ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.6, June 2008, pp.1519−1532
DOI: 10.3724/SP.J.1001.2008.01519

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于异常诊断的代码注入攻击自动分析和响应系统*

李 闻+, 戴英侠, 连一峰, 冯萍慧, 鲍旭华

(中国科学院 研究生院 信息安全国家重点实验室,北京 100049)

Code-Injection Attack Automatic Analyses and Response System Based on Abnormal Scene Diagnose

LI Wen+, DAI Ying-Xia, LIAN Yi-Feng, FENG Ping-Hui, BAO Xu-Hua

(State Key Laboratory of Information Security, Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: E-mail: wli@is.ac.cn

Li W, Dai YX, Lian YF, Feng PH, Bao XH. Code-Injection attack automatic analyses and response system based on abnormal scene diagnose. *Journal of Software*, 2008,19(6):1519−1532. http://www.jos.org.cn/1000-9825/19/1519.htm

Abstract: The paper presents a code-injection attack automatic analyses and response system. By means of analyzing abnormal scene and syntax of data payload, it generates vulnerability-oriented signatures which are used to filter off variant form of code injection attack based on same attack scheme of same vulnerability. By combining with protocol state and process state during signatures generating and attack responding process, very low false positive rate and lagging without elevating false negative rate can be gained. Some technical details of the protocol type system on Linux and Windows 2000 and experimental results are also presented.

Key words: abnormal scene diagnose; automatic attack signatures generation; automatic attack response; automaton

摘 要: 提出了一种基于进程异常场景分析的代码注入攻击自动分析和响应系统.该系统根据进程异常场景自动分析攻击载荷句法,并生成面向漏洞的攻击特征,由该攻击特征,可以识别和阻断基于同一未知漏洞同种利用方式的各种代码注入攻击的变形.通过在生成攻击特征以及响应攻击的过程中结合网络协议和进程的状态,可以在不升高检测漏警概率的前提下显著地降低响应虚警概率和系统对外服务的响应时间.另外,还简要介绍了基于 Linux 和 Windows 2000 的原型系统,并给出了功能和性能的实验结果.

关键词: 异常场景诊断;攻击特征自动提取;自动攻击响应;自动机

中图法分类号: TP393 **文献标识码**: A

代码注入攻击是指攻击者本地或者远程向进程的线性地址空间注入一段可执行的二进制代码,然后通过某种手段修改进程的正常控制流程,使进程执行这段代码,从而达到预定目的的攻击行为.早期文献中提到的“缓冲区溢出攻击(buffer overflow)”在严格意义上是指这一类攻击的一个子类,由于攻击者注入代码的位置通

* Supported by the National Natural Science Foundation of China under Grant Nos.60403006, 60503046 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2006AA01Z437 (国家高技术研究发展计划(863))

Received 2006-08-15; Accepted 2006-10-10

常在缓冲区,而且修改进程的正常流程往往需要利用对缓冲区缺乏边界检查的编程错误,所以,以“缓冲区溢出攻击”指代全体.其实,某些代码注入攻击从实现手段上,例如格式字符串误用、double-free 等等,与“溢出”都不直接相关,所以在近来的一些文献中都采用了“代码注入攻击(code injection)”这一比较严格的称谓,在本文中也将采用这个称谓取代“缓冲区溢出攻击”.

系统管理员面对的安全挑战主要是如何有效地防护服务器系统的机密性、完整性和可用性不被远程攻击者破坏.对于配置合理的服务器,安全漏洞主要来自服务器程序自身的编码缺陷,其中,由于程序内存管理缺陷导致的代码注入攻击危害最大.据统计,仅去年基于代码注入的攻击就占使 CERT/CC 提出建议的所有重大安全性错误的 50%以上.许多著名的蠕虫也是利用代码注入攻击实现传播的目的,例如冲击波(基于 RPC DCOM 远程溢出)和 CodeRed II(基于 IIS.IDA/IDQ ISAPI 远程溢出).对于这些漏洞,厂家会提供补丁供管理员下载和安装.随着计算机网络规模的增大以及新的漏洞挖掘技术的出现,这种基于静态补丁的传统响应方式越来越难以满足需要.

首先,由于漏洞挖掘及利用技术的成熟,厂家发布补丁的时间几乎总是落后于攻击者利用新漏洞(0day 漏洞)发动攻击的时间.厂家必须在每一个版本的产品上充分测试他们的补丁以确保补丁的可用性,而攻击者不必关心攻击程序是否会危害系统.其次,厂家发布补丁到补丁被应用于实际系统有一个时间间隔.对于商用系统,可用性比安全性更为重要,所以,管理员在安装系统之前需要充分测试补丁和系统的兼容性.这些因素导致了漏洞利用时间和补丁部署时间的间隔.

这就需要有一种实时的防护手段替代传统的静态补丁方案,这种手段必须能够:a) 抵抗大多数针对未知漏洞的代码注入攻击;b) 检测和响应攻击的过程不需要过多的人为干预;c) 虚警概率低,不会显著降低系统的可用性;d) 部署和拆除容易,不用重新编译现有的应用程序.国外的研究者做了大量相关的理论性和工程性的研究,一大批可以满足一个或者几个需求的原型系统和工具被提出.早期的研究着眼于如何加强程序的安全性,这些研究可以被分为如下两类:

(1) 静态的解决方案(static solution):这一类方案通过使用安全的编程语言[1−3]、对源代码的静态检查[4]、对编译器的增强[5]、使用特殊的安全函数库[6]等,从源代码级别消除代码注入攻击的危害,但是需要重写或者重新编译应用程序甚至操作系统的源代码;

(2) 运行时解决方案(run-time solution):这一类方案通过使用局部或者全局的砂箱或虚拟机[7]、指令随机化(randomized instruction set emulation,简称 RISE)[8]和线性地址空间随机化(address space layout randomization,简称 ASLR)[9],可以在极低的漏警概率下有效地阻断代码注入攻击,不需要重写重编译源代码,但在成功地阻断攻击后,系统往往因处于不确定状态而崩溃,系统的可用性被破坏.

近来的一些研究着重于如何阻断代码注入攻击,而且不会显著地降低系统可用性.Shield[10]根据漏洞的特性,在该攻击程序广泛传播之前手工定义攻击特征并基于网络协议状态阻断攻击,但是不能防护基于未知漏洞的攻击.TaintCheck[11]使用模拟器动态跟踪来自网络的数据在进程中的传播,同时,生成攻击特征并且过滤网络数据,效率比较低.DIRA[12]始终维护一个内存数据的更新日志,当攻击发生时做“回卷”.ARBOR[13]使用静态的方法检查缓冲区的边界,在攻击发生前截断网络数据流,但是,该系统只能防护代码注入攻击的一个子类——缓冲区溢出攻击,对格式字符串攻击等无能为力.

本文描述了一种使用基于异常场景诊断的代码注入攻击自动分析及响应系统.该系统根据进程受攻击时的异常场景分析得出攻击载荷的句法结构,并且结合进程状态以及网络协议的状态自动产生面向漏洞的攻击特征,通过把攻击特征反馈给前端的过滤模块.该系统可以对大多数基于未知漏洞的代码注入攻击做出自动响应.通过在分析和响应过程中结合进程状态和协议状态,该系统在不升高漏警概率的情况下获得几乎为 0 的虚警概率.本系统不需要重新改写或编译现有的应用程序,也不需要改写内核,从而易于部署和拆除.

