计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.200300151

通用代码 Shell 化技术研究

陈 涛 舒 辉 熊小兵
信息工程大学数学工程与先进计算国家重点实验室 郑州 450001
(498673466@qq.com)

摘 要 代码 Shell 化技术是一种实现程序从源码形态到二进制形态的程序变换技术。该技术可用于实现 Shellcode 生成,生成包括漏洞利用过程中的 Shellcode 及后渗透测试过程中的功能性 Shellcode。文中形式化地描述了程序中代码与数据的关系,提出了一种基于 LLVM(Low Level Virtual Machine)的通用程序变换方法,该方法可用于实现操作系统无关的代码 Shell 化。该技术通过构建代码内置全局数据表和添加动态重定位代码,将代码对数据的绝对内存地址访问转化为对代码内部全局数据表的相对地址访问,重构了代码与数据之间的引用关系,解决了代码执行过程中对操作系统重定位机制依赖的问题,使得生成的 Shellcode 代码具有位置无关特性。在验证实验中,使用适用于不同操作系统的不同规模的工程源码对基于该技术实现的 Shellcode 生成系统进行了功能测试,并对比了 Shell 化前后代码功能的一致性、文件大小、函数数量和运行时间,实验结果表明基于该技术的 Shellcode 生成系统功能正常,具有较好的兼容性和通用性。

关键词:LLVM;Shellcode;代码 Shell 化;内存加载;程序变换
中图法分类号 TP309.5

Study of Universal Shellcode Generation Technology

CHEN Tao,SHU Hui and XIONG Xiao-bing
State Key Laboratory of Mathematical Engineering and Advanced Computing,Information Engineering University,Zhengzhou 450001,China

Abstract Shellcode generation technology is a program transformation technology that transforms programs from source form to binary form. This technology can be used to implement Shellcode generation,including Shellcode used in exploitation and functional Shellcode used in post-penetration period. This paper formally describes the relationship between code and data in the program and proposes a LLVM-based program transformation technology,which can be used to generate system-independent Shellcode. By constructing a built-in global data table and adding dynamic relocation code,this technology converts the access form of the code to the data from absolute memory address to relative memory address,eliminates the dependence of the relocation mechanism provided by operating system during code execution,and makes the generated Shellcode have good position-independent characteristics. In the experimental part,we test the function of our shellcode generation system based on this technology with different source code of different sizes under different operating systems. We also compare the consistency of the code function before and after the shellcode generation,as well as the file size,number of functions and execution time. Experiment results show that the shellcode generation system functions normally and has strong compatibility and versatility.

Keywords LLVM,Shellcode,Shellcode generation,Memory loading,Program transformation

1 引言

二进制可执行程序的加载与运行离不开操作系统,由操作系统负责解析二进制程序文件,完成代码的内存映射、数据和函数地址重定位以及运行环境初始化等工作。Shellcode 是一段用于利用软件漏洞而执行的二进制代码[1],具备自定义的功能且短小精悍[2],能够不依赖特定的二进制可执行文件格式被内存加载执行,能够在执行期间完成数据和函数地址的重定位,具有较好的通用性。这种代码被称为“Shellcode”,原因是它通常启动一个 Shell 命令行终端。随着技术的发展,所有执行类似任务的代码片段都可以被称作 Shellcode,其应用场景不仅包括传统意义上的漏洞利用,还包括实现特定功能以用于对抗分析和检测等场景。近年来围绕 Shellcode 的生成[3-5]、变形[6-9]、检测[10-13]等问题的研究一直是漏洞利用与安全研究的热点。

借助编译器对程序代码进行处理来生成 Shellcode 是目前较为通用的 Shellcode 生成思路。该思路通过修改程序源码或者中间语言代码来完成程序代码的变换,并经过编译器和链接器的编译、链接以及优化处理,生成具备代码位置无关特性的 Shellcode 代码。然而,生成过程也要考虑一些存在的

到稿日期:2020-03-25 返修日期:2020-07-06 本文已加入开放科学计划(OSID),请扫描上方二维码获取补充信息。
基金项目:国家重点研发计划项目(2016YFB08011601)
This work was supported by the National Key R&D Program of China (2016YFB08011601).
通信作者:舒辉(415314938@qq.com)

问题：1）兼顾 Shellcode 的应用场景，即漏洞利用场景和要求 Shellcode 具备网络访问、文件读写、内存执行等复杂功能的后渗透利用场景；2）多平台架构适配问题，即适用于不同的操作系统环境，包括但不限于 Windows 和 Linux 系统，适用于不同的处理器架构，如 x86，x64，arm 等架构。

现有的 Shellcode 生成技术和方法存在许多不足，如对不同结构源码的通适性差；不支持多种应用场景；不支持多平台架构等问题。针对这些问题，本文提出了一种通用于多平台架构的代码 Shell 化技术，该技术具有现有 Shellcode 生成技术的优点，同时还很好地弥补了现有 Shellcode 生成技术的不足。本文首先介绍了现有 Shellcode 生成技术的研究现状；然后形式化地描述了程序中代码与数据的关系并构建一种新的代码 Shell 化模型，描述从源码到 Shellcode 生成的变换过程；接着介绍了基于该模型的原型系统的设计与实现；最后通过功能验证实验来评估该系统的功能，通过 Shell 化前后的目标文件大小、函数数量以及运行时间对比来评估该系统的性能。

