第33卷 第10期 计 算 机 工 程 2007年5月
Vol.33 No.10 Computer Engineering May 2007
·安全技术· 文章编号：1000—3428(2007)10—0144—03 文献标识码：A 中图分类号：TP301.6

基于系统调用的入侵检测系统研究

戴小鹏[1]，喻 飞[1,2]，张林峰[1]，沈 岳[1]
(1. 湖南农业大学计算机与通信学院，长沙 410128；2. 浙江大学人工智能研究所，杭州 310027)

摘 要：入侵检测是网络安全研究的热点技术之一，是新一代安全保障方案。该文实现了一种基于系统调用的异常入侵检测方法，使用系统调用作为输入，构建程序中函数的有限状态自动机，利用该自动机检测进程流程是否发生异常来确定是否发生了入侵。实验结果表明，该技术不仅能有效地检测出入侵行为，而且可以发现程序漏洞的位置，便于修改代码。

关键词：入侵检测；异常检测；系统调用；有限状态自动机

Research on Intrusion Detection System Based on System Call

DAI Xiaopeng[1], YU Fei[1,2], ZHANG Lingfen[1], SHEN Yue[1]
(1. School of Computer & Information Engineering, Hunan Agricultural University, Changsha 410128;
2. Institute of Artificial Intelligence, Zhejiang University, Hangzhou 310027)

【Abstract】 Intrusion detection is an efficient way to protect information system. This paper explains a new anomalous intrusion detection method. It uses system calls as input, and creates a FSA for the functions in the program. Then the FSA is used to detect the attack. It finds the place where the vulnerability exists in the program. This can help alter the source program. Experiment proves the method is effective in many intrusion events.

【Key words】 Intrusion detection; Anomalous detection; System call; Finite-state automation machine

1 概述

由于进程所要执行的关键操作都必须通过系统调用从用户态转换到内核态，因此通过查看系统调用序列可以基本了解进程的活动。因为系统调用及时准确，入侵者很难修改，所以目前该领域研究非常活跃。

通过监测系统调用来限制进程访问关键的资源[1]，这种方法具有异常和误用二者的优点，但却更像误用检测，它需要专家定义一些规范来检测入侵。而指定一个精确的规则非常复杂，即使是专家也会漏掉某些情况。而且新的攻击方式可能完全出乎专家预料之外，这种方法的漏报率比较高。

Forrest[2]将所有系统调用(只考虑名称)作为一个序列，又设置一个长度为n的窗口。算法首先经过一个训练阶段，此时多次在正常情况下运行程序，在系统调用序列中从第一个系统调用开始，依次截取长度为n的短序列，并将这个序列存入数据库。经过多次训练，形成一个正常短序列库。在之后的程序运行过程中，每当执行一个系统调用，就考察以该系统调用为最后一项的短序列是否在正常短序列库中出现过。如果没出现就认为可能出现了异常。这种算法是非常粗糙的，它不但无法保证正常的短序列都在正常短序列库中，而且无法保证入侵的短序列库不在正常短序列库中，但它首次提出了一个使用系统调用进行异常检测的思路。

Sekar提出使用有限状态自动机(Finite-state Automation Machine, FSA)描述正常系统调用的序列[3]。系统调用时的程序计数器(Program Counter, PC)作为状态，而把系统调用名称作为转移条件。通过训练阶段形成针对整个程序的FSA。在实际运行过程中，每当一个系统调用发生，首先检查当前程序计数器是否符合状态机中的程序计数器，然后会查看是否有一个符合该调用的转移路径。如果上述二者若有一个不符合，则认为可能有入侵产生。该算法的最大优点是能够用FSA描述进程的系统调用流程图，包括循环和分支，提高了准确性。但也有很多缺点，首先是仅用程序计数器很难确定一个真实的状态，多个状态可能对应着同一个程序计数器，可能会有多个函数调用同一个函数；然后这个函数再进行系统调用；结果当发生系统调用时，其程序计数器值是相同的，而这却对应着多个真实状态。算法中对DLL的处理非常粗糙，算法无法捕获递归调用。

Feng 在文献[4]中沿用了文献[2]的短序列，并且窗口大小设为 2。每一项不是使用系统调用名称，将使用函数堆栈＋程序计数器值。使用这种方式，能够更加精确地描述系统当时的状态。使用短序列无法把握进程的系统调用流程。