本文第 1 节描述系统组成和基本工作流程.第 2 节描述攻击载荷句法分析原理并简要介绍句法分析引擎模块.第 3 节构造应用状态自动机模型,并且阐述在攻击特征提取和攻击响应过程中作为一种降低虚警概率和系统响应时间的有效手段的应用.第 4 节给出对系统的功能和性能的测试结果.最后是对现有工作的总结和对未

来工作的展望.

1 系统结构

进程线性地址空间随机化技术(ASLR)[9]是近年来新出现的一种进程保护技术.在使用 ASLR 加固进程后,进程的栈(stack)、堆(heap)和代码段等关键数据结构将处于随机的地址,攻击者将不能准确定位这些地址,从而不能按预计方式运行恶意代码,结果是造成进程处于不确定状态从而崩溃.如果只想阻止恶意代码运行,这一类技术很好,但它无助于保持系统的正常运行.如果从系统的可用性角度考虑,这一技术远远不及厂家发布的补丁更可靠.

在我们的系统中,利用 ASLR 技术高敏感性的特点,把它作为一种检测而不是阻断代码注入攻击发生的手段,把进程异常事件作为触发攻击特征分析的信号.由于 ASLR 的检测漏警概率极低,所以我们的系统也可以获得极低的漏警概率.该系统由 5 个模块构成,如图 1 所示.

- 本地进程监测模块 LPMM(local process monitor module)启动一个 ASLR 化的服务器进程,以进程状态自动机(local process automaton,简称 LPA)实例记录进程的运行状态;
- 协议状态监测模块 NPMM(net protocol monitor module)在 TCP/IP 传输层之上以协议状态自动机 NPA(network protocol automaton)实例记录每一个协议会话的状态;
- 攻击载荷句法分析引擎 ASAE(attack signature analyses engine)在 NPMM 检测到进程异常时,根据 NPMM 提供的 NPA 状态、攻击载荷数据和 LPMM 提供的 LPA 状态、内存镜像分析得出攻击特征并保存在 ASDB 中;
- 攻击特征库 ASDB(attack signature data base)以树状结构保存攻击特征,按照漏洞类型、程序名、攻击句法类型作分类,便于攻击特征的快速存取;
- 网络数据过滤模块 NFM(network filter module)在 TCP/IP 传输层之上监测各个网络数据会话,根据当前 NPA、LPA 的状态、当前会话数据载荷和从 ASDB 反馈回的攻击特征之间的比较结果,检测并截断恶意会话.

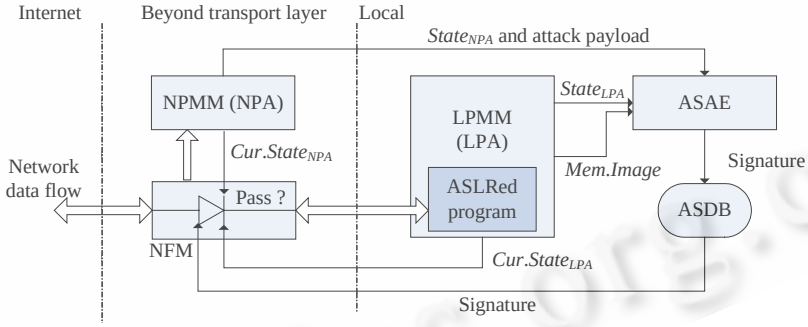


Fig.1 System architecture

图 1 系统结构

ASAE 根据三元组(NPA 状态,LPA 状态,内存镜像)产生攻击特征的过程,是一个高资源消耗非实时的过程,在我们的系统中,只有进程异常才会触发这一过程,只有遇到基于未知漏洞的攻击时才有可能发生这种情况.

在一个实用的自动化的响应系统中,对虚警概率的要求比漏警概率更为苛刻.如果虚警概率过高,系统虽然不容易漏过攻击数据流,但是很可能拒绝很多正常的数据会话,其结果是系统的可用性被显著降低.我们的系统在过滤网络数据流的,同时也兼顾 NPA 和 LPA 的状态,可以在不提高漏警概率的前提下显著地降低虚警概率.

由于 NFM 比较消息载荷是否符合攻击句法的过程是一个相对低效的过程,如果在阻断恶意会话中没有 NPA 和 LPA 的参与,系统就不得不在每一次收到网络数据的时候都与攻击句法比较.我们的系统只在 NPA 和

LPA 的状态都匹配时才开始一个比较操作,显著地提高了系统的数据吞吐率.

由于代码注入攻击与网络层无关,所以,NPMM 和 NFM 部署于网络传输层和应用层之间对系统功能没有影响,同时,部署于传输层之上使得系统可以处理像 HTTPS,SFTP 这样的使用传输层加密的应用协议.

2 攻击载荷的自动化句法分析

现有的一些自动化的攻击特征生成方法,如文献[15−17],其本质上是基于统计的方法,把消息载荷看作无差别的字节串,完全忽略了攻击载荷中包含的句法信息,所以生成的特征在更大程度上依赖于具体的攻击程序而不是面向漏洞,而且这些方法需要大量的攻击样本数据,不适用于对实时性要求很高的系统.

我们希望生成一种面向漏洞的攻击特征,从而可以识别和滤除基于该漏洞的各种攻击变化.一个思路是利用攻击载荷中包含的句法信息.同一个漏洞利用方式可能有多种,但是每一种利用方式有固定的句法.例如,一个栈溢出漏洞,可能的利用方式只有溢出某一特定的缓冲区,用特定的数值覆盖某几个特定的指针或者返回地址.缓冲区的位置,被覆盖的指针、返回地址等是确定的,所以,不同目的的攻击载荷必须按照某一确定的顺序包含这些确定的值.

我们把代码注入攻击按照漏洞类型分为 4 种,即基于栈溢出漏洞的攻击、基于堆管理信息的攻击、基于格式字符串滥用的攻击和基于其他数据结构的攻击.前 3 种可以分析出攻击类型和句法结构,最后一种可以得出类型信息,但是不能准确得出句法结构,此时,系统的漏警概率没有显著变化;而响应攻击时,虚警概率有所升高.幸而目前大多数代码注入攻击都是基于前 3 种类型的,因此,目前我们将主要的研究工作集中在针对这 3 种攻击类型的防护上.这里需要指出的是,对于攻击者来说,不是每一个漏洞都有多种可用的利用方式,大多数情况下只有一种方式是可能的.由于篇幅所限,对于每一种类型的漏洞,我们只示例性地给出对一种利用方式的攻击载荷句法分析过程.

代码注入攻击在实现过程上可以分为两步:首先,攻击者利用进程的某些缺陷(如对缓冲区缺乏检查、对格式字符串处理不当、存在悬疑指针等等)远程向进程的线性地址空间注入一段代码;其次,设法利用函数调用指令等控制流转移指令转而执行这段代码.我们注意到,进程的控制流程改变是代码注入攻击过程的分水岭,在执行控制流转移指令之前,进程执行的是本地代码,在执行这一指令之后,进程执行的是恶意代码.我们以符号 I_j 表示这一控制流转移指令,以符号 I_c 表示引起进程崩溃异常的指令.我们记 I_j 跳转的目标地址为 $Addr_{shellcode}$,这一数值必须在注入代码的过程中写入某个内存线性地址,记为 $Addr_{overwrite}$.记代码注入前该地址保存的数值为 $Addr_{redirect}$,即代码注入前$[Addr_{overwrite}]=Addr_{redirect}$,代码注入后$[Addr_{overwrite}]=Addr_{shellcode}$.