2　研究现状

现有的 Shellcode 构建有如下两种方法。1）直接由汇编代码编译链接生成 Shellcode。该方法编写难度大，需要手动维护堆栈平衡且与具体的操作系统和 CPU 架构相关，通用性不强。该方法用于编写代码规模较大的功能性 Shellcode 时基本不具备可行性。2）由 C/C++代码经集成开发环境编译生成 Shellcode。该方法的难度相对较低且效率高，由编译器完成诸如词法检查、语法检查、代码优化、维护栈和寄存器平衡等工作，极大地简化了 Shellcode 生成的过程。目前业内许多安全研究者已经基于第二种方法实现了不同的 Shellcode 构建方案，并针对 Shellcode 生成框架存在的代码位置无关、导入符号地址自定位问题提出了不同的解决方案。

BIONDI[14] 在 CanSecWest 2004 安全大会上公开的 ShellForge 框架支持包括 Linux，MacOS 等类 UNIX 操作系统，以及 i386，ARM，MIPS 等处理器架构。该框架通过封装不同处理器架构的系统调用来解决导入函数自定位的问题，通过编译器的-fpic 选项及合并数据段到代码段的方法来解决代码位置无关问题。Shellcode 生成框架具有一定的通用性，但也存在局限性，如只支持类 UNIX 系统上 32 位 C 程序的 Shell 化，且不支持导入变量引用。

Caillat[15] 在 BlackHatEurope 2009 上公开了 WishMaster 开源 Shellcode 生成框架，该 Shellcode 生成框架通过修改 C 语言代码的方式来改变代码与数据之间的地址引用关系，从而生成代码位置无关的 Shellcode，并通过添加自定义重定位处理函数来解决导入符号地址的自定位问题。该方法的局限性在于其仅支持 32 位 Windows 系统，通用性不足。

Zhu 等[16] 总结了现有的 Shellcode 生成方法，并提出了一种改进的生成方法，该方法可在 Windows 上实现 x86 和 x64 类型 Shellcode 的生成，但是由于未对全局变量、静态变量、常量字符、外部变量进行处理，其可用于生成 Shellcode 的程序代码内容受限，且不支持其他平台和架构。

这些构建方案都或多或少地解决了代码 Shell 化过程中的某些问题，具备一定的可用性，但同时也存在着兼容性差、源码内容和结构受限、易用性不足、目标平台通用性差等问题。本文提出的代码 Shell 化技术建立了一种数据聚合访问的通用程序变换模型，可实现程序从源码到 Shellcode 的生成。该技术具备适应 Windows 和 Linux 等操作系统、对源码内容的限制小、支持 x86，x64 和 arm 等处理器架构的诸多优势。

3　Shell 化技术的原理与模型

3.1　程序组织结构

PE（Portable Executable）是 Windows 平台上的可执行文件格式，ELF（Executable Linkable Format）则是类 UNIX 系统的可执行文件格式。PE 和 ELF 文件都采用了基于节的文件组织格式，其中代码节用于存储整个程序的代码；数据节用于存储程序代码运行过程中所依赖的数据（如全局变量、外部函数信息、常量字符串等）。操作系统在为程序创建进程实例时会根据目标程序的文件格式完成内存空间的开辟和参数的初始化等工作。

程序代码对全局变量、静态变量、常量字符串等数据的访问形式，是指向.data，.rodata 和.bss 数据段的绝对地址；对外部导入变量、外部导入函数的地址访问则是指向.idata 和.plt 表的绝对地址。这些绝对地址在程序被加载后由操作系统程序进行内存重定位修正，从而指向数据的准确内存地址。

本文的 Shell 化模型会改变程序的原有组织结构，图 1 给出了 Shell 化前源码经过正常编译链接生成的程序结构以及 Shell 化后输出的 Shellcode 代码的组成结构的对比。Shell 化生成的 Shellcode 包含原程序功能代码，用于聚合全局变量、静态变量、常量字符串等数据的全局数据表以及用于初始化该数据表的代码。

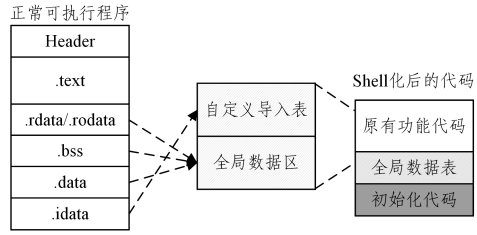


图 1　Shell 化前后程序结构的对比图

Fig. 1　Structure comparison before and after Shellcode generation

3.2　程序变换模型

现有的 Shellcode 生成技术对程序代码 Shell 化过程的描述仅停留在工程实现的层面，缺乏相应的理论模型和设计依据。针对该问题，本文通过借鉴代码（Position Independent Code，PIC）化思想提出并建立了一套通用的程序变换模型。该模型对程序中代码与数据之间的关系进行抽象并提出了一种数据聚合访问的程序变换方法，结合重建代码与数据之间引用关系的做法来解决程序中代码对数据的访问问题。其中，聚合数据的目的是实现“代码”包含“数据”，最终消除目标程序中的数据节，摆脱代码在运行时对文件结构和系统重定位机制的依赖；代码与数据之间引用关系的重建则是为了便于代码访问内置数据，变换了的访问方式不会破坏程序内部代码对数据的访问依赖，即代码仍旧可以正常地访问原有的数据和调用 API 函数。

