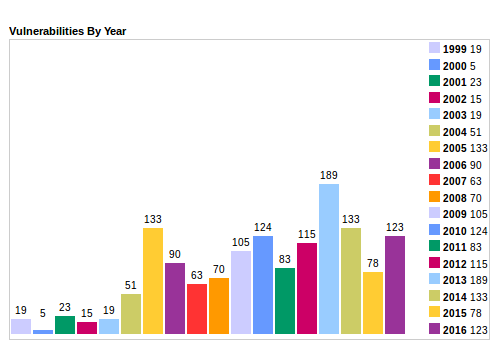
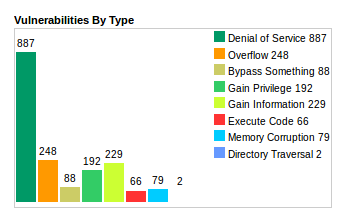
linus表达的想法有两个意思。其一，我们不能仅仅将安全寄托在Linux内核上，还有其他层实际上是应该要防止漏洞被利用的。第二，不管我们如何修复或者提升Linux内核，我们可能都不会看到它带来的真正好处，总会有其他的方法来达成这方面的提升。





内核自防护:

内核自防护是针对 Linux 内核对抗自身的安全缺陷的设计与实现。这个领域涉足广泛的问题，包括干掉一整个类型的 bug，阻止漏洞利用的方法和积极的检测攻击行为。这篇文档不讨论所有的议题，但这篇文档是整个内核自防护项目的一个起点和回答一些常见的问题。（欢迎提 交补丁）

在最糟糕的场景下，我们假设一名没有权限的本地攻击者拥有对内核内存任意读写访问的能力。在很多情况下，被利用的 bug 不会提供这个层面的访问能力，但在防御最糟糕的情况的同时我们也会讨论更多限制的攻击场景。root 用户大大的增加了攻击平面，而我们应该注意防御具有权限的本地攻击者则是更高的门槛（特别是当攻击者具有加载任意内核模块的场景）。

自防护系统的目标应该是让安全防护成为默认配置，不影响性能和调试以及经过严格的测试。要达到所有的目标可能不太容易，但值得我们都把这些问题罗列出来，再去解决或者接受。

降低攻击平面：

针对安全漏洞利用的最基础的防御是降低内核可被重定向执行的区域。这包含限制暴露给用户空间的 API，让内核自身的 API 更难以不正确的使用，可写内核内存区域的最小化，等等。

### 限制内核内存权限：

当所有的内核内存都是可写，重定向执行流变得容易。降低这些内核攻击的可用性需要针对内核内存更严格的权限。

#### 可执行的代码和只读数据必须不可写：

所有内核里可执行的内存区域都必须不可写。这很明显包含内核代码自身，但我们必须考虑所有的区域：内核模块，JIT 内存，等等。（有一些临时的例外以支持一些特性比如指令替换指令，断点，kprobes，等等。如果这些特性必须存在于内核，他们的实现应该在更新时的内 存变得临时可写，然后恢复到原有的权限。）

要支持这写特性，CONFIG\_DEBUG\_RODATA 和 CONFIG\_DEBUG\_SET\_MODULE\_RONX（名字起的太烂）保证代码不是可写，数据不是可执行以及只读数据即不能写也不能执行。

#### 函数指针和敏感变量必须不可写：

大量的内存区域都有函数指针用于内核查找和继续执行（例如， 描述符/向量表，文件/网络/etc 操作结构，等等）。这些变量都必须降低到最小。

很多这种变量可以通过设置 ”const” 而变成只读，所以他们可以存在于 .rodata 区域而不是 .data 区域来获得内核限制内存权限而带来的保护。

对于那些在 \_\_init 时初始化的变量可以标记为（新的和正在开发的）\_\_ro\_after\_init属性。

剩下的需要更新的变量就比较少见了（比如 GDT）。这些需要基础架构（类似上面提到的暂时性例外）的支持用于实现在例外更新以后的时间里为只读（比如被更新时，只有 CPU 线程执行更新操作会被赋予对内存的不可中断的写。

#### 从用户空间内存分离出内核内存：

内核必须永远不能执行用户空间的内存。内核也必须永远不能在没有显式预期的情况下访问用户空间内存。这些规则可以被基于硬件的限制（x86 的 SMEP/SMAP，ARM 的 PXN/PAN）或者通过模拟（ARM 的内存域）。这种方式阻断了执行和数据不能传递到被控制的用户空间内存里，只能强制攻击在内核内存中进行。