2.1 基于栈溢出漏洞的载荷句法分析

图 2 给出了函数调用栈的结构.攻击成功发生的前提条件是函数局部变量中必须包含一个足够长的、可以被越界改写的数组,并且该数组以上必须包含可被利用的地址数据,例如,函数指针或者函数返回地址.

如果进程对接受外界输入的数组 a 缺乏合适的边界检查,则攻击者写入包含恶意代码的数据到数组 a,并且用数值 $Addr_{shellcode}$ 覆盖地址 $Addr_{overwrite}$ 处的内容 $Addr_{redirect}$,当该数值由于函数调用(调用返回)被载入 EIP 时,恶意代码得以执行.

进程异常原因有两种:一种是攻击者不能猜测出恶意代码的入口地址,所以 $Addr_{shellcode}$ 指向的是一个无效的地址,当进程把这个值载入 EIP 后将发生内存访问违例,此时 $I_j=I_c$.通过分析 I_j 引用的线性地址,我们可以获得 $Addr_{shellcode}$;另一种是 $Addr_{shellcode}$ 被误指向进程的其他数据结构,进程在执行完 I_j 后的若干步内,执行 I_c 试图引用一块无效的内存区域发生异常,并且有 $I_j\neq I_c$.此时,仅仅从 I_c 反向推演出 I_j 是不可能的,我们采用的办法是在 LPMM 中从 LPA 记录下的进程状态为起点正向重新运行程序,同时跟踪进程控制转移指令的目的地址从而发现 I_j,进而得出 $Addr_{shellcode}$,这个过程可能需要重复多次.

无论哪种情况,在发生内存访问违例时,控制寄存器 CR2 存放违例线性地址,在内核栈上会保存当前 EIP 指向引起内存访问违例的指令.通过比较 EIP 和 CR2 的内容是否匹配就可以区分这两种情况.最后,我们可以得出此种栈溢出攻击载荷的句法结构:$\{SYNTAX\}=\{VLB|Addr_{shellcode}|VLB\}$,其中,$Addr_{shellcode}$ 是只与具体漏洞相关的确

定值.*VLB* 是指不定长度的字节串.

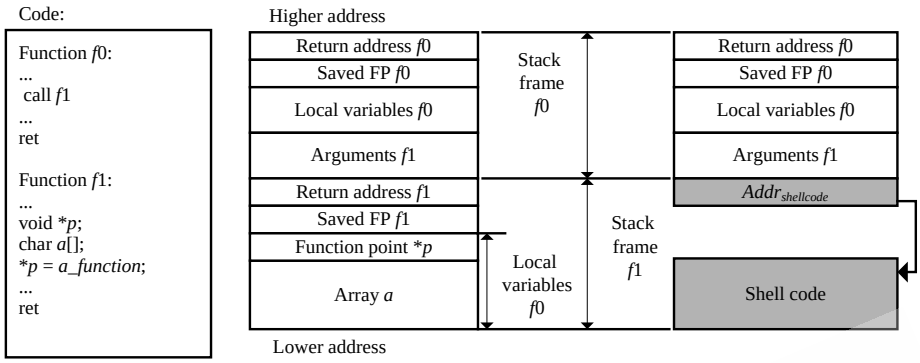


Fig.2 A stack-based code injection attack

图 2 一种栈溢出攻击

2.2 基于堆管理信息的攻击载荷句法分析

基于堆的溢出攻击一般通过改写堆管理数据块内的内容来实现.由于篇幅所限,本节给出对基于堆管理数据块的溢出攻击的最一般的利用方式的分析,其他基于堆管理数据块的高级利用方式,例如 double-free、构造虚假数据块等的利用方式细节,可参考文献[19],分析方法不在本文给出.

图 3 给出了 Linux 平台上的堆结构,堆由一系列不同大小的数据块组成,所有的空闲数据块使用双向链表连接起来.有些数据块已被分配给用户,例如 *chunk* 1;有些数据块处于空闲状态,例如 *chunk*2/*chunk* 3/*chunk* 4.两个空闲块在线性地址空间永远不会相邻,如果在堆内存分配/释放函数调用后有这种情况发生,则两个相邻的空闲块被合并为一个.

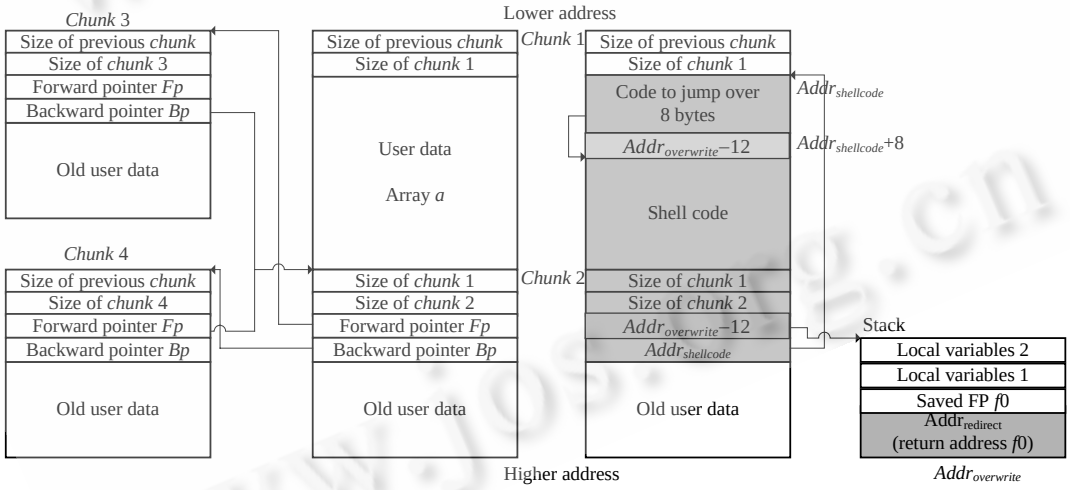


Fig.3 A heap-based code injection attack

图 3 一种堆溢出攻击

在图 3 中,如果 *chunk* 1 被释放,*chunk* 2 就必须要被从空闲块链表中摘出,于是有

$$chunk\ 2\rightarrow Fp\rightarrow Bp=chunk\ 2\rightarrow Bp,\text{相当于:}[chunk\ 2\rightarrow Fp+12]=chunk\ 2\rightarrow Bp \quad (1)$$

$$chunk\ 2\rightarrow Bp\rightarrow Fp=chunk\ 2\rightarrow Fp,\text{相当于:}[chunk\ 2\rightarrow Bp+8]=chunk\ 2\rightarrow Fp \quad (2)$$

如果攻击者在 *chunk* 1 释放前溢出了 *chunk* 1 内的数组,则导致改写了 *chunk* 2 的 *Fp* 和 *Bp* 指针,使得

$$Bp=Addr_{shellcode} \text{ 且 } Fp+12=Addr_{overwrite} \quad (3)$$

当块释放函数调用时,$Addr_{overwrite}$ 原来包含的地址数据 $Addr_{redirect}$ 将被 $Addr_{shellcode}$ 覆盖,于是,当控制流程转向 $Addr_{redirect}$ 时,恶意代码得以执行.

