第34卷 第13期 计 算 机 工 程 2008年7月
Vol.34 No.13 Computer Engineering July 2008
·博士论文· 文章编号：1000—3428(2008)13—0007—03 文献标识码：A 中图分类号：TP309

基于动态模拟的多态 Shellcode 检测系统

王兰佳[1]，段海新[2]，李 星[1]
(1. 清华大学电子工程系，北京 100084；2. 清华大学信息网络工程研究中心，北京 100084)

摘 要：通过分析多态 Shellcode 的行为特征，提出基于动态模拟的判决准则。以此准则为核心，针对现有方法的性能和应用性较差的问题，设计并实现了一个基于动态模拟的多态 Shellcode 检测系统，其模块采用多种优化技术以提高系统性能。使用 3.3 GB 实际网络数据和 11 000 个多态 Shellcode 样本对原型系统进行实验，其虚警和漏警率均为 0，提高了系统的吞吐量。
关键词：多态 Shellcode；动态模拟；入侵检测

Polymorphic Shellcode Detection System Based on Dynamic Emulation

WANG Lan-jia[1], DUAN Hai-xin[2], LI Xing[1]
(1. Dept. of Electronic Engineering, Tsinghua University, Beijing 100084; 2. Network Research Center, Tsinghua University, Beijing 100084)

【**Abstract**】Based on the analysis of the characteristics of polymorphic Shellcode's behavior, an dynamic emulation based detection criterion is proposed. Using the criterion, this paper designs and implements a dynamic emulation based polymorphic Shellcode detection system, which is highly optimized in each module. With 3.3 GB real network data and 11 000 polymorphic Shellcode samples, the experiment on prototype presents zero false positive and false negative, and it improves the throughput of system.
【**Key words**】polymorphic Shellcode; dynamic emulation; intrusion detection

1 概述

利用漏洞进行远程主机入侵是网络攻击的重要手段，近年来的许多蠕虫都是基于该手段进行传播的。由于Shellcode是漏洞攻击的重要组成部分，且对不同漏洞的攻击可以复用同样的Shellcode，因此Shellcode检测成为漏洞攻击检测的有效途径。近年来多态与变形技术[1]被广泛应用于Shellcode，使得传统的基于模式(特征)匹配的入侵检测方法不能对其进行有效检测。

研究者们提出了多种基于静态分析[2]的检测方法。其主要思路是将网络数据反汇编为机器指令代码，静态地分析代码特点，判断此代码是否为Shellcode。但是静态分析方法只能检测初期简单的多态Shellcode，对目前已采用了自修改等静态分析对抗技术的多态Shellcode无能为力。

最近出现的基于动态模拟的方法[3-4]克服了静态分析方法的缺陷，是检测多态Shellcode的最有效途径。但现有的方法在性能上十分有限，也缺少实际的系统实现与应用。本文提出一个基于动态模拟的检测判决准则，实现了一个多态Shellcode检测系统，大大提高了系统性能。

2 动态模拟检测的基本原理

2.1 多态 Shellcode 的动态行为分析

目前的多态 Shellcode 一般具有 2 个典型特征：加密与多态。加密是指原始 Shellcode 经加密后作为新的 Shellcode 的一部分——加密负载，而新 Shellcode 的另一部分则是解密头。当此新的 Shellcode 在被入侵系统中执行时，首先执行解密头，将负载解密还原为原始 Shellcode，即完成解密过程；然后执行原始 Shellcode。多态是指由同一个原始 Shellcode 可以生成多个不同的新 Shellcode 样本。通过采用不同的密钥或加密算法，这些样本的加密负载部分互不相同；而解密头则进一步利用寄存器替换、插入垃圾指令等技术，做到各样本间互不相同，从而实现 Shellcode 完全的多态。目前已存在多种自动生成多态 Shellcode 的工具——多态引擎。它以一个原始 Shellcode 作为输入，自动、快速地输出大量具有上述特征的多态样本。图 1 给出了一个简单样本的解密头部分。尽管目前的多态 Shellcode 可以做到不存在公共数据段(无法提取传统的字符串特征)，但是作为一段代码，其动态的执行行为存在一定的特征。

```
00:   jmp 12
02:   pop edx
03:   dec edx
04:   xor ecx, ecx
06:   mov cx, 017D
0A:   xor byte ptr [edx+ecx], 99
0E:   loop 0A
10:   jmp 17
12:   call 02
17:
 ...    <encrypted payload>
```

