Bootarmor 入门教程和用户手册

发布 *0.1.0*

赵俊德

2022 年 05 月 08 日

Contents

版本 0.1

主页 https://github.com/dashingsoft/bootarmor

联系方式 jondy.zhao@gmail.com

作者 赵俊德

Bootarmor 是以提供绝对安全为目标的一种操作系统，基于 Linux，结合不同 CPU 的特性，从系统引导开始接管所有内存管理，从而为用户提供一个安全运行环境。

用户通过命令行命令，可以将自己的应用程序转换成为安全应用，安全应用可以运行在安全系统 Bootarmor 中。运行在 Bootarmor 系统中的安全应用的代码和数据，操作系统中任何权限的用户（包括 root 在内）也无法读取和访问，无论是静态反编译，还是各种内核调试器和应用层调试器，都无法获取安全应用的代码和数据。

本文档适用于使用 Bootarmor 来保护自己应用程序的用户。

内容:

CHAPTER 1

了解 Bootarmor

Bootarmor 是以为软件产品提供绝对安全，确保软件产品发布之后，其代码不被使用者获取或者修改为目标的安全型嵌入式操作系统。

Bootarmor 使用 debian 包的形式提供了一个 Linux 内核，把原来的嵌入式系统的内核替换为 Bootarmor 提供的内核之后，就升级成为安全操作系统。

Bootarmor 可以直接保护二进制代码文件，例如可执行文件和动态库，对于 C#，Java，Python 等使用伪代码和虚拟机的语言，通过将解释器编译成为安全应用，间接的来进行保护。例如对 Python 脚本来说，首先将 Python 解释器使用 Bootarmor 保护起来，然后把 Python 脚本作为数据文件进行加密，最后使用安全的 Python 解释器来运行加密脚本。通过这种间接的方式来保护 Python 脚本。对于 Java 也类似，先将 Java 的运行库 jre 使用 Bootarmor 进行保护，然后把 Java 的.class 文件直接加密，最后使用安全的 jre 运行加密脚本。对于 C# 和其它脚本语言也类似。

传统的二进制代码保护工具，例如 VMProtect, Themedia 等相比较，它们不管采用虚拟指令，还是各种方式，都离不开操作系统的制约，无法实现绝对的安全。并且使用虚拟指令的方式虽然提高了安全性，但是大大降低了性能。

芯片级别的安全性，例如 INTEL® SGX INSTRUCTION 等，需要用户编写专门的程序来使用这些特性，对于普通用户来说，对代码要进行重整，工作量很大，并且都需要从厂商获得许可。

而使用虚拟机（VMX）的方式，对性能的影响则比较大。

Bootarmor 从系统引导开始接手，运行于芯片的最高权限级别，全面接管内存的管理和分配，然后在次高的权限级别装载并运行 Linux 操作系统，所以其不受操作系统的制约，可以提供绝对的安全性。

和虚拟机（VMX）的方式相比，Bootarmor 只接管内存，其它任务还交给操作系统来执行，基本不会对性能产生太大的影响。

Bootarmor 目前分为社区版和企业版两个版本。

CHAPTER 2

基本使用教程

Bootarmor 支持在原来的 Debian Linux（例如 Ubutnu 或者 RaspberryOS）中直接安装新的安全内核，重新启动之后就可以将原来的系统升级成为可以运行安全应用的操作系统。

Bootarmor 提供了命令行工具*btarmor*，可以帮助用户完成大部分的功能。

2.1 支持平台

目前 Bootarmor 内核只支持 arm64 架构的 Debian 11 (bullseye) 和 RaspberryOS，其它系统和版本的支持会在以后逐渐加入。

