         0069 - e0 7f 62 0a 11 47 15 5d-ee 0b 15 9b 03 f1 b5 ..b..G.]..
39         0078 - 5c 18 bb f5 24 31 3d 83-38 06 7d f4 c6 ba 23 \...$1=.8.
40         0087 - c4 40 ff 66 6d bf 62 28-90 6d fb eb bb 04 14 .@.fm.b(.m
41         0096 - 78 ee 84 0f 09 cb a8 2c-a0 e4 d9 1e a1 86 c8 x.....,..
42         00a5 - 2c 21 7a dd 10 d7 b3 cd-9b 43 be 2a ca 2f 01 ,!z.....C
43         00b4 - 22 f6 b6 86 72 31 bc d3-f1 43 5a 8e 2a 4d eb "...r1...C
44         00c3 - 2b 7f 39 a4 99 8b 38 16-42 3d 88 23 23 43 d6 +.9...8.B=
45         00d2 - 82 bc 19 d0 11 b9 41 a6-33 53 da 72 a6 8c db .....A.3S
46         00e1 - 6d c1 25 fe b2 56 ea 31-84 e2 5c 46 45 2c d6 m.%.V.1..
47         00f0 - b2 7c 03 f9 ff ea 92 b5-75 76 8b 9a dd 1f c8 .|.....uv
48         00ff - 3f ?
49     unsignedAttrs:
50         <ABSENT>

```

 Linux 内核中的 `verify_pkcs7_signature()` 函数能够以 PKCS #7 的格式解析签名，并使用内核内置密钥验证签名。

## 4.4 产生 ELF 文件的签名

ELF 文件签名程序需要使用 OpenSSL `libcrypto` 库中的函数。

首先，通过 `libcrypto` 的库函数，对签名过程进行初始化：

- 实例化所有的加密算法 - `OpenSSL_add_all_algorithms()`
- 实例化所有的摘要算法 - `OpenSSL_add_all_digests()`

然后，根据命令行参数指定的摘要算法、公钥证书文件路径、私钥文件路径，分别进行签名前的准备：

- 获得摘要算法实例 - `EVP_get_digestbyname(hash_algo)`
- 读取私钥 - `read_private_key(private_key_name)`
- 读取公钥证书 - `read_x509(x509_name)`

最终，通过下面的代码，产生最终的数字签名：