3.2.1　符号定义

可执行程序是一个由代码、数据、代码之间的引用关系及

代码与数据之间的关系组成的集合体，其中代码是指程序中指令集合，数据则是代码在执行过程中所依赖的变量、符号和函数地址等数据信息。表1列出了程序变换模型中用来描述代码、数据以及两者之间关系的各种符号。

表1 符号描述表

Table 1 Symbol description

符号	定义
P 和 P'	可执行程序
C 和 C'	程序中的可执行代码集合
D	程序代码运行时用到的数据集合
D'	Shell 化变形后引入的全局数据表
I	程序中可执行代码与数据之间的引用关系
Θ 和 Θ'	程序中可执行代码内部的引用关系集合
FI	程序代码中的所有内部函数集合
FO	程序代码中的外部导入函数集合
G	程序内定义的所有全局变量、静态变量
O	程序中引用的外部模块内定义的变量
ε	程序内部函数之间的相互引用关系集合
$\mathscr{R}$	程序内部函数对外部导入函数的引用关系集合
R	用于进行数据表初始化的重定位代码集合
T	程序功能等价变换，在本文中指对程序的 Shell 化
$\subseteq$	集合子集，$A\subseteq B$ 表示集合 A 是 B 的子集
$\cup$	集合的并
$\in$	元素属于集合

定义 1(代码集合) 程序中所有函数共同构成一个非空有限的可执行代码集合 C，函数 c 为集合中的元素，根据函数是否为内部定义将该集合 C 划分为内部函数集合 FI 和外部函数集合 FO，表达式为 $C=FI\cup FO$。

FI 代表了程序代码中的所有内部函数集合，表达式为 $FI=\{fi_0,fi_1,fi_2,\cdots,fi_n\}$，其中 n 表示函数个数，$fi_j\in FI$ $(1\leqslant j\leqslant n)$表示内部定义的某个具体函数，$fi_0\in FI$ 为程序的入口点函数，且对于任意 $fi_j,fi_k\in FI$，若 $j\neq k$，则有 $fi_j\neq fi_k$。

FO 代表了程序代码中的外部导入函数集合，表达式为 $FO=\{fo_1,fo_2,\cdots,fo_n\}$，其中 n 表示函数的个数，$fo_j\in FO$ $(1\leqslant j\leqslant n)$表示代码外部定义的某个函数，且对于任意 fo_j，$fo_k\in FO$，若 $j\neq k$，则有 $fo_j\neq fo_k$。

定义 2(数据集合) 程序中所用到的全局变量、静态变量以及外部导入变量共同构成一个非空有限的数据集合 D，这些数据在程序的二进制文件中被存储在各数据段内。根据数据是否为程序内部定义将该数据分为内部数据 G 和外部数据 O，用集合的并表示为 $D=G\cup O$。

其中，G 包含程序内定义的所有全局变量、静态变量，表达式为 $G=\{g_1,g_2,\cdots,g_n\}$，n 表示变量的个数，g_j $(1\leqslant j\leqslant n)$表示代码中的某个全局变量或静态变量，且对于任意 $g_j,g_k\in G$，若 $j\neq k$，则有 $g_j\neq g_k$。

O 表示程序中引用的外部模块内定义的变量，表达式为 $O=\{o_1,o_2,\cdots,o_n\}$，其中 n 表示变量的个数，o_j $(1\leqslant j\leqslant n)$表示代码中的某个外部变量，且对于任意 $o_j,o_k\in O$，若 $j\neq k$，则有 $o_j\neq o_k$。

定义 3(函数调用关系) Θ 表示程序中代码之间的引用关系集合，表达式为 $\Theta\subseteq C\times C=\{(c_1,c_2)\mid c_1\in C,c_2\in C\}$。$\Theta$ 可分为程序内部函数之间的相互引用关系集合 E 和程序内部函数对外部导入函数的引用关系集合 $\mathscr{R}$，用集合表示为 $\Theta=\varepsilon\cup\mathscr{R}$。

定义 4(数据引用关系) I 表示程序中可执行代码与数据之间的引用关系集合，表达式为 $I\subseteq C\times D=\{(c,d)\mid c\in C,d\in D\}$。具体可分为 I_1 和 I_2 两种，用集合的并表示为 $I=I_1\cup I_2$，其中 I_1 表示程序模块内部数据引用，如对全局变量、静态变量的引用；I_2 表示对模块外部的数据引用，比如引用其他模块中定义的导出变量。

定义 5(程序) 设符号 P 代表一个正常的程序，从数据、代码以及两者之间的相互引用关系的角度，用如下四元组来描述一个程序的组成，即 $P=\langle C,D,I,\Theta\rangle$，其中各个符号的定义和说明如定义1－定义4。

定义 6(功能等价变换) 设 T 是一个程序功能等价的变换，设符号 P' 代表 Shell 化后的程序，对程序 P 做功能保持的变换，表达式为 $P'=T(P)$。

本文将程序源码进行的 Shell 化视为一个功能保持的等价变换，后续将以符号 T 来代表对程序的 Shell 化变换过程。在 Shell 化过程中，虽然函数形态以及调用关系会产生变化，且会增加新的初始化函数，但是程序的功能并不会改变。源码在经过 Shell 化后生成的程序 P' 只包含“代码”，因此 Shell 化后的程序 P' 可用如下表达式表示：