命令行工具*btarmor* 可以跨平台运行，目前支持 arm64 和 x86_64 两种架构。

2.2 安装命令行工具 btarmor

可以根据使用习惯使用下面的任意一种方式安装

1. 使用 apt 安装

启动任意 Debian Linux 系统之后，使用下面的方式增加安装源之后，就可以使用常用的管理命令 *apt* 命令安装 Bootarmor 提供的各种包:

```
wget -O /etc/apt/trusted.gpg.d/btarmor-archive-keyring.gpg \
    https://btarmor.dashingsoft.com/apt/debian/btarmor-archive-keyring.gpg
```

(下页继续)

(续上页)

```
sudo echo "deb https://btarmor.dashingsoft.com/apt/debian bullseye contrib" \
          > /etc/apt/sources.list.d/btarmor.list

sudo apt update
```

命令行工具*btarmor* 在包 *btarmor-cli* 中，使用下面的命令直接安装:

```
sudo apt install btarmor-cli
```

查看命令*btarmor* 是否安装成功:

```
btarmor -v
```

安装成功的话，会显示相关的版本信息。

使用下面的命令清除 btarmor 相关的安装源配置:

```
sudo rm /etc/apt/trusted.gpg.d/btarmor-archive-keyring.gpg \
        /etc/apt/sources.list.d/btarmor.list
```

2. 使用 pip 安装

如果对 Python 比较熟悉，可以直接使用下面的命令安装:

```
pip install https://btarmor.dashingsoft.com/tools/btarmor-cli-1.0.1.tar.gz
```

安装完成之后同样会创建命令*btarmor*

2.3 安装安全操作系统

安装好命令行工具*btarmor* 之后，直接使用下面的命令安装安全操作系统:

```
sudo btarmor boot
```

然后重新启动系统就进入 *btarmor-os*

2.4 创建安全应用

创建安全应用就是把原来的可执行文件，动态库和数据文件使用命令转换成为安全应用。

例如，转换被保护的应用程序 `/opt/foo` 为安全应用:

```
btamor make -i /opt/foo
```

该目录下面的所有可执行文件和动态库，以及使用到的所有系统动态库都会被转换成为安全应用。

关于转换应用程序的更多方式，请参考命令手册*btarmor make*

对于使用 C 编写的应用程序，可以在源代码级别提供更多的安全性选项，具体请参考 C 用户使用手册

一般来说，转换成为安全应用之后应用程序的代码就无法使用逆向工程进行反编译，就可以发布给客户。

但是对于企业版用户来说，可以使用下面的发布功能来进一步加强内核的安全性。

2.5 发布安全应用

备注: 发布功能仅适用于企业版，社区版不提供这个功能。

发布功能会对内核进行加壳处理，从而提供更高的安全性，使用下面的命令转换安全内核为发布模式:

```
btarmor deploy
```

更多详细的使用方式，请参考命令手册*btarmor deploy*

CHAPTER 3

btarmor

安全操作系统*btarmor-os* 的命令行工具。

3.1 语法

```
btarmor <command> [options]
```

3.2 描述

btarmor 命令行工具主要的功能包括

- 将原来的 Debian 系统升级成为安全操作系统*btarmor-os*
- 将应用程序/动态库/数据文件转换成为加密保护的安全文件

常用的子命令包括:

```
boot     升级当前系统为安全系统
make     生成各种安全文件
deploy   发布安全系统
```

运行 `btarmor <command> -h` 可以查看各个命令的详细使用方法。

3.3 btarmor boot

子命令 `boot` 用于安装安全操作系统*btarmor-os*

直接运行下面的命令会安装新的安全内核，这个命令需要使用 `root` 权限，一般用于初始化安全操作系统:

```
sudo btarmor boot
```

安装完成之后需要重启系统。重新启动之后，内核就升级成为安全操作系统*btarmor-os*

下面的命令则显示安全系统的相关信息:

```
btarmor boot --status
```

3.4 btarmor make

子命令 `make` 用于将可执行文件，动态库和数据文件等，转换成为 Bootarmor 保护的安全文件，转换后生成的文件是经过加密处理的，只能在*btarmor-os* 系统上运行。