```
elf_sign.c

1  #ifndef USE_PKCS7
2  /* Load the signature message from the digest buffer. */
3  cms = CMS_sign(NULL, NULL, NULL, NULL,
4                CMS_NOCERTS | CMS_PARTIAL | CMS_BINARY |
5                CMS_DETACHED | CMS_STREAM);
6  ERR(!cms, "CMS_sign");
7
8  ERR(!CMS_add1_signer(cms, x509, private_key, digest_algo,
9                    CMS_NOCERTS | CMS_BINARY |
10                   CMS_NOSMIMECAP | use_keyid |
11                   use_signed_attrs),
12     "CMS_add1_signer");
13  ERR(CMS_final(cms, bm, NULL, CMS_NOCERTS | CMS_BINARY) < 0,
14     "CMS_final");
15
16  #else
17  pkcs7 = PKCS7_sign(x509, private_key, NULL, bm,
18                    PKCS7_NOCERTS | PKCS7_BINARY |
19                    PKCS7_DETACHED | use_signed_attrs);
20  ERR(!pkcs7, "PKCS7_sign");
21  #endif
```

产生的数字签名首先会被导入到一个临时文件，并在随后被注入到 ELF 文件中，成为 ELF 文件中的一个新的 section。

---

## 4.5 参考资料

The Linux Kernel - Manually signing modules

[RFC 2315 - PKCS #7: Cryptographic Message Syntax Version 1.5](#)

[Wikipedia - Digital signature](#)

[What Is a Digital Signature?](#)

[Introduction to Digital Signatures and PKCS #7](#)

[PKCS#7 - SignedData](#)

[OpenSSL 之 BIO 系列之15 — 内存 \(mem\) 类型 BIO](#)

[OpenSSL 中文手册之 BIO 库详解](#)

[OpenSSL Wiki - Main Page](#)

[OpenSSL Wiki - Libcrypto API](#)

# Chapter 5 - 将签名数据注入 ELF 文件

## 5.1 签名数据存放位置

要将签名数据附在原有的 ELF 文件中，且不破坏 **ELF 的结构**，使其能够在没有签名验证机制的 OS 上也能运行，有两种可能的方式：

- 直接在 ELF 文件的尾部附加签名
- 以符合 ELF 规范的方式以 ELF section 为单位添加签名数据

在尾部追加签名的方式比较简单，但从文件整体视角来看，该文件已经不是一个符合 ELF 格式的文件了。如果我们需要对 ELF 文件的多个 section 进行签名，在文件末尾追加的方法将具有较差的可扩展性和灵活性。此外，将无法通过 `readelf` 或 `objdump` 等工具读取签名数据。

以符合 ELF 格式的方式添加数据稍微复杂一些，因为其中涉及到对 ELF 的文件格式进行解析。另外，在 ELF 文件中添加若干新的 section，可能会涉及到对 ELF 文件中其它结构的修改。但这种方法使我们可以在 ELF 文件中添加任意多个 section，并可以在 ELF 的 section header table 结构中记录这些 section 的元信息。这样方便内核从签名后的 ELF 文件中快速获取有效的签名数据。

---

## 5.2 将签名数据 section 注入 ELF 文件

在向 ELF 文件插入数据时，我们尽可能地在每个 ELF 结构的尾部插入数据。在尾部插入能够最大程度地避免破坏已有的 ELF 结构之间的引用关系。要将数据注入到 ELF 文件中，并维持 ELF 文件的原有格式，需要修改如下几处结构：

1. 插入一个包含签名数据的新 section 数据区
2. Section header table 中应多出一个新的 entry 来描述签名数据在文件中的偏移和长度
3. 将新 section 的名称字符串添加到 `.shstrtab` 中，并更新该 section 的长度
4. 数据插入位置之后的所有 section 的偏移地址都需要被后移更新
5. 在 ELF header 中更新 section header table 的偏移位置和元素数量



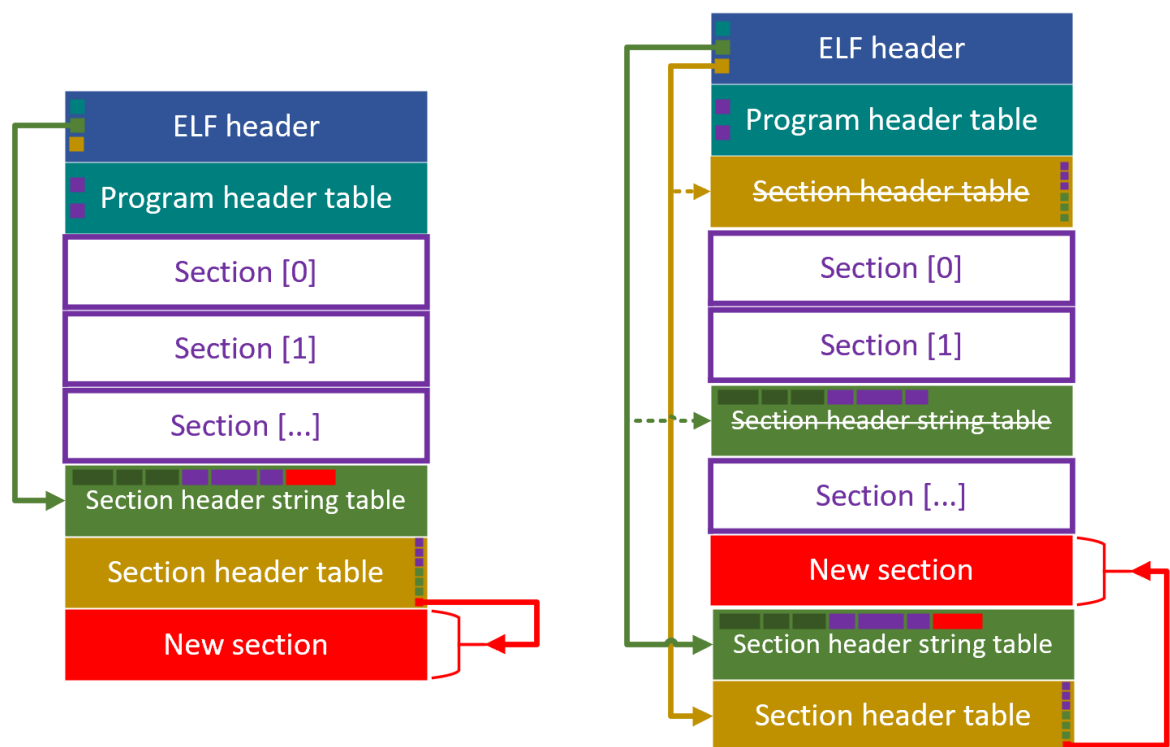
另外，还需要保证数据插入后，所有 section 的地址对齐要求得到满足。Section header table 的起始地址也需要对齐 8 字节地址，以充分利用总线宽度提升性能。

由于 ELF 文件的布局方式因编译器而异，在 [签名程序的代码仓库](#) 中，保存了 GCC 编译出的 ELF 示例文件与 [Golang](#) 编译出的 ELF 示例文件。它们有着完全不同的布局。

如果 section header string table 位于 section 的中间，向其中插入内容会使布局在插入位置之后的所有 section 在文件中后移。不仅会影响到 section header table 中的数据，还需要修正 program header table。为了避免对 program header table 中的内容产生影响，程序将在文件的最底部重新创建一个 section header string table，并使 section header table 中的数据结构指向该位置；原先的 section header string table 已经无效，但不被删除。

另外，如果 section header table 布局于位于任意 section 之前，在其中插入新的 section header entry 也会导致 program header table 中各 segment 的文件偏移与实际位置不一致。我们用了类似的处理方法，将原有 section header table 在文件末尾复制一份，并插入数据。如右下图所示。

图中红色部分为新插入的数字签名信息及其相关的描述信息：




注入签名后的 ELF 文件

在签名程序的具体实现中，我们将 ELF 文件中的以下部分装载到内存中。这两部分的内存副本将用于记录和修改各个 ELF 结构在文件中的偏移变化和长度变化，并最终被写回 ELF 文件，覆盖原有内容：

- ELF header
- Section header table

我们为 section header table 多分配了一段 section header entry 的内存，并在最后一个空出的 entry 中为新增的数字签名 section 设置信息。程序会将签名数据插入到从文件末尾开始的第一个能被 8 整除的地址，作为新增 section 的数据。这个 **地址** 以及 **签名数据的长度** 被记录到最后一个 section header entry 对应的结构体中。在这一过程进行的同时，程序顺带计算在文件中需要插入的字节偏移位置与字节数量。

 这里阐明文章中 **插入** 和 **覆盖** 的概念。对于普通的文件写入，可被理解为从文件中的某个偏移位置开始 **覆盖** 文件中的原有内容，(如果覆盖内容没有超出文件原有长度) 将不会改变文件的长度。而插入特指 **将插入位置之后的原有内容向后挤**，从而一定会引发文件长度的变化。

对于文件中位置、长度固定，而内容需要被修改的部分，使用覆盖操作 (比如 ELF header)；对于文件中必须新增的数据，使用插入操作 (比如新增 section 的字符串名)。


接下来将被插入文件的数据由两部分组成：

- 签名 section 的名称字符串，以及用于补齐 8 字节地址的无用字符 (因为 `.shstrtab` 长度任意)
- Section header table 内存副本中最后一个新增的 entry

基于上述工作，签名程序能够确定 ELF header 中记录的 section header table 偏移量 ( `e_shoff` ) 和 section header 数量 ( `e_shnum` ) 的最终值。另外，通过遍历 section header table，程序还可以确定在 `.shstrtab` 中插入新 section 名称字符串的位置，以及用于使插入位置之后所有 ELF 结构的地址对齐的填充字节个数。

最后一步，程序将内存中的 ELF header 副本与 section header table 副本 (不包含新增 entry) 中的偏移量修正后，覆盖原文件中的相同部分；然后，在计算好的文件偏移处插入准备好的

数据。

 在插入数据时，应当先处理在文件中插入位置靠后的数据。如果先处理在文件中插入位置靠前的数据，将会影响到之后插入的数据在文件中的插入偏移位置。

也就是说，插入的顺序与具体的 ELF 结构无关，只和 ELF 结构在文件中的偏移位置有关，插入位置靠后的数据先被插入。

签名程序会在被签名 section 名称的基础上加上 `_sig` 后缀，成为对应的签名数据 section。如，对于保存 ELF 程序指令的 `.text` section，签名程序会将签名数据作为一个名为 `.text_sig` 的 section 附加到 ELF 文件中。如果在解析 ELF section 的过程中发现已经存在名称以 `_sig` 为后缀的 section，则签名程序会中止退出，以防止重复签名。

