中国科学 E 辑：信息科学 2008 年 第 38 卷 第 10 期：1760 ~ 1773
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

基于动态模拟的多态 Shellcode 建模与网络层检测

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
* E-mail: wanglanjia@ccert.edu.cn

收稿日期: 2008-06-03; 接受日期: 2008-07-30

摘要 实践证明，通过模拟的方法对多态 Shellcode 进行检测是行之有效的. 然而，现有的基于模拟的方法在性能和抗攻击性上仍有不足. 提出了一种经过改进的基于模拟的检测方法，该方法包括一个基于自动机的多态 Shellcode 动态行为模型，以及基于该模型设计的具有高准确性的检测算法. 该算法同时还采用了多种优化技术，能够显著提高检测系统的运行性能，以及应对规避检测型 Shellcode 的能力. 通过一个原型系统实现了上述方法，且使用了大量真实网络数据，由多态引擎生成的多态 Shellcode 样本，以及手工构造的规避型 Shellcode，实验验证了该算法的有效性和先进性能.

关键词
多态 Shellcode
入侵检测
CPU 模拟

代码注入攻击是一种非常重要的恶意网络攻击活动. 通过利用远程主机运行的服务的某种漏洞，攻击者能够将攻击代码 Shellcode 注入其系统并获取系统的控制权. 许多著名的蠕虫(如 CodeRed, Blaster)，以及互联网上一些比较新的安全威胁(如 Botnet)，都利用了代码注入攻击. 由于 Shellcode 是代码注入攻击的关键组成部分，检测 Shellcode 应该可以成为检测代码注入攻击的有效途径.

特征匹配是基于网络的入侵检测的一种传统方法. 特征通常是一组字符串或正则表达式. 为了对抗基于特征的检测方法，病毒领域中的多态和变形技术[1]被用来构造多态Shellcode. 多态Shellcode包括一个解密子程序(也称为解码器)和一段加密负载. 图 1 显示了一个简单的多态 Shellcode. 在执行中，解码器将负载解密还原为原始的Shellcode. 使用不同的密钥或加密算法，原始Shellcode可以被转换为大量的多态样本，其解码器和负载都可以具有不同的形式. 这样，只要这些样本的多态程度足够高，就很难生成特征.

在基于网络的多态Shellcode检测方面已有很多研究. 很重要的一类方法是基于二进制代

码的静态分析 [2~4]. 这些方法多数使用反汇编技术将网络数据转换为二进制代码, 然后分析代码的程序流程以检测Shellcode的特性或模式. 然而, 如相关文献中所提到的 [5], 当前的多态Shellcode已应用了一些抗静态分析技术(如自修改技术), 能够躲避这些方法的检测.

为抵御抗静态分析Shellcode, Polychronakis等 [5]首先提出了基于模拟的方法, 将网络数据当作二进制代码来模拟执行, 以期发现多态Shellcode的执行行为. 他们的方法能够检测出目前最先进的多态Shellcode, 但仍然存在一些局限. 首先是开销. 由于Shellcode在网络流中的起始位置未知, 此方法使用一个模拟循环尝试从每一位置开始执行. 这是一个非常花费时间的过程, 尤其当流较大时更是如此. 其次是大多基于模拟的方法的一个本质性问题. 当运行网络数据时, 模拟器可能会遇到长循环(无限或有限但在短时间内不能终止的循环), 它会使模拟执行的性能降低或挂起. 他们的方法使用一个执行门限, 即执行指令的个数, 来终止模拟执行. 然而, 这使得攻击者可以在Shellcode中构造一个长循环来躲避检测. 这样, 在Shellcode被检测之前, 执行就已终止, 造成一个漏报. 另外一项研究 [6]提出了一种包含静态分析和动态模拟的混合型方法, 但同样存在类似的问题.

由于现有研究中这些局限的存在, 本文研究了多态 Shellcode 的特点, 提出了一种改进的基于模拟的多态 Shellcode 检测方法. 它包括一个多态 Shellcode 的动态行为抽象模型和一个基于此模型的检测算法. 与现有工作相比, 本文算法极大地优化了运行性能和针对规避检测型 Shellcode 的抗攻击能力. 本文的主要贡献包括:

```
00:  EB10          jmp12                          P5
02:  5A            pop edx                        P2
03:  4A            dec edx                        P5
04:  33C9          xor ecx, ecx                   P5
06:  66B97D01      mov cx, 017D                   P5
0A:  80340A99      xor byte ptr [edx+ecx], 99     P3 P4
0E:  E2FA          loop 0A                        P5
10:  EB05          jmp 17                         P5
12:  E8EBFFFFFF    call 02                        P1
17:
...      <encrypted payload>
```

图 1　一个简单的多态 Shellcode

1) 提出一个基于自动机的模型, 能够描述多态 Shellcode 的动态执行行为. 基于此模型, 形式化地定义了基于模拟的检测问题, 并推导出一个用于检测算法的检测准则.

2) 算法中, 对检测系统的性能进行了 3 个阶段的优化.

流分割: 通过某些规则将一个流分割为多个短的数据段.

数据段过滤: 在模拟前过滤掉某些不可能是攻击的数据段.

模拟循环优化: 减少了模拟循环次数.

3) 算法的抗攻击性能同样得到优化. 我们提出了一个新的策略, 它至少能部分的应对规避检测型 Shellcode.

4) 使用真实的网络数据和多态 Shellcode 样本, 我们通过实验对算法的准确性、性能和抗攻击能力进行了验证.

本文全篇结构如下. 第 1 节总结了相关工作; 第 2 节给出了基于自动机的多态 Shellcode 模型; 第 3 节中详细讨论了检测算法; 实验在第 4 节中进行; 第 5 节总结全文.

1　相关工作

上面已讨论了部分基于静态分析 [2~4]和动态模拟 [5,6]的方法. 一些方法 [2,7,8]集中于检测通

常位于Shellcode前的NOP-sled. 但很多攻击，尤其是对Windows的攻击，是很可能没有NOP-sled的. 另一些方法采用数据挖掘来区分Shellcode和正常网络数据 [2,9].

