doi:10.11887/j.cn.202002022 http://journal.nudt.edu.cn

ASLR机制脆弱性自动分析方法*

黄 宁,黄曙光,潘祖烈,常 超
(国防科技大学 电子对抗学院,安徽 合肥 230037)

摘 要:地址随机化是一种针对控制流劫持漏洞的防御机制。已有的漏洞自动分析与利用技术缺少对地址随机化机制影响的分析,导致生成的测试用例在实际环境中的运行效果受到极大限制。针对地址随机化的缺陷及其绕过技术的特点,提出了一种地址随机化脆弱性分析方法。该方法使用有限状态机描述程序运行路径中各关键节点的状态;针对常见的内存泄漏与控制流劫持场景建立约束条件;通过求解内存泄漏状态约束与控制流劫持状态约束的兼容性,分析地址随机化机制在特定场景下的脆弱性。实验结果表明,该方法可有效检测通过内存泄漏导致的地址随机化绕过及控制流劫持攻击,实现自动化的地址随机化脆弱性分析,提高针对软件安全性分析的效率。

关键词:地址随机化;控制流劫持;内存泄漏;有限状态机
中图分类号:TP311 **文献标志码**:A **文章编号**:1001-2486(2020)02-162-09

Automatic analysis to vulnerability of ASLR

HUANG Ning, *HUANG Shuguang*, *PAN Zulie*, *CHANG Chao*
(College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China)

Abstract: The ASLR (address space layout randomization) is a defense mechanism to prevent the control-flow hijack. The lack of analysis of the impact of ASLR in existed automatic vulnerability analysis and exploit technologies makes the test cases difficult to be used in actual environment. Aimed at the defects of address randomization and features of its bypass technologies, an analysis method was proposed to deal with the vulnerability of ASLR based on program states transition. The FSM (finite states machine) was used to describe the transition of each key state on the program path, the constraints for some common scenes of memory leakage and control-flow hijack were built, and the vulnerability of ASLR was analyzed by solving the compatibility of memory leakage state constraints and control-flow hijack state constraints. Experimental results show that the proposed method can effectively detect ASLR bypass and control-flow hijack attacks caused by memory leakage, realize the automatic vulnerability analysis of ASLR, and improve the efficiency of software security analysis.

Keywords: address space layout randomization; control-flow hijack; memory leakage; finite states machine

随着信息技术的发展,软件漏洞的挖掘与利用成了一个热点问题。针对不同类型的漏洞利用技术,各种保护机制也层出不穷。但是,多年的漏洞利用实践证明,由于各方面条件的限制,依然存在许多可绕过这些保护机制,成功实施漏洞利用的技术手段[1]。

一般情况下,通过劫持程序控制流,跳转至指定内存地址,实现任意代码执行,需要在触发程序控制流劫持状态的同时,注入目标内存地址。地址随机化(Address Space Layout Randomization, ASLR)机制对加载于程序内存空间中的各模块进行随机化布局,导致攻击者无法准确定位目标代码的内存地址,从而阻止控制流劫持攻击[2-3]。但是,ASLR依然存在不少局限性[4-5]。受地址随机化的影响,内存中各模块的加载地址随机分布,但各模块的内部结构依然相对固定,是导致内存信息泄漏的重要原因。

随着程序分析技术的发展,近年来,出现多种针对二进制程序漏洞自动化分析与测试用例生成技术。早期的漏洞自动利用技术多依赖于对程序漏洞补丁的分析[6-8]。近年来出现了多种针对特殊类型漏洞利用及保护机制绕过的技术方案[9-10],但仍然缺少一款针对ASLR机制脆弱性进行自动化分析的方案。

文献[11]提出了基于符号执行的自动化程序漏洞利用(CRash analysis of Automatic eXploit,

* **收稿日期**:2018-10-19
基金项目:国家重点研发计划"网络空间安全"重点专项资助项目(2017YFB0802905)
作者简介:黄宁(1990—),男,广东广州人,博士研究生,E-mail:tsukimurarin@163.com;
黄曙光(通信作者),男,教授,博士,博士生导师,E-mail:809848161@qq.com

CRAX)方法。该方法使用选择性符号执行技术,引导程序运行并触发控制流劫持状态,生成控制流劫持点可达的输入测试用例。该方案的局限性在于,缺少对 ASLR 影响的分析,导致测试用例难以应用于 ASLR 环境下。

文献[12]针对控制流处于有效防御的环境时,提出了面向数据流的漏洞自动分析与利用方案 FlowStitch。该方案在不改变程序控制流的前提下,利用已知漏洞,通过构造特殊数据,实现利用代码自动生成。但是,由于 ASLR 机制对内存模块布局的随机化处理,导致数据流分析无法应用于 ASLR 脆弱性分析过程。

文献[13]提出了首个针对 Android 系统的进行符号执行,实现漏洞自动利用的方案 Centaur。该方案着重分析程序控制流状态变化过程,同时构造漏洞触发点可达的路径约束。该方案的局限性在于,未考虑漏洞触发与控制流劫持状态间的程序状态依赖,导致生成的测试用例不能实现控制流劫持攻击。

为了准确分析漏洞程序能否在 ASLR 环境下实现控制流劫持攻击,本文结合已有的漏洞自动分析与利用技术,针对四种内存泄漏场景,提出了一种基于控制流状态检测的 ASLR 脆弱性自动分析方法。

本文首先建立了有限状态机模型,描述被测程序的动态运行状态。然后,根据内存泄漏技术特点,筛选出 ASLR 绕过过程的状态依赖。在程序有限状态机模型的基础上,使用符号执行工具 S2E[14-15]实现了地址随机化脆弱性分析(Address Randomization Vulnerability Analysis, ARVA)系统。原型系统检查内存泄漏状态的路径约束与控制流劫持状态的路径约束是否兼容,判断被测程序是否存在同时满足上述两种状态的路径。