$P'=\langle C',\Theta'\rangle$

(1) C' 表示变换后程序的代码集合，包含了内部函数集合 FI'、新建的全局数据表 D' 以及用于进行数据表初始化的重定位的代码集合 R，表达式如下：

$C'=FI'\cup R\cup D'$

(2) Θ' 表示变换后程序 P' 中可执行代码内部的引用关系集合。根据程序中对不同部分代码的引用情况可以分为 Θ'_1、Θ'_2 和 Θ'_3 3种，其中 Θ'_1 表示初始化代码对数据表的访问，比如填充外部导入函数和变量的实际内存地址；Θ'_2 表示模块内部函数对数据表的访问，如表内的原全局变量、静态变量等；Θ'_3 表示模块内部函数对初始化代码的调用，表达式如下：

$\Theta'=\Theta'_1\cup\Theta'_2\cup\Theta'_3$

变换过程以函数为基本单元，对程序中不同类型的函数的参数以及内部地址引用指令进行修改变换，结合添加自定义重定位函数来实现程序代码的 Shell 化变换。

3.2.2 地址引用变换

程序中代码的地址可分为变量地址和函数地址，其中代码与代码、代码与数据之间的地址引用关系可用如下4个表达式进行描述。

(1)模块内部函数之间的地址引用 ε，表达式如下：

$\varepsilon\subseteq FI\times FI=\{(fi,fj)\mid fi\in FI,fj\in FI\}$

(2)模块内部对全局变量和静态变量的地址引用 I_1，表达式如下：

$I_1\subseteq C\times G=\{(c,g)\mid c\in C,g\in G\}$

根据代码集合的划分，亦可用如下集合的并来表示：

$I_1=\{(fi,g)\mid fi\in FI,g\in G\}\cup\{(fo,g)\mid fo\in FO,g\in G\}$

(3)模块外部函数地址引用 $\mathscr{P}$，表达式如下：

$\mathscr{P}\subseteq FI\times FO=\{(fi,fo)\mid fi\in FI,fo\in FO\}$

(4)模块外部变量地址引用 I_2，表达式如下：

$I_2\subseteq C\times O=\{(c,o)\mid c\in C,o\in O\}$

根据代码集合的划分，亦可用如下集合的并来表示：

$I_2=\{(fi,o)\mid fi\in FI,o\in O\}\cup\{(fo,o)\mid fo\in FO,o\in O\}$

其中，E 代表模块内部函数之间的调用关系，其地址引用形式是相对地址，存在地址无关性，无需进行重定位处理。而 R 代表模块内部函数对模块外部函数的引用关系，I_1 代表模块内部全局变量引用，I_2 代表模块外部变量引用，这 3 种地址引用形式是绝对地址，需要进行重定位处理。

本文的 Shell 化变换模型中的地址引用变换是通过在程序代码中构建一个全局数据表来处理全局变量、静态变量和导入函数的地址引用问题，模型将代码对上述数据绝对地址形式的分散访问转化为对代码内置全局数据表相对地址形式的集中访问。变换之后的程序中地址引用关系将只包含代码对代码内置数据表的地址引用，即：

$\Theta' \subseteq C' \times C' = \{(c_1', c_2') \mid c_1' \in C', c'_2 \in C'\}$

根据不同的引用情况，可将其划分为以下 3 个部分：

(1)初始化代码在进行初始化时对全局数据表 D' 中数据项地址的引用；

$\Theta_1' \subseteq R \times D' = \{(r, d') \mid r \in R, d' \in D'\}$

(2)程序内部函数对数据表 D' 中原全局变量、静态变量和导入函数地址的引用；

$\Theta_2' \subseteq FI' \times D' = \{(fi', d') \mid fi' \in FI', d' \in D'\}$

(3)模块内部函数对初始化代码的调用。

$\Theta_3' \subseteq FI' \times R = \{(fi', r) \mid fi' \in FI', r \in R\}$

3.2.3　函数签名变换

函数签名包含一个函数的信息，该信息又包括函数名、函数参数类型、所在名称空间等。根据定义 1，$FI = \{fi_0, fi_1, \cdots, fi_n\}$ 代表一个程序中的内部函数集合，则 Shell 化过程在函数层面的变化可以用如下表达式进行描述：

$fi_m' = T(fi_m)(0 \leqslant m \leqslant n)$

从参数引用方式上，设一个函数的抽象表达为 F，函数参数为 $Arg_1, Arg_2, \cdots, Arg_n$，则函数表示为：

$F(Arg_1, Arg_2, \cdots, Arg_n)$

函数在经过 T 变换后表示为：

$F(Arg_1, Arg_2, \cdots, Arg_n, Args\_Struct)$

参数列表添加了一个新的参数 $Args\_Struct$，该参数即为全局数据表 D' 的起始地址指针，代码运行所需的全局变量、静态变量、常量字符串、导入变量、导入函数等被封装在该全局数据表中。将该表的起始地址以函数参数形式在函数间进行传递和引用，使得程序代码中对这些数据的地址引用方式由指向数据段和导入表的绝对地址变成对代码内指向该全局数据表的相对地址。

程序中的入口点函数为 fi_0，经过 Shell 化变换后的函数记为 fi_0'。对该入口点函数的变换包含函数签名的更新以及在函数起始位置新增调用初始化函数的指令。