一些基于特征的方法 [10~12]针对多态蠕虫的检测. 它们的基本假定是多态蠕虫中仍然包含固定的字符串，能够用做特征. 此假设在蠕虫使用了非常先进的多态技术的情形下是可能不成立的.

与本文工作相关的还有利用静态分析的多态恶意代码检测 [13,14]. SAFE[14]建立了基于静态模式的恶意代码的自动机模型，而本文的基于自动机的模型则是描述了多态Shellcode的动态行为.

2 多态 Shellcode 模型

检测器可以利用多态 Shellcode 的某些行为特征来对其进行检测. 本章将对多态 Shellcode 的行为进行更为形式化和一般性的描述，为基于模拟的检测方法提供理论基础.

2.1 多态 Shellcode 综述

现有研究已对多态 Shellcode 的特性进行了充分的说明. 在这一节中我们将对与本文方法相关的一些结论做以回顾.

为了对加密负载进行解码，多态Shellcode需要获取负载的虚拟内存地址. 一般利用GetPC code来实现这一功能. GetPC code必须包含指令call(如图 1 的 12 行)或fstenv[15].

解密子程序的第二部分为一解密循环(如图 1 的 0A 行). 循环的每一次运行中都会读取负载的若干字节，计算其原来的值并写回内存中(通常就写在读取的位置). 当解密结束时，程序转入解密后的原始 Shellcode 执行.

由于通常攻击者很难事先知道多态 Shellcode 在内存中的具体地址，多态 Shellcode 是与位置无关的. 类似的，它与 CPU 及内存的初始状态(即寄存器和内存的内容)也是无关的. 因此，我们能够脱离被攻击系统而利用模拟执行的方法对其进行检测.

2.2 指令序列变换

另 $S=\langle I_1,I_2,\ldots,I_J\rangle$ 为一段指令序列，即指令从 I_1 至 I_J 被依次执行. 任一指令 I_j，$j=1,2,\ldots,J$，可能执行内存读或写操作. 我们定义 $\mathrm{rw}(I_j)\in\{\mathrm{NULL,READ,WRITE,\ READ|WRITE}\}$ 为 I_j 的读写属性. 在此定义中，$\mathrm{rw}(I_j)=\mathrm{NULL}$ 表明指令 I_j 不包含内存读或写操作(mov eax,ecx)，而 $\mathrm{rw}(I_j)=\mathrm{READ|WRITE}$ 表明 I_j 同时包含内存读和写操作(如 xor [ecx],99). 显然，属性 READ 和 WRITE 都与 3 个参数(addr, size, value)相关，其中 addr 为被访问的内存块的低地址, size 为内存块大小，而 value 为所读写的内容，与 size 相一致，可能为一个字节、字、双字或字节序列. 当一条指令在某上下文环境中运行时，这 3 个参数是唯一确定的.

另 $\mathrm{pc}(I_j)$ 为指令 I_j 所在的内存地址. 由于 $\mathrm{pc}(I_k)$，$1\leqslant k\leqslant K$，能够通过 $\mathrm{pc}(I_j)$ 的偏移得到，为简洁起见，我们简单地称 $\mathrm{pc}(I_j)$，$1\leqslant j\leqslant J$，为 PC. 根据内存访问操作的结果不同，我

们将所有指令分为 5 类: $C_1 \sim C_5$. 每一类可由一个抽象模式(abstract pattern)来描述, 形如 $P=\left(\mathrm{Op},\mathrm{Cond}\left(\mathrm{addr},\mathrm{size},\mathrm{val}\right)\right)$, 其中 $\mathrm{Op}\in\left\{\mathrm{NULL, READ, WRITE}\right\}$, $\mathrm{Cond}\left(\mathrm{addr, size, val}\right)$ 为与 3 个参数有关的逻辑表达式. 形式化的, $\Sigma=\left\{P_1,P_2,P_3,P_4,P_5\right\}$ 为模式的一个有限集, 其中

$$
\begin{aligned}
P_1 &= \left(\mathrm{WRITE, size}=4\,\&\,\mathrm{value}=\mathrm{pc}(I_j), 1\leqslant j\leqslant J\right),\\
P_2 &= \left(\mathrm{READ, size}=4\,\&\,\mathrm{value}=\mathrm{pc}(I_j), 1\leqslant j\leqslant J\right),\\
P_3 &= \left(\mathrm{READ, addr}=\mathrm{pc}(I_j), 1\leqslant j\leqslant J\right),\\
P_4 &= \left(\mathrm{WRITE, addr}=\mathrm{pc}(I_j), 1\leqslant j\leqslant J\right),\\
P_5 &= \overline{P_1\mid P_2\mid P_3\mid P_4}.
\end{aligned}
$$

对此定义更为详细的解释如下: 一指令若将 PC 写入内存, 则属于类 C_1, 并可由 P_1 描述; 若它从内存中读取了 PC 值, 则属于 C_2; 若它在 PC 表示的内存地址(即指令序列所在的内存区域)进行了读操作, 则属于 C_3; 若它在 PC 表示的内存地址进行了写操作, 则属于 C_4; 否则, 此指令属于类 C_5. 注意一条指令可能属于多个类. 例如, 若指令 xor [esi], ecx 属于 C_3, 那么它也必然属于 C_4.

因此, 我们可以将指令 I_j 映射至 Σ 中的一个或多个模式. 若将此映射记为 $\mathrm{Tr}(I_j)=P_{j_1}P_{j_2}\cdots P_{j_h}\cdots P_{j_H}$, $j_h<j_{h+1}$, 那么 $S=\left\langle I_1,I_2,\ldots,I_J\right\rangle$, 可以得到变换 $\mathrm{Tr}(S)=W$, 它将 S 变换为一个模式序列 $W=\left\langle p_1,p_2,\ldots,p_k,\ldots,p_K\right\rangle$, $p_k\in\Sigma$, 且有 $K\geqslant J$. 在后文中, 我们将重点利用此变换得到的模式序列来分析多态 Shellcode 的行为.