```
1 $ readelf -a sign-target
2 ...
3 Section Headers:
4 [Nr] Name                Type                Address              Offset
5      Size                EntSize            Flags  Link  Info  Align
6 ...
7 [29] .text_sig             PROGBITS            0000000000000000    000020c0
8      000000000000001d1  0000000000000000    0      0      0      8
9 ...
```

在签名期间，签名程序会产生一些临时文件，用于保存被签名的若干个 section 的数字签名数据；在签名完成之后，这些临时文件将会被清除。

签名程序可以指定一个被签名文件名，以及一个可选的输出文件名。如果不指定输出文件名，对于一个名为 `elf` 的 ELF 文件，在签名成功后，签名程序会保留未被签名的旧版本 ELF 文件 `elf.old` 作为备份，而注入签名后的新 ELF 文件将会被命名为原先的 `elf`。具体的使用方法参见 [后续](#)。

## 5.3 参考资料

A tutorial introduction to *libelf*

Adding section to ELF file

ELF format manipulation

How to remove a specific ELF section, without stripping other symbols?

## 第三部分 - 使用方式

# Chapter 6 - 密钥/证书生成

## 6.1 密钥生成的配置文件

Linux 内核中，已经给出了一个可以生成 X.509 格式 **自签名证书** 的配置文件模板 ( `certs/x509.genkey` ) :

```
1 [ req ]
2 default_bits = 2048
3 distinguished_name = req_distinguished_name
4 prompt = no
5 string_mask = utf8only
6 x509_extensions = myexts
7
8 [ req_distinguished_name ]
9 0 = WatchDog
10 CN = ELF verification
11 emailAddress = mrdrivingduck@gmail.com
12
13 [ myexts ]
14 basicConstraints=critical,CA:FALSE
15 keyUsage=digitalSignature
16 subjectKeyIdentifier=hash
17 authorityKeyIdentifier=keyid
```

其中可自行修改的配置信息及其含义：

- `default_bits` 表示密钥长度 (2048-bit / 4096-bit / ...)
- `0` 表示密钥所属组织的名称
- `CN` 代表密钥的通用名称
- `emailAddress` 代表密钥所属组织的电子邮箱

---


## 6.2 密钥生成

有了 **上述配置文件** 之后，通过 OpenSSL 工具，使用如下命令产生 **公私钥** 并导入 **证书**：

```
1 $ openssl req -new -nodes -utf8 -sha256 -days 36500 -batch -x509 \  
2   -config x509.genkey -outform PEM -out kernel_key.pem \  
3   -keyout kernel_key.pem  
4 Generating a RSA private key  
5 .....+++++  
6 .....+++++  
7 writing new private key to 'kernel_key.pem'  
8 -----
```

命令将使用配置文件 `x509.genkey` 中的信息，产生一对 RSA 公私钥；生成一个名为 `kernel_key.pem` 的 PEM 格式的 X.509 自签名证书，过期日期为 36500 天后 (永不过期)；生成的 RSA 私钥与公钥证书全部导入到 `kernel_key.pem` 中。

该文件可以作为 ELF 签名程序的输入 (签名需要私钥和 X.509 格式的公钥证书)，同时也是编译内核时 `CONFIG_SYSTEM_TRUSTED_KEYS` 选项指向的文件。

 我们在 [内核源代码仓库](#) 与 [签名程序代码仓库](#) 放置了同一个 PEM 文件，其中的公私钥仅用于测试，请不要在生产环境中直接使用。暴露私钥会导致所有的机制失效。

## 6.3 参考资料

[OpenSSL 命令 - req](#)

[Generating signing keys](#)

[Administering/protecting the private key](#)

# Chapter 7 - 编译内核

## 7.1 内核编译配置

在 Linux 内核编译配置 `.config` 中，提供了 `SYSTEM_TRUST_KEYS` 的编译选项。其描述如下：

```
certs/Kconfig

1 config SYSTEM_TRUSTED_KEYS
2     string "Additional X.509 keys for default system keyring"
3     depends on SYSTEM_TRUSTED_KEYRING
4     help
5         If set, this option should be the filename of a PEM-formatted file
6         containing trusted X.509 certificates to be included in the default
7         system keyring. Any certificate used for module signing is implicitly
8         also trusted.
9
10        NOTE: If you previously provided keys for the system keyring in the
11        form of DER-encoded *.x509 files in the top-level build directory,
12        those are no longer used. You will need to set this option instead.
```

这个编译选项允许一个 PEM 格式的 X.509 证书被添加到系统默认的密钥环上。编辑这个选项，将其设置为 **PEM 格式公钥证书的路径**：

```
.config

1 #
2 # Certificates for signature checking
3 #
4 CONFIG_SYSTEM_TRUSTED_KEYRING=y
5 CONFIG_SYSTEM_TRUSTED_KEYS="<PATH_TO_CERT>/kernel_key.pem"
```

---

## 7.2 编译内核



完成上述选项的配置后，刷新配置并编译内核：

```
1 $ make oldconfig
2 $ make -j8
```

在编译过程中，应该可以看到如下信息：

```
1 ...
2 EXTRACT_CERTS    <PATH_TO_CERT>/kernel_key.pem
3 AS               certs/system_certificates.o
4 AR               certs/built-in.o
5 ...
```

编译完成后，安装编译后的新内核：

```
1 $ sudo make modules_install
2 $ sudo make install
```

重启电脑开机运行，查看 proc 文件系统中的 `/proc/keys`（需要 root 权限）。如果能够看到自行生成的密钥，那么说明该密钥已经被放置于内核的系统密钥环中。

```
0c87ab47 I----- 1 perm 1f030000 0 0 asymmetri WatchDog: ELF ver
```

---

## 7.3 参考资料

Public keys in the kernel

# Chapter 8 - 挂载模块

## 8.1 编译模块

在 [模块源代码仓库](#) 中，我们提供了 `Makefile` 用于编译该模块。编译模块时，需要使用到相对应的内核源代码目录。`Makefile` 中默认使用 **系统当前内核** 的源码路径 (当然也可以指定其它的内核源码路径)。

```
1 $ make
2 make -C /lib/modules/4.15.0+/build M=/home/mrdrivingduck/Desktop/linux-ker
3 make[1]: Entering directory '/home/mrdrivingduck/Desktop/linux-kernel-elf-
4 CC [M] /home/mrdrivingduck/Desktop/linux-kernel-elf-sig-verify/linux-ke
5 Building modules, stage 2.
6 MODPOST 1 modules
7 CC /home/mrdrivingduck/Desktop/linux-kernel-elf-sig-verify/linux-ke
8 LD [M] /home/mrdrivingduck/Desktop/linux-kernel-elf-sig-verify/linux-ke
9 make[1]: Leaving directory '/home/mrdrivingduck/Desktop/linux-kernel-elf-s
```


## 8.2 使用模块

使用 `modinfo` 命令查看模块附带信息：