Shawn: SMEP 是 2011 年在 Intel Sandybridge 中加入的特性，SMAP 是在 2014 年的 Broadwell 中加入，ARM 的 PXN 是在 armv7 中加入，PAN 会在 armv8.1 中加入。遗憾的是，PaX/Grsecurity 的 KERNEXEC 和 UDEREF 均领先厂商数年。Anyway，最终 SMEP/SMAP 把兵工厂逼上了 Kernel ROP 的道路 ;-)

### 减少对系统调用的访问：

一个简单的为64位系统消除很多系统调用的方式是编译时不带 CONFIG\_COMPAT。当然，这是一种很罕见的场景。

“seccomp” 系统为用户空间提供了减小针对运行的进程对内核入口的可选的功能。这限制了内核 codepath 的宽度从而降低了特定 bug 的攻击。

构建一些可行的方式只允许信任的进程访问像 compat，用户空间，BPF 创建和 perf 这类资源。这将内核入口点的范围限制到正常可用于非特权用户空间的更规则的集合中。

### 限制访问内核模块：

内核不应该让非特权用户有能力加载内核模块，因为这会增加攻击平面。（通过自定义子系统的按需加载模块，比如在这里 MODULE\_ALIAS\_\* 是可接受的。）比如，通过一个非特权 socket API 加载一个文件系统的模块是不应该的： 只有 root 或者物理的本地用户才能够触发文件系统的模块加载。（甚至这在一些场景下也是值得商榷的。）

要对抗特权用户，系统可能需要完全关闭模块加载（比如宏内核编译或者 modules\_disabled sysctl），或者提供带签名的模块（比如 CONFIG\_MODULE\_SIG\_FORCE 或者 dm-crypt 的 LoadPin）来保证 root 无法通过模块加载器接口加载任意内核代码。

## 内存完整性：

有很多内核的数据结构在攻击中是可以被滥用于获得执行控制的，至今最广为人知的是存储在栈上的返回地址被修改的栈缓冲区溢出。其他这类攻击的例子存在，相关用于对抗此类攻击的保护机制也存在。

### 栈缓冲区溢出：

经典的栈缓冲区溢出是越界的写一个存储在栈上的变量，最终写一个可控制的值到栈帧的存储返回地址。常用的防御方案是在栈和返回地址之间放 stack canary ( CONFIG\_CC\_STACKPROTECTOR)，以在函数返回前验证。其他防御方案包括 shadow stacks。

Shawn: 需要注意的是 kernel stack canary 被触发后系统通常会直接 panic，对于生产环境里性能和安全风险的 trade-off 由各位自己把握。

### 栈深度写出：

这是一种少见的攻击方式，用一个 bug 触发内核通过深度函数调用或者大量栈分配消耗掉栈内存。这种攻击可能覆盖内核预分配栈空间的末端和敏感结构体。为了更好的防护，有两个重要的改变需要做： 把敏感的结构体 thread\_info 移到别处，和增加一个内存错误机制在栈底用于捕捉这些溢出。

Shawn: 常见的漏洞利用会根据 rsp 算出 thread\_info 的地址从而去修改 addr\_limit，而 PaX/Grsecurity 的 x86 实现早在 2011 年以前就已经把 thread\_info 从 kernel stack 相邻的位置移走了;-)

### 堆内存完整性：

用于跟踪堆的链表的数据结构可以在分配和释放时做 sanity-check 以保证他们没有被用于篡改其他内存区域。

### 计数器完整性：

内核很多地方使用了原子计数器用于跟踪对象引用或者执行类似生命周期管理的操作。当这些计数器被 wrap 时(unsigned int 的 32 位会在 2^32 - 1 后出现）会暴露 user-after-free 的漏洞。通过捕获原子 wrapping 可以解决掉这类 bug。

Shawn: 来自以色列的 PERCEPTION POINT 公布的针对 [CVE-2016-0728 的 PoC](http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/) 算是第一个针对此种类型 bug 的公开漏洞利用，PAX\_REFCOUNT 可以防御。

### 大小计算溢出检测：

类似计数器溢出，整数溢出（通常是大小计算）需要在运行时被检测到来解决掉这类通常会导致内核缓冲区越界写的 bug。

## 统计性防御：

有很多防御是可以被认为是确定性的（比如只读内存不能被写入），一些防护在一些必须搜集足够信息的场景只提供统计性防御。虽然不完美，但也提供了有意义的防御。

### Canaries, blinding：

应该注意像之前讨论的 stack canary 是技术性的统计性防御，因为他们依赖于（可泄漏）的秘密值。致盲像在被用户空间控制的内容，类似 JIT 的逐字逐句的值，也需要一个类似的秘密值。关键是秘密值必须分离（比如每个 stack 不同的 canary）和高熵（比如，RNG 真的工作吗？)。