在部署了地址随机化之后,堆块的基地址是一个随机值,攻击者不能猜测出 $Addr_{shellcode}$ 和 $Addr_{overwrite}$ 的确切值而引起进程异常.第 1 种情况是,由于块释放函数执行时,$Addr_{overwrite}$ 指向的不是一个可写的线性地址,在执行块释放函数时异常,此时 I_j 还未执行且 $I_j \neq I_c$.通过分析 I_c 试图写入的地址可以获得 $Addr_{overwrite}$,进一步根据公式(1)、公式(3)获得 $Addr_{shellcode}$.第 2 种情况是,$Addr_{overwrite}$ 可写,但是 $Addr_{shellcode}$+8 指向的地址不可写,此时与情况 1 相同,可获得 $Addr_{shellcode}$,进一步由公式(2)、公式(3)获得 $Addr_{overwrite}$.第 3 种情况是,进程在执行完堆释放函数,执行指令 I_j 或者是执行 I_j 后继的某一指令时异常,对于前者有 $I_c=I_j$,我们直接分析 I_j 的目标地址可以获得 $Addr_{shellcode}$ 进而获得 $Addr_{overwrite}$,对于后者有 $I_c \neq I_j$,我们需要使用类似于第 2.1 节中的自动调试过程最终获得 $Addr_{shellcode}$ 和 $Addr_{overwrite}$.最后,我们可以获得攻击载荷的句法结构,可以表示为

$$\{SYNTAX\}=\{VLB|JMPAddr_{shellcode}+8|VLB|Addr_{overwrite}-12|Addr_{shellcode}|VLB\},$$

其中,$Addr_{shellcode}$ 和 $Addr_{overwrite}$ 是只与具体漏洞相关的确定值.

2.3 基于格式字符串滥用的攻击载荷句法分析

基于格式化函数攻击是 1999 年前后出现的一种攻击方法[18],程序中的格式函数,例如*printf,syslog 等的格式串参数如果部分地可以被攻击者控制,攻击者就有可能以任意值覆盖进程任意内存区段的内容.

一串“%”格式符指导格式化函数如何处理后继参数.在使用_stdcall 函数调用方式时,参数被由右到左依次压入栈中,如图 4 所示.格式参数中每一个格式符对应一个调用栈中参数,格式字符串函数根据格式符把对应参数位置处的 4 个字节解释为一个数值或者是一个地址.格式符“%*n*”使得格式化函数把当前打印的字节数以长整数的形式写入对应参数指向的内存区段中,通过巧妙地构造格式参数,攻击者可以利用“%*n*”结合其他的格式符在格式化函数调用中以任意值覆盖任意内存区段的 4 个字节.

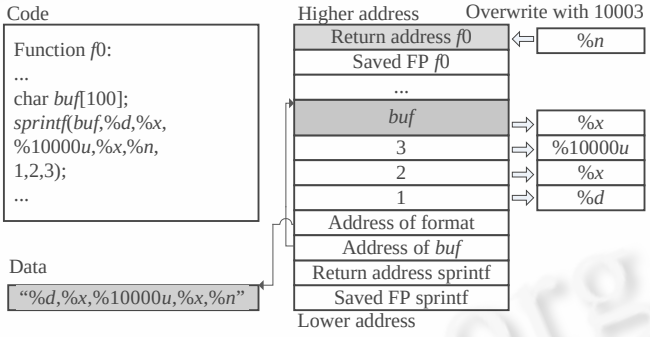


Fig.4 A format string attack

图 4 一种格式字符串滥用攻击

格式字符串攻击比较特殊的一点是,攻击者构造的跳转地址及其覆盖目的地址可能都不直接包含在攻击载荷中,而是通过“%*nx*”,“%*nu*”(*n* 为一个大整数)等格式符间接写入,所以很难把通过分析进程异常得出的特征地址 $Addr_{shellcode}$,$Addr_{overwrite}$ 和攻击载荷对应起来.但是,由于我们在 LPMM 中保存了函数调用的参数信息,所以,我们可以准确获得格式字符串参数.在实践中,我们检查消息载荷中是否包含“%*n*”格式符且该格式符之前是否包含形如“%*nx*”,“%*nu*”的格式符.由上所述,攻击载荷的句法结构为

$$\{SYNTAX\}=\{VLB|/\%n/|VLB|/\%(\backslash d)*u|\%(\backslash d)*x|\%(\backslash d)*d/|VLB\}.$$

2.4 攻击载荷句法分析引擎ASAE和自动调试

ASAE 根据 LPMM 提供的进程异常后的数据映像判断攻击类型 Type,然后调用句法分析逻辑分析句法

Syntax.进程发生异常时,某些情况下可以直接获得攻击类型、攻击载荷的句法结构信息(可辨别的异常),其他情况下需要重新启动进程,改变进程栈、堆和数据段的基地址,反复这一过程直到可辨别的异常出现为止,这一过程称为自动调试.改变数据结构的基地址,相当于改变这些结构在进程的线性地址空间中的布局,在调试过程中,每两次布局相互不重叠.如果始终没有可辨别的异常出现,则把整个攻击载荷作为攻击特征,此时,该攻击特征不能用来识别攻击变种.所幸,在绝大多数情况下,经过自动调试都可以出现可辨别的异常.目前采用的这种自动调试策略是一种比较低效的枚举策略,虽然不影响系统的性能(自动调试过程只有在进程异常时才可能被启动),但是会降低攻击特征的分析效率.基于先验知识的高级调试器正在研究中.

在不同的硬件平台/软件平台/函数库版本上,同一种利用方式的分析逻辑可能有所不同.如果对于一个漏洞类型可能的利用方式有多种,相应的句法结构也有多种.每一种句法的分析逻辑被做成一个插件.在目前的系统已经实现的插件中,基于栈溢出漏洞的有 4 个,基于堆溢出漏洞的有 8 个,基于格式字符串漏洞的有 18 个,基于其他数据结构的有 2 个.

3 进程自动机和协议自动机

本质上,服务器进程就是一个对网络消息自动作出响应的状态自动机,我们用应用自动机模型(application state automaton)对它作一般性的抽象.一个应用自动机对网络消息作出响应的同时,改变外部协议的状态(protocol state),反映网络协议进行到了哪一步.而它的每一个协议状态可以细化为若干个进程状态(process state),进程状态的转移,反映了该自动机处理某个网络消息时进程内部数据的改变,如图 5 所示.在我们的模型中,分别以进程自动机和协议自动机加以抽象.ASLR 化的服务器进程在受到攻击而异常之前将停留在某个状态,我们定义该状态为受攻击状态(vulnerable state).我们的模型基于以下两点假设:

(1) 攻击者必须使被攻击进程达到受攻击状态时,才有可能实施代码注入攻击;

(2) 有效攻击代码是一次注入的.