```
1 $ modinfo binfmt_elf_signature_verification.ko
2 filename: /home/mrdrivingduck/Desktop/linux-kernel-elf-sig-verify/li
3 alias: fs-binfmt_elf_signature_verification
4 version: 1.0
5 description: Binary handler for verifying signature in ELF section
6 author: zonghuaxiansheng <zonghuaxiansheng@outlook.com>
7 author: mrdrivingduck <mrdrivingduck@gmail.com>
8 license: Dual MIT/GPL
9 srcversion: 24C778301DE1DD13C1BB3CF
10 depends:
11 name: binfmt_elf_signature_verification
12 vermagic: 4.15.0+ SMP mod_unload
```

使用 `insmod` 命令装载模块：

```
$ sudo insmod binfmt_elf_signature_verification.ko
```

 如果内核中的模块签名验证选项 ( `CONFIG_MODULE_SIG` ) 被开启，则内核会在装载模块之前，验证模块中的数字签名。如果模块签名验证失败，内核将会被标记为 tainted (被污染)。

因此，在装载模块之前，需要使用内核源码树下的 `scripts/sign-file` 手动对模块进行签名。更多信息可见：<https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html#configuring-module-signing>

使用 `lsmod` 命令确认模块是否被装载：

```
1 $ lsmod | grep binfmt_elf_signature_verification
2 binfmt_elf_signature_verification    16384  0
```

使用 `rmmod` 命令卸载模块：

```
$ sudo rmmod binfmt_elf_signature_verification
```

---

## 8.3 参考资料

[Stackoverflow - How to compile a kernel module](#)

[CSDN - 如何编译 Linux 内核模块](#)



# Chapter 9 - ELF 签名


## 9.1 ELF 文件签名程序的构建

首先，需要安装签名程序依赖的库。基于 Debian 系列的 Linux 发行版可以使用 **APT** (Advanced Package Tool) 工具轻易安装这些依赖：

```
$ sudo apt install libssl-dev
```

在 GitHub 上克隆 **ELF 文件签名程序** 的代码仓库，通过 `make` 命令自动编译、构建签名程序，并对构建出的签名程序本身进行签名：

```
1 $ make
2 cc -o elf-sign elf_sign.c -lcrypto
3 ./elf-sign.signed sha256 certs/kernel_key.pem certs/kernel_key.pem elf-sign
4 --- 64-bit ELF file, version 1 (CURRENT).
5 --- Little endian.
6 --- 29 sections detected.
7 --- Section 0014 [.text] detected.
8 --- Length of section [.text]: 9200
9 --- Signature size of [.text]: 465
10 --- Writing signature to file: .text_sig
11 --- Removing temp signature file: .text_sig
```

 由于自行构建得到的 `elf-sign` 也是一个 ELF 程序，因此，在它可以用于对其它 ELF 文件进行签名之前，其自身必须先被签名，否则内核将拒绝执行这个 ELF 程序。在仓库中，我们提供了一个已经被测试密钥 (`certs/kernel_key.pem`) 签名后的签名程序 `elf-sign.signed`，并用这个签名程序对 `make` 命令构建的 `elf-sign` 进行签名。

如果您自行生成了私钥与公钥证书，那么您可能需要在一个 **未装载 ELF 签名验证模块** 且 **确认安全** 的内核上，对自行构建得到的 `elf-sign` 进行自签名。然后，这个被签名后的签名程序可以使用在装载 ELF 签名验证模块的内核上。

如果一切正常，通过 `readelf` 或 `objdump` 命令，可以在构建成功的 `elf-sign` 中看到名为 `.text_sig` 的 section；而签名前被备份的原始版本 ELF 文件 `elf-sign.old` 中则没有这个 section：

```
1 $ readelf -a elf-sign
2 ...
3 [29] .text_sig          PROGBITS          0000000000000000  00006cb0
4          00000000000001d1 0000000000000000   0      0      0      8
5 ...
```

```
1 $ objdump -s elf-sign
2 ...
3 Contents of section .text_sig:
4 0000 308201cd 06092a86 4886f70d 010702a0 0.....*.H.....
5 0010 8201be30 8201ba02 0101310d 300b0609 ...0.....1.0...
6 0020 60864801 65030402 01300b06 092a8648 `.H.e....0...*.H
7 0030 86f70d01 07013182 01973082 01930201 .....1...0.....
8 0040 01306e30 56311130 0f060355 040a0c08 .0n0V1.0...U....
9 0050 57617463 68446f67 31193017 06035504 WatchDog1.0...U.
10 0060 030c1045 4c462076 65726966 69636174 ...ELF verificat
11 0070 696f6e31 26302406 092a8648 86f70d01 ion1&0$...*.H....
12 0080 09011617 6d726472 6976696e 67647563 ....mrdrivingduc
13 0090 6b40676d 61696c2e 636f6d02 144879f3 k@gmail.com..Hy.
14 00a0 22f1671d f368169f 7f3ed0f8 4150a2cc ".g..h...>..AP..
15 00b0 86300b06 09608648 01650304 0201300d .0...`.H.e....0.
16 00c0 06092a86 4886f70d 01010105 00048201 ...*.H.....
17 00d0 00ad85d9 206b0473 2742b31b 0a22e22f .... k.s'B..."/
18 00e0 27693ef2 c5b128d1 699adef7 0217b02d 'i>...(.i.....-
19 00f0 5296daf7 cc3bc6a6 b876928f a459d69f R....;...v...Y..
20 0100 625fc1ae dbcb383d f7070aad a41dd3a1 b_....8=.....
21 0110 820054d3 1c971e96 be2c858b cc439625 ..T.....,...C.%
22 0120 5e228ef2 623f2087 3ab349d9 4c3906db ^"..b? ..:I.L9..
23 0130 58ecbbac 43ccd826 8ca5bd8e 16194514 X...C..&.....E.
24 0140 021b8b5d 72a67370 bf5f33f3 a4b4a824 ...]r.sp._3....$
25 0150 78505c7d 4ef054a6 2b622152 589008fe xP\}N.T.+b!RX...
26 0160 121d3d32 2aca10c3 4d6800c7 386887e4 ..=2*...Mh..8h..
27 0170 af9b3649 c77a7813 40a75d55 64f69c2a ..6I.zx.@[.]Ud...*
28 0180 51d34e1e cd242dec bda586b5 2027071b Q.N..$-..... '..
29 0190 8f1d903b c84197ee ce293e23 187fbf8c ...;.A...)>#....
30 01a0 5f7c3df3 4055130f c769a797 3b9b2b63 _|=.@U...i...;.+c
31 01b0 a6f4754f 8e32036d d27243f5 a6a38017 ..u0.2.m.rC.....
32 01c0 2745de68 5723b263 c17f9921 5571c6da 'E.hW#.c...!Uq..
```