### 内核地址空间布局随机化（KASLR）：

内核内存的地址几乎是成功攻击的重要工具，让地址变得不确定性会增加漏洞利用的难度。（注意，这反过来会让泄漏值更高从而可能发现所需的内存地址。）

#### 代码段和模块基地址：

通过在启动时( CONFIG\_RANDOMIZE\_BASE)对内核的物理和虚拟地址的基地址进行重定位，需要内核代码的攻击会受挫。另外，offsetting 模块的加载基地址意味着系统每次启动以相同的顺序加载相同的模块不会共享同样的基地址。

#### 栈基：

如果内核栈的基地址对于不同的进程甚至系统调用都不一样，攻击将变得很困难。

#### 动态内存基：

根据早期启动的初始化，太多的内核动态内存（比如 kmalloc, vmalloc, etc）都有相对确定性的内存布局。如果这些区域的基地址在不同的启动是不同的，攻击会受挫，从而需要特定区域的信息泄漏。

## 组织泄漏：

敏感结构体的地址是攻击的首要目标，很重要的是去防御对内核内存地址和内核内存内容（他们包含内核地址或者其他敏感信息比如 carnary 值）的泄漏。

Shawn: 对于敏感结构体最好的方式是 code diversification，但这对生产环境的取证工作会带来一些麻烦-\_-

### 唯一的标示符：

内核内存地址不能作为标示符暴露给用户空间。相反，应该使用一个原子计数器，一个 idr 或者类似唯一的标示符。

### 内存初始化：

内存拷贝到用户空间必须总是完全初始化的。如果没有显式的 memset()，这需要修改编译器确保这些结构体是清空的。

### 内存污染：

当释放内存是，最好去污染内容（在 syscall 返回是清空栈，在释放后清空堆），以防止依赖于旧内存内容重用的攻击。这会受挫很多未初始化变量的攻击，栈泄漏，堆信息泄漏和 UAF 攻击。

### 目的跟踪：

为了解决掉内核地址被写到用户空间的 bug，写的目的地址需要被跟踪。如果缓冲区是用户空间（比如 seq\_file 的 /proc 文件），就应该检查敏感值。

在栈可能出现问题之前：有一个值是保护栈的，我们发现这一随机值是放在了函数的局部变量和保存的指令指针（译注：此处指返回地址和EBP）之间。这个值被称作金丝雀(“canary”)值，（指的是矿工曾利用金丝雀来确认是否有气体泄漏，如果金丝雀因为气体泄漏而中毒死亡，可以给矿工预警），但有一些漏洞，远远超出了“金丝雀”的区间，因此，提出了堆栈位置随机化---并行堆栈（用来存储诸如返回地址的重要的值）

对于堆溢出，放置保护页可以捕获堆溢出漏洞。设置隐藏符号，避免了内核只针对用户空间的泄漏,内核地址空间布局随机化并不是一个完美的屏蔽，设置内存保护是更好的方法（硬件支持）

INTEL—0day：

通过与社区合作，进一步提高内核的性能。在这个网站上提供的数据允许社区成员密切跟踪每一个版本的内核的性能。最终我们希望这些测试数据导致性能的提升。我们正在运行一个大规模的基准测试，覆盖内核核心组件（虚拟内存管理，I / O子系统，进程调度程序，文件系统，网络，设备驱动程序等），具体来说：提供深入的内核测试服务，同时还监视各种内核树，也包括内核邮件列表，当上述监控对象有变化时，会第一时间对启动、性能、功能、功率等进行检测，每当启动检测时，开发者会自动收到相关信息的邮件。同时，该项目建立并测试使用超过100种不同的内核配置来对内核进行测试。开机测试是基于x86的裸机或在虚拟机上做的。对于已通过构建测试的内核，将运行超过80个功能测试套件和基准测试程序。对于每一个测试用例或基准测试，可以有几十到数百个设置和参数的组合。