3.2.4　代码 Shell 化算法

本文程序变换模型的核心思想是通过将聚合代码中分散的数据访问变换为统一的对数据表的访问来实现代码地址无关化。这个变换过程包含全局数据表的创建、函数签名变换、地址引用变换和入口点更新这几个部分。通过创建全局数据表来聚合程序中的数据访问是整个变换的基础；入口点的更新是为了实现在程序起始处加入用于实现全局数据表初始化的重定位代码；函数签名变换是为了向函数中引入全局数据表地址；地址引用变换根据函数中传入的全局数据表地址来修改数据访问方式。算法 1 描述了 Shell 化的基本处理流程。

算法 1　Shell 化算法

```
1. Procedure ShellTransform
2.   for f in all functions do //遍历程序所有函数
3.     if Isentry point(f) then //若当前函数为入口点函数
4.       Insert Initializationcode(f)//插入数据表初始化代码
5.     else
6.       Update Signature(f) //更新函数签名
7.       for var in f do //遍历函数中所有变量
8.         if Is GlobalData(var) then //若当前变量为全局数据
9.           Change CallRef(var) //修改当前变量引用
10.          AddTo Table(var)//将当前变量加入全局数据表
11.        end if
12.      end for
13.    end if
14.  end for
15. end Procedure
```

Shell 化的变换过程具体包含如下几个步骤。

步骤 1　根据程序中表示代码的全局变量、静态变量、常量字符串、导入变量、导入函数等的全局数据建立全局数据表 D'。

步骤 2　对用到上述全局数据的函数进行签名变换，引入全局数据表指针。

步骤 3　对用到上述全局数据的函数中的指令进行地址引用方式修改，指向全局数据表。

步骤 4　在程序的入口点函数 fi_0 中插入完成数据表建立和初始化的函数调用指令。

4　系统的设计与实现

基于本文提出的 Shell 化程序变换模型来实现原型系统的方法有两种：1)在源码层面通过修改源码的方式来实现；2)通过 LLVM 编译系统来实现。本文选择 LLVM 编译系统的原因是它是目前国内外最新、最完善的开源编译系统，其前端能够良好地兼容如 C/C++等多种高级编程语言，其后端能够支持如 x86，x86-64 等多种 CPU 架构，具备良好的前端语言无关和目标平台无关特性，还提供了丰富的各类优化接口，这些优势为本文研究内容的展开提供了良好的理论和工程实践基础。然而，通过修改源码的方式来实现 Shell 化则受限于编译器的选择，只能适用于特定的编程语言和特定平台架构，通用性有所缺乏。

本文将设计实现的 Shellcode 生成系统命名为 ShellVM。该系统是本文通用 Shell 化技术的具体实现，具备较好的通用性，其通用性不仅包含程序变换模型带来的方法通用性，即能够支持不同源码结构的代码 Shell 化变换，还包含 LLVM 编译器支持多种编程语言、目标系统平台、CPU 架构上的通用性。具体来说，其通用性包括以下 3 个方面：1)编程语言通用性，即支持的源程序编程语言包括但不限于 C/C++；2)平台通用性，即支持在包括 Windows，Linux，MacOS 等操作系统平台上的 Shellcode 生成；3)架构通用性，即支持包括 x86，x86-64，PowerPC，Mips 等 CPU 架构。ShellVM 系统是本文通用 Shell 化技术的具体实现，能够支持对不同源码结构的代码 Shell 化，具备较好的兼容性。

4.1　设计框架

本文的 Shellcode 生成系统的框架如图 2 所示。

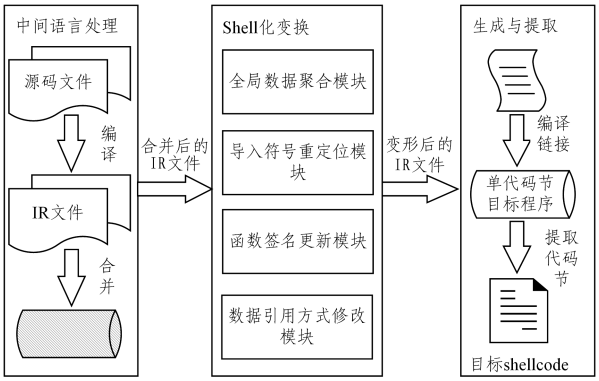


图 2 Shellcode 生成系统的框架图

Fig. 2 Framework of Shellcode generation system

ShellVM 系统包括以下 3 个部分。

(1)中间语言处理部分。该部分利用 Clang 编译器对程序源码进行编译,生成对应的 IR 文件,然后将其合并成一个包含程序所有中间语言代码的 IR 文件。

(2)Shell 化变形处理部分。该部分是整个系统的核心部分,用于对合并后生成的 IR 文件进行变形处理,可以分为以下 4 个模块:

1)全局数据处理模块。该模块负责提取程序 IR 代码中的全局变量、静态变量、常量数据初始值、导入变量、函数名称和所在模块信息,并依此构建全局数据表。

2)导入符号动态重定位模块。该模块负责在程序代码运行之初完成代码中导入符号的动态地址重定位,获取指定符号名对应的地址并填充全局数据表中对应的私有导入表项。

3)函数签名更新模块。这里的函数签名修改是通过添加全局数据表地址参数来实现函数对全局数据以及外部导入符号的地址的访问。

4)数据引用方式修改模块。该模块对代码中用到的全局数据和导入符号的指令操作数进行替换,用全局数据表基地址加上表内对应项的偏移来替换原有的操作数访问。

(3)该部分首先将变形处理后的 IR 中间语言文件与运行时动态重定位代码的中间语言文件进行合并、编译、链接,然后生成重构后只具有单个代码节的二进制可执行文件,最后提取该文件的代码节作为最终的 Shellcode 代码。