```
33 01d0 54 T
34 ...
```

## 9.2 签名程序的使用方式

```
1 $ ./elf-sign
2 Usage: elf-sign [-h] <hash-algo> <key> <x509> <elf-file> [<dest-file>]
3   -h,          display the help and exit
4
5 Sign the <elf-file> to an optional <dest-file> with
6 private key in <key> and public key certificate in <x509>
7 and the digest algorithm specified by <hash-algo>. If no
8 <dest-file> is specified, the <elf-file> will be backup to
9 <elf-file>.old, and the original <elf-file> will be signed.
```

elf-sign 的参数含义：

1. hash-algo - 摘要算法 (可以选用其它 内核支持的摘要算法)
2. key - 存放用于签名的私钥的文件路径
3. x509 - 存放用于签名的公钥证书的文件路径
4. elf-file - 待签名的目标 ELF 文件
5. dest-file (可选) - 签名后的输出文件名

比如，用测试证书中的 RSA-2048 私钥与 SHA-256 摘要算法，对仓库中的示例 ELF 文件进行签名：

```
1 $ ./elf-sign sha256 \
2   certs/kernel_key.pem certs/kernel_key.pem \
3   test/func/hello-golang hello-golang
4 --- 64-bit ELF file, version 1 (CURRENT).
5 --- Little endian.
6 --- 23 sections detected.
7 --- Section 0001 [.text] detected.
8 --- Length of section [.text]: 528699
9 --- Signature size of [.text]: 465
10 --- Writing signature to file: .text_sig
```

```
11 --- Removing temp signature file: .text_sig
```

或将一个已有的 ELF 文件 ( `/bin/ls` ) 签名为一个自定义文件名的 ELF 文件 ( `myls` ) :