首先,在现实世界中,大多数漏洞利用起来需要严格的前置条件,攻击者必须使服务器处于某一个特定握手阶段或者获得一定的权限,例如,Serv-U FTP 服务器的 MDTM 栈溢出必须在攻击者以合法账户登陆之后才能利用;其次,攻击者必须精心构造消息载荷保证消息载荷的完整和有序性.由于两次注入间隔之间进程的状态不能完全由攻击者控制,所以,分次注入实现困难而且成功率很低.不排除攻击者首先注入一段有效攻击代码,下载另一段有效代码的情况.在这种情况下,第 1 段攻击代码必须能够生效,从而必须被一次注入.

基于上述两个假设,我们可以仅仅检查攻击状态之前该会话所接受的一个网络消息载荷是否匹配攻击语法.这样做可以极大地提高使用复杂网络协议的服务器系统的效率,同时,消息载荷匹配算法的设计可以大幅度简化,而不会提高响应虚警概率.有些服务器使用非常简单的网络协议例如 HTTP.此时,仅仅把受攻击协议状态包含在攻击特征中对提升系统性能和降低虚警概率没有太多帮助,所以,我们需要在攻击特征中引入更细粒度的状态——进程状态.

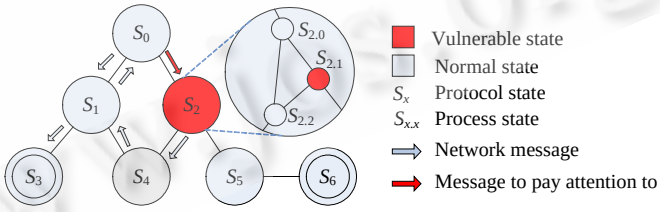


Fig.5 Application state automaton

图 5 应用状态自动机

3.1 网络协议自动机和协议状态监测模块

3.1.1 网络协议自动机定义语言

通过分析网络协议,我们发现应用层协议的复杂性使得仅仅从收到的网络数据无法自动地区分 NPA 的各

个状态,必须事先对每一个需要关注的应用层协议定义对应的 NPA.目前,没有自动化的方法替代手工的定义过程,所幸这一过程可以根据现有的 RFC 文档来完成.我们必须定义:

- 应用程序标识(application identification):NPMM 如何区分消息数据属于何种应用程序.
- 消息标识(event identification):如何从收到的消息数据中获得消息类型.
- 会话标识(session identification):如何得知该消息属于哪一个会话.
- 自动机定义(NPA specification):定义 NPA 的状态、消息以及状态的转移函数.

这些定义与具体的应用程序及攻击事件无关,而仅与应用层协议相关.所有的定义脚本只用写一次,然后随着我们的系统一起发布给用户.事实上,可以分多次定义一个 NPA,例如,在 HTTP 协议中只定义"GET"和"POST"这两个方法(method)的相关状态,本质上分多次定义了整个协议状态图的各个子图,在处理过程中需要对各个子图加以合并.附录 1 给出了用我们自己开发的协议自动机定义语言(process state definition language,简称 PSDL)对 FTP 协议用户认证阶段的定义.

3.1.2 消息识别和会话分发

会话(session)是在 NPMM 中对客户端和服务器通信单位的一种抽象,一个会话可能包含多次消息传递和状态的转移.所以,如果 NPMM 的状态超过 1 个,必须有手段识别消息类型以及该消息属于哪一个会话.

一个服务器可能同时为多个客户端提供服务,每一次服务可能包含多个网络连接.在 NPMM 中为每一个会话维护了一个 NPA 实例.由消息数据中的 TCP/UDP 端口字段可以得知,该消息属于哪一个应用程序(application identification).根据 PSDL 中定义的消息标识(event identification),我们可以得知该消息的消息类型.根据每一个 NPA 实例维护的会话列表/端口列表,我们可以得知该消息的会话 ID.当 NPMM 收到一个消息后,根据应用程序名、消息类型和会话 ID 把消息分发给对应的应用程序,同时驱动 NPA 实例的状态变化,NPMM 不必保存所有消息数据,而只用维护当前的状态和最后一个消息载荷即可.

NPA 实例的寿命不长于与之相联系的会话的寿命,现代的网络服务器应用层都有超时机制用来防止过多的未完成会话存在过长的时间而耗尽系统资源.在我们的系统中,由于 NPA 要占用额外的资源,每一个 NPA 的寿命可能还会小于这个超时时间,如果 NPA 提前终止,相应的会话也被终止.该机制可以防止攻击者恶意生成许多"半开"会话而造成服务器资源耗尽.

3.1.3 数据块重组

在传输层之上,IP 分片已经被重组,NPA 建立于网络传输层和应用层之间,所以无须关心底层的 IP 分片细节.虽然如此,每一块被 NPA 接收的数据不一定代表一个完整的应用层消息,这种现象可能是由于 TCP 拥塞或者是应用程序的特殊消息处理实现机制引起的.NPA 必须区分每一块数据属于哪一个消息.在应用层协议中,一个消息不可能跨过多个网络连接,所以,我们为每一个网络连接维护一个缓存,在接收到一块数据时复制一份并加以组装,当能够辨别出该消息的类型时,把消息传送给 NPA 并且清除缓存.与第 3.1.2 节的原因相同,每一个缓存也有不大于对应连接的超时时间.

3.2 进程状态自动机和进程状态监测模块

3.2.1 函数调用返回动作到下推自动机的对应

引入进程状态自动机的难点是如何定义进程自动机的状态,我们不可能像对待网络协议自动机那样为每一个应用程序通过预定义脚本的方式预定义,这样做需要为每一个应用程序写一个脚本,在没有程序源码时是几乎不可能的,即使有源代码,工作量也是巨大的.一个想法是,通过某种对应方式在进程的函数调用——返回动作和进程自动机之间建立某种对应关系,自动化地构造进程状态自动机.

如果进程某子函数 f_i 被调用的动作用符号 f_i^{call} 表示,该函数返回调用者的动作以 f_i^{ret} 表示,所有的符号构成字母表 Σ.假设程序除 main 函数包含的子函数数目为 N,则理想情况下,一个服务器在从启动到正常退出过程中的函数调用返回动作可以用形如 $f_{main}^{call} f_i^{call} \ldots f_j^{call} f_j^{ret} f_{j+1}^{call} f_{j+1}^{ret} \ldots f_i^{ret} f_{main}^{ret}, \forall i, j \in [0, N-1], f_i^{call}, f_i^{ret}, f_j^{call}, f_j^{ret} \in \Sigma$ 的符号串表示,满足

(1) $\forall i \in [0, N-1], f_i^{call}, f_i^{ret}$ 必须成对出现(子函数必须返回).

(2) $\forall i \in [0, N-1], f_i^{call}, f_i^{ret}$ 之间必须有成对出现的 f_j^{call}, f_j^{ret} (函数不能交叉嵌套).

该文法 G 可形式化地表示为 $G=(V,T,P,S)$,其中:

文法变量集合 $V=\{A\}\cup\{B_i|i\in[0,N-1]\}$.

终极符号集合 $T=\{f_i^{call}, f_i^{ret} \mid \forall i \in [0, N-1], f_i^{call}, f_i^{ret} \in \Sigma\}$.

产生式集合 $P=\{A \to f_{main}^{call} B_i f_{main}^{ret}, B_i \to f_i^{call} B_j f_i^{ret} \mid f_i^{call} f_i^{ret} B_j \mid \varepsilon\}, i, j \in [0, N-1]$.

