第 33 卷 第 5 期 计 算 机 工 程 2007 年 3 月
Vol.33 No.5 Computer Engineering March 2007
·软件技术与数据库· 文章编号：1000—3428(2007)05—0047—03 文献标识码：A 中图分类号：TP311

基于 Linux 内核的通用软件封装器的设计与实现

赵文进，石昭祥，胡荣贵
(解放军电子工程学院网络工程系，合肥 230037)

摘 要：基于 Linux 系统设计并实现的通用软件封装器旨在操作系统内核中嵌入一个安全框架，该框架通过对指定的任何软件进行封装，实时监控软件与操作系统之间的系统调用，并依据封装器对其进行处理可实现多种安全策略(如访问控制、入侵检测)，从而保护主机资源。实验结果表明，通用软件封装器在系统内核中运行稳定，达到了预期效果。

关键词：通用软件封装器；封装器支持子系统；封装器激活原则子系统

Design and Realization of Generic Software Wrapper Based on Linux Kernel

ZHAO Wenjin, SHI Zhaoxiang, HU Ronggui
(Department of Network Engineering, Electronic Engineering Institute of PLA, Hefei 230037)

【Abstract】 Generic software wrapper (GSW) system is realized in Linux kernel. The building of the GSW system aims to provide a security framework in Linux kernel, on which many security polices, such as access control and intrusion detection, can be realized in order to protect host resources by wrapping any specified program, monitoring system calls between program and operating system, taking corresponding measures according to wrappers. The GSW runs smoothly in Linux kernel and reaches anticipated results.

【Key words】 Generic software wrapper(GSW); Wrapper support subsystem(WSS); Wrapper activation criteria subsystem(WACS)

保护主机资源不受恶意软件(木马、病毒等)的窃取和破坏一直是研究的热点，目前采用的防范措施也是多种多样，如入侵检测、密码验证、访问控制、审计监控等，它们的共同特点是在操作系统用户模式下运行，本身就容易受到恶意软件的攻击。而通用软件封装器是在系统内核模式下运行，并作为系统内核的一部分，它可避免模式的切换开销和恶意代码的攻击。GSW 还为多种安全策略在操作系统内核中的实现提供了一个平台，通过对指定的任何程序进行封装，监控被封装程序与操作系统之间的系统调用并根据封装器的指示作出反应来实现对主机资源的保护。GSW 能够动态地装卸载出系统内核，并能动态地装卸载其中用自然描述语言描述的安全策略。

考虑到现在很多主流服务器安装的都是 Linux 操作系统，且 Linux 是开放源码的，并且支持 LKM 机制，因此在 Linux 内核中构建了通用软件封装器。GSW 按其内部功能可分为 3 个子系统：封装器支持子系统(WSS)，封装器激活原则子系统(WACS)和封装器(WS)子系统。

目前美国NAI实验室也对GSW进行了研究，但考虑问题的出发点不同，NAI 研究GSW的目的是为了加强COTS (Commercial off-the-shelf)软件的安全[1~3]，而本文的研究出发点是在内核的系统调用级构建一个能实现多种安全策略以保护主机资源的通用平台。

1 通用软件封装器的功能结构

GSW 可对运行于 Linux 系统中的任何程序进行封装。它主要由 3 个子系统构成：封装器支持子系统，封装器激活原则子系统和封装器子系统，其中 WACS 子系统含有封装器激活原则，原则中指明了要实施封装的程序及所采用的封装器；WS 子系统中包含多个封装器，而封装器中指定了封装哪些系统调用，并指示如何处理；封装器支持子系统则为 WACS 和 WS 提供了一个执行环境，并负责管理。

在每个新进程执行前都由 WSS 自动检查激活原则链表，查看新进程是否是要封装的进程，若是，则激活相应的封装器。而在封装进程退出时，则由 WSS 负责撤消其所对应的被激活的封装器。一个封装器可被多个被封装程序所复用，而一个被封装程序也可能对应着多个封装器。

