



(12) 发明专利申请

(10) 申请公布号 CN 103473508 A

(43) 申请公布日 2013.12.25

(21) 申请号 201310425645.5

(22) 申请日 2013.09.17

(71) 申请人 肖楠

地址 710068 陕西省西安市电子三路 3 号西
京国际电气中心 A 座 1308

(72)发明人 肖楠

(74) 专利代理机构 西安智邦专利商标代理有限公司 61211

代理人 王少文

(51) Int. Cl.

G06F 21/57 (2013.01)

G06F 21/51 (2013.01)

G06F 21/56 (2013.01)

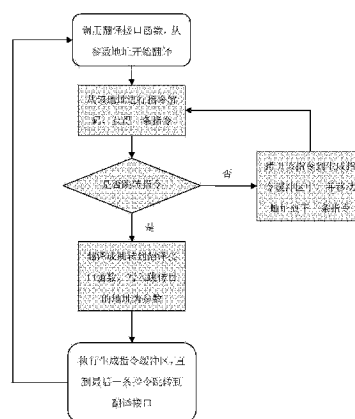
权利要求书2页 说明书11页 附图1页

(54) 发明名称

操作系统内核运行时安全验证方法

(57) 摘要

本发明一种操作系统内核运行时安全验证方法,包括确认目标操作系统的所有内核模块文件的合法性、验证内核中已经加载的模块的合法性、对随时加载的新内核模块进行合法性、验证重新设定处理器中所有从用户等级切换到特权等级的入口、对二进制翻译所要翻译的每条指令进行检查等步骤。这种方法可以用于计算机安全软件,能够从根本对运行中的计算机操作系统进行检测,探知其中存在的内核病毒;并能对现有的操作系统进行防范,发现试图入侵的内核病毒,并有条件地进行清除。是对rootkit的彻底解决之道。



1. 一种操作系统内核运行时安全验证方法,其特征在于:包括以下步骤:

步骤 1] 确认目标操作系统的所有内核模块文件的合法性:

静态确认目标操作系统的所有内核模块文件的合法性,并保存目标操作系统中所有的合法内核模块的完整性验证信息;

步骤 2] 验证内核中已经加载的模块的合法性:

通过枚举操作系统已经加载的内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证内核中已经加载的模块的合法性,同时获得每个模块的每个可执行节的地址范围;

步骤 3] 对随时加载的新内核模块进行合法性验证:

对操作系统随时加载的新内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证该新加载模块的合法性,同时获得该模块的每个可执行节的地址范围;

步骤 4] 重新设定处理器中所有从用户等级切换到特权等级的入口:

拦截处理器中所有从用户等级切换到特权等级的指令,并重新设定这些指令的入口,使之指向二进制翻译入口;

步骤 5] 对二进制翻译所要翻译的每条指令进行检查:

对二进制翻译所要翻译的每条指令进行检查,如果该条指令不在某个合法内核模块的某个可执行节中,则按照异常指令处理;如果该条指令位于某个合法内核模块的某个可执行节中,则执行步骤 6;

所述按照异常指令处理是指:

若系统处于最高安全要求下,则停止系统运行并收集相关信息自动上报,经人工检测无毒后再确认新的合法模块;

若系统处于较高安全要求下,则允许系统继续运行或回避该指令并设法让系统能够继续正常运行,同时搜集相关信息上报进行安全性分析;

若系统处于低安全性要求下,允许异常指令继续运行,同时搜集信息进行安全性分析;

步骤 6] 对步骤 5] 中经过检查的每条指令进行动态二进制翻译。

2. 根据权利要求 1 所述的操作系统内核运行时安全验证方法,其特征在于:所述步骤 6 中的动态二进制翻译是采用直拷贝动态二进制翻译,其具体步骤如下:

6. 1] 调用翻译接口函数,从某个本应执行指令的地址开始翻译;

6. 2] 从该地址进行指令解码,获得一条指令;

6. 3] 如果该指令为跳转指令,则翻译成跳到翻译接口函数,写入跳转目的地址为参数,并保留原跳转指令的跳转目的,这组指令跳转到翻译接口进行翻译,翻译结束后,跳转到生成指令缓冲区开始执行,直到最后一条指令执行完后回到原跳转指令的跳转目的;对于不带跳转的指令,不做任何变动,直接拷贝到生成指令缓冲区开始执行,同时移动地址到下一条指令;

6. 4] 返回步骤 6. 1 开始翻译下一条指令。

3. 根据权利要求 1 所述的操作系统内核运行时安全验证方法,其特征在于:所述步骤 6 中的动态二进制翻译是采用慢缓冲技术、快缓冲技术和 / 或直链技术的动态二进制翻译。

4. 根据权利要求 2 所述的操作系统内核运行时安全验证方法,其特征在于:所述步骤 6

中的直拷贝动态二进制翻译是采用慢缓冲技术、快缓冲技术和 / 或直链技术的直拷贝动态二进制翻译。

操作系统内核运行时安全验证方法

技术领域

[0001] 本发明涉及计算机安全领域,尤其涉及对计算机操作系统内核病毒的检测、防范和清除,并确保操作系统内核安全运行的相关技术。

背景技术

[0002] 现有的计算机信息安全包括很多方面。其中对病毒、木马等恶意程序的检测和防止为计算机安全的重要一环。其中,运行于计算机操作系统内核层(kernel level),使用了被称为 rootkit 技术的病毒是计算机病毒中技术含量最高,最难以检测、防止和清除的。