4.2 系统实现

4.2.1 全局数据聚合模块

该模块在中间语言层面实现代码中的数据聚合,将分散的地址访问聚合成对全局数据表的集中访问,包含以下 3 个部分:

(1)全局变量、静态变量、常量数据聚合

遍历程序 IR 文件中的所有全局变量、静态变量、常量数据,并根据有无初始化值来进行区分和初始化处理,以完成代码中全局数据的聚合。对于已初始化的变量和常量字符串,在全局表中存放其初始值;对于程序中未初始化的变量,在表中记录该类未初始化全局变量的大小,后续由初始化函数负责动态分配内存。

(2)导入变量、导入函数聚合

首先遍历程序 IR 文件中具有 dllimport 或 declare 属性的导入变量和导入函数,并根据其所属的不同动态链接库来完成归类和信息提取工作;然后在全局数据表中添加对应的表项,完成全局数据表中私有导入表部分的建立;最后实现导入变量和导入函数的聚合。

(3)全局数据表的构建

全局数据表中包含全局数据和私有导入表。根据前面两个部分中各全局数据、导入变量、导入函数的处理顺序,构建相应的数据表。在该表中,全局数据部分的表项内容对应全局数据的初始值,私有导入表部分的表项内容对应导入变量、导入函数的名称。该表的规模不仅与程序中所用到的全局变量、静态变量、常量字符串等全局数据有关,还与程序中用到的外部变量和 API 函数数量有关。

4.2.2 动态地址重定位模块

因为 Shellcode 缺乏正常二进制程序的文件结构,无法依赖系统的程序加载器来实现导入符号重定位,所以需要添加动态重定位代码来替代系统的程序加载器。该模块结合全局数据聚合模块建立私有导入表,在 Shellcode 初始化时动态解析导入符号地址,完成导入函数和导入变量的动态重定位处理。

该模块由于涉及具体的平台架构,需要针对目标 Shellcode 运行所依赖的目标操作系统和 CPU 处理器框架给出对应的代码实现,本系统集成了针对 Windows 操作系统上 x86 和 x64 架构及针对 Linux 操作系统上 x86,x64,arm,arm64,mips 和 mips64 架构的导入符号动态重定位实现。

4.2.3 函数签名更新模块

函数签名包含一个函数的声明信息,该信息由函数名称、参数类型、参数个数、返回值及其类型以及函数调用约定等组成。本文中函数签名修改的目的是为函数添加一个可访问全局数据表的指针。根据不同的函数类型采取不同的方式,如对于显式调用的函数采取添加额外参数的方式来实现;对于通过指针调用的特殊函数如线程函数,则通过在函数起始位置添加用于获取全局数据表指针的代码来实现。

4.2.4 数据引用方式修改模块

该模块负责对签名更新后的中间语言文件进行处理,对代码中数据引用方式进行修正。全局数据表保存了程序中所有全局数据的值和外部符号的地址指针。该模块通过替换中间语言代码中的指令或常量表达式操作数的方式,实现了将原有的对全局数据和外部导入符号的引用替换成对全局数据表项的值或地址的引用。

5 实验测试情况

本文设计的 Shellcode 生成系统能够实现 C/C++源码到 Shellcode 的自动化生成,即支持在 Windows 和 Linux 平台上的 C/C++源码文件或工程的 Shell 化编译并生成具有完整功能的 Shellcode 代码。本文实验使用的 C/C++程序源码均来自于 Github 开源代码仓库,既包括 Windows 平台上的 Visualstudio 2010,Visualstudio 2013,Visualstudio 2015 的工程,也包括 Linux 平台上的 Cmake 工程,源码文件代码行数从几百行到上万行不等。

实验分为功能测试和对比测试两个部分。

(1)功能测试。该部分对源码进行 Shell 化编译测试,测试系统的功能是否正常,即能否对给定的源码工程进行 Shell 化,并实现了一个 Shellcode 内存加载工具,用于测试由该系

统生成的 Shellcode 的功能是否正常。

(2)对比测试。该部分对比经过源码正常编译链接和经过该 Shell 化系统处理后的目标文件的大小、函数数量和运行时间，以从侧面体现该系统 Shell 化的效果。

5.1　实验条件

实验所采用的环境配置如表 2 所列。

表 2　代码 Shell 化的测试环境

Table 2　Shellcode generation test environment

PC 主机配置	OS：Windows10 64 bit；内存：32 GB；处理器：IntelCorei7-7700，3.6 GHz
Windows 虚拟机	OS：MicrosoftWindows 7；内存：2 GB
Linux 虚拟机	OS：Ubuntu 16.04；内存：2 GB
LLVM 版本	LLVM 3.8.0
Windows 开发环境	Visual studio 2010，Visual studio 2013 和 Visual studio 2015
Linux 开发环境	GCC7.4