图 1 中所示的进程 A 是不打算被封装的进程，进程 B 是打算被封装的进程。粗线箭头表示运行进程 B 所经过的流程，细线箭头表示进程 A 所经过的流程。当进程 A 或 B 执行时，通过 0x80 中断陷入内核模式，并执行 execve 系统调用。WSS 截取到这个系统调用，通过查询 WACS 子系统判断当前要执行的进程是否是要封装的。如果是(如进程 B)，则再查询 WS 子系统，获得对应进程 B 的封装器，把这些封装器加入活动的封装器结构中，并激活这些封装器，然后正常执行进程 B。进程 B 开始执行后，这些封装器就开始发挥作用，按封装器中的内容对进程 B 实施封装；而进程 A 由于不打算被封装，则不需要查询 WS 子系统，直接正常执行。

封装器是通过截获进程中部分或全部的系统调用来实现对进程的封装，它完全控制了进程与操作系统之间的通信。而且，封装器运行不存在环境切换的开销，因此工作效率高，并且是自保护的。封装器与封装进程同步工作，对每个系统调用，封装器可以观察到参数值和返回值，并可以拒绝、登

作者简介：赵文进(1975－)，女，讲师、博士生，主研方向：计算机仿真，信息安全；石昭祥，教授、博导；胡荣贵，副教授
收稿日期：2006-03-20 **E-mail：**l-w-x-@sohu.com

记或正常执行该系统调用。

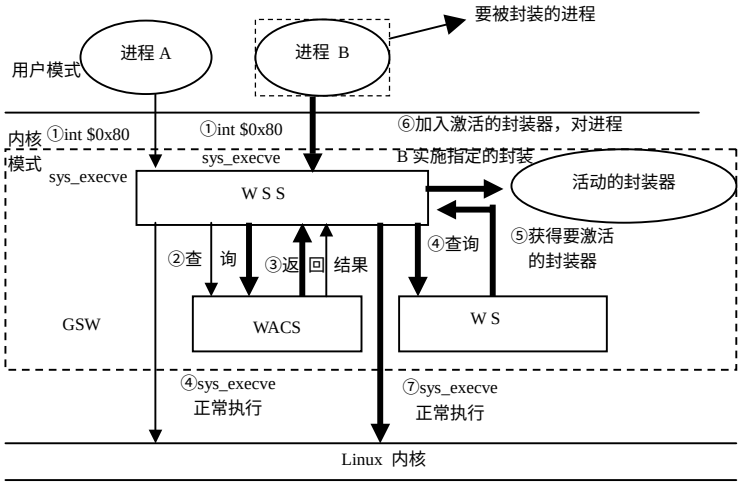

图 1 GSW 对进程实施封装的流程

2 Linux 系统中 WSS 的原理及设计

WSS 作为内核模块动态装入系统，也就是说它根据需要可随时装载入或卸载出内核。这需要操作系统提供这样一种机制，在 Linux 系统中称之为可装载内核模块机制(LKM)。

WSS作为LKM来实现，它跟踪要运行的进程，并在适当的时候根据激活原则为进程激活新的封装器。WSS作为内核模块在内核级运行，它实际上是Linux内核的一种扩展，它的实现使用了插入(Interposition)技术[4]。插入是捕获通过接口的事件并把这些事件传递到接口扩展部分的过程。扩展对事件进行处理，再把事件传递到原来的目的地，或强制事件返回。

图 2 中实线表示进程和内核之间的系统调用接口，虚线表示在接口上插入扩展部分。扩展代码截取原接口的事件，并把这些事件传递出去。图中，左边原来的系统调用接口在右边的每一层中都得到保持，这样就使插入的扩展代码对应用程序、内核和其它扩展都是透明的。

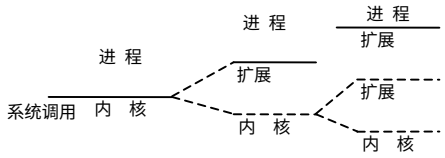

图 2 在接口处插入扩展

这种插入原理有几个有用的特性：

(1)透明性。因为插入代码既使用又实现了原来的接口，所以应用程序和内核并不觉察到插入代码的存在。

(2)递增性。扩展只需要捕获它们所感兴趣的事件，而并不需要处理通过接口的所有事件，例如，要登记 fork( )系统调用的扩展可以使用操作系统提供的 write( )系统调用来存储到日志上。因此扩展代码只需要实现所需的扩展功能，而不需要实现整个接口的功能。