0-day的关键特征：

1：提供小时的全天候响应时间，因此称为0-day

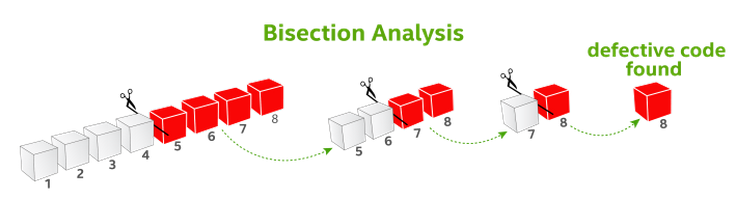
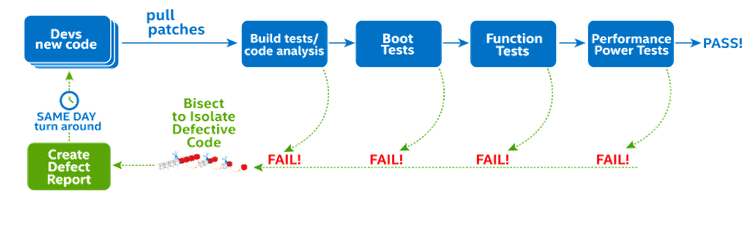
2：通过补丁对补丁进行测试

3：覆盖了开发者树的所有分支

4：使用静态源代码分析器执行内核编译和静态语义级测试

5：进行基于各种平台的启动测试、功能测试、和性能测试

6：当测试错误或是性能退化，可以自动知道是哪些补丁导致了上述问题



当出现错误或显示性能回退时，0-day自动指出造成上述问题出现的补丁。这其中最为关键的是使用了代码的二分法----识别引起特定行为的变化的方法。0-day把补丁集和重新测试的代码段一分为二，直到它指出确切的出现问题的补丁，然后发送缺陷报告给开发人员，整个过程的耗时往往在不到一天的时间

FIO是一种工具，将产生大量的线程或进程做一个特定类型的由用户指定的IO操作。该采取了一些全局参数，运行该工具测试磁盘的IO

FFSB是一个文件系统性能测量工具，这是一个多线程应用程序，它不同于其他的文件系统基准，用户可以提供一个配置文件来创建自定义的工作负载，而大多数其他的文件系统基准测试使用一组固定的工作负载

OLTP：系统在负载量大的情况下运行数据库（mysql）时对系统参数测试。

swap（Linux系统中的交换分区基准测试），在一个内存使用紧张的系统中测量系统的性能。

INTEL-UEFI

新型[UEFI](http://baike.baidu.com/view/196503.htm)，全称“统一的可扩展固件接口”(Unified Extensible Firmware Interface)， 是一种详细描述类型接口的标准。这种接口用于操作系统自动从预启动的操作环境，加载到一种操作系统上。可扩展固件接口（Extensible Firmware Interface，EFI）是 Intel 为 PC 固件的[体系结构](http://baike.baidu.com/view/1188494.htm)、接口和服务提出的建议标准。其主要目的是为了提供一组在 OS 加载之前（启动前）在所有平台上一致的、正确指定的启动服务，被看做是有近20多年历史的 BIOS 的继任者。

BIOS即Basic Input/Output System，翻成中文是“基本输入/输出系统”，是一种所谓的“[固件](http://baike.baidu.com/view/33738.htm)”，负责在开机时做硬件启动和检测等工作，并且担任操作系统控制硬件时的中介角色。

因为硬件发展迅速，传统式（Legacy）BIOS 成为进步的包袱，现在已发展出最新的UEFI（Unified Extensible Firmware Interface）可扩展固件接口，相比传统 BIOS 的来说，未来将是一个“没有特定 BIOS”的电脑时代。与legacy BIOS 相比，UEFI最大的几个区别在于：

1. 编码99%都是由C语言完成；

2. 一改之前的中断、硬件端口操作的方法，而采用了Driver/protocol的新方式；

3. 将不支持X86实模式，而直接采用Flat mode（也就是不能用DOS了，现在有些 EFI 或 UEFI 能用是因为做了兼容，但实际上这部分不属于UEFI的定义了）；

4. 输出也不再是单纯的二进制code，改为Removable Binary Drivers；

5. OS启动不再是调用Int19，而是直接利用protocol/device Path；

6. 对于第三方的开发，前者基本上做不到，除非参与[BIOS](http://baike.baidu.com/view/361.htm)的设计，但是还要受到ROM的大小限制，而后者就便利多了。

7.弥补BIOS对新硬件的支持不足的问题。

优点:

纠错特性

与BIOS显著不同的是，UEFI是用模块化、C语言风格的参数堆栈传递方式、[动态链接](http://baike.baidu.com/view/945740.htm)的形式构建系统，它比BIOS更易于实现，容错和纠错特性也更强，从而缩短了系统研发的时间。更加重要的是，它运行于32位或64位模式，突破了传统16位代码的寻址能力，达到处理器的最大寻址，此举克服了BIOS代码运行缓慢的弊端。