2.3 多态 Shellcode 的一般自动机模型

图 1 显示了一个简单的多态 Shellcode 的解密子程序, 其加密负载的长度为 0x17D 字节(由图 1 的 06 行). 图 1 中右边的一列表示每一条指令变换后的模式. 由解密子程序的执行过程, 可得到它的模式序列 $W=\left\langle P_5,P_1,P_2,P_5,P_5,P_5,P_3,P_4,P_5,P_3,P_4,P_5,\ldots,P_3,P_4,P_5\right\rangle$. 若将 W 作为输入符号串, 可以构造一个自动机来描述此多态 Shellcode 的行为. 图 2 显示的状态转移图代表了这一特殊自动机(specific automaton), $M=\left(Q,\Sigma,\delta,q_0,F\right)$, 其中:

1) $Q=\left\{q_0,q_1,\ldots,q_{2l+2}\right\}$, l=0x17D, 为状态集;
2) $\Sigma=\left\{P_1,P_2,P_3,P_4,P_5\right\}$ 为字母表;
3) $\delta: Q\times\Sigma\rightarrow Q$ 为状态转移函数;
4) q_0 为初始状态;
5) $F=\left\{q_{2l+2}\right\}$ 为终止状态集合.

当 GetPC code 执行 call 或 fstenv 时(对应于 P_1), 此自动机从初始态转移到状态 q_1; 当取得 PC 时, 从状态 q_1 转移到 q_2; 当 Shellcode 开始读负载时从 q_2 转移到 q_3; 当解密循环的第 1 次

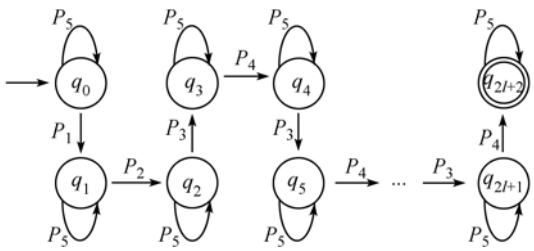

图 2　图 1 中 Shellcode 的特殊自动机

运行结束时，从 q_3 转移到 q_4；以此类推. 可知 $P_1 \sim P_4$ 代表了解密子程序执行时的关键步骤. 另外，易知图 2 中的自动机同样可以接受图 1 中 Shellcode 的一些变种.

尽管图 2 中的自动机能够精确描述一类特定的多态 Shellcode 的行为，对于检测问题来说需要一个更为一般性的模型，可适用于更多类型的多态 Shellcode. 不同多态 Shellcode 之间的区别包括解密循环的迭代次数，以及每次迭代中内存访问操作的次数. 因此，我们将特殊自动机扩展为一般多态 Shellcode 自动机(GPSA)，$M=(Q,\Sigma,\delta,q_0,F)$，如图 3 所示，其中:

1) $Q=\{q_0,q_1,q_2,q_3,q_4\}$ 为状态集;
2) $\Sigma=\{P_1,P_2,P_3,P_4,P_5\}$ 为字母表;
3) $\delta: Q\times\Sigma\to Q$ 为状态转移函数;
4) q_0 为初始状态;
5) $F=\{q_4\}$ 为终止状态集合.

对比图 2 的特殊自动机, GPSA 在状态 q_3 之前的部分与之相同，代表 GetPC code. 在 GPSA 中，状态转移 $\delta(q_3,P_3)=q_3$ 的意思是解码器在写操作之前可能对 Shellcode 所在内存区域进行多次读操作，例如，读取一个字节作为解密密钥. 另外，状态转移 $\delta(q_4,P_3)=q_4$ 和 $\delta(q_4,P_4)=q_4$ 说明解密循环可迭代任意多次. 当然，在多数情形下，由于加密负载通常至少有几十字节，因而循环次数不会太少. 因此，对于现有多态 Shellcode 来说，GPSA 具有足够的一般性，能代表所有的典型多态 Shellcode，即 GPSA 能够接受它们所变换而成的模式序列.

2.4　改进的 GPSA 与问题定义

尽管 GPSA 能够接受大部分多态 Shellcode，但由于要将其应用于检测问题，我们希望它能具有更强的鲁棒性，以抵抗攻击者对其的攻击. 若攻击者将一些“GetPC-like”指令插入解码器中, GPSA 可能会失效. 为抵抗类似攻击，我们将 GPSA 扩展为改进的 GPSA(IGPSA)，如图 4 所示. 在 IGPSA 中，P_1 和 P_2 可以在多种状态下被输入，且不会触发状态转移. 这样，在用于本文的检测算法中时, GPSA 的应用范围和鲁棒性优于 GPSA.

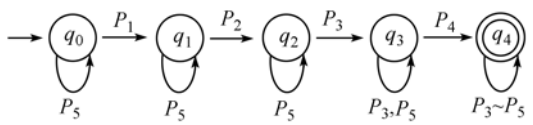

图 3　一般多态 Shellcode 自动机(GPSA)

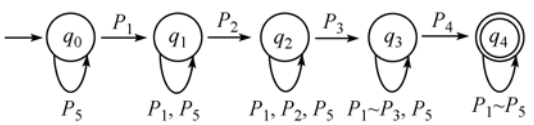

图 4　改进的 GPSA(IGPSA)

一方面几乎所有的多态 Shellcode 都能被 IGPSA 所接受，而另一方面的问题则是是否所有

被IGPSA接受的模式序列都对应于多态Shellcode. 由于IGPSA描述的是一类非常特殊的行为, 我们相信这样的行为出现在随机数据中的概率非常小. 第 4 节的实验也将验证这一点. 因此, 基于 IGPSA 模型, 多态 Shellcode 检测的问题可以形式化地定义如下:

问题定义 1 给定指令序列 $S=\langle I_1,I_2,\ldots,I_J\rangle$. 判定其是否包含多态 Shellcode 的问题等价于判定其变换后的模式序列 $\mathrm{Tr}(S)=W=\langle p_1,p_2,\ldots,p_k,\ldots,p_K\rangle$, $p_k\in\Sigma$ 是否被 IGPSA 所接受.