本文实验部分对 Windows 系统上的 Visual Studio 工程和 Linux 系统上的 Cmake 工程代码进行代码 Shell 化测试，以验证该系统是否能够对上述 C/C++源码工程进行 Shell 化并生成无外部依赖的目标 Shellcode 代码。实验选取了 Github 上具有代表性的开源程序进行测试，测试内容包括：验证能否正常 Shell 化，比较系统处理后的代码与正常编译生成的目标文件大小，输出的 Shellcode 功能是否正常。用于进行测试的工程如表 3、表 4 所列，分别为 Windows 上的 5 个 Visualstudio 工程和 Linux 上的 5 个 Cmake 工程，这些工程都具备特定的功能并能够正常编译、链接，且生成的二进制目标程序都能够正常执行。

表 3　用于 Windows 上测试的 C/C++开源工程

Table 3　C/C++open source project on Windows

序号	程序	规模(行)	功能描述
1	TinyMet	217	支持 TCP，HTTP，HTTPS 协议的 Meterpreterstager
2	Remotecmd	313	一个开启本地端口监听的远程 cmd 命令行工具
3	Netcat	1866	一款集端口监听、文件传输和远程 Shell 的渗透测试工具
4	Lcx	2846	一款 windows 上的端口转发渗透测试工具
5	UacElevator	10550	一个利用 COM 白名单接口绕过系统 UAC 以实现权限提升的工具

表 4　用于 Linux 上测试的 C/C++开源工程

Table 4　C/C++open source project on Linux

序号	程序	规模(行)	功能描述
6	TcpSocket	52	一个简易的 TCPsocket 连接测试程序
7	Tinyhttpd	399	一个超轻量型的 HTTPServer 程序
8	Linux-rc	687	一个简易的 Linux 远程控制软件
9	Miniftp	1857	一个轻量型的命令行 ftp 服务器
10	MiniGzip	2127	一个小型的命令行 gzip 压缩软件

5.2　功能测试

功能测试过程分为 3 个部分：1)对源码工程进行正常编译、链接，得到可执行文件，并记录运行情况；2)将源码工程作为 Shellcode 生成系统的输入，由系统完成中间语言层面的 Shell 化变形处理，经过编译、链接得到具有位置无关的单代码节的可执行程序，提取代码节得到最终的 Shellcode；3)通过内存加载工具对 Shellcode 进行测试，并记录加载运行情况。具体测试情况如表 5 所列。由表 5 可知，该 Shellcode 生成系统能够支持 Windows 和 Linux 系统上的工程源码的 Shell 化；不同代码量的样本测试以及 Shell 化前后功能的对比结果也表明该系统的鲁棒性较好。

表 5　Shell 化前后的代码功能测试

Table 5　Code function test before and after Shellcode generation

序号	Shell 化前的功能	能否被 Shell 化(Y 表示成功)	生成的 Shellcode 功能
1	正常	Y	正常
2	正常	Y	正常
3	正常	Y	正常
4	正常	Y	正常
5	正常	Y	正常
6	正常	Y	正常
7	正常	Y	正常
8	正常	Y	正常
9	正常	Y	正常
10	正常	Y	正常

5.3　对比测试

本文从文件大小、函数数量以及运行时间 3 个方面对比了实验样本 Shell 化前后的数据。其中，文件大小对比如图 3 所示，函数数量对比如图 4 所示，运行时间对比如图 5 所示。

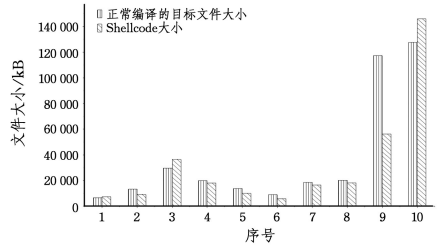


图 3　Shell 化前后生成的目标文件大小对比

Fig. 3　Comparison of file size before and after Shellcode generation

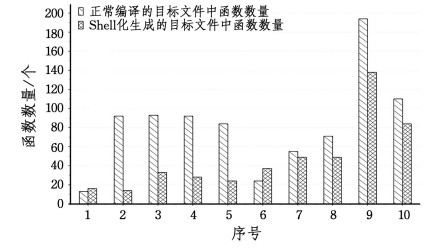


图 4　Shell 化前后生成的目标文件中函数数量对比

Fig. 4　Comparison of function number before and after Shellcode generation

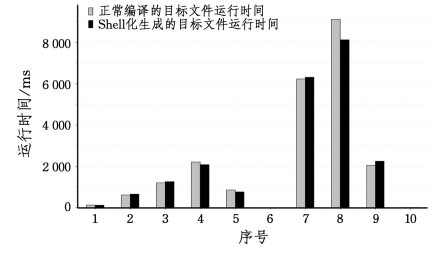


图 5　Shell 化前后生成的目标文件运行时间对比

Fig. 5　Comparison of execution time before and after Shellcode generation

由图3可知，实验1、实验3和实验10源码Shell化后生成的目标Shellcode稍大于正常编译生成的二进制文件，这是因为内建的全局数据表为某些静态变量分配了存储空间；其余的源码经过本系统生成的目标Shellcode都小于正常编译生成的二进制文件，这表明数据聚合在缩小目标代码上有一定作用。

由图4可知，实验1源码Shell化生成的目标Shellcode中的函数数量稍多于正常编译生成的目标二进制文件，这是因为该测试用例的源码中函数数量本身较少，且系统Shell化过程中增加了重定位部分代码；其余的测试源码经过本系统Shell化生成的Shellcode中函数数量明显少于正常编译生成的目标二进制文件。