```
1 $ ./elf-sign sha256 \  
2     certs/kernel_key.pem certs/kernel_key.pem \  
3     /bin/ls myls  
4 --- 64-bit ELF file, version 1 (CURRENT).  
5 --- Little endian.  
6 --- 28 sections detected.  
7 --- Section 0014 [.text] detected.  
8 --- Length of section [.text]: 74969  
9 --- Signature size of [.text]: 465  
10 --- Writing signature to file: .text_sig  
11 --- Removing temp signature file: .text_sig
```

---

## 9.4 参考资料

Manually signing modules



## 第四部分 - 评估与测试

# Chapter 10 - 内核签名验证开销评估

## 10.1 开销分析

我们的解决方案在内核为执行 ELF 文件进行准备的过程中，引入了额外的签名验证机制，从而引入了 CPU、内存、I/O 上的开销：

- CPU 的开销：摘要算法和 *RSA* 算法所带来的计算开销
- 内存开销：将 ELF 文件中必要的 table 和 section 保存在内存缓冲区所带来的开销
- I/O 开销：将 ELF 文件中必要的 table 和 section 从磁盘装入内存的开销

这里的开销主要分为两个维度：**时间**上的开销，以及 **空间**上的开销。

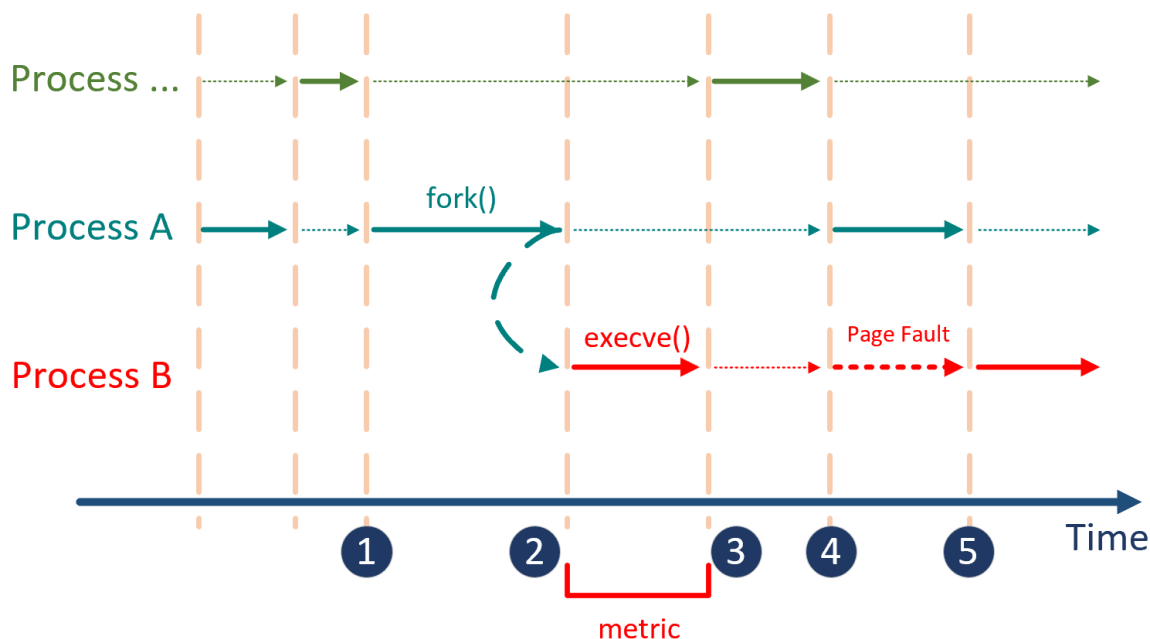
---

## 10.2 时间开销

我们评估了签名验证机制所带来的时间开销。具体地，评估带有签名验证机制的内核，在执行一个 ELF 文件前的准备工作中，比不进行签名验证的内核慢多少。

我们选用的测试对象是 `/bin` 下的一些系统内置命令，它们实际上都是 ELF 文件，如 `ls` `mv` 等。我们对它们进行签名后，分别在签名验证模块已装载和未装载的条件下，以空参数执行这些 ELF 程序足够多次。虽然以空参数执行 ELF 文件会导致一些程序错误退出，但我们只关心 **内核为执行二进制文件而进行准备工作** 的时间，而并不关心二进制文件真正开始执行后的时间与状态。

我们以 `fs/exec.c` 文件中实现 `execve` 系统调用具体逻辑的 `do_execveat_common()` 函数作为评估对象。该内核函数主要负责为可执行文件准备运行环境，包括在进程控制块中设置新进程的入口地址、内存等相关工作。最重要的是，该函数中包括了对二进制格式处理程序链表的遍历和使用。



时间开销的衡量指标

如图所示，虚线指示的区间表示进程没有被 CPU 运行；实线指示的区间表示进程正在被 CPU 运行。Process A 代表启动执行 ELF 文件的进程 (比如 `bash`)，Process B 代表执行 ELF 文件的新进程。

- 在 1 号时刻，A 进程接收了执行一个 ELF 文件的命令，于是调用 `fork()`，开始从自身复制一个新的进程 B
- 在 2 号时刻，`fork()` 结束，新复制的进程 B 准备就绪。为了避免 `copy-on-write` 优化机制带来额外的开销，内核通常会调度新的进程 B 首先执行。进程 B 开始执行 `execve()`，载入新的 ELF 文件
- 在 3 号时刻，`execve()` 结束。此时，进程 B 的下一条将被执行的指令被设置为 ELF 文件的入口指令。但由于 `load-on-demand` 机制，第一条指令所在的内存页还没有被载入内存。此时，所有进程全部处于就绪状态，可以被调度器调度
- 在 4 号时刻，调度器选择执行 B 进程。在访问 ELF 文件的第一条指令时，发生 **缺页异常 (Page Fault)**，由异常处理程序从磁盘上将该指令所在的页装入内存；在此过程中，调度器可能会选择其它进程执行
- 在 5 号时刻，B 进程所要访问的第一条指令已经在内存中；当调度器调度到 B 进程时，B 进程真正开始运行

我们评估的是签名验证机制在 `execve()` 的执行过程中引入的开销，即图中 2 号时刻与 3 号时刻之间的时间。我们将每个待测试的 ELF 文件分别在签名验证模块装载和未装载的内核

上运行足够多的次数 (1000 次)，并统计前后 `execve()` 执行时间增加的百分比。软硬件环境与统计单位：

- Machine : Lenovo® R720
- CPU : Intel® Core™ i7-7700HQ CPU @ 2.80GHz \* 8
- Memory : 15.5 GiB
- Disk : 60GB SSD
- OS : Deepin 15.11, x64
- 时间单位：微秒 (μs)

ELF 文件	平均执行时间 (无签名验证)	平均执行时间 (签名验证)	开销倍数
cp	94.94	869.132	9.1545
df	97.087	712.007	7.3337
echo	94.597	511.307	5.4051
false	91.869	494.750	5.3854
grep	98.181	1275.288	12.9892
kill	96.786	466.024	4.8150
less	96.456	907.837	9.4119
ls	97.489	823.131	8.4433
mkdir	95.568	704.105	7.3676
mount	100.155	520.235	5.1943
mv	93.213	909.927	9.7618
rm	92.616	628.854	6.7899
rmdir	95.957	531.206	5.5359
tar	97.198	1838.067	18.9105
touch	94.066	698.057	7.4209

true	93.025	491.792	5.2867
umount	106.101	503.504	4.7455
uname	94.898	579.819	6.1099

测试脚本与结果位于 [签名程序的代码仓库](#) 中。

我们对测试结果进行了分析与解读：

1. 没有签名验证的执行时间基本稳定在 90-100  $\mu$ s，因为内核内置的 ELF 处理模块仅需要访问 ELF header 和 program header table，而 program header table 在布局上紧接 ELF header 之后，因此当内核将 ELF 的头 128 字节装入主存时，紧接其后的 program header table 也已经位于主存 **高速缓冲 (buffer)** 中，访问时不再需要额外的 I/O 开销
2. 加入签名验证机制后，`execve` 系统调用的执行时间与程序代码段长度相关 - 比如 `tar` 程序的代码段特别长，因此需要更长的时间分配内存、计算代码段摘要
3. 加入签名验证机制后，`execve` 系统调用的执行时间平均翻了 5 - 7 倍，主要原因是将 ELF 文件的 section header table、section header string table、签名 section 与被签名 section 载入内存引入了额外的 I/O 开销。由于 `execve` 系统调用原本就能够在 90-100  $\mu$ s 左右的时间内结束，是一个相对较快的过程；而额外的 I/O 操作会将 `execve` 系统调用的执行时间提升至少半个数量级，因为 I/O 的速度远比内存慢

## 10.3 空间开销

这里主要关注内核为待验证的 ELF section 所动态分配的内存空间。由于内存空间会在使用完毕后被回收，这里只关心动态分配内存空间的 **极限值**。即，内核最大能够支持分配多大的内存将一个 ELF section 装入内存进行验证。

在所有目前经过测试的程序中，具有最长的 `.text` section 的 ELF 文件为 **chromium** (Version 71.0.3578.80, Developer Build, 64-bit, deepin 15.11 x64)。该 ELF 文件的大小为 134147808 字节 (127.9333 MB)，其中，`.text` section 的长度为 80549925 字节 (76.8184 MB)。ELF 数字签名验证模块成功地验证了其中的数字签名。

## 10.4 参考资料

[Wikipedia - Copy-on-write](#)

[Mr Dk.'s blog - linux-kernel-comments-notes/Chapter 12 - 文件系统/Chapter 12.15 - exec.c 程序](#)

[Stackoverflow - Does Linux load program-pages on demand?](#)

# Chapter 11 - ELF 签名开销评估

## 11.1 开销分析

ELF 签名程序所带来的开销主要来自于：

- 遍历 ELF 文件中的每个 section，将需要被签名的 section 装入内存
  - 对装入内存的 section 计算摘要，并用公私钥生成数字签名
  - 将数字签名作为新的 section 加入 ELF 文件中
- 

## 11.2 时间开销

时间开销与 ELF 文件中需要被签名的 section 个数呈正相关。目前版本中，我们仅对 `.text` 进行签名和验证。但 ELF 数字签名验证模块与 ELF 数字签名程序都是以验证或签名多个 section 为目标设计的。

对于单个 section 的签名来说，时间开销与 section 的数据长度呈正相关。越长的 section 需要越多的时间把数据装入内存并计算摘要。由于对 section 计算出的摘要是一个定长的序列，因此通过摘要计算数字签名的时间开销可被认为是固定的。

由于计算出的 section 摘要长度固定，并且长度较短，因此 ELF 文件的签名工作基本在每一台机器上都能快速完成。

---

## 11.3 空间开销

这里主要关注签名数据作为新的 section 被加入原 ELF 文件后，会使 ELF 文件大小增加多少。

上面已经提到，由于对一个 section 计算出的摘要是一个 **定长** 序列，那么计算得到的数字签名也是定长的。以 SHA-256 和 RSA-2048 作为签名工具链，我们测试得到每个被签名保护的 section 会为 ELF 文件带来 560 字节左右的额外开销，包含了：

- 签名数据
- Section header table 中新的 section 项
- Section header string table 中该 section 的名称字符串

对于现代计算机的硬件来说，这个空间开销基本上可以忽略。规模越大的 ELF 文件，这个开销所占的百分比越小。以 **chromium** 的 ELF 文件为例 ( `/usr/lib/chromium/chromium` )，签名前的程序大小为 134147808 字节 (127.9333 MB)，签名后的程序大小为 134148352 字节 (127.9338 MB)，产生的额外开销为 544 字节 (0.0004%)。



## Chapter 12 - 适配不同的 ELF 文件布局

## 12.1 目的

ELF 文件的布局因编译器不同而大相径庭。由于签名程序需要对 ELF 文件中的结构进行调整，因此应当适配不同的 ELF 文件布局。在签名程序代码仓库的 `test/func/` 目录下，我们试图收集布局尽可能不同的合法 ELF 文件，用于测试签名程序 `elf-sign` 的健壮性。

## 12.2 ELF by GCC

上述目录中的 `hello-gcc` 是一个由极其简单的 C 程序通过 GCC 编译后得到的 ELF 文件：

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello world\n");
6     return 0;
7 }
```