2 FSA 算法的改进

本文使用 FSA 来描述系统调用序列，不同于文献[4]中针对整个进程构造一个状态机，对每个跟系统调用相关联(直接或间接调用系统调用)的函数构造一个状态机。状态机中的转移条件是程序计数器或者函数堆栈中的返回地址，一般状态(除了开始、结束状态)并没有实际意义。

通过这种方法，解决了文献[4]中程序计数器无法准确描述状态的缺点；由于针对函数构造状态机，因此捕获递归调用将变得非常容易；使用了函数堆栈，DLL 将变得非常容易解决；该算法实现了之前算法从未实现的功能，可以根据异常发生的函数确定源程序漏洞出现的地点，这样有利于修改源代码、消除漏洞。

基金项目：湖南省自然科学基金资助项目(03JJY3103)；湖南农业大学人才引进基金资助项目(03YJ08)
作者简介：戴小鹏(1964－)，男，博士生、副教授，主研方向：网络安全，农业灾害治理；喻 飞，博士生、讲师；张林峰、沈 岳，学士、副教授
收稿日期：2006-06-02 **E-mail：**dxp_18@163.com

2.1 两次系统调用间的函数调用情况

函数堆栈是一个先进后出的、存放函数调用时相关信息的内存数据结构。每次函数调用时，函数都会将参数、局部变量、返回值和紧接在下面的帧的地址作为一个帧存入堆栈，如图 1 左部所示。当函数返回时，将此帧弹出。帧中很重要的一项是返回地址，表示当前函数返回后应该执行的下一条指令的地址。因为返回地址总是属于该函数的调用者的，所以可以通过查看该地址属于哪个函数的地址范围确定系统调用是通过哪些函数一层层调用的。

为了便于说明，本文假设系统调用A发生时其堆栈从底到顶的帧依次是$a_0,a_1,\ldots,a_m$，系统调用B发生时其堆栈从底到顶的帧依次是$b_0,b_1,\ldots,b_m$。下面涉及的函数调用都是和系统调用相关的，即直接或间接调用系统调用的。跟系统调用无关的函数调用，本文将不作考虑。

两次系统调用之间函数调用的情况如下(如图 1 右部)：第 1 个系统调用 A 发生后，帧 am 到 a(k+1)对应的函数返回，帧弹出；当返回到 ak 对应的函数时，ak 执行一条语句调用帧 b(k+1)对应的函数；然后继续一直调用到 bn 对应的函数，由 bn 直接发起系统调用。帧 ak 和 bk 对应着同一个函数，只不过是一次是调用 ak 对应的函数，另一次是 bk 对应的函数。

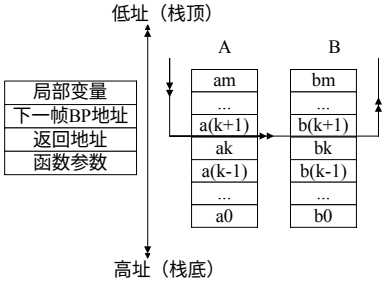

图 1 堆栈帧结构及堆栈转换

为了确定每个函数的状态机，必须确定新的系统调用 B 是从哪个函数开始重新发起的，或者说堆栈中未被弹出的顶层帧对应的函数。这对图 1 而言，就是求 k 等于多少。

分析图 1 可以发现，因为 a0~a(k-1)对应的函数没有执行过任何其中的指令，所以它们对应的帧即 a0~a(k-1)不会发生变化，即 $a_i = b_i$，$0 \le i < k$。ak 执行了其中的语句而重新调用了函数(别的或同一个)，这是第 1 个可能发生变化的帧，变化量可能是函数参数和局部变量。第 k+1 帧(从 0 计数)则是第 1 个返回地址可能发生变化的帧，因为 ak 对应的函数重新调用了一次函数。

如果 ak 对应函数调用了不同的位置的函数，则 a(k+1)和 b(k+1)的返回地址必然发生变化，如果发现某一帧的返回地址发生了变化，那么其下面帧是重新发起系统调用的帧。如果没有变化，说明 ak 对应函数调用了同一个位置的函数，只能是 ak 对应函数中有一个调用一个第 k+1 帧对应函数的循环。循环发生时，该调用者的局部变量或函数参数一般都会发生变化。本文对 Snort 源代码做过统计，总共 34 000 行左右的核心代码，共出现循环 250 次左右，其中两种形式的循环为主。一种是使用局部变量 i 递增或递减进行循环判断，大约共有 120 次；另一种使用局部链表指针的形式，根据指针是否为 NULL 来进行循环的条件判断，共出现大约 90 次。这两种方式都是局部变量或函数参数发生变化。另外还有 30 多次也会导致局部变量或函数参数变化。只有 5 次是由于循环使用指针指向的结构中的数据作为判断条件，使得局部变量或函数参数不发生变化，只占循环总数的 0.25%，完全可以忽略。