btarmor make 可以运行在*btarmor-os* 系统，直接将应用程序转换成为安全应用。

btarmor make 也可以运行在普通的 Linux 系统，将应用程序转换成为安全应用，然后再把安全应用发布到*btarmor-os* 系统中。

3.4.1 语法

```
btarmor make <options> <PATTERN>
```

3.4.2 选项

-i, --inplace	生成的文件直接覆盖原来的文件
-O, --output PATH	生成的文件保存在另外一个路径，默认是 *dist*
-sys, --share	使用共享模式进行保护，一般用于保护系统动态库
-sh, --safe-heap	不允许内核访问应用程序申请的堆空间
-ss, --safe-stack	不允许内核访问运行栈（局部变量）
-f FILE	从文件里面读取需要处理的文件名称

3.4.3 描述

子命令 make 用于将命令行列出的一个或者多个文件转换成为安全文件，可以使用通配符。

例如:

```
cd /home/jondy/myapp
btarmor make foo *.so config.txt
```

还可以把所有需要处理的文件名称都存放到一个文件里面，例如 `files.list`

```
foo
libtoo.so
config.txt
```

然后在命令行使用 `-f` 指定这个文件，例如:

```
cd /home/jondy/myapp
btarmor make -f files.list
```

默认情况下生成的新文件会保存在另外一个输出目录 `dist` ，使用选项 `--output` 可以指定输出目录。

如果指定了选项 `--inplace` ，那么输出文件会直接覆盖原来的文件。

选项 `-sys` 一般用于加密系统动态库，主要是为了保护系统命令不能被修改，并且在高级模式下面能够执行系统命令。因为专用模式下面，没有加密的任何可执行文件都无法运行。使用该选项加密的文件允许内核访问，同时这个可执行文件也只能运行在应用程序层，而没有运行在安全应用层。

选项 `--safe-heap` 用于保护内存堆，这里面一般是程序使用 `malloc` 申请的内存空间，默认情况下是允许内核访问的。如果这些数据不需要被内核访问，使用该选项可以提高安全性。

选项 `--safe-stack` 用于保护运行栈，这里面一般是程序的调用框架和局部变量，默认情况下是允许内核访问的。如果这些数据不需要被内核访问，使用该选项可以提高安全性。

3.4.4 示例

- 转换目录 `foo` 下面的所有文件，保存生成的安全文件到到默认输出目录 `dist`:

  ```
  btarmor make foo/*
  ```

- 只转换两个动态库文件，转换之后直接覆盖原来的文件:

  ```
  btarmor make -i dist/*.so
  ```

- 没有使用任何内核服务的程序，对内存堆和运行栈也进行保护:

```
btarmor make -sh -ss myapp
```

- 加密保护数据文件到 `dist/data.txt`:

```
btarmor make data.txt
```

- 使用共享模式加密系统动态库:

```
sudo btarmor make -i --share /usr/lib/*.so
```

- 加密系统命令 `ls`:

```
sudo btarmor make -i -sys /usr/bin/ls
```

3.5 btarmor deploy

用于产品开发完成之后，发布运行安全应用的系统到最终用户。

这个命令仅适用于企业版，社区版没有提供此功能。

3.5.1 语法

```
btarmor deploy <options>
```

3.5.2 选项

-s, --senior	安全内核使用高级模式
-e, --exclusive	安全内核使用专用模式

3.5.3 描述

在产品准备交付给用户之前，需要使用该命令把内核功能进行加壳固化，清理开发辅助的文件等。

选项 `--senior` 可以启用安全内核的高级模式。启用高级模式之后，Linux 内核将不能访问安全应用的内存，如果需要访问，必须得到安全应用的授权。这样能提高安全性，但是也需要更多的了解操作系统的内存管理，ELF 文件的组成，以及编译器的工作原理等相关知识。否则，安全应用可能无法正常运行。

选项 `--exclusive` 可以启用安全内核的专用模式。启用专用模式之后，则普通可执行文件，包括系统命令都无法执行，只有加密后的安全应用可以运行，这样可以进一步提高安全行。