[0003] 内核病毒是指运行于计算机操作系统内核层的病毒。使病毒运行于计算机操作系统内核层的技术称之为 rootkit 技术。

[0004] 一般而言,现代处理器架构都有不同的特权层级,以便运行具有不同权限的软件。根据处理器架构的不同而具有不同的名称,但是其设计本意是相同的。比如广泛应用于个人计算机(PC)的英特尔的 IA32 处理器,其具有 r0、r1、r2、r3 四个不同的特权等级。操作系统的内核运行于 r0 等级,从而保证了操作系统内核对计算机有最高特权。而其他的应用软件以及操作系统的用户态部分的程序都运行于 r3 等级,对计算机硬件只拥有有限的权限,从而保证计算机的安全性。即使恶意代码入侵运行于 r3 的软件,也只能造成有限的破坏。

[0005] 同样,广泛应用于移动领域的 ARM 处理器架构则拥有“用户模式”、“系统模式”、“特权模式”等多种工作模式。同样以此来确保操作系统的内核的最高权限,并限制用户软件的权限,来确保计算机系统的安全。

[0006] 所谓的 rootkit 技术则是设法获取和操作系统内核一样的特权,侵入操作系统内核的技术。使用了 rootkit 技术的病毒则获取了对计算机操作的最高权限。从理论上来讲,在一个已经运行内核病毒的系统上来检测或者清除内核病毒,在现有的技术上是是不可能彻底完成的工作。因为内核病毒自身具有计算机操作系统中的最高权限。而试图检测内核病毒的安全软件也只具有与之相同的权限。内核病毒可以使用多种技术隐藏自己,干扰甚至停止安全软件的运行,比如:

[0007] 1. 接管系统调用。如果系统管理员试图使用工具去检测硬盘是否存在内病毒时,由于系统调用已经被病毒接管,因此工具通过系统调用返回看到的系统情况很可能是病毒伪造的。比如无法看到其实存在的被病毒感染的文件等。

[0008] 2. 杀死或者伪造安全软件进程。由于病毒运行于内核层,可以在内核中强行终止安全软件进程的运行,甚至可进一步生成假的安全软件进程来防止被发现。亦可从内核中清除安全软件的部分功能使之成为不具备安全性的进程。

[0009] rootkit 在各种架构的处理器、以及各种操作系统上都可能出现。比如现有的运行 Windows 各种版本的 PC,以及运行 Linux 的服务器,以及安装了 Android 系统的移动设备(如手机、平板等),都有使用 rootkit 技术的内核病毒出现的报导。

[0010] 在许多重大的网络安全事故、计算机系统攻击中,都有 rootkit 技术和内核病毒

的出现。

发明内容

[0011] 本发明的目的是提供一种操作系统内核运行时安全验证方法,这种方法可以用于计算机安全软件,能够从根本上对运行中的计算机操作系统进行检测,探知其中存在的内核病毒;并能对现有的操作系统进行防范,发现试图入侵的内核病毒,并有条件地进行清除。是对 rootkit 的彻底解决之道。

[0012] 本发明的技术解决方案是:

[0013] 一种操作系统内核运行时安全验证方法,其特殊之处在于:包括以下步骤:

[0014] 步骤 1] 确认目标操作系统的所有内核模块文件的合法性:

[0015] 静态确认目标操作系统的所有内核模块文件的合法性,并保存目标操作系统中所有的合法内核模块的完整性验证信息;

[0016] 步骤 2] 验证内核中已经加载的模块的合法性:

[0017] 通过枚举操作系统已经加载的内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证内核中已经加载的模块的合法性,同时获得每个模块的每个可执行节的地址范围;

[0018] 步骤 3] 对随时加载的新内核模块进行合法性验证:

[0019] 对操作系统随时加载的新内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证该新加载模块的合法性,同时获得该模块的每个可执行节的地址范围;

[0020] 步骤 4] 重新设定处理器中所有从用户等级切换到特权等级的入口:

[0021] 拦截处理器中所有从用户等级切换到特权等级的指令,并重新设定这些指令的入口,使之指向二进制翻译入口;

[0022] 步骤 5] 对二进制翻译所要翻译的每条指令进行检查:

[0023] 对二进制翻译所要翻译的每条指令进行检查,如果该条指令不在某个合法内核模块的某个可执行节中,则按照异常指令处理;如果该条指令位于某个合法内核模块的某个可执行节中,则执行步骤 6;

[0024] 所述按照异常指令处理是指:

[0025] 若系统处于最高安全要求下,则停止系统运行并收集相关信息自动上报,经人工检测无毒后再确认新的合法模块;

[0026] 若系统处于较高安全要求下,则允许系统继续运行或回避该指令并设法让系统能够继续正常运行,同时搜集相关信息上报进行安全性分析;

[0027] 若系统处于低安全性要求下,允许异常指令继续运行,同时搜集信息进行安全性分析;