1　ASLR 环境下的程序有限状态机模型

根据常见的内存泄漏技术特点,结合污点数据传播和程序动态运行状态特征,本文提出了一种基于程序状态变迁的地址随机化脆弱性分析方法。该方法以上述各程序状态为分析单元,结合污点数据传播驱动程序动态运行过程,设计了针对 ASLR 环境下程序运行状态的有限状态机模型,如图 1 所示。该有限状态机(Finite States Machine, FSM)可用式(1)所示五元组进行描述:

$$M=(U,\Sigma,T,\delta,u_0) \tag{1}$$

其中:U 为有限状态集合,即全体程序状态集合;Σ 为有穷的事件输入集,即驱动程序动态运行的污点数据集合;T 为非空的转换集合;δ 为映射函数,$\delta=U_x\Sigma\rightarrow T$;$u_0$ 为初始状态,$u_0\in U$。其中,T 中的每个元素又可表示为式(2)所示五元组:

$$T=\{S\_State,T\_State,Event,Constraint,Action\} \tag{2}$$

式中:*S_State* 和 *T_State* 分别表示 T 的初始状态和目标状态;*Event* 表示 *S_State* 和 *T_State* 状态变迁依赖的输入事件,每次状态变迁依赖的 *Event* 属于 Σ 中的一个子集;*Constraint* 表示监护条件或事件 *Event* 的参数等约束条件;*Action* 表示状态变迁过程中的执行动作。*Action* 对于状态变迁过程不是必需的,当条件满足时,也可以在不执行任何动作的情况下实现程序状态变迁。

状态集合 U' 中各状态的状态变迁过程所依赖的事件输入 σ、约束条件 c 和执行动作 a 如下:

1)$State_{\text{Input}}$ 状态表示程序从开始接收外部数据(即污点源),到下一次任意污点数据传播或约束改变时刻之间的程序状态。外部输入事件可视为一种特殊的针对污点数据的操作事件 σ_{Input}。事件 σ_{Input} 与 $State_{\text{Input}}$ 状态转换的映射关系可如式(3)所示:

$$U_x\sigma_{\text{Input}}\rightarrow T:T\_State=State_{\text{Input}} \tag{3}$$

由于程序执行过程可能不止一次地触发事件 σ_{Input},且每次触发事件 σ_{Input} 均会驱动程序进入 $State_{\text{Input}}$ 状态,因此事件 σ_{Input} 和事件输入集合 Σ 的关系为:$\{\sigma_{\text{Input0}},\cdots,\sigma_{\text{Inputn}}\}\subseteq\Sigma$。

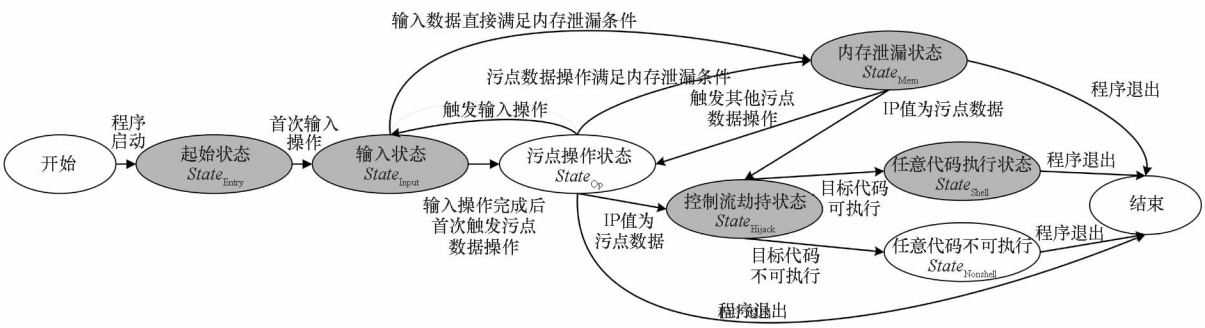


图 1　ASLR 环境下程序有限状态机模型

Fig. 1　Program finite state machine model in ASLR environment

前置状态向输入状态变迁过程中的执行动作 a_{input}：程序通过输入功能，将输入数据写入内存地址。

2）$State_{\text{Mem}}$是从程序满足内存泄漏条件的时刻到目标内存信息泄漏时刻间的程序状态。本文总结了四种典型内存泄漏场景的 $State_{\text{Mem}}$状态依赖条件。

①容错攻击[16-17]。

事件输入 σ_{Mem}：循环执行事件集合 $Event_{\text{Loop}}$；内存读取事件 $Event_{\text{ReadMem}}$；异常处理操作 $Event_{\text{Except}}$。

集合 $Event_{\text{Loop}}$ 中的元素为二元组（Pos，$Event$），其中 Pos 表示事件 $Event$ 的执行优先度，Pos 越小，优先度越高。

约束条件 c_{Mem}：$\{(Pos_1, Event_{\text{ReadMem}}), (Pos_2, Event_{\text{Except}})\} \subseteq Event_{\text{Loop}} \wedge (Pos_1 < Pos_2) \wedge Event_{\text{Loop}}$ 执行次数 $count$ 足够大。

执行动作 a_{Mem}：$Event_{\text{Loop}}$集合中的事件根据各自的执行优先级循环执行，直至内存读取事件 $Event_{\text{ReadMem}}$读取目标地址。

②格式化字符串[18]。

事件输入 σ_{Mem}：调用格式化字符串函数事件 $Event_{\text{Format}}$。

约束条件 c_{Mem}：格式化字符串函数的格式化控制符参数为污点数据。

执行动作 a_{Mem}：通过格式化字符串函数输出目标地址。

③任意地址读取。

事件输入 σ_{Mem}：内存读取事件 $Event_{\text{ReadMem}}$。

约束条件 c_{Mem}：事件 $Event_{\text{ReadMem}}$ 的读取对象为带长度信息的数据结构 $\wedge$（待读取数据结构长度信息为污点数据 $\vee$ 待读取数据结构指针为污点数据）。