3.6 btarmor patch

下载安全内核的源代码补丁，用于定制自己的安全内核。

3.6.1 语法

```
btarmor patch <options> [VERSION]
```

3.6.2 选项

-O, --output PATH 保存到指定目录，默认是 patches

-l, --list 列出所有可用版本的补丁名称

3.6.3 描述

直接下载和当前系统内核版本相同的补丁，保存到目录 *patches/*:

```
btarmor patch
```

下载应用于内核版本 *5.10.73* 的补丁:

```
btarmor patch 5.10.73
```

查看所有支持的内核版本的补丁:

```
btarmor patch --list
```

也可以从下面的链接查看和下载对应版本的*btarmor-os* 补丁

https://btarmor.dashingsoft.com/kernel/patches/

CHAPTER 4

C 用户使用手册

本章详细说明了如何保护 C 语言开发的应用程序，适用于 C 开发人员，详细了解如何使用*btarmor* 保护的 C 应用程序的各个组成部分。

4.1 默认保护模式

对于 C 开发的应用程序来说，基本的保护包括

- 代码段
- 数据段，存放全局变量和函数体内使用 `static` 声明的变量
- 字符串常量
- 内存堆，程序申请的内存
- 运行栈，存放局部变量
- 数据文件

受保护内存只允许应用程序本身访问，不允许任何外部访问，包括 Linux 内核，所以提供了最大限度的安全性。当然这样也意味在调用系统服务的时候，如果这些数据需要被内核访问，就需要进行一些额外的设置和处理。

默认情况下，代码段，数据段，字符串常量是被保护的，而内存堆，运行栈和数据文件是没有被保护的。

例如，在默认编译链接选项下面

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

// 全局变量存放被保护
int ga;
int gb = 1;
static int sa;
static int sb = 2;

// 字符串常量 "abc" 也受到保护
char *gc = "abc";

int main(int argc, char *argv[])
{
    // 局部变量存放在内存栈中，没有被保护
    int a;

    // 字符串常量 "efg" 受到保护
    char *b = "efg";

    // 使用 static 声明的变量受到保护
    static int d;
    static int e = 2;

    // 下面的所有可执行代码都受到保护
    d = ga + sa + gb + sb + a + d + e;

    printf("Got result %d\n", d);

    return 0;
}
```

对于使用 C 语言开发的可执行文件来说，如果使用默认保护模式，那么不需要修改源代码。

默认保护模式的基本使用步骤

1. 正常编译一个可执行文件:

```
gcc -o myapp myapp.c
```

2. 使用*btarmor* 转换成为安全应用:

```
btarmor make -i myapp
```

对于动态库，和可执行文件的保护是一样的。下面的例子是一个包含可执行文件和私有动态库的应用程序的

基本的使用步骤

1. 正常编译动态库和可执行文件到发布目录 *dist*:

```
mkdir dist
gcc -o dist/myapp myapp.c
gcc -shared -o dist/mydll.so mydll.c
```

2. 使用*btarmor* 转换整个目录下面的文件为安全文件:

```
btarmor make -i dist/
```

如果需要使用非默认的保护模式，则需要对源代码进行一定的修改，这时候需要导入的头文件 `btapp.h`，并在源代码中使用相应的宏，来实现需要的功能。

4.2 共享字符串和全局变量

当内核需要访问字符串或者全局变量的时候，因为默认保护模式下它们受到保护，直接访问会出错。例如系统调用 `open` 需要把文件名称传递到内核，下面这个示例中默认模式下是无法正常运行的，因为字符串常量默认是处于保护模式，无法被内核访问。

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

char *filename1 = "data1.txt";

int main(int argc, char *argv[])
{
    int fd;
    char *filename2 = "data2.txt";

    // 出错
    fd = open("data.txt", 0);
    if (fd > 0)
        close(fd);

    // 出错
    fd = open(filename1, 0);
    if (fd > 0)
        close(fd);

    // 出错
    fd = open(filename2, 0);
```