[0028] 步骤 6] 对步骤 5] 中经过检查的每条指令进行动态二进制翻译。

[0029] 上述步骤 6 中的动态二进制翻译是采用直拷贝动态二进制翻译,其具体步骤如下:

[0030] 6.1] 调用翻译接口函数,从某个本应执行指令的地址开始翻译;

[0031] 6.2] 从该地址进行指令解码,获得一条指令;

[0032] 6.3] 如果该指令为跳转指令,则翻译成跳到翻译接口函数,写入跳转目的地址为

参数,并保留原跳转指令的跳转目的,这组指令跳转到翻译接口进行翻译,翻译结束后,跳转到生成指令缓冲区开始执行,直到最后一条指令执行完后回到原跳转指令的跳转目的;对于不带跳转的指令,不做任何变动,直接拷贝到生成指令缓冲区开始执行,同时移动地址到下一条指令;

[0033] 6.4] 返回步骤 6.1 开始翻译下一条指令。

[0034] 上述步骤 6 中的动态二进制翻译是采用慢缓冲技术、快缓冲技术和 / 或直链技术的动态二进制翻译。

[0035] 上述步骤 6 中的直拷贝动态二进制翻译是采用慢缓冲技术、快缓冲技术和 / 或直链技术的直拷贝动态二进制翻译。

[0036] 与现有技术相比,本发明具有以下优点:

[0037] 1、相对于传统的用病毒特征码进行扫描,来发现病毒的方案,本发明是一种针对一切内核病毒的通用型方案,并不针对任何病毒的特例进行防御或者清除,而是针对操作系统内核病毒这一大类进行彻底的防御和清除,是一种用于操作系统内核安全的通用技术。

[0038] 2、与传统的静态文件安全性保证技术(如通过计算完整性码比如 md5 值之类,或者是对可执行文件进行签名验证等)相比,本发明是一种动态验证的方案。本发明利用了现有的文件安全性保证技术来验证所有的操作系统的内核模块是合法的。但是在此之外,本发明独创的是在操作系统内核实际运行时,确保实际运行的每条指令与经过验证的文件中的代码的一致性。因而是一种更全面和彻底的安全性验证技术。

[0039] 3、与现有的市面上大量的安全软件采用的“主动防御”的技术相比,本发明并不是集中于操作系统内核中有限的若干个“关键点”进行检查和防御。实际上是在操作系统内核中的任何点,也就是每条指令运行之前,进行检查和防御。因此,本技术相对于已有的“主动防御”技术来说,是一种更全面和彻底的技术。

[0040] 4、本发明使用类似虚拟机的方式,但是简单高效得多的方案对在特权等级上运行任何指令进行检查,确保每条指令都是已知的内核模块中的合法指令。

附图说明

[0041] 图 1 是简化过的直拷贝二进制翻译流程图,图中灰色的部分代表翻译过程,白色的部分代表执行过程。

具体实施方式

[0042] 本发明独创运行时内核模块合法性检测方法,仅仅通过检测在内核中实际运行的指令与已经确认合法内核模块可执行文件之间的同一性来确定内核中是否有非法指令在运行,以确保内核的高度安全性,达到彻底检测以及完全阻止内核病毒入侵的目的。

[0043] 一种操作系统内核运行时安全验证方法,包括以下步骤:

[0044] 步骤 1] 确认目标操作系统的所有内核模块文件的合法性:

[0045] 静态确认目标操作系统的所有内核模块文件的合法性,并保存目标操作系统中所有的合法内核模块的完整性验证信息;

[0046] 步骤 2] 验证内核中已经加载的模块的合法性:

[0047] 通过枚举操作系统已经加载的内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证内核中已经加载的模块的合法性,同时获得每个模块的每个可执行节的地址范围;每个合法的可执行模块的每个可执行节的地址确定之后,内核中所有合法指令的地址都已经确定。凡是出现这些地址之外执行的特权指令,均可认为是病毒或者其他攻击。

[0048] 步骤 3] 对随时加载的新内核模块进行合法性验证:

[0049] 对操作系统随时加载的新内核模块,与步骤 1 中保存的完整性验证信息进行对比,从而验证该新加载模块的合法性,同时获得该模块的每个可执行节的地址范围;

[0050] 步骤 4] 重新设定处理器中所有从用户等级切换到特权等级的入口:

[0051] 拦截处理器中所有从用户等级切换到特权等级的指令,并重新设定这些指令的入口,使之指向二进制翻译入口;这样处理器一旦切换到特权等级,所有特权指令都将被翻译。

[0052] 步骤 5] 对二进制翻译入口的每条指令进行检查:

[0053] 对二进制翻译入口的每条指令进行检查,如果该条指令不在某个合法内核模块的某个可执行节中,则按照异常指令处理;如果该条指令位于某个合法内核模块的某个可执行节中,则进行步骤 6;

[0054] 所述按照异常指令处理是指:

[0055] 若系统处于最高安全要求下,则停止系统运行并收集资料自动上报,经人工检测无毒后再确认新的合法模块;

[0056] 若系统处于较高安全要求下,则允许系统继续运行或回避该指令并设法让系统能够继续正常运行,同时搜集相关信息上报进行安全性分析;