图 1 简单的多态 Shellcode

先进行解密操作，解密头必须取得负载所在的内存地址。在不同的系统中，此地址往往是不同的，因此，无法预先将其硬编码入 Shellcode 中，只能动态获取。目前存在 2 种方法

基金项目：国家“973”计划基金资助项目(2003CB314805)
作者简介：王兰佳(1981－)，女，博士研究生，主研方向：计算机网络安全；段海新，副教授、博士；李 星，教授、博士生导师
收稿日期：2007-08-28 **E-mail**：wanglanjia@ccert.edu.cn

取得这一地址：利用 call 指令(如图 1 的行 12)或 fstenv 指令。call 指令将其下一指令的地址压入栈中，随后解密头即可读取此地址(如图 1 的行 02)，并由此计算得到负载的地址。

解密头通常通过一个解密循环(如图 1 的行 06 和 0A)对负载进行解密。在此循环中，将对负载进行反复的读取和写入操作。解密操作结束后，程序指针跳入已解密的原始 Shellcode(如图 1 的行 17)。

2.2　网络数据的模拟执行

一段网络数据流事实上为一段字节序列。利用反汇编技术,可以在任意位置按 X86 指令的编码规则得到相应的指令。因此，以任意位置作为起始点，可将数据流看作二进制可执行代码，给定一个初始的执行上下文(包括 CPU 的各寄存器组、内存等)，即可执行此代码。本文系统的基础为 CPU 模拟器，它用高级语言实现，能够模拟 CPU 对二进制代码进行执行，根据指令动态改变模拟的 CPU 及内存状态。

在进行远程漏洞攻击时，Shellcode 被嵌入到网络攻击数据流中。在被攻击主机中，当漏洞被成功利用时，程序指针(EIP)会指向 Shellcode 从而执行 Shellcode，并按照 2.1 节的解密过程，首先执行解密头部分。此解密过程一般具有与当前执行上下文无关的特点，即它不依赖初始的寄存器值和内存中的内容，以及其他初始的系统或主机相关信息。因此，任意给定模拟器的 CPU 及内存状态，解密头在模拟器中的执行与在真实系统中的执行是完全一致的(即能够完全再现上述解密过程)，这就为基于动态模拟进行多态 Shellcode 检测提供了基础。

如果今后有更先进的技术使得 Shellcode 依赖于模拟器无法获知的系统初始状态或信息，那么本文提出的基于模拟的方法就会失效。另外，若 Shellcode 以某种编码形式出现于网络数据中，漏洞系统会在执行前对其进行还原(如 Unicode 编码的 Shellcode)，本文方法也不能检测。但这类特殊的 Shellcode 的应用环境非常有限。由于其目前形式单一，检测系统也能够较容易地增加特殊模块对其进行解码后再检测。

2.3　多态 Shellcode 的判决准则

如上文所述，以网络数据流的任意位置作为起点，可以得到一段可执行代码，并可在模拟器中执行。由于事先不知道 Shellcode 在网络数据中的位置，因此需要对这些以不同的位置作为起始的代码分别进行判决，并判断其为多态 Shellcode 或由网络数据形成的随机代码。这种多次的执行称为“模拟循环”。

根据前文所分析的解密头的动态行为特征，在模拟循环的每次执行中，若顺序出现以下执行结果，则判决为 Shellcode：

(1)遇到 call 或 fstenv 指令，使得当前代码所在地址(称为 *PC* 值)写入内存中；

(2)读出(1)所写的 *PC* 值；

(3)对代码自身所在内存区域的读、写操作次数大于阈值 *T*。

在此判决准则中，步骤(1)、步骤(2)对应于获取 *PC* 值的过程，而步骤(3)对应于解密循环。*T* 应当小于原始 Shellcode 的可能最小长度。由于代码在模拟环境中动态的执行，上述结果可以直接取得。

若执行过程中遇到以下情况之一，则执行停止，当前代码判定为正常：

(1)遇到异常指令。异常指令的情况包括：当前位置的数据无法反汇编为指令、特权级指令以及其他不可能出现于 Shellcode 解密头中的指令(如 in 指令，用于读取 I/O 端口)。