根据此定义, 检测算法的一项重要功能为生成模式序列并检验其是否为 IGPSA 所接受. 尽管随着多态技术的发展, 图 4 中的 IGPSA 可能无法表示一些未来的多态 Shellcode 的行为, 但本文提供了对检测问题进行分析的一种理论途径, 并且适用于基于模拟的方法. 我们相信由此途径, 能够很容易的在未来的应用中对 IGPSA 进行扩展, 能够应对多态 Shellcode 的发展.

3 改进的基于模拟的检测算法

基于模拟的检测系统的基本设计是非常直接的, 但是在检测准确性、运行性能及抗攻击(规避检测)方面都十分有限. 因此, 需要一个改进的算法, 使这些方面得到提高.

3.1 检测器的基本设计

我们可以总结基于模拟的检测系统的基本设计所必要的组成部分如下. 事实上, 现有方法 [5,6]也是基于此设计的.

流重组. 由于攻击者可以任意地将 Shellcode 分片并置于不同的 IP 包中, TCP 流重组是一项必要的工作. TCP 流可定义为四元组(IP_{src}, $Port_{src}$, IP_{dst}, $Port_{dst}$).

模拟循环. 重组得到的流即为模拟器的输入. 由于 Shellcode 真正的起始位置未知, 模拟为一循环, 以每一位置作为起始点进行重复的执行.

检测准则. 我们称模拟循环的每次迭代为一次执行. 在每次执行中, 需要有一个执行准则来决定所执行的 code 是否为 Shellcode. 此准则应该能够有效地平衡虚警与漏报, 通常基于多态 Shellcode 的某些特征而建立.

执行终止. 当遇到某预定义的条件时, 执行被终止, 例如, 遇到非法指令. 一个比较难处理的问题是当遇到一个循环长时间运行而不能结束时, 如何将其终止或应当采取什么措施. 本文中, 称此问题为长循环问题.

在本章的其他部分, 我们将给出一个改进的算法. 它采用基于 IGPSA 的检测准则(3.2 小节), 对检测器性能(3.3 小节)以及抗攻击能力(3.4 小节)进行了很大程度的优化.

3.2 基于 IGPSA 的检测准则

在模拟循环的一次执行中, 从流的某一位置起反汇编得到机器指令并执行, 因而一次执行会生成一个指令序列 S. 根据 2.4 小节的问题定义, 算法需要判断 S 变换得到的模式序列是否会被 IGPSA 所接受.

由于代码是由模拟的 CPU 和内存来执行, 我们可以直接获知每一个执行的指令是否进行了内存的读或写操作, 以及相应的参数(addr, size, value). 因此, 模式序列 W 随执行而生成, 同时 IGPSA 的状态根据 δ 进行转移.

为保证非常低的虚警率, 我们事实上采用了一个比 IGPSA 更强的检测准则. 首先, 对 P_2 增加一个额外的要求: 其参数 addr 必须等于某一之前的 P_1 的 addr, 即 P_2 对应的指令必须在 P_1 存储 PC 的内存地址进行读操作. 其次, 在 IGPSA 中, 一个 P_3 和 P_4 就可以使状态转移至 q_4, 但在我们的算法中, P_3 和 P_4 的个数不能太少. 我们定义阈值 TH_w, 其值小于原始 Shellcode 的可能最小长度. 只有当由 P_4 所写内存大于 TH_w 字节时, 算法才输出一个肯定判决, 即检测到一个多态 Shellcode. 在我们实现的原型系统中, 设置 $\mathrm{TH}_w = 15$.

执行终止. 当某些条件满足时, 算法输出否定判决(即当前执行的代码非Shellcode), 当前执行结束. 这些条件与文献 [5]中所用条件有部分相似, 如下所示:

1) 模式序列被 IGPSA 拒绝. 这主要是由于随机代码中恰好包含 $P_2 \sim P_4$ 造成的.

2) 遇到非法指令. 这里, 非法的意思是所遇到的字节无法反汇编为指令.

3) 遇到特权级指令或其他不可能用于解密子程序的指令(如, in eax,dx, 其从指定的 I/O 端口拷贝数据).

4) 寄存器 EIP 指向非法的内存地址. Shellcode 可能动态地在某些位置(如栈)写入数据, 跳转至这些位置. 因此, EIP 只能指向这些地址或 Shellcode 本身.

3.3 性能优化

3.1 小节的检测器基本设计的时间开销是非常大的. 在本文算法中, 我们分 3 个阶段对检测器的运行性能进行优化.

3.3.1 流分割

由于执行是从流的每一个位置开始的, 模拟总的时间复杂度约为 $O(L^2)$. 因此, 处理较大流(如大于100KB)的开销是非常大的. 为降低开销, 我们设计使用如下方法, 将一个流分割为多个数据段. 假定多态 Shellcode 可能的最大长度为 n, 我们将流(或步骤一生成的数据段)分割成长度为 l 相互重叠的连续数据段, 并且任意相邻数据段相互重叠的长度为 n. 换句话说, 令流中所有字节的序号为 $\langle 1,2,\ldots,L\rangle$, 那么第 1 个数据段则为 $\langle 1,2,\ldots,l\rangle$, 第 2 个为 $\langle l-n+1, l-n+2,\ldots,l,l+1,\ldots,2l-n\rangle$, 以此类推. 重叠部分保证了小于 n 的 Shellcode 能够完整的包含在至少一个数据段中.