执行动作 a_{Mem}：事件 $Event_{\text{ReadMem}}$ 读取指定内存信息。

④部分地址定位。

事件输入 σ_{Mem}：返回地址覆盖事件 $Event_{\text{RetCover}}$。

约束条件 c_{Mem}：事件 $Event_{\text{RetCover}}$可通过污点数据覆盖函数返回地址。

执行动作 a_{Mem}：覆盖返回地址的低地址部分。

3）$State_{\text{Hijack}}$表示了程序控制流被劫持时刻的程序状态。程序处于控制流劫持状态，可实现任意地址跳转。

状态所依赖的事件输入 σ_{Hijack}：指令指针（Instruction Pointer，IP）寄存器值为污点数据。

约束条件 c_{Hijack}：目标内存地址已泄漏。

执行动作 a_{Hijack}：将目标内存地址写入 IP 寄存器。

后置状态：$State_{\text{Shell}}$状态。

4）任意代码执行状态 $State_{\text{Shell}}$。该状态抽象描述了程序开始非法执行一段攻击者指定的代码到该段代码执行结束时刻间的程序状态。控制流劫持攻击的结果取决于跳转目的地址的可执行属性，当且仅当目标地址是可执行的，程序转入任意代码执行状态 $State_{\text{Shell}}$，表示控制流劫持攻击成功。

状态所依赖的事件输入 σ_{Shell}：IP 寄存器值指向目标内存地址。

约束条件 c_{Shell}：目标地址内存可执行。

前置状态 S_State_{Shell}：$State_{\text{Hijack}}$状态。

ASLR 环境下，基于内存泄漏的控制流劫持攻击要求程序状态集合 U'是全体程序状态集合 U 的子集。集合 U'至少要包含以下几种状态：

$$\{State_{\text{Entry}}, State_{\text{Input}}, State_{\text{Mem}}, State_{\text{Hijack}}, State_{\text{Shell}}\} \subseteq U'$$

2 原型系统实现

本文使用符号执行技术，实现目标程序动态运行过程中的污点数据跟踪、程序状态监测与地址随机化脆弱性分析等工作。分析工作架构如图 2所示。

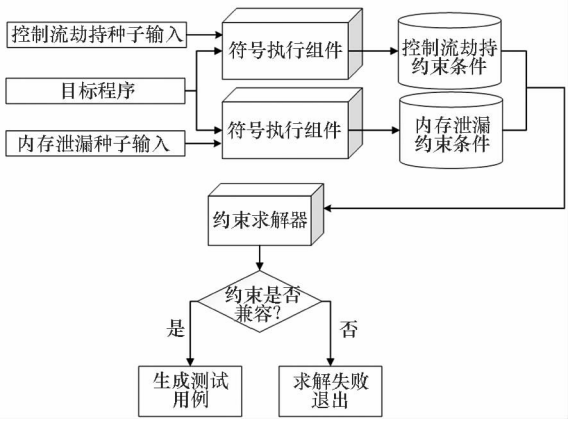


图 2 ASLR 脆弱性分析架构

Fig. 2 Structure of ASLR vulnerability analysis system

分析架构通过符号执行组件对动态运行的目标程序状态进行监视与分析，构造满足地址随机化机制绕过的程序状态约束，并由约束求解器求解约束，生成测试用例。符号执行组件需要三项输入：可执行文件格式的目标程序，可触发目标程序内存泄漏状态的种子输入以及可触发控制流劫持状态的种子输入。

内存泄漏种子文件为可触发程序内存泄漏状态的输入数据。通过该种子文件,引导程序在动态运行过程中触发内存泄漏,并收集从入口点到内存泄漏状态的约束条件,构建目标程序的内存泄漏约束。

控制流劫持种子文件为可触发目标程序控制流劫持状态的输入数据。该过程不考虑 ASLR 对控制流劫持攻击的影响。本文将所有由外部传入目标程序的污点数据标记为符号化数据。目标程序在动态运行过程中,所有被污点数据污染的内存空间或寄存器会被标记为符号化内存或寄存器。通过检查某一内存地址或寄存器的符号化属性,可判断内存地址或寄存器的可控性。

针对目标程序进行分析时,分别通过内存泄漏种子文件和控制流劫持种子文件驱动目标程序动态运行,生成内存泄漏状态约束 $Constraint_{\text{Mem}}$ 与控制流劫持状态约束 $Constraint_{\text{Hijack}}$。约束求解器求解上述约束间的兼容性,判断两者是否具备在同一程序路径触发的可能性。当且仅当上述两种约束满足式(4)所示关系时,表示两者互相兼容。

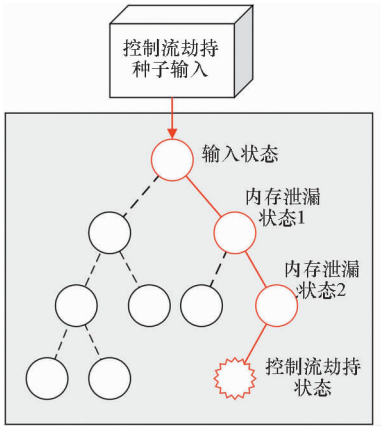
$$Constraint_{\text{Mem}} \wedge Constraint_{\text{Hijack}} = True \quad (4)$$