文法的开始符号 $S=A$.

转化为 Greibach 范式:

$V=\{A\}\cup\{B_i|i\in[0,N-1]\}\cup\{C_i|i\in[0,N-1]\}\cup\{C\}$.

$T=\{f_i^{call}, f_i^{ret} \mid \forall i \in [0, N-1], f_i^{call}, f_i^{ret} \in \Sigma\}$.

$S=\{A\}$.

$P=\{A \to f_{main}^{call} B_i C, C \to f_{main}^{ret}, B_i \to f_i^{call} B_j C_i \mid f_j^{call} C_j \mid \varepsilon, C_i \to f_i^{ret}\}$.

引理 1. G 的产生语言 $L(G)$不是正则语言,G 不是正则文法.

证明:见附录 2. □

由于该文法不是正则文法,所以不存在有穷状态自动机与它对应.但是,该文法显然是上下文无关文法,所以,我们可以构造如下所述的下推自动机与其对应.

$M=(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$,$N(M)=L(G)$,其中:

状态集合 $Q=\{q_0\}$, $\Sigma=\{f_i^{call}, f_i^{ret} \mid i \in [0, N-1]\}\cup\{f_{main}^{call}, f_{main}^{ret}\}$,终止状态集合 $F=\varnothing$.

栈符号表 Γ,开始符号 $Z_0=A$.

状态转移函数定义为:

$\delta(q_0, f_{main}^{call}, A)=\{(q_0, B_i C)\}, \delta(q_0, f_{main}^{ret}, C)=\{(q_0, \varepsilon)\}$,

$\delta(q_0, f_i^{call}, B_i)=\{(q_0, B_j C_i)\}, i, j \in [0, N-1], i \neq j$,

$\delta(q_0, f_j^{call}, B_i)=\{(q_0, C_j)\}, i, j \in [0, N-1], i \neq j$,

$\delta(q_0, f_i^{ret}, C_i)=\{(q_0, \varepsilon)\}, i \in [0, N-1]$.

由于集合 Σ 是有限的,我们可以在集合 Σ、集合 $\{C_i|i\in[0,N-1]\}$ 以及集合 $\{D_i|i\in[0,N-1]\}$ 之间建立一一映射的关系,从而 Γ 也是一个有限集合.于是,我们可以用下推自动机当前的栈符号串表示某一时刻进程的状态,即 $State_{LPA} \in \Gamma^*$.

3.2.2 进程状态监测模块

LPMM 为每一个进程(实际是线程,见第 3.2.3 节)维护一个下推自动机实例,我们通过对进程作动态分析,实时地跟踪进程的函数调用返回动作来推动该自动机的状态转换.由于子函数调用可能通过间接 call 指令完成(如 *call*[*eax*]),在间接 call 执行之前,我们不可能知道全部以函数入口地址表示的符号 f_i^{main}, f_i^{ret} 以及集合 Σ,也不可能完全知道栈符号表 Γ,但是已知 Σ 和 Γ 是有限的,所以,我们可以比较新出现的字母 f_i^{main}, f_i^{ret} 是否已在 Σ 中,然后为新出现的字母标号、对栈符号表 Γ 也做类似的处理.其结果是,任意时刻 LPMM 维护的实际是进程状态机的一个“子集”.为了减少重复计算的开销,我们在进程结束时把本下推自动机的字母表的“意义”,即字母和函数入口地址的对应关系保存下来重复使用.由于使用了 ASLR 技术,所以每次实际的入口地址是上一次保存值加/减一个偏移量.

3.2.3 对多进程/多线程的处理

Linux 平台和 Windows 平台对进程(process)和线程(thread)有不同的理解.在 Linux 平台上使用系统调用 fork 生成子进程执行分支任务,使用系统调用 exec 使子进程执行新的可执行文件.在 Windows 平台上,进程调用 API CreateThread 生成一个新的属于进程的线程执行分支任务,功能相当于 fork,不过没有新创建进程.调用 API

CreateProcess 生成一个新的进程,功能相当于 fork+exec.

不论何种情况,如果只是执行分支任务,我们把当前的 LPA 实例完整地复制一份给新创建的线程或者进程;如果产生了分支并且执行新的可执行文件,我们就为子进程创建一个新的 LPA 实例.

4 实验结果

当前,我们在 Linux-x86 和 Windows 2000-x86 上实现了原型系统,该系统目前可以对使用 HTTP,HTTPS,FTP,SFTP,SMTP 和 POP3 协议的服务器提供保护.通过编写 PSDL 脚本,可以扩展系统对其他协议的支持能力.实验硬件平台为 Pentium 4 2.5G/1G RAM 的服务器.

4.1 攻击检测/特征自动生成能力测试

我们选择 Unix/Linux 平台上广泛使用的一种 FTP 服务器——wu-ftpd-2.60 以及在 Windows 系统上自带的 HTTP 服务器——IIS5.0 作为被保护对象,所有软件未打任何补丁以模拟未知漏洞.我们选择了一些具有代表性的漏洞并使用现有的攻击程序加以攻击,结果见表 1 和表 2.

Table 1 Typical vulnerabilities of wu-ftpd-2.60

表 1 wu-ftpd-2.60 的一些典型漏洞

CVE/BID No.	Type	Description	May create signature?
CVE-2000-0573	Format string overflow	SITE EXEC format string remote overflow	Yes
CVE-2001-0187	format string overflow	debug mode format string overflow	Yes
CVE-2001-0550	Heap overflow	heap corruption in file glob	Yes
CAN-2003-0466	Integer overflow	fb_realpath remote off-by-one	Yes
(BID8668)	Stack overflow	SockPrintf remove stack over run	Yes
CVE-2004-0148	Logic error	restricted gid gain access	No
CVE-2004-0185	Stack overflow	S/key remote stack based buffer over run	Yes

Table 2 Typical vulnerabilities of IIS 5.0

表 2 IIS 5.0 的一些典型漏洞

CVE/BID No.	Type	Description	May create signature?
CVE-2000-0884	Logic error	Unicode translation heap corruption	No
CVE-2002-0150	Heap overflow	ASP HTTP header buffer overflow	Yes
CVE-2002-0364	Heap overflow	IIS htr chunked encoding buffer over flow	Yes
CVE-2002-1310	Heap overflow	JRun long url buffer over flow	Yes
CVE-2002-1744	Logic error	CodeBrws.asp content disclosure	No
CVE-2003-0224	Heap overflow	SSI filename buffer overflow	No
CVE-2004-1135	Stack overflow	w3who ISAPI remote buffer overflow	Yes

实验结果表明,原型系统对基于栈溢出、堆溢出以及格式字符串滥用的攻击能够自动生成攻击特征.对于其他类型的攻击,如果是间接造成代码注入的,例如 CAN-2003-0466,则可以生成攻击特征;对于程序设计的逻辑错误,例如漏洞 CVE-2004-0148,CVE-2000-0884,由于利用这些漏洞时不会向进程内部注入代码,不属于代码注入攻击范畴,所以不能自动生成攻击特征.