通过推导可知, 模拟运行一个流的计算复杂度在 $l = 2n$ 时取最小值 $O(4nL)$ (证明见附录 A). 由于 n 事实上为常数, 复杂度与 L 成线性关系. 在我们的原型系统中, 设置 $n = 2048$.

3.3.2 数据段过滤

GetPC code是多态Shellcode的必要组成部分. GetPC code中所使用的call指令为相对调用(如图 1 中的call 02)或绝对间接调用(形如call reg)[15], 其操作码分别为E8 和FF. 指令fstenv的操作码则为D9.

因此, 在进行模拟之前, 我们可以先在数据段中搜索 E8, FF 及 D9. 若均未找到, 立即将此数据段标记为正常并删除; 否则, 在这 3 个字节所在位置进行反汇编, 以判断是否为合法的

call 或 fstenv 指令. 如果不存在 call 或 fstenv 指令, 则数据段标记为正常; 否则, 此数据段可能包含 GetPC code, 因而被输入到模拟器中进行进一步的分析.

我们可以从理论上估计任一数据段在这一阶段被过滤掉(标记为正常)的概率为 $P=\left(1-l/2^{40}-5/2^{12}\right)^{l}$ (详见附录 B). 若 l = 100, 则 P = 0.89; 若 l = 500, 则 P = 0.54. 因此, 数据段越短, 则被过滤的可能性越大. 由于短的流在网络流量中所占比例很大, 同时上节中的流分割技术也将长的流分割成短的数据段, 数据段过滤的策略能够很大程度地减少模拟器的输入, 提高整体运行性能.

3.3.3 模拟循环优化

过滤之后, 只有一部分数据段输入到模拟器中执行. 如前文所述, 在不做任何优化的情形下, 模拟循环的时间开销是 $O(l^2)$. 通过研究多态 Shellcode 的特性, 我们发现事实上可以减少模拟循环的循环次数, 优化模拟过程.

多态 Shellcode 的一个重要特征是它的执行与 CPU 和内存的初始状态无关. 换句话说, CPU 和内存的初始状态是完全随机的. 我们将所有指令分为两类: 随机指令和初始化指令. 随机指令保持 CPU 和内存完全随机的状态, 因此不可能作为多态 Shellcode 的第一条指令(注意, 若多态 Shellcode 的起始部分为一系列垃圾指令, 我们不将这些指令算在 Shellcode 内). 例如, 指令 add eax,20, xor eax,ebx, 等等. 初始化指令则会对完全随机的 CPU 和内存状态进行部分初始化. 例如, 指令 mov eax, 20, xor eax, eax, mov eax, ebx 等. 反汇编之后, 在位置 i 的指令属于哪一类就可以确定了. 在模拟循环中, 执行只从初始化指令开始. 通过检查 X86 指令集, 我们可以大致估计出执行中遇到的任一指令为随机指令的概率约为 70%. 因此, 这一策略能很大程度地减少模拟循环次数.

除了上述优化, 我们还设计了另外一个策略来进一步减少模拟循环次数. 此策略同样基于多态 Shellcode 与 CPU, 和内存的初始状态无关. 我们有如下定理:

定理 1　假定指令序列 $S=\langle I_1,I_2,\ldots,I_J\rangle$ 是一个数据段在模拟循环中一次执行的轨迹, 且在这次执行中未检测到 Shellcode. 若它不是漏报, 那么所有 S 中的指令都不可能是此数据段中任何多态 Shellcode 的第一条指令.

证明　由于多态 Shellcode 与 CPU 和内存的初始状态无关, 若 I_j 为一多态 Shellcode 的第一条指令, 那么后面接着的指令必然为 I_{j+1}, $I_{j+2},\ldots$, 因此 S 包含多态 Shellcode. 但是 S 非漏报. 因此, I_j 不可能为多态 Shellcode.

根据这一定理, 当 S 执行后, 其他 $J-1$ 次分别起始于 $I_2,I_3,\ldots,I_J$ 的执行可以省去. 这样, 模拟循环的次数减少了 $J-1$ 次. 在实验中我们将看到, 本节中的这两个优化策略平均能将模拟循环次数减少 0.15l.

3.4 抗攻击性优化

简单使用门限的方法 [5]不能够很好的处理长循环问题. 攻击者可以很容易地构造一个有

限的循环, 其执行指令的个数大于门限, 这样就绕过了检测器. 而若将此门限调高(如从 256 升至 2048), 并不能增加攻击的难度, 因为攻击者只需相应增加其所构造的循环的循环次数, 同时花稍多时间等待攻击成功; 但这种情形下检测器的性能将会降低很多.

本文设计了一个新的方案来解决此问题. 首先, 使用一个较小的阈值 TH_e (原型系统中 $TH_e = 20$ 来检测循环. 当一个后向控制转移指令(如 jmp, je 等)执行次数大于 TH_e 时, 则认为一个循环被检测到. 然后我们终止此循环, 从循环后的指令继续执行. 若此循环是无限的, 它必然不可能属于任何 Shellcode. 然而, 若此为一有限循环且属于一 Shellcode, 那么对它错误的终止就可能扰乱 Shellcode 的正确模拟执行, 导致 Shellcode 不能被检测到. 为避免漏报发生, 需要对此循环进行更多必要的分析.

攻击者所构造的循环可能会定义一些寄存器或内存值, 在多态 Shellcode 的后续部分使用. 因此, 若循环被错误地终止, 解密子程序的某些关键部分可能会错误地运行, 例如: (ⅰ) 获取 PC 时读取了错误的内存地址; (ⅱ) 或者在对加密负载进行解码时读取了错误的内存地址. 这两种错误会直接影响模式 P_2, P_3 和 P_4 的生成, 从而使本文基于 IGPSA 的检测准则失效. 除此之外, 错误终止的循环还可能从其他方面影响 Shellcode, 例如: (ⅲ) 造成错误的解密循环次数; (ⅳ) 错误的程序流程, 绕过 Shellcode 的某些部分.