(2)寄存器 EIP 指向异常地址。异常地址指非当前代码所在的内存区域，且非代码之前所写的内存区域。

这样，通过上述准则可判定执行的代码段为正常数据或多态 Shellcode 的解密头。

3　检测系统体系结构

以上述基于 CPU 模拟的判决方法为核心，本文设计了如图 2 所示的多态 Shellcode 检测系统体系结构。

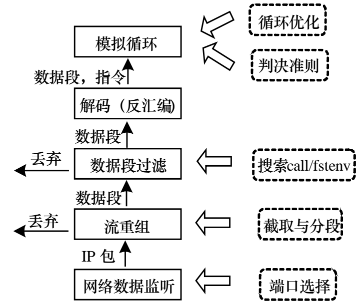

图 2　检测系统体系结构

此结构显示了网络数据在系统中的处理流程，整个系统包括 5 个主要的模块。在本系统中，对每个模块的设计除了实现基本的功能之外，还采用了不同的优化策略或方法，以增强系统性能。

3.1 网络数据监听

如同一般的基于网络的入侵检测系统，本系统最底层的模块为网络数据的监听模块。一般可利用交换机的镜像端口监听到被监测子网的全部出、入流量。监听到的 IP 数据包输出给下一模块，以进行后续处理。

“端口选择”是这一模块采用的一个优化策略。根据实际被监测网络的特点，例如流量大小、检测系统的硬件性能，可以只选择一些特定端口的流量，如易被攻击的端口 135、端口 139、端口 445 等，以及提供常见应用的端口 80 等。

3.2　流重组

由于攻击者可能将Shellcode分散于多个IP包中，因此流重组(主要针对TCP流)是本系统中必要的模块。对于TCP协议，流定义为四元组 $<IP_{src}, Port_{src}, IP_{dst}, Port_{dst}>$。在一些应用中，可能出现非常大的流，如文件传输。对于这些流，考虑到在其中间嵌入Shellcode的概率较小，可以只截取其开始(及结束)的一部分，如 64 KB大小，这样能极大地减少后续模块的输入数据量。

在这一模块中，一项重要的优化技术是流的分割。输入到模拟循环中的数据长度对系统性能有很大影响，因此，在流重组过程中，一个流以一定规则被分割为多个数据段，这些数据段分别作为后续模块的输入。分割的规则包括 2 条。由于 Shellcode 通常作为字符串输入应用程序，不能包含字节 0。因此，字节 0 作为分段标志，一旦流中出现字节 0，表示一个数据段结束。若由字节 0 分割的数据段仍较长，可进一步分割为长度为 *l* 的连续数据段，相邻两数据段有一部分重叠，重叠部分长度为 *n*，且 *n* 大于多态 Shellcode 可能的最大长度。重叠部分的作用是保证任一 Shellcode 必然完整地包含在至少一个数据段中，即避免由于 Shellcode 被分割而检测不到。

3.3　数据段过滤

前文已提到，多态 Shellcode 通常利用 call 或 fstenv 指令。

在X86指令系统中，call的操作码为E8或FF，fstenv为D9。因此，在对数据段的整体反汇编之前，首先搜索这3个字节是否存在，若存在，则在相应位置进行反汇编，检查是否为call或fstenv。如果数据段中没有搜索到call/fstenv指令，说明其必然不可能包含多态Shellcode，将其丢弃，否则，输出到下一模块进行后续处理。

3.4 解码(反汇编)

在模拟执行中，某一位置的指令可能被执行多次，因此，在本系统中，反汇编作为一单独模块。数据段每一位置反汇编得到的指令一定格式存储在一段缓存中，每次执行某指令时去缓存中读取，而不必反复执行反汇编操作。对于自修改的代码，系统记录被修改的位置，再次执行到此时，将重新对被修改的数据进行反汇编，这样就保证了正确的CPU模拟。

3.5 模拟循环

模拟循环是系统的核心部分。以数据段的每一位置作为起始，都可以得到一段不同的代码，在模拟循环中对每一位置执行一次。这样，若数据段长度为L B，那么循环次数为L。而本系统对此模块同样采取了优化方法，以减少实际的循环次数。

多态Shellcode的特点与初始执行环境无关，其自身对所需寄存器进行初始化。因此，许多类型的指令，如inc eax，是不可能作为多态Shellcode的第1条指令的(垃圾指令不必算在Shellcode内)，这样，遇到这些指令，可以直接跳过，而不必以其作为起始进行模拟执行。

对于已出现在某次执行中的指令，不必再执行以其作为起始的代码。原因同样是由于多态Shellcode与初始环境的无关性。如果以此指令作为起始的代码是Shellcode，那么在上述执行中必然会检测到。