(3)组合性。扩展对上保持了原有接口不变，而对下可递归地应用扩展，这样就使得多个独立的扩展可组合它们的功能。图 2 的右边显示了在原有的扩展下又增加了一层扩展，形成了一个扩展栈。

这种插入技术的透明性、递增性和可组合性使得它很适合于扩展已存在接口。WSS 把可信任代码有效地插入到 Linux 内核中去，而无须改变系统内核代码。它的功能就是监视程序的执行，检查激活原则链表，看是否是我们要封装的进程，若是，则激活此进程对应的封装器(可能有多个)，根据封装器中指定的内容对进程进行封装。WSS 插入到内核后，要监控程序的运行，采用的方法是拦截系统调用，修改 Linux 的系统调用表中的指针，使它指向 WSS 的处理函数，WSS 根据封装器中的封装原则进行处理，对原有的系统调用消息否定、接受或修改。接受或修改过的消息继续传递给原来的系统调用处理函数或下一个封装器。

WSS 为了对 WACS 和 WS 进行管理，提供了一些接口供它们调用，以下是部分接口函数：

```
(struct wrapper_struct) *wss_alloc_wrapstrt(char* wrappername,kmem_cache_t *wrapcache);//分配一个封装//器结构
int wss_free_wrapstrt(int id,kmem_cache_t *wrapcache); //释放一个封装器结构
(struct criteria_struct) *wss_alloc_criteriastrt(char* criterianame, kmem_cache_t *criteriacache);//分配一个激活//原则结构
int wss_free_criteriastrt(int id,kmem_cache_t *criteriacache); //释放一个激活原则结构
```

WSS 在内核中作为一个模块引出了这些接口函数，WACS 和 WS 通过调用这些接口函数向 WSS 发出请求，WSS 把指向结构的指针或执行状态返回给 WACS 和 WS，完成它们之间的交互。

3 WACS 和 WS 的实现及封装器的状态

WACS 子系统由激活封装器原则构成，它指明了要对哪些程序实施封装，封装程序所对应的封装器(即要激活的封装器)；而 WS 子系统则由很多封装器构成，封装器中包含的是安全策略，它指定了对程序封装哪些内容，并指示如何处理。这两个子系统在内核中都作为模块实现并运行。

因为这两个子系统需要频繁地分配和回收激活原则结构和封装器结构，所以它们使用了Linux的slab 层(它是Linux的通用数据结构缓存层)，WACS和WS子系统若要分配或释放内部的数据结构，都是调用WSS提供的接口函数，再由接口函数调用slab分配器的接口函数[5,6]：

```
kmem_cache_t *kmem_cache_create(const char *name, size_t size,size_t offset,unsigned long flags,void (*ctor)(void*,kmem_cache_t *,unsigned long);
void (*dtor)(void *,kmem_cache_t *,unsigned long));
int kmem_cache_destroy(kmem_cache_t *cachep);
void *kmem_cache_alloc(kmem_cache_t *cachep,int flags);
void kmem_cache_free(kmem_cache_t *cachep,void *objp);
```

WS 子系统中的封装器在系统内核中经历了 5 种状态的变化：初始状态是封装器刚装入到系统内核时的状态；准备状态是封装器已经装入到内核中但还没有被激活时的状态；激活状态：封装器要封装的进程已执行，WSS 使其对应的封装器进入工作状态，进入工作状态的封装器就称为激活封装器；撤消激活状态：被封装进程退出执行，WSS 使其对应的封装器停止工作状态，停止工作状态的封装器就处于撤消激活状态；退出状态：封装器从系统内核中卸载的状态。

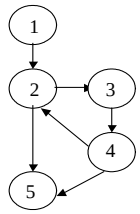

图 3 封装器状态变化

在图 3 中，1 表示封装器的初始化状态，2 表示封装器的准备状态，3 表示封装器的激活状态，4 表示撤消激活状态，5 表示封装器的退出状态。从 1 到 2 是封装器状态的自然变