求解 k 值的方法是：当发现一个帧的局部变量或函数参数发生变化时，就认为该帧对应的函数重新进行了系统调用(直接或间接)；如果一个帧的返回地址发生变化，就认为该帧的下面帧对应的函数重新进行了系统调用。

2.2 函数状态机

每个跟系统调用相关的函数都用一个状态机来描述其行为。每个状态机都有一个起始状态 S 和终止状态 E。状态机还必须至少有一个中间状态，中间状态并没有具体含义。转移条件可以是“指令地址”或“函数结束”。“指令地址”具体是下一帧的返回地址或程序计数器值。而“函数结束”只能由中间状态指向 E 状态。

2.3 训练阶段

该阶段的任务是对每个跟系统调用相关联的函数生成一个 FSA。在每个系统调用发生时，我们都会在检测程序中存储当前的函数堆栈，保存到下个系统调用发生的时候。这样，当一个系统调用发生时，都能得到当前和最近两个函数堆栈。

本文首先确定哪个函数重新进行了系统调用，即求 k 值，然后从 am 对应的函数回溯到 ak 对应的函数(同时也是 bk 对应的函数)，再向下一直到 bm 对应的函数。这个过程中，这些函数的状态机都可能进行修改。根据函数帧在堆栈中位置不同，分为 4 类：

(1)am~a(k+1)对应的函数。函数状态的当前状态发出一条指向 E 状态的状态转移，转移条件是“函数结束”。S 状态称为当前状态。

(2)ak 对应的函数。函数状态机中生成一个新的状态，让当前状态发出一条指向该状态的状态转移，转移条件是 b(k+1)帧中的返回地址。新生成状态称为当前状态。

(3)b(k+1)~b(n-1)对应的函数。函数状态机中生成一个新的状态，让当前状态(即 S 状态)发出一条指向新状态的状态转移，转移条件是该帧的下一帧中的返回地址。新状态称为当前状态。

(4)bn 对应的函数。函数状态机中新生成一个状态，让当前状态(即 S 状态)发出一条指向新状态的状态转移，转移条件是当前 PC 值。新状态称为当前状态。

在这些过程中，如果发现到新状态的转移已经在状态机中出现，则不生成新状态和状态转移，只是改变当前状态。

2.4 检测阶段

被监测进程在实际运行过程中，根据训练阶段生成的状态机进行检测。检测阶段跟训练阶段非常类似，区别在于：在训练中，如果发现之前未曾出现的新的转移条件，则在状态机中添加该转移和新状态；而检测阶段如果出现这种情况，则认为是一个异常，当作一种入侵。

根据异常发生的状态机的所属函数，可以判断可能的函数漏洞是在哪个函数中出现的。通过这种方式，可以很容易地查看该函数源代码或汇编代码来判断是否构成入侵。如果是，本文就可以修改代码；如果不是，则说明之前的训练阶段做得不够充分，需要将新的转移加入到状态机中。

3 改进后算法的实现

考虑到 Linux 具有灵活性、稳定性和开放性等特点，本文在 Linux 下实现改进后的新算法。

3.1 捕获系统调用

有多种方式捕获系统调用，最常见的两种是修改系统内核和使用 ptrace 函数。在具体实现中，本文采用了 ptrace 函数，因为这样可以充分利用 Linux 提供的机制而不需要自己修改内核。

在监视进程中调用 ptrace 函数的 PTRACE_ATTACH 参数来监测被监视进程。这样每次被监视进程的系统调用发生时，被监视进程会给监视进程发送一个信号，然后转入睡眠。监视进程获得 CPU 控制权后可以使用 ptrace 的 PTRACE_PEEKDATA 和 PTRACE_GETREGS 参数来获得被监视进程的函数堆栈和程序计数器值。处理完毕后，它通过 PTRACE_CONT 参数给被检测进程发送信号进行激活。

3.2 查找每个函数起止地址

Linux 中的可执行文件和动态共享库文件的格式都是 ELF。ELF 格式中有一个.symtab 段可以确定该文件中定义的函数的开始位置和长度。当进程运行时，可执行文件中定义的函数的开始位置和可执行文件中说明的开始位置相同。对于动态共享库文件定义的函数，稍微复杂一点，因为动态共享库在不同次运行间可能导入到不同的位置。此时本文需要查看/proc/被监视进程 ID/maps。这个文件中可以看到导入的共享库的起止位置。而共享库文件中的函数起始位置是以 0 为基址的。共享库的起始位置和共享库文件中函数的起始位置相加就可以得到进程内存中该函数的起始地址。