该 ELF 文件的布局如下，特点为：

- Section header table 布局在所有 section 数据之后
- Section header string table 是布局在最后的 section

[illegible]

```

11 Version: 0x1
12 Entry point address: 0x530
13 Start of program headers: 64 (bytes into file)
14 Start of section headers: 6440 (bytes into file)
15 Flags: 0x0
16 Size of this header: 64 (bytes)
17 Size of program headers: 56 (bytes)
18 Number of program headers: 9
19 Size of section headers: 64 (bytes)
20 Number of section headers: 29
21 Section header string table index: 28

```

## 23 Section Headers:

24 [Nr]	Name	Type	Address	Offset
25	Size	EntSize	Flags Link Info	Align
26 [ 0]		NULL	0000000000000000	00000000
27	0000000000000000	0000000000000000	0 0	0
28 [ 1]	.interp	PROGBITS	0000000000000238	00000238
29	000000000000001c	0000000000000000	A 0 0	1
30 [ 2]	.note.ABI-tag	NOTE	0000000000000254	00000254
31	0000000000000020	0000000000000000	A 0 0	4
32 [ 3]	.note.gnu.build-i	NOTE	0000000000000274	00000274
33	0000000000000024	0000000000000000	A 0 0	4
34 [ 4]	.gnu.hash	GNU_HASH	0000000000000298	00000298
35	000000000000001c	0000000000000000	A 5 0	8
36 [ 5]	.dynsym	DYNSYM	00000000000002b8	000002b8
37	00000000000000a8	0000000000000018	A 6 1	8
38 [ 6]	.dynstr	STRTAB	0000000000000360	00000360
39	0000000000000082	0000000000000000	A 0 0	1
40 [ 7]	.gnu.version	VERSYM	00000000000003e2	000003e2
41	000000000000000e	0000000000000002	A 5 0	2
42 [ 8]	.gnu.version_r	VERNEED	00000000000003f0	000003f0
43	0000000000000020	0000000000000000	A 6 1	8
44 [ 9]	.rela.dyn	RELA	0000000000000410	00000410
45	00000000000000c0	0000000000000018	A 5 0	8
46 [10]	.rela.plt	RELA	00000000000004d0	000004d0
47	0000000000000018	0000000000000018	AI 5 22	8
48 [11]	.init	PROGBITS	00000000000004e8	000004e8
49	0000000000000017	0000000000000000	AX 0 0	4
50 [12]	.plt	PROGBITS	0000000000000500	00000500
51	0000000000000020	0000000000000010	AX 0 0	16
52 [13]	.plt.got	PROGBITS	0000000000000520	00000520
53	0000000000000008	0000000000000008	AX 0 0	8
54 [14]	.text	PROGBITS	0000000000000530	00000530
55	000000000000001a2	0000000000000000	AX 0 0	16
56 [15]	.fini	PROGBITS	00000000000006d4	000006d4
57	0000000000000009	0000000000000000	AX 0 0	4
58 [16]	.rodata	PROGBITS	00000000000006e0	000006e0
59	0000000000000011	0000000000000000	A 0 0	4
60 [17]	.eh_frame_hdr	PROGBITS	00000000000006f4	000006f4
61	000000000000003c	0000000000000000	A 0 0	4

```

62  [18] .eh_frame      PROGBITS      00000000000000730  00000730
63      00000000000000108  0000000000000000  A      0      0      8
64  [19] .init_array    INIT_ARRAY    0000000000200db8  00000db8
65      00000000000000008  00000000000000008  WA      0      0      8
66  [20] .fini_array    FINI_ARRAY    0000000000200dc0  00000dc0
67      00000000000000008  00000000000000008  WA      0      0      8
68  [21] .dynamic       DYNAMIC       0000000000200dc8  00000dc8
69      000000000000001f0  00000000000000010  WA      6      0      8
70  [22] .got           PROGBITS      0000000000200fb8  00000fb8
71      00000000000000048  00000000000000008  WA      0      0      8
72  [23] .data          PROGBITS      0000000000201000  00001000
73      00000000000000010  00000000000000000  WA      0      0      8
74  [24] .bss           NOBITS        0000000000201010  00001010
75      00000000000000008  00000000000000000  WA      0      0      1
76  [25] .comment       PROGBITS      0000000000000000  00001010
77      00000000000000029  00000000000000001  MS      0      0      1
78  [26] .symtab        SYMTAB        0000000000000000  00001040
79      000000000000005e8  00000000000000018      27     43     8
80  [27] .strtab        STRTAB        0000000000000000  00001628
81      000000000000001ff  00000000000000000      0      0      1
82  [28] .shstrtab      STRTAB        0000000000000000  00001827
83      00000000000000fe  00000000000000000      0      0      1
84  Key to Flags:
85      W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
86      L (link order), O (extra OS processing required), G (group), T (TLS),
87      C (compressed), x (unknown), o (OS specific), E (exclude),
88      l (large), p (processor specific)
89
90  There are no section groups in this file.
91
92  ...