[0057] 若系统处于低安全性要求下,允许异常指令继续运行,同时搜集相关信息进行安全性分析;

[0058] 步骤 6] 步骤 5] 中经过检查的每条指令进行直拷贝动态二进制翻译:

[0059] 6.1] 调用翻译接口函数,从某个本应执行指令的地址开始翻译;

[0060] 6.2] 从该地址进行指令解码,获得一条指令;

[0061] 6.3] 如果该指令为跳转指令,则翻译成跳到翻译接口函数,写入跳转目的地址为参数,并保留原跳转指令的跳转目的,这组指令跳转到翻译接口进行翻译,翻译结束后,跳转到生成指令缓冲区开始执行,直到最后一条指令执行完后回到原跳转指令的跳转目的;对于不带跳转的指令,不做任何变动,直接拷贝到生成指令缓冲区开始执行,同时移动地址到下一条指令;

[0062] 6.4] 返回步骤 6.1 开始翻译下一条指令。

[0063] 本发明原理:

[0064] 发明人经过研究认为,内核病毒之所以理论上难以被检测、防止和清除,其根本原因是内核病毒与安全软件运行于同样的特权等级(注意,本说明中,无论各种处理器架构,都将内核代码运行的权限级别称为特权等级。与之对应的普通应用软件运行的权限级别称为用户等级)。在这种情况下,彻底检测、防范与清除内核病毒是理论上不能做到的。所以本发明引入另一层权限控制,为彻底检测、防范与清除内核病毒开辟了道路。

[0065] 注意内核病毒有且必须有二个特征,缺一不可:

[0066] 1. 内核病毒必须有指令在特权等级运行。

[0067] 2. 上述的指令必定不是操作系统内核以及其他合法的内核模块中的指令。而是被利用任何方法“额外”注入到内核中的。

[0068] 发明人经过研究认为,利用以上 2 个特征,就足以精确识别出内核病毒。

[0069] 运行时内核模块合法性检测方法

[0070] 本发明使用独特的“运行时内核模块合法性检测方法”来对内核病毒进行检测。这完全有别于现在市面上大部分安全软件采用的静态扫描病毒码、实时监控扫描病毒码、主动防御等方案。

[0071] “内核模块合法性检测方法”的基本思想是：

[0072] 1、操作系统内核由若干个可提取的可执行文件组成。如 Windows 上,由一组 PE 格式的文件组成。而在 Linux 上,由一组 ELF 格式的文件组成。这些文件的完整性和合法性是可以通过已有的方法进行验证的(比如 Windows 上的签名验证等)。本发明中将这些经过验证,确定未被修改的模块称为合法模块。

[0073] 2、合法模块可以被静态解析。一般而言,合法模块中的可执行节(executable sections)中含有所有的可执行指令。这些指令的内容是可以通过静态解析来得到并确认的。

[0074] 3、合法模块的加载可在运行时监控。所有的操作系统在加载合法模块的时候,都有合法的入口可以进行监控。比如 Windows 上,可以通过接口 PsSetLoadImageNotifyRoutine 注册一个回调函数来进行监控。而 Linux 或者 Android 可以通过可加载模块(LKM)的加载接口 ins_mod 来进行监控。在监测到系统中有内核模块加载时,可以通过检查模块映像(Image)来确认这个模块的确是未经修改的合法模块。在加载之后,可确定模块加载的内存地址范围。

[0075] 4、非法的指令(比如 rootkit 病毒)何时被加载运行则难以掌控。因为病毒不一定通过合法的接口加载(当然,病毒也完全有可能通过合法的接口加载)。此外,非法的模块的文件结构等也非已知。病毒的指令可能是动态生成的,不存在明确的文件结构,也可能和合法的模块结构并没有什么不同。但是无论如何,在一个有着严格安全要求的操作系统内核内,如果有实际要运行在特权等级的任何指令,无法通过上述 1、2、3 进行确认,则可以怀疑是内核病毒指令。此时,可以根据不同的安全要求设定不同的策略。

[0076] 市面上安全软件的现有技术的问题是无法做到精确监控内核中每条运行的指令并对其进行检查。因为一条运行在特权等级的指令在运行的时候,实际已经掌握了计算机的全部控制权。外部无从对它加以干涉。因此,在特权等级上运行任何指令之前,必须要加入一道新的机制对指令进行检查。本发明的独创一点,就是使用类似虚拟机的方式,但是简单高效得多的方案对这些指令进行检查。也就是所谓直拷贝动态二进制翻译技术。

[0077] 动态二进制翻译(Binary Translation)

[0078] 动态二进制翻译(Dynamic Binary Translation)并非本发明的内容。这是一种已经公开存在并且被广泛应用的技术。是一种在运行时直接翻译可执行二进制程序的技术,能够把一种处理器上的二进制程序翻译到另外一种处理器上执行。

[0079] 其原理为:一方面读入某种处理器架构的二进制指令流,另一方面对这些指令流进行解码,获取其功能含义,并对照同样的功能生成另一种适合于另一种处理器架构的指令流并执行。当执行到未翻译过的指令时,则继续进行翻译。执行与翻译交叉进行。