若两种约束互相兼容,表明目标程序存在至少一条路径,可同时抵达内存泄漏状态和控制流劫持状态,即目标程序满足 ASLR 环境下的控制流劫持攻击条件。

为了降低符号执行对目标程序控制流劫持点检测的时间开销,符号执行组件采用了经过路径选择算法优化的导向式符号执行技术。以种子输入作为目标程序的输入文件,引导目标程序沿着确定的程序路径动态运行,直至触发相应的程序状态。图 3 是通过种子输入引导符号执行触发相应程序状态的过程。

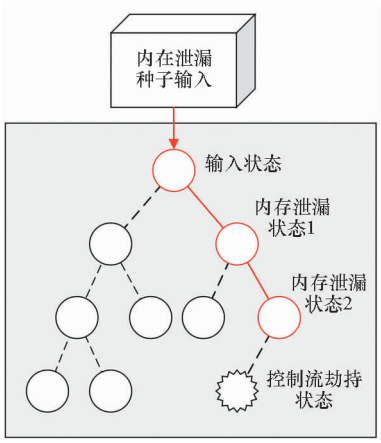
符号执行组件用于监视程序动态运行状态,并分析程序状态变迁过程是否满足基于内存泄漏的地址随机化绕过的依赖条件。在此过程中,符号执行组件收集程序运行的路径约束,并根据程序状态,构造满足状态变迁条件的数据约束。符号执行组件的工作架构如图 4 所示。

符号执行组件对程序状态分析的过程,是一种面向过程模式的 FSM 结构实现过程。当程序状态满足变迁约束,并触发状态变迁操作时,当前状态(源状态)执行退出动作。每个源状态的退出动作由两部分组成:根据源状态构造数据约束;根据事件及事件约束选择目标状态。源状态退出动作过程如算法 1 所示。



(a) 控制流劫持种子驱动程序运行过程

(a) Control-flow hijack seed input direct the running path of program



(b) 内存泄漏种子驱动程序运行过程

(b) Memory leakage seed input direct the running path of program

图 3　种子输入引导符号执行触发程序状态的过程

Fig. 3　Seed input trigger the program states in symbolic execution

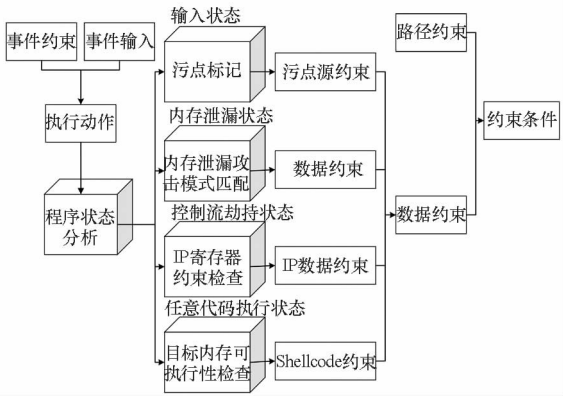


图 4　符号执行组件工作架构

Fig. 4　Structure of symbolic execution parts

算法 1 当前程序状态(源状态)退出过程

Alg. 1 Exit process of current program state (source state)

输入:事件输入 σ;事件约束 c;源状态 srcState
输出:数据约束 dataConstraint;目标状态 targetState

```
switch( srcState )
{
    case State_Input :
        data←外部数据
    targetState←执行算法 2 ( σ, c )
        break
    case State_Mem :
        data←被篡改的关键数据
        targetState ←执行算法 2 ( σ, c )
        break
    case State_Hijack :
        data←EIP 寄存器值
        targetState←执行算法 2 ( σ, c )
        break
    case State_Shell :
        data←shellcode 代码
        targetState←执行算法 2 ( σ, c )
        break
    default :
        targetState←执行算法 2 ( σ, c )
        break
}
dataConstraint←Eq(data, addr)
return dataConstraint, targetState
```

算法 1 通过下述操作为程序状态退出过程建立数据约束:

$$Constraint = Eq(Value, Addr)$$

其中:*Value* 为关键内存地址在相应的状态退出过程中需要满足的条件值;*Addr* 表示约束构建的目标内存地址或寄存器。

完成数据约束构造后,通过事件输入判断程序状态,实现程序状态变迁逻辑。该过程如算法 2所示。

3 实验评估

本文使用 S2E 作为原型系统 ARVA 的符号执行引擎,针对快速模拟器(Quick EMUlator, QEMU)下运行的目标程序及其系统环境进行全系统模拟的基础上,实现程序状态监测与 ASLR 脆弱性分析。本文选取了 8 个实际的漏洞利用攻击样本对 ARVA 原型系统进行评估。实验中,原型系统运行在配备了3.40 GHz Intel Core i7 - 6700 CPU、8 GB 内存、250 GB 硬盘的计算机上。

算法 2 程序状态变迁过程

Alg. 2 States conversion process

输入:事件输入 σ;事件约束 c
输出:目标状态 targetState

```
switch(σ )
{
    case σ_Input :
        a_Input ;
        targetState = State_Input ;
        break;
    case σ_Mem :
        a_Mem ;
        targetState = State_Mem ;
        break;
    case σ_Hijack :
        a_Hijack ;
        targetState = State_Hijack ;
        break;
    case σ_Shell :
        a_Shell ;
        targetState = State_Shell ;
        break;
    case σ_Op :
        a_Op ;
        targetState = State_Op ;
        break;
    default:
        break;
}
return targetState
```

目标程序与系统均运行于 QEMU 虚拟机中,各样本的漏洞编号、运行环境与目标程序如表 1 所示。

表 1 实验样本信息

Tab. 1 Information of experimental sample

实验样本	目标程序	运行环境
MS06 - 055	IE 6	Windows 2000
CVE - 2010 - 3333	Office 2003	Windows XP
CVE - 2012 - 1876	IE8	Windows 7
CVE - 2014 - 0322	Flash ActiveX 13, IE9	Windows 7
CVE - 2014 - 6332	IE10	Windows 7
CVE - 2015 - 3090	Flash ActiveX 13, IE9	Windows 7
CVE - 2015 - 5119	Flash ActiveX 13, IE9	Windows 7
CVE - 2015 - 5122	Flash ActiveX 13, IE9	Windows 7