由图5可知，实验源码在Shell化后生成的目标Shellcode的运行时间与正常编译生成的目标二进制文件的差异不大，可见Shell化过程基本不会对程序的运行效率造成影响。

结束语 本文提出的代码Shell化技术为Shellcode生成提供了一种通用的技术方法。该技术根据代码地址无关的思想建立了数据聚合访问的通用方法模型，并通过全局数据表的构建及导入符号自定位代码来重建代码与数据之间的引用关系。本文的Shellcode生成系统是基于LLVM编译框架设计实现的，具有良好的语言无关、平台无关特性，理论上可以同LLVM所支持的所有前端编程语言协调工作，支持包括C/C++等编程语言的代码Shell化，可面向如x86，x64，PowerPC，ARM，ARM-64以及MIPS等处理器架构生成位置无关Shellcode，具有适用多语言、多架构的通用性。此外，该系统还支持第三方LLVM Pass组件的使用，用以实现对Shellcode的指令替换、虚假控制流插入、控制流扁平化等混淆保护，因此应用前景十分可观。

当然，基于该技术实现的Shellcode生成系统也存在一些不足。首先由于该系统的设计与实现是基于中间语言的，存在语言层级的局限性，如不支持引用第三方lib库的源码；然后，最终生成的代码中所有变量和函数处理都在栈上实现，因此存在代码健忘性，即代码中的全局变量等只在当前执行周期内有效，在下一次运行该代码时，这些变量将会恢复为默认值；最后，该系统输出的Shellcode在特定的应用场景中可能存在限制，如缓冲区漏洞利用过程中的零字符截断等问题，该问题可以通过Shellcode编码变形[7-8]等手段来解决。

参 考 文 献

[1] WANG Y, LI X H, GUANG L, et al. Attack and Defending Technology of shellcode [J]. Computer Engineering, 2010, 36(18): 163-165, 168.

[2] NÉMETH Z L, LÁSZLÓ E. When Every Byte Counts—Writing Minimal Length Shellcodes[C]// Proceedings of the 13th International Symposium on Intelligent Systems and Informatics. Washington D. C., USA: IEEE Press, 2015: 269-274.

[3] ARCE I. The shellcode generation[J]. IEEE Security & Privacy Magazine, 2004, 2(5): 72-76.

[4] NICKHAR B. Writing Shellcode with a C Compiler[EB/OL]. (2010-07-01) [2019-01-28]. https://nickharbour.wordpress.com/2010/07/01/writing-shellcode-with-a-c-compiler.

[5] MATT G. Writing Optimized Windows Shellcode in C[EB/OL]. (2013-08-16)[2019-01-22]. http://www.exploit-monday.com/2013/08/writing-optimized-windows-shellcode-in-c.html.

[6] ROSCHKE S, CHENG F, MEINEL C. BALG: Bypassing Application Layer Gateways using multi-stagedencrypted shellcodes [C]// IEEE 2011 IFIP/IEEE International Symposium on Integrated Network Management (IM). New Jersey: IEEE, 2011: 399-406.

[7] MASON J, SMALL S, MONROSE F, et al. English shellcode [C]// Proceeding of the 16th ACM Conference on Computer and Communications Security (CCS'09). New York: ACM, 2009: 524.

[8] BASU A, MATHURIA A, CHOWDARY N. Automatic generation of compact alphanumeric shellcodes for x86[C]//LNCS 8880: Information Systems Security. Berlin: Springer, 2014: 399-410.

[9] TAMBOLI T, AUSTIN T H, STAMP M. Metamorphic code generation from LLVM bytecode[J]. Journal of Computer Virology and Hacking Techniques, 2013, 10(3): 177-187.

[10] VERMA N, MISHRA V, SINGH V P. Detection of alphanumeric shellcodes using similarity index[C]// Proceeding of 2014 International Conference on Advances in Computing, Communications and Informatics. 2014: 1573-1577.

[11] GU B X, BAI X L, YANG Z M, et al. Malicious shellcode detection with virtual memory snapshots[C]// Proceeding of the 29th Conference on Information Communications. 2010: 974-982.

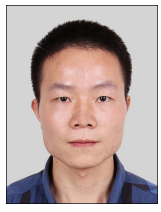
[12] WANG L J, DUAN H X, LI X. Dynamic emulation based modeling and detection of polymorphic shellcode at the network level [J]. Science China Information Sciences, 2008, 51(11): 1883-1897.

[13] ZHAO Z M, AHN GAIL-JOON A. Using instruction sequence abstraction for shellcode detection and attribution[C]// Proceeding of 2013 IEEE Conference on Communications and Network Security. 2013: 323-331.

[14] BIODI P. Shell Forge [EB/OL]. (2005-07-04) [2019-01-30]. http://www.secdev.org/projects/shellforge.

[15] CAILLAT B. WiShMaster-Windows Shellcode Mastery. [EB/OL]. (2007-05-29)[2019-01-30]. http://benjamin.caillat.free.fr/wishmaster.php.

[16] ZHU S, LUO S L, KE D X. Research on Automatic Building Approach of Windows Shellcode[J]. Information System and Security, 2017(4): 15-25.



CHEN Tao, born in 1992, postgraduate. His main research interests include cyber security and reverse engineering.

SHU Hui, born in 1974, Ph. D, professor, Ph.D supervisor. His main research interests include cyber security and reverse engineering.