```

## 12.3 ELF by Golang

上述目录中的 `hello-golang` 是一个由极其简单的 Go 程序编译后得到的 ELF 文件：

```

1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Println("Hello world!")
7  }

```

该 ELF 文件的布局如下，特点为：

- Section header table 紧随 program header table 之后，布局在所有 section 数据之前
- Section header string table section 布局在其它 section 中间

```
1 $ readelf -a hello-golang
2 ELF Header:
3   Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
4   Class:                               ELF64
5   Data:                                   2's complement, little endian
6   Version:                               1 (current)
7   OS/ABI:                               UNIX - System V
8   ABI Version:                           0
9   Type:                                   EXEC (Executable file)
10  Machine:                               Advanced Micro Devices X86-64
11  Version:                               0x1
12  Entry point address:                    0x44f4d0
13  Start of program headers:               64 (bytes into file)
14  Start of section headers:              456 (bytes into file)
15  Flags:                                   0x0
16  Size of this header:                    64 (bytes)
17  Size of program headers:                56 (bytes)
18  Number of program headers:              7
19  Size of section headers:                64 (bytes)
20  Number of section headers:              23
21  Section header string table index: 3
22
23 Section Headers:
24  [Nr] Name                               Type                               Address                               Offset
25      Size                               EntSize                             Flags Link Info Align
26  [ 0]                                     NULL                                0000000000000000 00000000
27      0000000000000000 0000000000000000 0 0 0
28  [ 1] .text                               PROGBITS                           0000000000401000 00001000
29      0000000000008113b 0000000000000000 AX 0 0 16
30  [ 2] .rodata                             PROGBITS                           0000000000483000 00083000
31      00000000000041a3a 0000000000000000 A 0 0 32
32  [ 3] .shstrtab                           STRTAB                              0000000000000000 000c4a40
33      0000000000000010b 0000000000000000 0 0 1
34  [ 4] .typelink                           PROGBITS                           00000000004c4b60 000c4b60
35      00000000000000b44 0000000000000000 A 0 0 32
36  [ 5] .itablink                           PROGBITS                           00000000004c56a8 000c56a8
37      00000000000000040 0000000000000000 A 0 0 8
38  [ 6] .gosymtab                           PROGBITS                           00000000004c56e8 000c56e8
39      00000000000000000 0000000000000000 A 0 0 1
40  [ 7] .gopclntab                          PROGBITS                           00000000004c5700 000c5700
```

```

41      0000000000004ded8 0000000000000000 A      0      0      32
42  [ 8] .noptrdata      PROGBITS      00000000000514000 00114000
43      000000000000cbdc 0000000000000000 WA      0      0      32
44  [ 9] .data          PROGBITS      00000000000520be0 00120be0
45      00000000000006b10 0000000000000000 WA      0      0      32
46  [10] .bss           NOBITS       00000000000527700 00127700
47      0000000000001c688 0000000000000000 WA      0      0      32
48  [11] .noptrbss      NOBITS       00000000000543da0 00143da0
49      00000000000002698 0000000000000000 WA      0      0      32
50  [12] .debug_abbrev   PROGBITS      00000000000547000 00128000
51      000000000000001b5 0000000000000000      0      0      1
52  [13] .debug_line     PROGBITS      000000000005471b5 001281b5
53      00000000000010539 0000000000000000      0      0      1
54  [14] .debug_frame    PROGBITS      000000000005576ee 001386ee
55      00000000000012054 0000000000000000      0      0      1
56  [15] .debug_pubnames PROGBITS      00000000000569742 0014a742
57      00000000000007d9c 0000000000000000      0      0      1
58  [16] .debug_pubtypes PROGBITS      000000000005714de 001524de
59      0000000000000a50d 0000000000000000      0      0      1
60  [17] .debug_gdb_script PROGBITS      0000000000057b9eb 0015c9eb
61      0000000000000002d 0000000000000000      0      0      1
62  [18] .debug_info     PROGBITS      0000000000057ba18 0015ca18
63      000000000000063dcf 0000000000000000      0      0      1
64  [19] .debug_ranges   PROGBITS      000000000005df7e7 001c07e7
65      00000000000005f80 0000000000000000      0      0      1
66  [20] .note.go.buildid NOTE         00000000000400f9c 00000f9c
67      00000000000000064 0000000000000000 A      0      0      4
68  [21] .symtab         SYMTAB       00000000000000000 001c7000
69      00000000000012018 00000000000000018      22    101      8
70  [22] .strtab         STRTAB       00000000000000000 001d9018
71      000000000000121c4 0000000000000000      0      0      1
72  Key to Flags:
73      W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
74      L (link order), O (extra OS processing required), G (group), T (TLS),
75      C (compressed), x (unknown), o (OS specific), E (exclude),
76      l (large), p (processor specific)
77
78  There are no section groups in this file.

```

## 12.4 更多 ELF 布局

后续我们将会收集更多不同布局的 ELF 文件，作为 `elf-sign` 程序的回归测试集合。

## 后记

# 后记

本项目开始于 2020 年 4 月。因 COVID-19 疫情，整个春天基本上除了蹲在家里哪儿也去不了。蹲在家里并不意味着没有事情干，从二月上旬开始，我已经在家中恢复了科研状态，另也有一些项目在忙。但是看到 第九届“中国软件杯” 的这道题目后，我们开了一个小时的钉钉会议，决定要把它做出来。

在过去的一年里，我已经阅读了不少 Linux 内核代码。心里一直有个小小的愿望：读了这么些代码，也得写一点内核代码了。通过这个项目，算是小小地满足了心里的那点意难平。本科一年级以后，我已经很久没有写过这么多 C 代码了。Happy kernel hacking! 🧑🔧

这个项目只是窥探了 Linux 内核中极微小的一部分，功能也不算很底层很核心，其规模也就与一个简单的设备驱动程序相当。2020 年初，Linux 内核代码量已经达到了 2780 万行。近年来，西方发达国家亡我之心不死，前有中兴芯片事件，后有华为被禁用 Android GMS 服务，近日又有中国高校和科研单位被禁用 MATLAB。可以说，现在中国的科技环境很严峻。无论是国产芯片，还是国产操作系统、国产基础软件，我们都还有很长很长的路要走。

希望国家也好、企业也好、个人也好，能有越来越多的人关注这些相对冷门的领域。争夺 AI、5G、物联网的主动权固然重要，但 自主可控的核心 才是我国免遭别国卡脖子的底牌。

张靖棠 (@mrdrivingduck)

2020 年 6 月 20 日 于南京