3.1　ASLR 脆弱性分析结果

本文选取了近年来影响较大的控制流劫持攻击漏洞,对 ARVA 原型系统及其理论的有效性进行验证评估。实验中,内存泄漏种子文件触发各样本程序内存泄漏状态信息如表 2 所示。

表 2 中,ASLR 环境表示样本程序的依赖系统对程序内存空间的随机化效果。由于早期的 Windows 系统未设置 ASLR 机制,因此样本程序不受地址随机化的影响。Windows XP 系统的 ASLR 机制只能实现堆栈等部分内存空间的随机化效果,本文将该环境称为部分随机化环境。在 Windows 7 以后的系统中,ASLR 机制基本实现了针对全部内存空间的随机化效果,本文将实现这一效果的随机化环境称为整体随机化环境。最终泄漏目标地址表示内存泄漏种子文件引导样本程序执行至最后一次内存泄漏状态时,泄漏的目标地址。

由控制流劫持种子文件触发各程序控制流劫持状态信息如表 3 所示。

表 3 中,控制流劫持目标地址表示样本程序进入控制流劫持状态时,下一指令的跳转目的地址;目标跳转指令为该目的地址处对应的汇编指令;控制流劫持攻击类型表示样本程序在控制流劫持种子文件驱动下执行的攻击类型。

表 4 列举了各样本程序从初始状态到控制流劫持状态变迁过程中,$State_{Mem}$状态的触发次数以及每次触发该状态时泄漏的内存信息。

根据表 2、表 3 与表 4 的信息,仅 MS - 06 - 055 与 CVE - 2010 - 3333 的控制流劫持攻击选用了覆盖返回地址的方式。原因如下:

1) Windows 2000 未设置任何地址随机化机制。MS - 06 - 055 的控制流劫持种子可通过 JavaScript 实现堆喷射布局 shellcode 以及固定值 0×0c0c0c0c 覆盖返回地址,导致任意代码执行。这一过程不涉及内存泄漏。

表 2　样本程序内存泄漏状态信息

Tab. 2　Information of memory leakage of experimental sample

实验样本	ASLR 环境	内存泄漏类型	内存泄漏状态的前置操作	最终泄漏目标地址
MS06 - 055				
CVE - 2010 - 3333	堆栈	部分地址定位	确定跳转指令 jmp esp	栈顶指针
CVE - 2012 - 1876	整体	任意地址读写	修改 BSTR 字符串长度	mshtml. dll 基地址
CVE - 2014 - 0322	整体	容错攻击	重复搜索可读内存	Kernel32. dll 基地址
CVE - 2014 - 6332	整体	任意地址读写	篡改 Safe Array 数组长度	[COleScript] + 0×174
CVE - 2015 - 3090	整体	任意地址读写	修改 Flash vector 数组长度	Flash32. dll 基地址
CVE - 2015 - 5119	整体	任意地址读写	修改 Flash vector 数组长度	Flash32. dll 基地址
CVE - 2015 - 5122	整体	任意地址读写	修改 Flash vector 数组长度	Flash32. dll 基地址

表 3　样本程序控制流劫持状态信息

Tab. 3　Information of control-flow hijack of experimental sample

实验样本	控制流劫持目标地址	目标跳转指令	控制流劫持攻击类型	任意代码执行类型
MS06 - 055	0×0c0c0c0c	shellcode	覆盖返回地址	shellcode
CVE - 2010 - 3333	0×7d1f5fb7	jmp esp	覆盖返回地址	shellcode
CVE - 2012 - 1876	mshtmlBase + 0×1031	retn	ROP	ROP
CVE - 2014 - 0322	Kernel32Base + 0×4c43a	VirtualAlloc 函数地址	ret2libc	执行库函数
CVE - 2014 - 6332			DVE	脚本执行任意功能
CVE - 2015 - 3090	Kernel32Base + 0×4c43a	VirtualAlloc 函数地址	ret2libc	执行库函数
CVE - 2015 - 5119	Flash32Base + 0×4fe4b	mov eax, [edx]	ROP	ROP
CVE - 2015 - 5122	Flash32Base + 0×4fe4b	mov eax, [edx]	ROP	ROP

表4 基于内存泄漏的样本程序状态变迁过程

Tab. 4 States conversion of memory leakage of experimental sample

实验样本	第一次	第二次	第三次	内存泄漏约束与控制流劫持约束兼容性
MS06 – 055				
CVE – 2010 – 3333	栈顶指针			兼容
CVE – 2012 – 1876	CButtonLayout 虚表指针	mshtml 基地址		兼容
CVE – 2014 – 0322	Kernel32 基地址			兼容
CVE – 2014 – 6332	CScriptEntryPoint 对象指针	COleScript 对象指针	IE 安全模式标志位	兼容
CVE – 2015 – 3090	sound 对象虚表指针	Flash32 基地址	Kernel32 基地址	兼容
CVE – 2015 – 5119	sound 对象虚表指针	Flash32 基地址		兼容
CVE – 2015 – 5122	sound 对象虚表指针	Flash32 基地址		兼容

2) Windows XP SP3 设置了针对堆栈等部分内存区域的地址随机化。CVE – 2010 – 3333 实验中,控制流劫持种子将 Shellcode 布局于栈内存中。但由于堆栈随机化影响,需要通过跳板指令 jmp esp 确定栈内存的位置(即定位栈空间地址的前四位)。Office 2003 中,存在不受随机化影响的地址 0x7d1f5fb7 固定为 jmp esp 指令。当程序处于控制流劫持状态时,将返回地址覆盖为 0x7d1f5fb7,执行跳板指令后,可定位栈空间前四位地址,驱动样本程序开始执行栈内存中的 Shellcode,触发任意代码执行状态。