### 兼容性

与BIOS不同的是，UEFI体系的[驱动](http://baike.baidu.com/view/43111.htm)并不是由直接运行在CPU上的代码组成的，而是用EFI Byte Code（EFI[字节代码](http://baike.baidu.com/view/185293.htm)） 编写而成的。Java是以“Byte Code”形式存在的，正是这种没有一步到位的中间性机制，使Java可以在多种平台上运行。UEFI也借鉴了类似的做法。EFI Byte Code是一组用于UEFI驱动的虚拟机器指令，必须在UEFI驱动运行环境下被解释运行，由此保证了充分的向下兼容性。

一个带有UEFI驱动的扩展设备既可以安装在使用安卓的系统中，也可以安 装在支持UEFI的新PC系统中，它的UEFI驱动不必重新编写，这样就无须考虑系统升级后的兼容性问题。基于解释引擎的执行机制，还大大降低了UEFI 驱动编写的复杂门槛，所有的PC部件提供商都可以参与。

### 鼠标操作

UEFI内置图形驱动功能，可以提供一个高分辨率的彩色图形环境，用户进入后能用鼠标点击调整配置，一切就像操作Windows系统下的应用软件一样简单。

### 可扩展性

UEFI将使用[模块化设计](http://baike.baidu.com/view/189730.htm)，它在逻辑上分为硬件控制与OS（操作系统）软件管理两部分，硬件控制为所有UEFI版本所共有，而OS软件管理其实是一个可编程的开放接口。借助这个接口，主板厂商可以实现各种丰富的功能。比如我们熟悉的各种备份及诊断功能可通过UEFI加以实现，主板或[固件](http://baike.baidu.com/view/33738.htm)厂商可以将它们作为自身产品的一大卖点。UEFI也提供了强大的联网功能，其他用户可以对你的[主机](http://baike.baidu.com/view/23880.htm)进行可靠的远程故障诊断，而这一切并不需要进入操作系统。

### 图形界面

目前UEFI主要由这几部分构成：UEFI初始化模块、UEFI驱动执行环境、UEFI驱动程序、兼容性支持模块、UEFI高层应用和GUID[磁盘分区](http://baike.baidu.com/view/1514781.htm)组成。

UEFI初始化模块和驱动执行环境通常被集成在一个[只读存储器](http://baike.baidu.com/view/132973.htm)中，就好比如今的BIOS固化程序一样。UEFI初始化程序在系统开机的时候最先得到执行，它负责最初的CPU、北桥、南桥及[存储器](http://baike.baidu.com/view/87697.htm)的初始化工作，当这部分设备就绪后，紧接着它就载入UEFI驱动执行环境（Driver Execution Environment，简称DXE）。当DXE被载入时，系统就可以加载硬件设备的UEFI驱动程序了。DXE使用了枚举的方式加载各种总线及[设备驱动](http://baike.baidu.com/view/547398.htm)，UEFI[驱动程序](http://baike.baidu.com/view/1048.htm)可 以放置于系统的任何位置，只要保证它可以按顺序被正确枚举。借助这一点，我们可以把众多设备的驱动放置在磁盘的UEFI专用分区中，当系统正确加载这个磁 盘后，这些驱动就可以被读取并应用了。在这个特性的作用下，即使新设备再多，UEFI也可以轻松地一一支持，由此克服了传统BIOS捉襟见肘的情形。 UEFI能支持[网络设备](http://baike.baidu.com/view/1158081.htm)并轻松联网，原因就在于此。

值得注意的是，一种突破传统MBR（[主引导记录](http://baike.baidu.com/view/418401.htm)）[磁盘分区](http://baike.baidu.com/view/1514781.htm)结构限制的GUID（全局唯一标志符）磁盘分区系统将在UEFI规范中被引入。MBR结构磁盘只允许存在4个[主分区](http://baike.baidu.com/view/1304004.htm)，而这种新结构却不受限制，分区类型也改由GUID来表示。在众多的分区类型中，UEFI[系统分区](http://baike.baidu.com/view/115542.htm)用 来存放驱动和应用程序。很多朋友或许对这一点感到担心：当UEFI系统分区遭到破坏时怎么办？而容易受病毒侵扰更是UEFI被人诟病的一大致命缺陷。

事实上，系统引导所依赖的UEFI驱动通常不会存放在UEFI系统分区中，当该分区的驱动程序遭到破坏，我们可以使用简单方法加以恢复，根本不用担心。

X86处理器能够取得成功，与它良好的兼容性是分不开的。为了让不具备UEFI引导功能的操作系统提供类似于传统BIOS的[系统服务](http://baike.baidu.com/view/685551.htm)，UEFI还特意提供了一个兼容性支持模块，这就保证了UEFI在技术上的良好过渡。