[0080] 这种技术广泛应用于以某种处理器硬件来运行为另一种处理器架构所开发的情况。比如苹果公司的 Mac 笔记本,在将核心处理器的架构由 PowerPC 换成 IA32 之后,为了能够兼容以前的为 PowerPC 架构处理器的软件,就使用了动态二进制翻译技术。

[0081] 有时,也需要使用同种处理器架构指令的翻译。比如,从 IA32 指令翻译到 IA32 指令。虽然是同种指令,但是,翻译的过程对指令进行了一些改变。曾用到这种技术的有著名的虚拟机软件 VMware。VMware 在一台 IA32 架构的 PC 上,使用一个处理器,虚拟多个同样架构的处理器,因而用到了二进制翻译技术(在 Intel 提供 VT 技术之后,VMware 使用 VT 技术提供的便利替代了动态二进制翻译技术)。

[0082] 发明人经过研究发现,可以专门针对操作系统的内核进行动态二进制翻译,从而做到:

[0083] 1、对操作系统内核已有的、运行中的指令进行检测,确保每条指令都是已知的内核模块中的合法指令。

[0084] 2、对操作系统内核中已有的、运行中的指令进行改变,对非法代码进行屏蔽和清除等。

[0085] 3、不改变操作系统内核原有的功能。对用户透明。

[0086] 这是因为进行二进制翻译之后,每条即将要运行的指令都必须先经过翻译才能运行。而翻译的过程就可以对指令的合法性进行检查。而翻译之后,如果原始指令被恶意攻击所修改,可以很容易地被发现。因此以上几点为本发明对内核病毒的检测、防范、与清除提供了技术基础。

[0087] 直拷贝(direct copy)翻译技术

[0088] 普通的动态二进制翻译牵涉到地址变换(从某种地址空间变换到另一种地址空间,用于用主机上某段地址空间去虚拟一台完整的计算机的情况)、硬件设备虚拟机化(用于对硬件操作指令的虚拟),状态的记录与恢复等。所牵涉之技术非常复杂,同样,对计算机性能的损耗也大。其原因是,动态二进制翻译一般用于异种平台之间,或者某种平台上,一对多的处理器的虚拟。因此不得不实现地址变换、硬件虚拟等复杂的功能。

[0089] 发明人经研究发现,在以对抗内核病毒为目的的本发明中,无需进行任何硬件虚拟和地址变换。这是因为:

[0090] 1、这并非异种处理器架构之间的翻译。而是同种处理器架构之间的翻译。既不需要改变指令的种类,也不需要变换指令所操作的数据的地址。

[0091] 2、不需要一对多的虚拟,只需要一对一地进行翻译。所以不需要特别额外地保存多个虚拟处理器的状态。可以直接在操作系统的原生状态下运行。只需要确保翻译前后处理器寄存器状态不变即可。

[0092] 相对于传统的动态二进制翻译,本发明所需要的使用的二进制翻译相对简单,只需要实现直拷贝,即国外文献中常称为“direct copy”的翻译方式即可。最简单的直拷贝翻译方法为(不带任何优化的情况下):

[0093] 1、翻译接口从某个本应执行的地址开始翻译。翻译之前先保存处理器运行的当前状态。

[0094] 2、对于大部分普通的指令(不带跳转),则直接拷贝到生成指令缓冲区,不做任何变动。这是最简单的翻译方式,也就是直接拷贝。(但有少部分相对地址寻址的指令,由于

翻译后,自身地址已经改变,所以相对寻址的寻址偏移应该给予修改。此种指令必须进行简单的翻译修改)。

[0095] 3、对于跳转指令,则翻译成一组指令:这组指令跳转到翻译接口,并保留原跳转指令的跳转目的为参数。然后结束翻译,恢复之前保存的处理器状态,跳转到生成指令缓冲区开始执行。由于非跳转指令在翻译前后完全是等效的,所以执行这个缓冲区的效果和执行原始指令的效果一致。

[0096] 4、拷贝的指令执行完后,执行到最后,会调用翻译接口,并从原跳转指令的跳转目的开始翻译。这就回到了最初的一步。

[0097] 上面四步循环执行,实现了翻译和执行交叉进行的循环过程。而且执行的效果和原始过程(不加翻译时)是一样的,此处的“一样”系指面向用户的效果等效。处理器在执行过程中有中间步骤的状态不同,比如所执行的指令所在地址和原始不同等等,并不影响对用户所展示的结果。

[0098] 当然,以上两步忽略了一些其他需要翻译的情况,比如当前指令地址相对地址寻址的情况。同时也省略了各种优化。优化在后面的内容中详述。

[0099] 在进行直拷贝二进制翻译的情况下,翻译执行的流程如图 1。

[0100] 注意,以上翻译过程由于其简单性,可以直接插入到操作系统内核入口运行。当然,必须要确保:

[0101] 1、翻译的过程前后处理器的处理器中的面向用户的寄存器(具体到 x86,系指通用寄存器、标志寄存器这些和运算结果相关的寄存器)状态是一样的。这点可以在翻译之前进行备份,翻译之后执行之前恢复状态来实现。