为发现这些情形, 当检测到一循环时, 我们先判断其是否为一明显的无限循环. 如果是, 则当前执行结束. 否则, 我们记录循环中定义的寄存器, 并在后续执行中为之维护一个 define-use 链. 此 define-use 链用于在执行中记录受循环影响的所有寄存器. 当遇到内存读操作时, 若寻址寄存器受循环影响, 且当前状态非 q_0, 上述情形(ⅰ)和(ⅱ)则有可能发生. 此时我们将当前数据段标记为可疑. 为解决情形(ⅳ), define-use 链同样也要记录受循环影响的 EFLAGS 寄存器. 若发现程序流程可能被错误终止的循环所改变, 数据段标记为可疑. 对于情形(ⅲ), 由于只是解密循环次数错误, 一旦解密循环开始, 我们就将当前数据段标记为可疑.

当前数据段的模拟循环结束后, 若没有检测到 Shellcode, 且数据段被标记为可疑, 则将其加入一个可疑队列. 另一个线程对此队列中的数据段重新进行处理, 以一个很高的阈值对错误结束的循环进行执行. 利用此阈值, 循环需执行相当长的时间, 超过攻击者可能忍受的范围. 因此, 只要数据段加入可疑队列的频率足够低, 包含攻击者所构造循环的 Shellcode 就能够被检测出来. 在实验中将看到, 事实上此频率是非常低的.

图 5 给出了算法的整体结构, 其中包括了检测准则及所有优化策略.

4 实验验证

在本章中, 我们使用由多态引擎生成的多态 Shellcode 样本来检验漏报情形, 使用真实的网络流量数据来检验虚警、运行性能以及优化策略的作用. 我们还人为构造了一个具有长循环的 Shellcode 来验证本文算法的抗攻击性能.

4.1 实现

对本文提出的基于模拟的方法, 已实现了一个原型系统. 流重组由 libnids 实现. 指令的反

汇编主要由 libdasm 完成, 根据本文方法的需要进行了相应的一些修改. 对模拟器, 我们实现了除 FPU/MMX/SSE 外的所有常用指令. 这几类指令对 Shellcode 检测没有影响, 模拟中仅仅将其跳过. 检测器用 C 语言编写.

```
Input: Segment (length l)                                   // After Flow Segmentation
Output: True or False

Valid[] = (1...l);                                  // Valid instruction, for each position
Start[] = (1...l);                  // Can be the first instruction of polymorphic shellcode
if (!search_GetPC_code())                            // Segment Preselection (Section 4.3.2)
    return False;
for (pos = 0; pos < l; pos++)
    (Valid[pos], Start[pos]) = decode(pos);                 // Disassembly at each position
for (pos = 0; pos < l; pos++) {                                          // Emulation loop
    if (!Start[pos])                              // Emulation Loop Reduction (Section 4.3.3)
        continue;
    PC = pos;
    while (!check_ending_condition()) {                     // Ending Execution (Section 4.2)
        Pattern = execute(PC);                   // Execute and output abstraction pattern
        Suspicious = endless_loop_checking();   // Optimization of Resilience (Section 4.4)
        if (state_transition(Pattern))                  // Detection Criterion (Section 4.2)
            return True;
        Start[pos] = 0;                           // Emulation Loop Reduction (Section 4.3.3)
    }
}
if (Suspicious)
    add_to_suspicious_queue();
return False;
```

图 5　算法整体结构

4.2　漏报

我们使用了 11 个多态引擎生成多态 Shellcode 样本来检验漏报率, 包括 Clet, ADMmuate, Jempiscodes, TAPioN 以及 Metasploit Framework 的解码器模块 Alpha_mixed, Alpha_upper, Call4_dword_xor, Countdown, Fnstenv_mov, Jmp_call_additive 和 Shikata_ga_nai. 这些引擎采用了不同的, 甚至比较高级的多态技术.

实验中, 每一个引擎对一 Shellcode 生成 1000 个变种. 为接近 Shellcode 在数据流中的真实情形, 我们在每一 Shellcode 前加上了 100 字节随机数据, 这样会增加检测难度. 实验结果是原型系统检测到了全部 11000 个样本, 没有漏报.

4.3　虚警

我们使用真实的网络流量数据来检验算法的虚警率. 流量数据采集自清华大学校园网出口网关, 如表 1 所示, 共包含 2 种类型流量. 一类主要是常见的网络应用, 包括 HTTP(80 端口)和 HTTPs(443 端口); 另一类包括端口 135, 139 和 445. 与此 3 个端口相关的服务在广域网范围内应用较少, 因此采集到的流量也很少, 但我们将看到其很大一部分都是恶意攻击.

由于远程代码注入攻击通常攻击一些公开的服务中的漏洞, 在实验中我们只处理client-to-server方向的流. 另外, 为方便与现有工作 [5]的对比, 我们还测试了对较大流只取前 64 KB 的情形(尽管我们认为这样做会错过某些位于流后部的Shellcode).

实验中, 80 和 443 端口的数据中没有检测到Shellcode, 但在 135, 139 和 445 端口的数据中检测到了很多, 如表 2 所示. 通过人工分析我们发现它们全部的确是Shellcode. 因此虚警率为 0, 此结果优于现有工作 [6]. 我们对实验过程进行了跟踪, 发现在输入到模拟循环的数据段中,

有 65%进入到IGPSA的状态q_1, 0.03%进入到状态q_2(即从内存中读取PC), 而没有任何一个进入解密循环. 因此, 事实上所有数据段都还距离成为虚警很远.