(下页继续)

(续上页)

```
    if (fd > 0)
        close(fd);

    return 0;
}
```

这时候需要修改源代码，可以在函数中使用 BTS 声明字符串常量。这种字符串会被存放到当前函数的运行栈中，而运行栈默认情况是允许内核访问的，从而避免内核无法访问字符串的问题。下面的例子就可以解决上面的问题

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#include "btapp.h"

int main(int argc, char *argv[])
{
    int fd;
    fd = open(BTS("data.txt"), 0);
    if (fd > 0)
        close(fd);
    return 0;
}
```

宏 BTS 可以应用于字符串常量，字符串数组，结构体等，但是不能用于字符串指针

```
char arr[10];
struct {
    int a;
    int b;
} stv;

BTS("hello");
BTS(arr);
BTS(stv);

char *s = "hello";
BTS(s);                 // 错误
```

宏 BTS 只支持内核读取数据，并不支持内核回写数据。如果需要内核写回数据到共享变量，那么需要使用一组宏 BTPS 和 BTPE 。在需要共享的时候调用 BTPS ，结束共享调用 BTPE 。

例如，

```
// 全局变量 gt 受到保护，内核无法直接访问
struct T {
    int a;
    int b;
} gt = { 1, 2 };

// 20 号系统调用需要修改 gt 里面的数据，原来的调用方式
// syscall(20, &gt);

// 使用 BTPS/BTPE 向内核共享可写的全局变量
BTPS(&gt);
syscall(20, &gt);
BTPE(&gt);
```

备注: 宏 BTS 和 BTPS/BTPE 都使用通过把数据拷贝到运行栈来实现和内核共享，如果应用程序使用 --safe-stack 对运行栈进行了保护，那么这些宏是无法共享数据到内核的。

4.3 保护内存堆

如果需要保护内存堆中的数据，可以使用下面的方式

1. 使用宏 mmap 分配空间，并使用 MAP_BTMDATA 标志。例如，

```
#include "btapp.h"

int main(int argc, char *argv[])
{
    ssize_t len = 1024;
    char *ps = (char*)mmap(
        0,
        len,
        PROT_READ | PROT_WRITE,
        MAP_PRIVATE | MAP_MDATA,
        -1,
        0);

    ...

    munmap(ps, len);
    return 0;
}
```

2. 使用宏 BTMMAP 分配空间。例如，

```
#include "btapp.h"

int main(int argc, char *argv[])
{
    ssize_t len = 1024;
    char *ps = (char*)BTMMAP(len);

    ...

    munmap(ps, len);
    return 0;
}
```

3. 在生成安全应用的时候指定选项 --safe-heap:

```
btarmor make --safe-heap myapp
```

4.4 保护内存栈

默认情况下内存栈没有进行额外保护，如果需要保护内存栈，可以在生成安全应用的时候使用选项 --safe-stack:

```
btarmor make --safe-stack myapp
```

4.5 保护数据文件

如果需要对数据文件的进行保护，那么需要对源代码进行相应的修改，必须使用 mmap 的方式来读写文件，而不能使用 fread 和 fwrite 等方式读写文件。

首先对数据文件进行加密，下面使用命令 make 对数据文件 data.txt 进行加密，加密后的文件保存为 dist/data.txt:

```
btarmor make data.txt
```

然后使用 mmap 分配内存，同时指定标志 MAP_BTARMOR 。下面是一个简单的示例

```
#include "btapp.h"

int main(int argc, char *argv[])
{
```

(下页继续)

(续上页)