[0102] 2、翻译无需动用除了寄存器、自己独有的内存之外的其他的硬件资源。

[0103] 3、翻译不需要调用操作系统本身提供的任何代码。

[0104] 由于发明人采用了直拷贝二进翻译这种简单的翻译形式,导致以上 3 点都是变成了可实现的。由此,本发明所描述的技术可以作为操作系统的内核模块直接安装到操作系统中运行,而无需像虚拟机一样,运行在操作系统之下。利用直拷贝二进制翻译技术进行仅仅针对操作系统内核代码的翻译,这是本发明的独创之处之一。

[0105] 直拷贝动态二进制翻译(direct copy)属于动态二进制翻译中最简单的一种,是公知技术。一般用于开发动态二进制翻译项目的早期阶段进行简单的模拟开发,但在实际应用中极少出现。

[0106] 直拷贝(direct copy)中可利用的动态二进制翻译优化技术

[0107] 注意本说明中介绍的直拷贝翻译技术并未介绍其优化过程。因此,如果仅仅按上述说明的方法来进行实现,会导致运行效率极其低下。但是,所有动态二进制翻译的优化方法,都可以同样地运用在本发明上,而且不影响本发明所提供的安全功能。这些优化技术有早已被广泛、公开地使用,比如:慢缓冲(slow cache)、快缓冲(fast cache)、直链(chaining)等等。

[0108] 注意下面对这些优化技术的介绍以直拷贝动态二进制翻译的情况为例。在复杂的动态二进制翻译(比如异种平台指令之间的翻译)中的具体情况可能有所不同,但因为和本发明没有关系所以可忽略。

[0109] 为了介绍优化的具体方法,这里必须先明确少量的概念。在动态二进制翻译中,我

们将被翻译之前的指令,称为“原始指令”。而翻译之后的指令,称为“翻译后指令”。进行具有翻译功能的程序,称为“翻译程序”。而调用翻译程序的接口,称为“翻译接口”。一段在内存中连续出现的以跳转指令为结束的指令(中间不能有跳转指令),称为一个“基本块”。

[0110] 在正常的翻译情况下,原始指令的基本块大概是如下的样子:

[0111] 非跳转指令 1

[0112] 非跳转指令 2

[0113] ...

[0114] 跳转到目标 A

[0115] 在操作系统上加载了翻译程序之后,实际上第一条指令“非跳转指令 1”执行之前就会被拦截进行翻译,翻译之后变成如下的样子:

[0116] 非跳转指令 1

[0117] 非跳转指令 2

[0118] ...

[0119] 保存状态

[0120] 调用“翻译接口”(翻译目标 A 开始的基本块)

[0121] 可以想象,一个原始的程序由无数个基本块组成。翻译之后,每个基本块执行完毕,都会跳转到“翻译接口”开始对原始跳转目标进行翻译。由于翻译过程相对执行一个基本块而言极为耗时,所以整个过程效率极为低下。

[0122] 所谓慢缓冲,就是指将已经翻译好的翻译后指令基本块,保存在一张可高效查找的表中。当有新的目标要进行翻译,先在表中检查是否曾经翻译过。如果翻译过的话就直接执行了,没有必要再次翻译。上面的翻译结果加入慢缓冲之后,变成如下的样子:

[0123] 非跳转指令 1

[0124] 非跳转指令 2

[0125] ...

[0126] 保存状态

[0127] 调用慢缓冲查找(目标 A)

[0128] 如果找到结果:

[0129] 恢复状态,跳转 A 对应的翻译后代码执行

[0130] 如果没有找到:

[0131] 跳转到“翻译接口”(翻译目标 A 开始的基本块)

[0132] 上面的方式看起来似乎更复杂,但是由于对于每个基本块,翻译其实只执行一次。大部分情况都变成了慢缓冲查找就找到结果。所以变得更快了。

[0133] 慢缓冲的是一个可以不断扩充的可以用于查找的表(比如哈希表)。由于可以不断扩充,所以容量巨大。但是也限制了查找的速度。当表比较小的时候,查找速度快。当翻译过的基本块越来越多,速度也就降低了。这时候快缓冲就有必要了。

[0134] 所谓的快缓冲是指将最近使用过的一组从原始指令地址到翻译后指令地址的对应关系,保存在一张有固定大小的表中。由于表的大小是固定的,所以查找速度也是固定的,相对于可扩充的慢缓冲而言,速度要快很多。

[0135] 但其缺点是大小固定。所以只能容纳最近最新的缓冲数据。老的数据在缓冲区中

自动被覆盖。但是由于数据的使用往往具有集中性,所以合理设置快缓冲的大小,就可以在高速的情况下获得极高的命中率。

[0136] 根据发明人的具体测试表明,一般快缓冲的命中率能达到 90% 以上。对效率提高效果显著。

[0137] 一般在快缓冲命中失败之后进行慢缓冲。如果慢缓冲命中,则将该项缓冲加入快缓冲。

[0138] 加入快缓冲之后,翻译的结果变成如下的样子:

[0139] 非跳转指令 1

[0140] 非跳转指令 2

[0141] ...