每一个漏洞的每一种利用方式对应一个句法结构,大多数漏洞只有一种利用方式,所以大多数攻击特征只包含一个句法结构.每一个攻击特征可以用五元组(程序名,异常时 LPA 状态,异常时 NPA 状态,攻击载荷句法,攻击类型)表示.例如,针对漏洞 CVE-2001-0550 的攻击特征为:

({*name*}={*in.ftpd*,md5=1534485451cdd11fc6ed6e13fc606126},

{*LPA_VSTATE*,*BASE*={08040000,40000000}}={*B*_0804bd4d,*B*_08056c18,*B*_08059019,*B*_4207ac55},

{*NPA_VSTATE*}={*S_FTPCkeckCommand*},

{*SYNTAX*}={*VLB*|0XFFE59000|*VLB*|0X405F0000/16|0X46CF0000/16|*VLB*,$2=$3~12},

{*TYPE*}={*HEAP_BASED_BO*})

4.2 响应能力测试

针对表 1、表 2 的所有可以自动生成攻击特征的漏洞,除了现有的攻击程序,我们还写了 1~5 个不同的攻击

程序,使用不同功能的攻击载荷,模拟来自不同攻击者的攻击行为,它们是添加特权用户、绑定监听端口并在接到连接时生成交互 shell、反向连接式交互 shell、上传文件和下载文件.在实验环境中,所有的攻击会话都被阻断.

4.3 虚警概率测试

我们使用上述的 IIS 服务器,配置原型系统使用第 4.1 节中生成的攻击特征.然后配置 NFM 在过滤会话数据时不匹配 LPA 状态、不匹配 NPA 状态、两者都不匹配以及两者都匹配 4 种情况下,编写 perl 脚本和 IIS 服务器之间正常地交互 1G 左右的数据,期望没有数据会话被截断.表 3 给出了测试结果.

Table 3 Result of false positive testing

表 3 虚警概率测试结果

E ıble NPA?	E able LPA?	Total H TP request	Chunked H TP request	False p sitive (%)
No	No	24 573	458	1.86
Yes	No	24 573	400	1.62
No	Yes	24 573	52	0.21
Yes	Yes	24 573	0	0.0

实验表明,LPA 和 NPA 降低虚警概率的作用是显著的:在仅使用攻击载荷句法匹配所有会话而不结合 LPA 和 NPA 的状态时,虚警概率在本实验环境下为 1.9%;在部署了 LPA 后降为 0.21%;如果同时结合两者状态时,则在本实验中没有虚警发生.由于 HTTP 协议状态数很少,所以 NPA 降低虚警概率的作用没有 LPA 显著.

4.4 性能测试

我们使用 LoadRunner 8.1 从系统响应时间的角度对性能作一个评估性的测试.LoadRunner 配置为使用不同数目的线程并发访问 HTTP 和 FTP 服务器,对每一种服务器运行 10 组策略脚本访问不同的资源,并且测试平均响应时间.对每一个服务器在部署了 ASLR 以及部署原型系统并且按照第 4.1 节中 4 种配置共 5 种情况下加以测试.图 6 给出了测试结果.

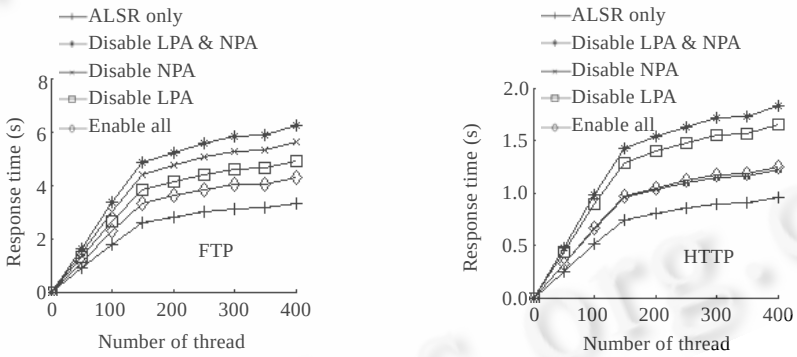


Fig.6 Responding of FTP server and HTTP server

图 6 FTP 服务器和 HTTP 服务器响应时间

可以看出,LPA 和 NPA 降低系统响应时间的作用是显著的.如果禁止 LPA 和 NPA,系统不得不在每一次收到网络数据时都与句法结构作比较,结果比仅仅部署 ASLR 的服务器增大了大约一倍的响应时间.HTTP 协议比 FTP 协议的协议状态少得多,所以,禁用 NPA 对 HTTP 服务器的响应时间影响更显著.反之,HTTP 服务器结构远比 FTP 服务器复杂,所以,禁用 LPA 对 FTP 服务器的负面影响更大.正常配置的原型系统大约使响应时间增大 10%~35%.

5 总结和展望

本系统自动化地生成攻击特征,解决了 Shield[10]不能防护基于未知漏洞的代码注入攻击的缺陷,同时,通过

引入进程状态自动机,解决了使用简单的网络协议时 Shield 的虚警概率过高的问题.本系统只有在进程异常时才会分析异常场景,免去了 TaintCheck[11]和 DIRA[12]实时跟踪进程数据、改写或者记录内存数据的更新日志的开销,同时也解决了 ARBOR[13]不能防护格式字符串滥用攻击的弱点.

文献[14]提出的系统同样是利用进程异常场景分析攻击载荷的某些特征,与本系统最接近.但是,本系统明确地区分各种不同的攻击类型,逻辑更清晰,从而易于通过编写插件扩充系统的句法分析能力,而且本系统进一步地通过状态自动机降低了虚警概率和响应时间.

本系统能够对代码注入攻击提供有效的防护和自动响应,目前,对于利用程序逻辑错误发起的攻击(例如 IIS Unicode 解码错误)、基于高层语言语法缺陷的注入攻击(例如 SQL 注入)、CGI(common gateway interface)滥用等无防护和响应能力.

目前,我们正在改进这一原型系统,未来的工作主要集中在:

(1) 原型系统功能的扩充.包括编写更多的 PSDL 协议自动机定义脚本,从而支持更多的网络协议以及对基于其他数据结构(例如 C++虚表等)攻击的句法分析研究和插件的编写.

(2) 异常恢复能力的研究.目前,本系统只能通过重启被保护进程来恢复正常的业务,这样会丢失正在进行中的会话,我们正在研究进程异常前现场的恢复能力.

References:

[1] Grossman D, Hicks M, Jim T, Morristee G. Cyclone: A type-safe dialect of C. C/C++ Users Journal, 2005,23(1):112−139.

[2] Necula G, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy code. In: John L, ed. Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. ACM, 2002. 128−139.

[3] Larus JR, Ball T, Das M, DeLine R. Righting software. IEEE Software, 2004,21(3):92−100.

[4] Chen H, Wagner D. MOPS: An infrastructure for examining security properties of software. In: Atluri V, ed. Proc. of the ACM Computer and Communications Security (CCS) 2002 Conf. ACM, 2002. 18−22.

[5] Dinakar D, Vikram A. Backwards compatible array bounds checking for C with very low overhead. In: Osterweil LJ, ed. Proc. of the 28th Int'l Conf. on Software Engineering (ICSE 2006). ACM, 2002. 162−171.

[6] Baratloo A, Singh N, Tsai R. Transparent run-time defense against stack smashing attacks. In: Ley M, ed. Proc. of the USENIX 2000 Annual Technical Conf. USENIX Association, 2000. 251−262.