化过程，从封装器刚装入系统内核开始，完成了初始化过程，就进入了状态 2。当 WSS 监测到被封装程序执行时，就激活其所对应的封装器，封装器从状态 2 变为状态 3。当 WSS 监测到被封装进程退出执行时，就使其所对应的封装器退出激活状态，也就是从状态 3 变为状态 4，若还要使用这些封装器，则从状态 4 变为状态 2，若不打算使用这些封装器，就直接从 4 变为 5；若封装器所对应的封装一直都未被执行或 GSW 系统停止运行，就可以把封装器从内核中卸载掉，封装器的状态就从状态 2 变为状态 5。这些状态变化的假设前提是在一个 CPU 的情况下。

4 实验结果分析

我们的软件实验平台是 Fedora Core 3(Linux 内核版本为 2.6.10)，shell 是 Bourne shell，编程语言是 C。

使用 GSW 系统对 gedit 程序(一个编辑程序)进行封装。先把 GSW 系统装入到内核中，在 GSW 系统中装入了 2 个规则和 2 个封装器，如图 4 所示，它显示了 GSW 系统内部的状态，此时，在 GSW 系统中装入了 2 个规则 c1 和 c2 及 2 个封装器 w1 和 w2，规则 c1 指定了对 gedit 程序使用 w1 封装器进行封装，规则 c2 指定了对 vi 程序使用 w2 封装器，w1 封装器中指定的封装内容是 open 系统调用，处理措施是 pass(通过)，w2 封装器指定了对 read 系统调用进行封装，处理措施是 deny(拒绝)，由于此时 gedit 程序和 vi 程序都没有执行，因此封装器 w1 和 w2 都处于未激活状态。

```
Nov 17 11:02:05 localhost kernel:query:Totally 2 criterias loaded in GSW at present
Nov 17 11:02:05 localhost kernel:query:*****************************************
Nov 17 11:02:05 localhost kernel:query: criterianame    program-to-wrap    wrapper-name
Nov 17 11:02:05 localhost kernel:query:*****************************************
Nov 17 11:02:05 localhost kernel:query:    c1                gedit                w1
Nov 17 11:02:05 localhost kernel:query:-----------------------------------------------------------
Nov 17 11:02:05 localhost kernel:query:    c2                vi                   w2
Nov 17 11:02:05 localhost kernel:query:-----------------------------------------------------------
Nov 17 11:02:05 localhost kernel:query:*****************************************
Nov 17 11:02:05 localhost kernel:query:Totally 2 wrappers loaded in GSW at present
Nov 17 11:02:05 localhost kernel:query:*************************************************
Nov 17 11:02:05 localhost kernel:query: wrapname      operation-name     action        state
Nov 17 11:02:05 localhost kernel:query:*************************************************
Nov 17 11:02:05 localhost kernel:query:    w1                open               pass        inactivated
Nov 17 11:02:05 localhost kernel:query:-----------------------------------------------------------
Nov 17 11:02:05 localhost kernel:query:    w2                read               deny        inactivated
Nov 17 11:02:05 localhost kernel:query:-----------------------------------------------------------
Nov 17 11:02:05 localhost kernel:query:*************************************************
Nov 17 11:02:05 localhost kernel:query:Totally 0 activated wrapper instances at present
Nov 17 11:02:05 localhost kernel:query:*************************************************************
Nov 17 11:02:05 localhost kernel:query:wrapname    wrapped-program    process-id    operation-name action
Nov 17 11:02:05 localhost kernel:query:*************************************************************
```