CVE – 2014 – 0322、CVE – 2015 – 3090、CVE – 2015 – 5119 与 CVE – 2015 – 5122 实验分别验证了整体随机化环境下的漏洞可利用性。上述漏洞均可通过 Action Script 脚本触发 IE 的 Flash 插件漏洞并实现漏洞利用。本文分别选取 Kernel32. dll 与 Flash32. dll 模块作为最终内存泄漏目标。内存泄漏种子文件可覆盖 Flash Vector 数组长度,导致任意内存读写。上述样本程序中,基于容错攻击的 CVE – 2014 – 0322 触发了一次内存泄漏状态;其余样本至少触发两次以上内存泄漏状态。

与其他实验相比,CVE – 2014 – 6332 实验涉及特殊的 IE 浏览器沙盒机制。本文针对该机制构造的控制流劫持种子文件在触发样本控制流劫持状态后,不会转入汇编指令层面的任意代码执行状态。控制流劫持种子文件通过将 IE 浏览器安全标志置于位置 0,使用 VBScript 脚本可调用系统的任意功能。该攻击方法被称为数据虚拟执行 DVE。

根据表 4 的信息,除 MS – 06 – 055 实验不涉及 ASLR 环境外,其余各实验的内存泄漏约束与控制流劫持约束的判定结果均为互相兼容。该结果表明,这些实验样本均至少拥有一条程序路径,同时满足内存泄漏和控制流劫持的条件。在此路径下,ASLR 机制可被绕过。

上述实验表明, ARVA 原型系统可有效识别部分地址定位、容错攻击、任意地址读写等内存泄漏状态,以及覆盖返回地址、ROP、ret-to-libc 等控制流劫持类型。另一方面,ARVA 通过求解内存泄漏状态约束与控制流劫持状态约束,可判断两者的约束是否兼容。实验结果表明,基于内存泄漏的 ASLR 绕过过程中,可能需要触发不止一次内存泄漏状态,才能非法获取目标内存信息。

3.2 案例分析

本文挑选了 CVE – 2014 – 6332 案例来对 ARVA 的检测过程与效果进行详细阐述。此漏洞利用任意地址读写实现 ASLR 绕过;通过 DVE, Heap Spray 等攻击技术实现控制流劫持。本文的系统能准确识别样本程序的内存泄漏状态与控制流劫持状态,构造并判断两种状态约束的兼容性。

案例 CVE – 2014 – 6332 漏洞程序 ASLR 脆弱性分析。此漏洞是一个整数溢出漏洞。该漏洞可通过篡改 IE 浏览器的安全模式标志位,绕过浏览器沙盒保护,进而导致脚本文件获取任意功能执行权限。

CVE – 2014 – 6332 实验的种子文件为 VBScript 脚本文件。其中,内存泄漏种子输入的目标为:引导漏洞程序在 ASLR 环境下准确定位 IE 安全模式标志位的地址;控制流劫持种子文件的目标为:引导漏洞程序执行任意功能。

IE 浏览器通过 OLEAUT32. dll 对 VBScript 中的数组进行管理。假设有数组 *arr*,其初始长度为

$a0$;后又将数组 *arr* 的长度更改为 $a1 = a0 + 0 \times 80000000$, OLEAUT32 将根据 $a1 - a0 = 0 \times 80000000$ 计算数组 *arr* 的新索引值。由于 0×80000000 被系统默认为有符号整数,换算成十进制数为 0,因此数组 *arr* 的实际大小仍为 $a0$。此时,通过 *arr* 的索引值,可实现越界读写。

内存泄漏种子文件触发的程序状态变迁过程为:

$State_{Input}$状态:向进程内存中布置两个错位分布的污点数据数组 *arr*A 与 *arr*B。两个数组在内存中的布局如图 5 所示。

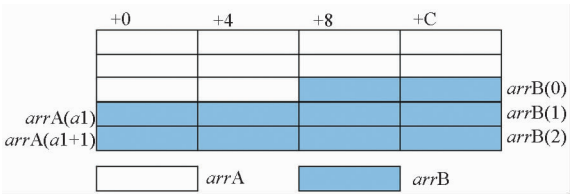


图 5　数组 *arr*A 与 *arr*B 结构分布

Fig. 5　Layout of *arr*A and *arr*B

构建数据约束: $Constraint1 = Eq(arrA, \&arrA) \wedge Eq(arrB, \&arrB)$。

VBScript 语言的数据存储时,每个数据都占据 16 字节,其中,前 8 个字节表示数据类型,后 8 个字节表述数据值。

$State_{Input}$状态:通过种子文件布局一个 sub 类型函数 *func*,并将函数地址 *&func* 存储于 $arrA(a1)$。

构建数据约束: $Constraint2 = Eq(\&func, \&arrA(a1))$。

$State_{Input}$状态: $typeof(\&func) = 0x1$(NULL 类型)。构建数据约束 $Constraint3 = Eq(0x1, \&arrA(a1-1))$。

$State_{Input}$状态: $arrB(2) = 1.740\,885\,347\,313\,24E-310$($arrB(2)$数据值 = $arrA(a1+2)$类型值 = $0 \times 200C$)。

构建数据约束: $Constraint4 = Eq(1.740\,885\,347\,313\,24E-310, \&arrB(2))$。

$State_{Input}$状态:布局 Safe Array 型数组 *myArr* 起始地址为 0×0,包含 $0 \times 7fffff0$ 个元素,每个元素大小为 1 Byte。