[0142] 保存状态

[0143] 调用快缓冲查找(目标 A)

[0144] 如果找到结果:

[0145] 恢复状态,跳转 A 对应的翻译后代码执行

[0146] 如果没有找到:

[0147] 调用慢缓冲查找(目标 A)

[0148] 如果找到结果:

[0149] 将这个慢缓冲项加入快缓冲

[0150] 恢复状态,跳转 A 对应的翻译后代码执行。

[0151] 如果没有找到:

[0152] 跳转到“翻译接口”(翻译目标 A 开始的基本块)

[0153] 在使用了快缓冲之后,动态翻译系统的效率会得到显著的提高。但是如果再加上“直链”技术的话,则可以使动态翻译系统运行的效率提高到接近原始未翻译系统的效率。回头看一下原始指令的形状:

[0154] 非跳转指令 1

[0155] 非跳转指令 2

[0156] ...

[0157] 跳转到目标 A

[0158] 如果明确目标 A 已经被翻译,翻译后指令的地址为 B,那么上述基本块其实可以翻译成下面的样子:

[0159] 非跳转指令 1

[0160] 非跳转指令 2

[0161] ...

[0162] 跳转到目标 B

[0163] 所谓直链技术就是在翻译后指令的尾部直接加入一条跳转到另一个翻译后指令的基本块。中间即不进行任何指令的查找,也不进行翻译。这样一来,运行的速度就基本与翻译之前所差无几了。

[0164] 大约 90% 的跳转都可以用直链完成。但是对于“不定跳转”(本文档中不定跳转是指跳转目标不固定,需要结合处理器状态才能知道跳转目标的跳转指令)无法直接使用,但

亦有技术手段可以折衷利用。

[0165] 经过发明人的实践研究,发现在 Linux 上,使用本发明进行过实时安全验证的情况下,操作系统内核的运行效率依然可达直接运行时的 70%。而整个操作系统的整体效率可达原生状态的 90%。因此对用户几乎没有可感知的影响。请注意,上述数据根据测试所用的实际硬件平台的不同而有所不同。

[0166] 本发明的技术可以适用于现在存在的各种操作系统,包括但不限于:各种版本的 Linux、Android,以及 Windows。同时也可以适用于各种不同的处理器架构,包括但不限于:IA32, ARM 等。但是对于不同的处理器架构,需要不同的代码实现;在不同的操作系统上,也有不同的实施方法。但是,在各种系统以及各种处理器平台上,本发明都实现为一个软件:既可能作为一个独立的软件(但需要在特权等级下运行),安装在已有的操作系统上;亦可能作为操作系统的一部分,与操作系统一起提供给用户;亦可能以固化在某个硬件比如 ROM 中的形式,作为机器的一部分提供给用户。下面的内容都已作为独立的软件进行安装为例作为讲解。但是,应该注意的是,即便是作为操作系统的一部分直接提供,或者是固化在硬件存储器中以硬件的方式提供给客户,也是完全可行的。

[0167] 本发明作为独立的软件安装进行实施的时候,需要如下的步骤:

[0168] 步骤一:静态确认目标系统的所有内核模块文件的合法性。注意这一点不需要在目标操作系统运行时进行,只需要读取硬盘上的文件进行检查就行了。有已知的多种途径可以进行。比如检查操作系统的内核有厂商的合法签名(比如微软对 Windows 内核的签名),各种驱动程序也有合法的签名或者是可从官方网站上下载合法的版本。Linux 可用代码编译出无毒的内核,并对其进行完整性计算,保留一个完整性验证码(比如 md5 值)。此时,需要将目标系统中所有的合法模块的完整性验证码保存起来,以便模块加载时验证。

[0169] 步骤二:验证内核中已经加载的模块的合法性。所有操作系统都提供了枚举已经加载的内核模块的公开方法。通过枚举这些模块,和步骤一中保留的完整性信息进行对比,可以确认这些已经加载到内存中的模块是否是完整的,未被破坏的。同时亦可获得每个模块的每个可执行节(executable sections)的地址范围。

[0170] 步骤三:安装可加载模块加载监控。这一点在不同的操作系统上有所不同,但是各自都有已知的方法。而且已经广泛应用于现在市面上的各类安全软件。对所有加载的内核模块进行验证。验证的方法和步骤二相同。所不同的是,步骤二是对本软件起作用之前系统中已经加载的内核模块进行验证。而本步骤是对本软件起作用之后,操作系统随时加载的新内核模块进行验证。

[0171] 步骤四:拦截所有的可能导致处理器的运行状态从用户等级到特权等级的切换点。这一点在 Windows 上和 Linux 系统,以及不同的处理器架构(IA32 和 ARM)都有所不同。但是提供用户等级和特权等级两种运行模式的处理器架构必定提供了从用户等级切换到特权等级的切换入口,且这种入口只能在特权等级下进行修改。由于本软件自身运行在特权等级下,因此总是可以修改这些入口。具体的操作在不同的架构下有所不同。比如 IA32 处理器,所有的中断都可能导致从用户等级切换到特权等级,此外,指令 sysenter(或 64 位的 syscall)也会切换到特权等级,同时指令指针指向一个保存在 MSR 中的值。本发明的实施方法,就是重新设定这些入口,使之指向二进制翻译入口,对原本入口的代码进行直拷贝动态二进制翻译。