表 1　实验数据集

端口	持续时间/d	大小/B	流数量	Client-to-Server 流	
				总量	<64KB 截取后总量
80	7	28G	710K	3.3G	1.1G
443	33	2.2G	190K	400M	240M
135	17	25M	7.5K	7.5M	—
139	17	720M	31K	41M	—
445	17	96M	11K	20M	—

表 2　流量数据集的实验结果

端口	检测数量	吞吐量/Mbps	可疑数量
80	0	44	9
80(64K)	0	132	1
443	0	40	6
443(64K)	0	42	5
135	4434	—	0
139	658	—	0
445	2269	—	0

进一步的, 我们还手工分析了在 135 端口数据集中检测到的 Shellcode. 根据这些流和 Shellcode 的相似情形, 我们将这些 Shellcode 分为几类. 每一类中, 所有的 Shellcode 都位于流的同一位置, 流中除 Shellcode 外的其他部分相同, 而 Shellcode 完全相同或解密子程序具有相似的结构. 由此, 我们可以估计, 每一类对应于一个或一些相似的攻击来源, 如一种蠕虫, 一种攻击工具或一个攻击者. 表 3 总结了这些类的特征. 其中, 两个类使用了改变密钥的多态技术. 此结果说明 Shellcode 的加密甚至多态已在真实网络攻击中被广泛使用.

表 3　135 端口 Shellcode 分类

序号	百分比	GetPC code	漏洞	多态
1	68%	call	unsure	yes (key)
2	24%	call	MS03-026	no
3	4.0%	call	unsure	yes (key)
4	2.8%	call	MS03-026	no
5	1.3%	fstenv	MS03-026	no

4.4　性能优化

本文中实验均在一台 2.80 GHz Intel Xeon, 运行linux kernel 2.4.20 的PC机上运行. 表 2 给出了原型系统对各数据集的吞吐量. 由于 135/139/445 的数据集较小, 难以准确统计吞吐量,表中未显示. 在现有工作中, 文献 [6]的吞吐量约为 12 Mbps, 文献 [5]中 80 端口在最优情形下(执行门限为 256)约为 15 Mbps.

由于各种外部原因(如硬件环境, 数据集特性等), 绝对的吞吐量事实上并不能有很强的说服力. 我们又进行了进一步的分析来验证算法中采用的优化技术的有效性.

首先, 我们分析了 80 端口第 2 个数据集(保留 64KB)的流长度分布情形(如图 6 所示). 图 7 的两条曲线中, Y 分别表示长度大于 X 字节的流所占的百分比, 以及它们总的大小在所有流总的大小中所占比例. 由图 7 可知, 尽管较大流的比例很低, 但它们占据了大部分的数据量. 因此, 很大一部数据能够通过流分割策略而得到优化. 其次, 在数据段过滤阶段, 只有 6.8%的数据段被保留输入模拟器. 尽管较长的数据段更可能被保留, 但它们对性能的影响也更大. 因此, 被保留数据段的字节量只有 4.3%, 且数据段过滤后, 平均数据段长度从 440 降为 280 字节. 再次, 模拟循环的平均循环次数为 35, 而平均数据段长度为 280, 即优化技术使得循环次数降低了 87%.

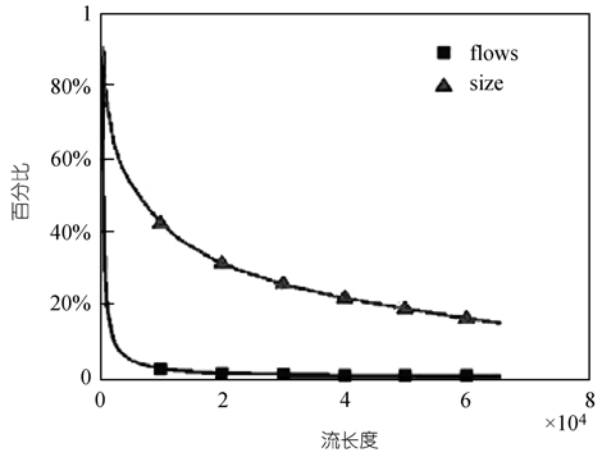

图 6　流长度分布

综上所述, 本文的算法通过这 3 个阶段对检测器进行了很大程度的优化. 尽管我们只做了基于数据集的离线实验, 但根据目前的实验结果, 我们相信它能够进一步地实现在实时的基于网络的入侵检测系统中.

```
00:  EB35          jmp 00000037
-----------------------------------------------
02:  83EC64        sub esp, 00000064
05:  31C9          xor ecx, ecx
07:  83C164        add ecx, 00000064    - loop 1
0A:  44            inc esp
0B:  E2FD          loop 0000000A
-----------------------------------------------
0D:  31DB          xor ebx, ebx
0F:  83C37F        add ebx, 0000007F
12:  4B            dec ebx              - loop 2
13:  75FD          jne 00000012
15:  753A          jne 00000051
-----------------------------------------------
17:  5A            pop edx
18:  4A            dec edx
-----------------------------------------------
19:  89D0          mov eax, edx
1B:  6681C21412    add dx, 1214
20:  4A            dec edx              - loop 3
21:  39C2          cmp edx, eax
23:  75FB          jne 00000020
-----------------------------------------------
25:  33C9          xor ecx, ecx
27:  41            inc ecx
28:  6681F97D01    cmp cx, 017D         - loop 4
2D:  75F8          jne 00000027
-----------------------------------------------
2F:  80340A99      xor byte ptr [edx+ecx], 99
33:  E2FA          loop 0000002F
35:  EB05          jmp 0000003C
37:  E8C6FFFFFF    call 00000002
3C:
...     <encrypted payload>
```

图 7　包含构造长循环的多态 Shellcode

4.5　抗攻击能力

本节中我们将验证系统的抗攻击能力. 虽然存在很多循环, 但可疑队列中的数据段却很少, 如表 2 所示. 注意数据集的时间长度至少为 7 天, 因此判为可疑的数据段能够轻易的由另一线程处理.

另外, 我们构造了一个包含长循环的多态Shellcode来测试系统的抗攻击能力. 基于图 1 中的简单解密子程序, 我们在其不同位置又加入了 4 个循环, 构造了一个新的解密子程序, 它可能使现有基于模拟的方法 [5,6]失效. 构造的解码器如图 7 所示, 其 4 个循环的循环次数都超过TH_e, 因此都被错误终止. 循环 1 通过改变ESP影响PC的获取, 循环 2 影响程序流程, 循环 3 影响负载的读取, 循环 4 影响解密循环次数. 使用这些循环, 就能绕过简单的基于门限的方法. 在原型系统中, 4 个

循环都被检测到, 被发现影响了内存读操作和程序流程, 因此此Shellcode被加入可疑队列, 最终被检测到.