[7] Provos N. Improving host security with system call policies. In: Ley M, ed. Proc. of the 12th USENIX Security Symp. USENIX Association, 2003. 257−272.

[8] Barrantes EG, Ackley DH, Forrest S, Palmer TS, Stefanovic D, Zovi DD. Randomized instruction set emulation to disrupt binary code injection attacks. In: Atluri V, ed. Proc. of the 10th ACM Conf. on Computer and Communications Security (CCS 2003). ACM, 2003. 281−289.

[9] The PaX Team. Documentation for the PaX project. http://pax.grsecurity.net/docs/index.html

[10] Wang HJ, Guo C, Simon DR, Zugenmaier A. Shield: Vulnerability-Driven network filters for preventing known vulnerability exploits. In: Raj Y, ed. Proc. of the 2004 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications. ACM, 2004.193−204.

[11] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In: Harder E, ed. Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS 2005). ISOC, 2005. 134−150.

[12] Smirnov A, Chiueh T. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In: Harder E, ed. Proc. of the 12th Annual Network and Distributed System Security Symp. (NDSS 2005). ISOC, 2005. 398−405.

[13] Liang Z, Sekar R. Automated, sub-second attack signature generation: A basis for building self-protecting servers. In: Ley M, ed. Proc. of the USENIX 2005 Annual Technical Conf. USENIX Association, 2005. 200−213.

[14] Xu J, Ning P, Kil C, Zhai Y, Bookholt C. Automatic diagnosis and response to memory corruption vulnerabilities. In: Atluri V, ed. Proc. of the 12th ACM Computer and Communications Security (CCS). ACM, 2005. 223−234.

[15] Kim HA, Karp V. Autograph: Toward automated, distributed worm signature detection. In: Ley M, ed. Proc. of the 13th USENIX Security Symp. USENIX Association, 2004. 271−286.

[16] Kreibich C, Crowcroft J. Honeycomb—Creating intrusion detection signatures using honeypots. In: Balakrishnan H, ed. Proc. of the 2nd Workshop on Hot Topics in Networks (HotNets-II). ACM, 2003. 51−56.

[17] Singh S, Estan C, Varghese G, Savage S. The EarlyBird system for real-time detection of unknown worms. Technical Report, CS2003-0761, San Diego: University of California, 2003.

[18] Cowan C. Software security for open-source systems. IEEE Security and Privacy, 2003,1(1):38−45.

[19] Younan Y, Joosen W, Piessens F. A methodology for designing countermeasures against current and future code injection attacks. In: John L, ed. Proc. of the 3rd IEEE Int'l Workshop on Information Assurance (IWIA 2005). IEEE Press, 2005. 3−20.

附录 1. FTP 协议认证部分的 PSDL 定义

```
###Version&PolicyID
VERSION(2,0,1); ID(0x20046);
###Application identification
FTP(TCP,(20,21,[4000~6000])); #4000~6000 is passive port range defined by user
###Event identification
#Event ID location
EVENT_ID_LOCATION=(0b,(1,"/–/")w); # offset 0 bytes, 1 words, split by blank or "–"
#Event Type&Direction
EVENT E_FTPSendUserName=(0,/USER/i,INCOMING); #case insensitive
EVENT E_FTPSendUserNameAck_Err=(1,/1\d\d/,OUTGOING);
EVENT E_FTPSendUserNameAck_Ok=(2,/2\d\d/,OUTGOING);
EVENT E_FTPSendUserNameAck_WaitPassword=(3,/3\d\d/,OUTGOING);
EVENT E_FTPSendUserNameAck_Fail=(4,/4\d\d|5\d\d/,OUTGOING);
EVENT E_FTPSendPassword=(5,/PASS/i,INCOMING); #case insensitive
EVENT E_FTPSendPasswordAck_Err=(6,/1\d\d/,OUTGOING);
EVENT E_FTPSendPasswordAck_Ok=(7,/2\d\d/,OUTGOING);
EVENT E_FTPSendPasswordAck_Fail=(8,/4\d\d|5\d\d/,OUTGOING);
###Session identification
#FTP is tcp-based, so follow tcp connections table, udp-based follow ports table
SESSION(FOLLOW_CONNECTION_TABLE);
###NPA specification
#NPA States
STATE INITIAL_STATE=(0,S_FTPWaitForUsername);
STATE FINAL_STATE={(1,S_FTPSuccess),(2,S_FTPError),(3,S_FTPFail)};
STATE STATE4=(4,S_FTPCheckUsername);
STATE STATE5=(5,S_FTPWaitForPassword);
STATE STATE6=(6,S_FTPCheckPassword);
#NPA Transport Functions
# state, events, jump-to-state
STATE_MACHINE={(0,{0},4),(0,{[1~8]},2),(4,{2,3},5),(4,{4},3),(4,{0,1,[5~8]},2),
                (5,{5},6),(5,{[0~4],[6~8]},2),(6,{7},1),(6,{[0~5],6},2),(6,{8},3)};
```

附录 2. 引理 1 的证明

引理 1. G 的产生语言 $L(G)$不是正则语言,G 不是正则文法.

证明:假设 $L(G)$是正则语言,不妨设 M 为满足泵引理的仅依赖于 $L(G)$的正整数.

取 $z = f_{main}^{call}(f_0^{call})^M(f_0^{ret})^M f_{main}^{ret}$,显然 $z\in L(G)$.

按照 RL 泵引理所述,必然存在 u,v,w 满足: $z = f_{main}^{call}uvwf_{main}^{ret}$,$|uv|\le|M|,|v|\ge 1$.

$\forall i\ge 0$, $f_{main}^{call}uv^iwf_{main}^{ret}\in L(G)$ 且 M 不大于接受 $L(G)$的最小确定有穷自动机的状态数.

由于$|uv|\le|M|,|v|\ge 1$,故 v 只可能是由 f_0^{call} 组成的非空串.不妨设: $v=(f_0^{call})^k, k\ge 1$.

此时有: $u=(f_0^{call})^{M-k-j}, w=(f_0^{call})^j(f_0^{ret})^M$.

从而有: $uv^iw=(f_0^{call})^{M-l-j}((f_0^{call})^k)^i(f_0^{call})^j(f_0^{ret})^M=(f_0^{call})^{M+(i-1)k}(f_0^{ret})^M$.

当 i=2 时有: $uv^2w=(f_0^{call})^{M+(2-1)k}(f_0^{ret})^M=(f_0^{call})^{M+k}(f_0^{ret})^M$.

由于 $k\ge 1$,所以 $M+k>M$.

即 $f_{main}^{call}(f_0^{call})^{M+k}(f_0^{ret})^M f_{main}^{ret}\notin L(G)$,与泵引理矛盾.

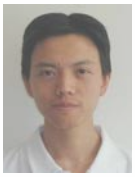
所以,$L(G)$不是正则语言,G 不是正则文法. □

李闻(1980－),男,青海西宁人,博士生,主要研究领域为安全信息系统,入侵检测系统.

冯萍慧(1979－),女,博士生,主要研究领域为脆弱性分析.

戴英侠(1942－),女,教授,主要研究领域为信息安全.

鲍旭华(1977－),男,博士生,主要研究领域为入侵检测,报警关联.



连一峰(1974－),男,博士,副研究员,主要研究领域为网络与系统安全.