在模拟循环的每次执行中，根据2.3节中的判决准则对当前代码进行判定。若输出肯定判决，则说明包含Shellcode，为恶意流；否则，模拟直到循环结束，判定为正常流。

对长为l的数据段，模拟循环的计算复杂度为$O(l^2)$。若整个流的长度为L，分割时的重叠部分长度为n，则总的对整个流的计算复杂度近似为$O(Ll^2/(l-n))$。可推导得，当$l=2n$时，此复杂度最小，为$O(4nL)$。由于n为常数，因此它是线性复杂度。而如果在流重组模块中不采用分割的优化方法，模拟循环的复杂度为$O(L^2)$。由此可知，对流的分割可以有效提高系统性能。

4 系统实现与实验结果

4.1 系统实现

目前已实现一个基于上述体系结构的原型系统。网络数据采集基于libpcap。流重组利用工具libnids。反汇编模块以工具libdasm为基础，对其进行了一定修改，加入了与系统其他功能或模块相关的代码。原型系统以C语言实现。相关参数设置为T=15, n=2 048, l=4 096。

4.2 实验结果

为对原型系统的性能进行评估，采集了清华大学某一子网出口一周的流量数据，如表1所示。实验数据包括常用端口80和端口443，易被攻击端口135、端口139和端口445。实验中对以这些端口为目的端口的流(如表1中的C-to-S)进行了测试。

表1 实验数据集

端口	字节数	流数量/k	C-to-S 单向字节数	
			完整流	前64 KB流
80	28 GB	710.0	3.3 GB	1.1 GB
443	470 MB	40.0	85.0 MB	51.0 MB
135	10 MB	3.1	3.1 MB	…
139	300 MB	13.0	17.0 MB	…
445	40 MB	4.5	8.2 MB	…

对各个数据集，实验检测到的Shellcode数量和对80端口及443端口数据的吞吐量如表2所示。通过对135, 139和445端口数据检测到的Shellcode的手工分析，确认虚警为0。这些Shellcode中的一部分是多态的，解密头中采用了不同的密钥；另一部分事实上并非多态，而只是进行了加密。显然，本系统对此类Shellcode同样有效。实验同时说明多态或加密的Shellcode在网络攻击中被广泛使用。

表2 检测结果与吞吐量

端口	流类型	检测#	吞吐量/(Mb•s^{-1})
80	完整流	0	44
80	64 KB流	0	132
443	完整流	0	42
443	64 KB流	0	40
135	完整流	4 434	—
139	完整流	658	—
445	完整流	2 269	—

同时，在现有相关工作中[3-4]，对80端口数据的处理吞吐量在最优条件下约为15 Mb/s，而本系统则高得多，约为132 Mb/s。实验测得，在模块“数据段过滤”中丢弃的数据段占总数的93%，而“模拟循环”的平均循环次数仅为13%×l，因此，各模块采用的优化方法对提高系统性能具有重要作用。

为评估系统的检测率，实验收集了互联网上现有的11个多态引擎(Clet, ADMmuate, Jempiscodes, TAPioN以及工具Metasploit Framework 的7个多态模块)，使用每一个引擎生成1 000个多态Shellcode样本。实验利用这11 000个样本对系统进行测试，检测率为100%。

5 结束语

本文提出的检测系统对目前绝大多数多态Shellcode具有很好的检测效果。由于攻击技术总是不断更新发展以对抗现有的检测系统，因此未来工作的目标是在目前判决准则的基础上建立普适性更强的检测模型，进一步增强系统的鲁棒性和可扩展性。同时，结合协议分析等方法，通过加强数据的预处理过程来实现更高的运行性能。

参考文献

[1] Szor P, Ferrie P. Hunting for Metamorphic[C]//Proc. of the 11th Virus Bulletin Conference. Prague, Czech Republic: [s. n.], 2001.

[2] Chinchani R, Berg E. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows[C]//Proc. of International Symposium on Recent Advances in Intrusion Detection. Washington D. C., USA: [s. n.], 2005.

[3] Polychronakis M, Anagnostakis K G, Markatos E P. Network-level Polymorphic Shellcode Detection Using Emulation[C]//Proc. of International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment. Berlin, Germany: [s. n.], 2006.

[4] Zhang Qinghua, Reeves D S, Ning P. Analyzing Network Traffic to Detect Self-decrypting Exploit Code[C]//Proc. of ACM Symposium on Information, Computer and Communications Security. Singapore: [s. n.], 2007.