[0172] 步骤五：直拷贝动态二进制翻译时，对待翻译的每条指令进行检查。确保这条指令的确位于某个合法模块的某个可执行节中。如果这一点是否定的，根据不同的安全策略采取不同的措施，比如但不限于：

[0173] 在最高安全要求下，出现这种情况就停止系统运行，机器送有关部门检测。确认无毒后再放行。

[0174] 在较高安全要求下，可以允许系统继续运行，但是搜集足够的信息发送到有关技术部门进行分析，以确认其安全性。

[0175] 在有时，对于嫌疑指令可以设法回避，并设法让系统能够继续正常运行。同时收集信息警告用户有潜在的安全风险，发送到有关技术部门进行分析。

[0176] 在低安全性要求的场合下，可以允许非法指令继续运行，但是收集相关的信息发送到有关技术部门进行分析。

[0177] 注意，在合法的操作系统中，的确有极少数动态生成的指令是不能通过步骤五的验证的。但是由于这是已知的而且极少数的情况，完全可以当做特例排除掉。

[0178] 本技术依赖于对用户权限等级到特权等级切换的硬件入口的掌握。理论上，rootkit 病毒也可以更改这些入口来达到获取控制权，破坏本发明的技术的目的。但是，由于只有运行特权等级的指令能够更改这些入口并破坏本技术，而特权等级上运行的指令都会被本技术进行二进制翻译分析，而发现对方的意图，因此理论上可以保护自己不受到攻击。

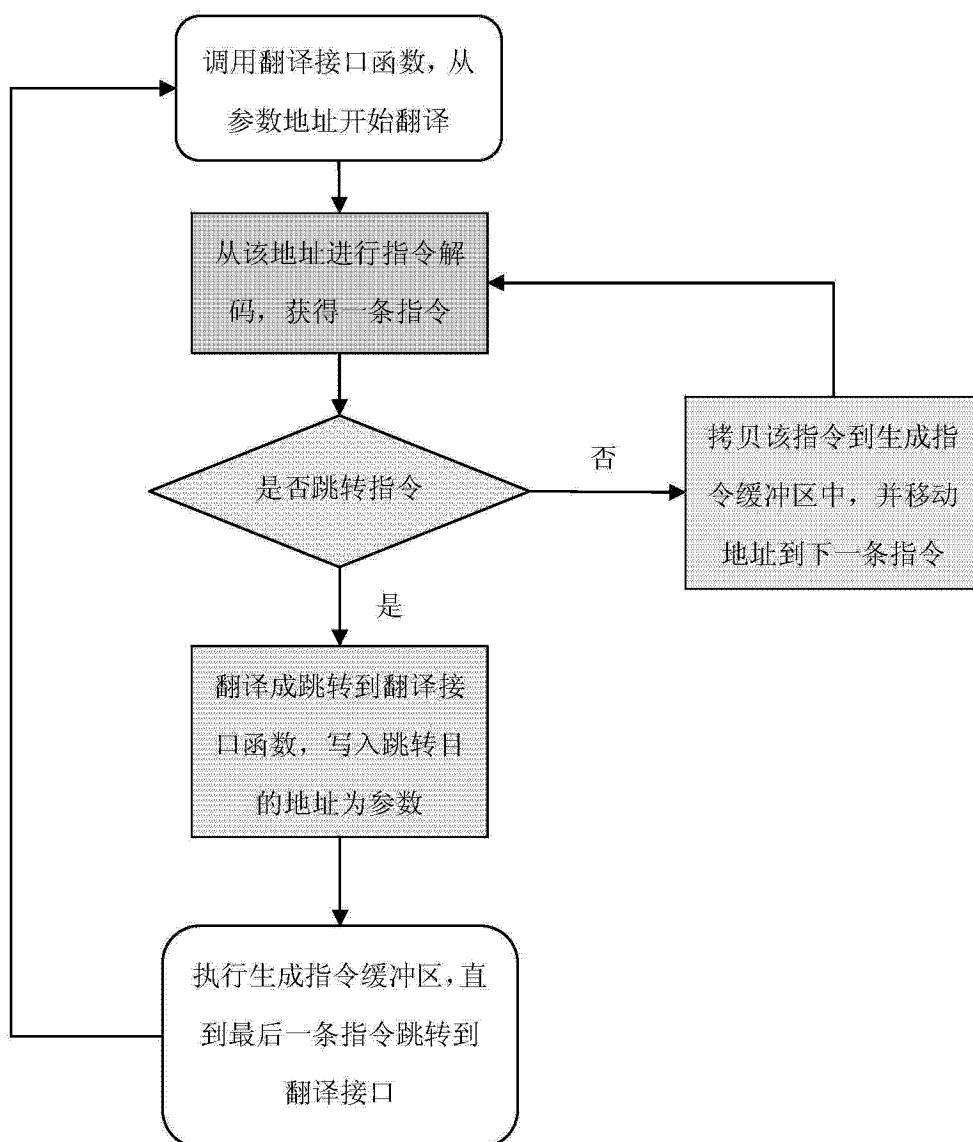


图 1