当然由于攻击者可能构造出更复杂的, 目前我们尚未意识到的循环, 我们并不能说本文算法能够应对所有构造的长循环. 但是, 至少我们能够检测到其中一部分, 并因此提高了构造能绕过检测的多态 Shellcode 的难度. 同时, 基于我们所设计的解决长循环问题的基本思路(即检测错误终止循环对解密子程序的可能影响), 本文算法具有可扩展而应对更多类型的构造长循环的能力.

5 结论

本文提出了一种改进的基于模拟的多态 Shellcode 检测方法. 提出了抽象模型 IGPSA 对多态 Shellcode 的动态行为进行描述. 基于此模型, 推导出了作为检测算法一部分的检测准则. 算法中通过几项优化技术显著提高了运行性能和对规避检测型 Shellcode 的应对能力. 针对本文算法我们设计了一个原型系统. 实验中, 原型系统实现了无虚警和漏报, 同时具有较高的吞吐量和抗攻击能力. 未来工作主要针对如何更早地过滤出正常流量, 以进一步减小开销.

参考文献

1 Szor P, Ferrie P. Hunting for metamorphic. In: Proceedings of the Virus Bulletin Conference. Ofordshire: Virus Bulletin Ltd, 2001. 123—144

2 Payer U, Lamberger M, Teufl P. Hybrid engine for polymorphic code detection. In: Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05). Berlin: Springer-Verlag, 2005. 19—31

3 Chinchani R, Berg E. A fast static analysis approach to detect exploit code inside network flows. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005. 284—308

4 Kruegel C, Kirda E, Mutz D, et al. Polymorphic worm detection using structural information of executables. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05). Berlin: Springer-Verlag, 2005

5 Polychronakis, M, Anagnostakis K G, Markatos E P. Network-level polymorphic shellcode detection using emulation. In: Proceedings of SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment. Berlin: Springer-Verlag, 2006

6 Zhang Q, Reeves D S, Ning P, et al. Analyzing network traffic to detect self-decrypting exploit code. In: Proceedings of the 2nd ACM Symposium on InformAtion, Computer and Communications Security (ASIACCS'07), 2007. 4—12

7 Akritidis P, Markatos E, Polychronakis M, et al. Stride: Polymorphic sled detection through instruction sequence analysis. In: Proceedings of the 20th IFIP International Information Security Conference (SEC'05), 2005. 375—392

8 Toth T, Kruegel C. Accurate buffer overflow detection via abstract payload execution. In: Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'02), 2002. 274—291

9 Payer U, Teufl P, Kraxberger S, et al. Massive data mining for polymorphic code detection. In: Mathematical Methods, Models and Architectures for Computer Network Security Workshop (MMM-ACNS'05), 2005. 25—27

10 Pasupulati A, Coit J, Levitt K, et al. Buttercup: On network-based detection of polymorphic buffer overflow vul-

nerabilities. In: Proceedings of Network Operations and Management Symposium (NOMS), 2004
11 Li Z, Sanghi M, Chen Y, et al. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In: Proceedings of 2006 IEEE Symposium on Security and Privacy (S&P'06), 2006. 32—47
12 Newsome J, Karp B, Song D. Polygraph: Automatically generating signatures for polymorphic worms. In: Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05), 2005. 226—241
13 Christodorescu M, Jha S, Seshia S A, et al. Bryant. Semantics-aware malware detection. In: Proceedings of 2005 IEEE Symposium on Security and Privacy (S&P'05), 2005. 32—46
14 Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, 2003. 169—186
15 Intel Corporation. IA-32 Intel Architecture Software Developer's Manual. 2006

附录 A　数据段长度优化

另 L 为流长度, l 为数据段长度, n 为数据段间重叠部分的长度. 在 $L \gg l > n$ 的情形下, 一个流中的数据段平均数量为 $L/(l-n)$. 由于模拟一个数据段的计算复杂度是 $O(l^2)$, 对流的复杂度为 $L/(l-n)\times O(l^2)$. 另 $C = Ll^2/(l-n)$. 为得到最优的 l, 使得 C 最小, 将 C 求导 $C'(l) = (Ll^2/(l-n))' = Ll(l-2n)/(l-n)^2$. 另 $C'(l)=0$, 可得 $l=2n$. 此时 $C(l)$ 取最小值 $C(2n)=4nL$. 因此, 当 $l=2n$ 时, 模拟一个流的计算复杂度最小, 其值为 $O(nL)$.

附录 B　数据段过滤的概率估计

另 $x_1, x_2, \ldots, x_i, \ldots, x_l$ 为长l的数据段, 其中 x_i 是位置i处的字节. 假定 x_i 在区间[0, 0xFF]上均匀分布, 则有 $P(x_i = \text{0xE8}) = P(x_i = \text{0xFF}) = P(x_i = \text{0xD9}) = 1/256$. 在位置 i 的指令为合法的相对call的概率 $P_1 = 1/256 \times l/2^{32} = l/2^{40}$. 根据对call指令格式的定义 [15], 操作码FF为call指令的概率为 1/8, 因此在位置 i 的指令为绝对相对 i 的概率为 $P_2 = 1/256 \times 1/8 = 1/2^{11}$. 类似可得, 指令为fstenv的概率 $P_3 = 3/2^{12}$. 最后, 假定 $x_i, i = 1, 2, \ldots, l$ 为独立同分布, 那么一个数据段中不包含这 3 种指令(标记为正常)的概率可计算得 $P = (1 - P_1 - P_2 - P_3)^l = (1 - l/2^{40} - 5/2^{12})^l$.