4 实验结果分析

本文在 Linux 8.0 上对 ftp 程序进行了测试，整个过程使用了 3 个程序：跟踪程序，分割及学习程序，优化程序。系统调用跟踪进程与超级服务器进程 in.inet 同步运行，它随后会派生出一个子进程来跟踪 ftp 的运行，并将系统调用序列送给分割程序；分割程序产生 ftp 程序正常行为的原始文法描述，并将它写到一个文件，由优化程序生成最终的文法。

本文首先运行 ftp 的常用指令进行训练，生成状态机，然后进入检测阶段。当进行正常的操作时，一切都按照之前的状态机进行。当利用该缓冲区漏洞进行人工模拟攻击时，发现在 fb_realpath()函数的状态机中发生异常，而溢出正是在这个函数中产生的。

5 结束语

本文实现了一种使用有限自动状态机描述程序中与系统调用相关的函数流程的方法，通过检测进程流程是否发生异常来确定是否发生了入侵。实验结果表明这对当前大多数的入侵都有良好的检测效果。今后将对更多的入侵方式进行训练，并提高系统的执行效率。

参考文献

1 Calvin K, George F, Karl L. Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring[C]//Proceedings of the 10th Annual Computer Security Applications Conference. 1994.

2 Forrest S, Hofmeyr S A, Somayaji A, et al. A Sense of Self for Unix Processes[C]//Proc. of IEEE Symposium on Computer Security and Privacy. 1996.

3 Sekar R, Bendre M, Dhurjati D, et al. A Fast Automation-based Method for Detecting Anomalous Program[C]//Proc. of IEEE Symposium on Security and Privacy. 2001.

4 Feng H H, Kolesnikov O M, Fogla P, et al. Anomaly Detection Using Call Stack Information[C]//Proc. of IEEE Symposium on Security and Privacy. 2003.

(上接第 135 页)

"\xCF\xDA\xCD\x66\xCC\x61\xCF\xF3\xF7\xF1\xFF\xF6\xE9\xFC\xCD\xC9"

"\x66\xCC\x65\x10\xEC\x4D\x10\xEC\x41\x10\xEC\x45\x10\xEC\x79\x5F"

"\xDC\x4D\xFA\x5F\xDC\x4C\xA3\x5F\xDC\x4F\xC5\x5F\xDC\x4E\xEE\x5F"

"\xDC\x45\xEE\x14\xC4\x45\xCA\x14\xC4\x4D\xCA\x66\x49\x12\x7C\xC4"

该测试 Shellcode 代码能在 Windows 2000 sp1 系统和 Windows XP sp2 系统下测试通过。

2.3 实验结果

拥有了测试 Shellcode 和“覆盖-测试”算法，本文对实际蠕虫的特征提取进行了测试。测试环境是 CPU 1.6GHz，虚拟机 Vmare，系统是 Windows2000 sp1，测试结果如下：

Red Code I:

GET default.ida?NNN%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a HTTP/1.0 0Dh,0Ah

Content-type: text/xml',0Ah

HOST:www.worm.com 0Ah

Accept: */*,0Ah

Content-length: 3569 0Dh,0Ah 0Dh,0Ah

从测试结果中可以看出，自动提取生成的蠕虫特征值和安全厂商手工分析生成的特征值具有很大的可比性。这是首次特征值自动提取能和手工提取相对比的算法。也证明了该算法的有效性。

3 结论

本文提出了一种基于蠕虫模型先验知识，采用语义分析自动提取蠕虫特征的新方法。提取的效果和人工生成的特征串基本一致，到达了很好的效果。下一步的工作将集中在提高语义识别速度方面。

参考文献

1 Kreibich C, Crowcroft J. Honeycomb-creating Intrusion Detection Signatures Using Honeypots[C]//Proceedings of the 2nd Workshop on Hot Topics in Networks. 2003.

2 Kim H A, Karp B. Autograph: Toward Automated, Distributed Worm Signature Detection[C]//Proceedings of the 13th USENIX Security Symposium. 2004.

3 Newsome J, Song D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software[C]//Proceedings of the 12th Annual Network and Distributed System Security Symposium. 2005.

4 Pasupulati A. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities[C]//Proc. of IEEE/IFIP Network Operation and Management Symposium. 2004.