```
    char *fmap;
    char buf[200];
    int size = 100;
    int offset = 0;

    int fd = open("data.txt", 0);
    if (fd < 0) {
        perror("open file");
        return -1;
    }

    fmap = mmap(0, size, PROT_READ, MAP_PRIVATE | MAP_BTARMOR, fd, offset);
    if (fmap == MAP_FAILED) {
        perror("map file");
        close(fd);
        return -1;
    }
    memcpy(buf, fmap, size);
    buf[size] = 0;

    printf("file %d bytes: %s\n", size, buf);

    close(fd);
    return 0;
}
```

CHAPTER 5

附录

5.1 btarmor-os

btarmor-os 是一个基于 Debian Linux 安全操作系统。

btarmor-os 目前提供两个不同的版本:

- *Btarmor 社区版*
- *Btarmor 企业版*

5.1.1 Btarmor 社区版

默认安装的版本是社区版，任何人可以自由下载使用。

5.1.2 Btarmor 企业版

企业版需要注册之后得到授权文件，把授权文件部署在社区版上，就可以升级成为企业版。

社区版和企业版的区别在于

- 社区版使用共享的相同的 Key
- 企业版使用私有的 Key
- 企业版的安全内核经过加壳来保证安全，社区版的安全内核没有经过加壳

5.1.3 Btarmor 内核模式

- 高级模式
- 专用模式

高级模式的主要特征:

- 高级模式不允许 Linux 内核直接访问安全应用的内存。如果需要访问，必须得到安全应用的授权。

默认情况下，Linux 内核是可以访问安全应用的内存，高级模式能够提供更高的安全性，但是也需要更多的了解操作系统的内存管理，ELF 文件的组成，以及编译器的工作原理等相关知识。否则，安全应用可能无法正常运行，或者即便运行，也降低了安全性。

专用模式的主要特征:

- 只有经过加密的安全应用才可以运行

默认情况下，普通的可执行文件和动态库还可以继续使用，安全应用也可以正常运行。但是在专用模式下面，普通的可执行文件和动态库都无法使用，必须转换成为相应的安全文件才可以继续使用。

这两个模式可以同时启用，并不互斥。

5.1.4 安全应用

使用 btmake 命令转换之后的应用程序称之为安全应用。

安全应用是经过加密后的应用程序。

安全应用的代码，数据和堆栈都是可以设置成为不允许内核访问的。

安全应用无法运行普通的 Linux 系统中，只能运行在*btarmor-os* 中的安全层。

在同一个系统中，每一个安全应用使用自己独有的 Key 被加密。

5.1.5 认证应用

使用 btmake 命令的同时指定选项 `--sys` ，这样转换之后的可执行文件和动态库称之为认证应用和认证动态库。

认证应用无法运行普通的 Linux 系统，只能运行在*btarmor-os* 中。

认证应用运行在应用层，而不是运行在安全层。

认证应用的代码和数据是能够被内核直接访问的。

认证动态库被安全应用调用的时候运行在安全层，被认证应用调用的时候运行在应用层。

其它用户无法修改认证应用。

在同一个系统中，所有的认证应用使用相同的 Key 被加密。

5.1.6 安全层

运行安全应用的最高权限层。

5.1.7 内核层

运行操作系统内核的次高权限层。

5.1.8 应用层

运行一般应用程序的最低权限层。

5.2 Debian Packages

btarmor 提供了下列 Debian 包

开发环境的功能包，目前支持 amd64 和 arm64 两种架构

- btarmor-cli 提供了命令行工具 btarmor ，大部分的开发相关的工作都可以使用这个命令完成
- btarmor-toolkit 提供了头文件和示例源文件，用于帮助生成具备高级保护功能的 c 应用程序

运行环境的包，主要包括两个

一是用于替换原来的 Linux 内核包:

- raspberrypi-btarmor (适用于 raspberry os，替换原来的 raspberrypi-kernel)
- linux-image-bt-arm64 (适用于 debian 系统，替换原来的 linux-image-arm64)
- linux-btarmor (适用于 ubuntu 系统，替换原来的 linux-generic)

二是用于装载安全应用和动态库的包，主要是替换原来的 ld.so 的功能，支持安全库的装载

- btarmor-runtime