图 4 GSW 系统的内部状态示例

再执行 gedit 程序，GSW 系统中的 WSS 一侦测到 gedit 程序开始运行，就按封装器 w1 中的内容对指定的系统调用 open 进行监控。

图 5 是 GSW 对 gedit 程序实施封装过程的一部分内容：图中“wssmodule is loaded in kernel”是指 WSS 模块装入到 Linux 内核中，这时 GSW 系统的运行环境被初始化，准备装入激活封装器原则和封装器；criteria c1 is loaded in kernel”是指规则名为 c1 的激活封装器原则被装入到内核，接下来的 3 条语句表示规则 c2，封装器 w1 和 w2 装入到内核中，再接下来的语句说明对位于不同目录下的 gedit 程序都进行了封装(应用的封装器是 w1)，进程号是 5285，封装的系统调用是 open，接着，gedit 程序所产生的所有打开文件的系统调用(open)都得到了拦截，并根据 w1 封装器的指示，放它们通过。

```
Nov 17 11:00:53 localhost kernel:zwj-gsw:wssmodule is loaded in kernel !
Nov 17 11:01:04 localhost kernel:zwj-gsw:criteria c1 is loaded in kernel !
Nov 17 11:01:18 localhost kernel:zwj-gsw::criteria c2 is loaded in kernel !
Nov 17 11:01:27 localhost kernel:zwj-gsw::wrapper w1 is loaded in kernel !
Nov 17 11:01:55 localhost kernel:zwj-gsw::wrapper w2 is loaded in kernel !
Nov 17 11:03:03 localhost kernel:zwj-gsw::program /usr/Kerberos/sbin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /usr/Kerberos/bin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /usr/local/sbin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /usr/local/bin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /sbin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /bin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /usr/sbin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: :program /usr /bin/gedit(process id 5285)is wrapped ,wrapped operation is open.
Nov 17 11:03:03 localhost kernel:zwj-gsw:: pass:let the process 5285----/etc/ld.so.cache to execute the open operation normally!
Nov 17 11:03:03 localhost kernel:zwj-gsw:: pass:let the process 5285----/usr/lib/libeel-2.so.2 to execute the open operation normally!
```

图 5 GSW 对 gedit 程序的部分封装过程

从图 5 中可见，GSW 系统实现了对 gedit 程序的封装，除了 open 系统调用，GSW 可根据需要拦截到 gedit 程序产生的任何系统调用，只要对它使用相应的封装器即可。通过这一结果显示，GSW 系统完全达到了预期的目的，可对任何指定的软件进行封装，封装的具体内容由封装器指示。而且，被封装程序的运行速度与未封装前相比，没有任何停滞。

5 结束语

在 Linux 平台下实现了通用软件封装器系统，它实际上在内核的系统调用级构建了一个能实现多种安全策略的通用平台，通过对任何指定的程序进行封装，监控程序与操作系统之间的通信并根据封装器中的内容对之进行相应处理来达到保护主机资源的目的。而且封装器乃至整个 GSW 系统可动态装卸载，可根据安全要求在系统内核中同时实现多种安全策略，从而多层次、多方面地维护主机资源的安全。

GSW 系统的特点是具有易配置、不可回避性(即封装的程序没法避开系统强制执行的安全功能，如仲裁、审核，这意味着 GSW 不受在它控制下运行的软件的攻击)。而且由于 GSW 在内核模式运行，可避免模式的切换开销和恶意代码的攻击。实现的 GSW 系统在内核运行稳定并达到预期效果，使用者根本不能从时间上的延迟感觉 GSW 的存在。

下一步还要考虑它的移植性问题(如移植到 Windows 系列，OS/2 等)以及开发安全策略的自然描述语言和对应的编译器等一系列问题。

参考文献

1 Fraser T, Badger L, Feldman M. Hardening COTS Software with Generic Software Wrappers[C]//Proceedings of the 1999 IEEE Symposium on Security and Privacy. 1999.

2 Bershad B N, Savage S. Extensibility, Safety and Performance in the SPIN Operating System[C]//Proceedings of the 15th ACM Symposium on OS Principles. 1995: 267-284.

3 Biba K J. Intergrity Considerations for Secure Computer Systems[R]. USAF Electronic System Division, ESD-TR: 76- 372, 1977.

4 Ghormley D P, Petrou D, Steven H, et al. SLIC: An Extensibility System for Commodity Operating Systems[C]//Proceedings of the USENIX Annual Technical Conference'98. 1998-06.

5 Love R. Linux 内核设计与实现[M]. 北京: 机械工业出版社, 2004-11.

6 范 磊. Linux 内核源代码[M]. 北京: 人民邮电出版社, 2002-01.