构建数据约束: $Constraint5 = Eq(myArr, \&myArr)$。

$State_{Input}$状态: $arrA(a1+2) = \&myArr$。

构建数据约束: $Constraint6 = Eq(\&myArr, arrA(a1+2))$。

检查任意地址读取类型依赖的约束条件,数组 *arr*A 满足下述条件:

数组 *arrA* 带长度信息的数据结构 ∧ 数组 *arr*A 长度信息 $a1$ 为污点数据。

因此,ARVA 判断数组 *arr*A 满足任意地址读取的触发条件。

$State_{Mem}$状态:利用 *arr*A 数组越界读写泄漏 *&func* + 12 处 CScriptEntryPoint 对象指针。

构建数据约束: $Constraint7 = Eq(\&CScriptEntryPoint, \&func + 12)$。

$State_{Mem}$状态:泄漏 &CScriptEntryPoint + 20 处 COleScript 对象指针。

构建数据约束: $Constraint8 = Eq(\&COleScript, \&CScriptEntryPoint + 20)$。

$State_{Mem}$状态:泄漏 &COleScript + 0×174 处的 IE 安全模式标志位地址。

构建数据约束: $Constraint9 = Eq(\&SecurityBit, \&COleScript + 0 \times 174)$。该地址在正常权限下仅可通过 Safe Array 数组进行写操作。

$State_{Input}$状态: $myArr[\&SecurityBit] = 0$。

构建数据约束: $Constraint10 = Eq(0, myArr[\&SecurityBit])$。

内存泄漏的路径约束构建过程中,共触发 6 次 $State_{Input}$ 状态,3 次 $State_{Mem}$ 状态,并在触发 $State_{Mem}$状态时针对状态依赖的事件输入与约束条件进行了 1 次检查。

ARVA 在该过程中建立的污点源数据约束如式(5)所示。

$$srcConstraint = Constraint1 \wedge Constraint2 \wedge Constraint3 \wedge Constraint4 \wedge Constraint5 \wedge Constraint6 \wedge Constraint10 \tag{5}$$

ARVA 建立的内存泄漏数据约束如式(6)所示。

$$memConstraint = Constraint7 \wedge Constraint8 \wedge Constraint9 \tag{6}$$

当该漏洞程序处于任意代码执行状态 $State_{Shell}$时,指定的代码为 VBScript 脚本文件中任意功能调用代码。由于 DVE 利用技术的特殊性,可认为该案例中的控制流劫持状态 $State_{Hijack}$ 与 $State_{Shell}$是同时触发的。控制流劫持种子文件触发的程序状态变迁过程为:

$State_{Hijack}$状态:ShellExecute "cmd. exe"。

CVE - 2014 - 6332 的最终数据约束如式(7)所示:

$$dataConstraint = Constraint_{Mem} \wedge Constraint_{Hijack} = srcConstraint \wedge memConstraint \wedge$$

$$Constraint_{\text{Hijack}} \tag{7}$$

其中，$Constraint_{\text{Hijack}}$因不涉及 IP 寄存器的污点传播，其默认值为 *True*。求解约束 *dataConstarint* 值为 *True*，表示内存泄漏约束 $Constraint_{\text{Mem}}$与控制流劫持约束 $Constraint_{\text{Hijack}}$兼容，因此可得出结论，CVE－2014－6332 可通过内存泄漏绕过地址随机化，实现控制流劫持攻击。

3.3 时间开销分析

在原型系统 ARVA 中，测量时间 $t1$ 定义为内存泄漏种子输入驱动样本程序开始运行，至求得内存泄漏约束所用时间；$t2$ 定义为控制流劫持种子输入驱动程序开始运行，至求得控制流劫持约束所用时间。同时，本文也在原生 S2E 系统中，以结合了 ASLR 绕过和控制流劫持功能的脚本文件为输入，运行同样的样本程序，并记录样本程序的运行时间 t'。具体数据如图 6 所示。

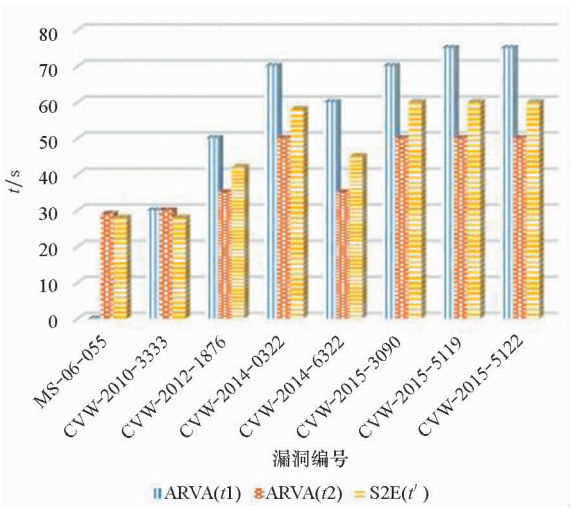


图 6 ARVA 与原生 S2E 运行时间开销对比

Fig. 6 Comparison of running time of ARVA and S2E

根据图 6 数据，MS－06－055 不涉及地址随机化绕过，ARVA 针对其控制流劫持约束求解时间 $t2$ 大于 S2E 的运行时间。CVE－2010－3333 的使用部分地址定位，内存泄漏约束求解过程与控制流劫持约束求解过程基本一致，两者的时间 $t1$ 与 $t2$ 基本相等，与 S2E 的运行时间 t'也基本持平。

CVE－2012－1876、CVE－2014－0322、CVE－2014－6332、CVE－2015－3090、CVE－2015－5119 与 CVE－2015－5122 等实验时间开销基本呈现下述两种特点：$t1 > t' > t2$；$t1 + t2 \approx 2t'$。

上述特点说明，ARVA 针对样本程序的地址随机化脆弱性检测的时间开销主要集中于内存泄漏约束的构建与求解方面。

此外，由于 ARVA 的控制流劫持约束求解过程不考虑地址随机化对样本程序的影响，而原生 S2E 系统在运行同一样本程序时，种子文件结合了地址随机化绕过与控制流劫持两方面功能，导致 ARVA 的控制流劫持约束求解时间 $t2$ 小于 S2E 运行时间 t'。

4 结论

本文提出了一种基于有限状态机的 ASLR 机制脆弱性分析方法。该方法能够分析目标程序是否可通过内存泄漏技术绕过 ASLR 保护，实现控制流劫持攻击。本文使用有限状态机模型描述程序状态变迁过程，分析通过内存泄漏导致 ASLR 绕过的程序状态依赖，在此基础上，针对四种常见的内存泄漏场景分别建立了内存泄漏状态的进入与退出条件，使该模型更好地适用于不同的漏洞攻击模式。根据上述理论与模型，本文实现了一套基于符号执行工具 S2E 的 ASLR 脆弱性分析原型系统 ARVA。该系统针对目标程序的内存泄漏状态与控制流劫持状态分别进行约束构建，并对二者的兼容性进行求解，检查目标程序是否满足地址随机化环境下实施控制流劫持攻击的条件。通过对若干实际漏洞程序的实验表明，本文方法能够准确检测到被测程序的动态运行状态，求解相关程序状态约束，并有效识别 ASLR 环境下漏洞程序的可利用性。

参考文献(References)

[1] 王明华，应凌云，冯登国. 基于异常控制流识别的漏洞利用攻击检测方法[J]. 通信学报，2014，35(9)：20－31.
WANG Minghua，YING Lingyun，FENG Dengguo. Exploit detection based on illegal control flow transfers identification[J]. Journal on Communications，2014，35(9)：20－31.(in Chinese)

[2] 邵思豪，高庆，马森，等. 缓冲区溢出漏洞分析技术研究进展[J]. 软件学报，2018，29(5)：1177－1198.
SHAO Sihao，GAO Qing，MA Sen，et al. Progress in research on buffer overflow vulnerability analysis technologies[J]. Journal of Software，2018，29(5)：1177－1198.(in Chinese)

[3] Bhatkar E，Duvarney D C，Sekar R. Address obfuscation：an efficient approach to combat a broad range of memory error exploits[C]//Proceedings of the 12th Conference on USENIX Security Symposium，2003：105－120.

[4] Payer M，Gross T R. String oriented programming：when ASLR is not enough[C]// Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop，2013：1－9.

(下转第 185 页)

with measurement error [J]. Energies, 2014, 7: 520 – 547.
[11] Feng L, Wang H L, Si X S, et al. A state-space-based prognostic model for hidden and age-dependent nonlinear degradation process [J]. IEEE Transactions on Automation Science and Engineering, 2013, 10(4): 1072 – 1086.
[12] Wang H X, Jiang Y. Performance reliability prediction of complex system based on the condition monitoring information[J]. Mathematical Problems in Engineering, 2013: 1 – 7.
[13] Wang X L, Jiang P, Guo B, et al. Real-time reliability evaluation based on damaged measurement degradation data[J]. Journal of Central South University, 2012, 19(11): 3162 – 3169.
[14] Wang W B, Carr M, Xu W J, et al. A model for residual life prediction based on Brownian motion with an adaptive drift[J]. Microelectronics Reliability, 2010, 51(2): 285 – 293.
[15] Wei M H, Chen M Y, Zhou D H. Multi-sensor information based remaining useful life prediction with anticipated performance [J]. IEEE Transactions on Reliability, 2013, 62(1): 183 – 198.
[16] Peng W, Coit D W. Reliability and degradation modeling with random or uncertain failure threshold [J]. Institute of Electrical and Electronics Engineers, 2007: 392 – 397.
[17] Usynin A, Hines J W, Urmanov A. Uncertain failure thresholds in cumulative damage models[C]// Proceedings of the Reliability and Maintainability Symposium, 2008: 334 – 340.
[18] Huang J B, Kong D J, CuiL R. Bayesian reliability assessment and degradation modeling with calibrations and random failure threshold [J]. Journal of Shanghai Jiaotong University (Science), 2016, 21(4): 478 – 483.
[19] Goebel K, Agogino A. Mill data set [DB/OL]. [2018 – 10 – 12]. USA: NASA, 2007. http://ti. arc. nasa. gov/project/prognostic -data-repository.

(上接第 170 页)

[5] Evtyushkin D, Ponomarev D, Abu-Ghazaleh N. Jump over ASLR: attacking branch predictors to bypass ASLR [C]// Proceedings of 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016: 1 – 13.
[6] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: techniques and implications [C]// Proceedings of IEEE Symposium on Security and Privacy (sp 2008), 2008: 143 – 157.
[7] Wang M H, Su P R, Li Q, et al. Automatic polymorphic exploit generation for software vulnerabilities [C]// International Conference on Security and Privacy in Communication Systems. Springer, 2013: 216 – 233.
[8] Long F, Rinard M. Automatic patch generation by learning correct code [C]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming, 2016, 51(1): 298 – 312.
[9] Schwartz E J, Avgerinos T, Brumley D. Q: exploit hardening made easy[C]//Proceedings of the 20th USENIX Conference on Security, 2011: 25 – 41.
[10] Heelan S, Melham T, Kroening D. Automatic heap layout manipulation for exploitation[C]// Proceedings of the 27th USENIX Security Symposium, 2018: 763 – 779.
[11] Huang S K, Huang M H, Huang P Y, et al. Software crash analysis for automatic exploit generation on binary programs[J]. IEEE Transactions on Reliability, 2014, 63(1): 270 – 289.
[12] Hu H, Chua Z L, Adrian S, et al. Automatic generation of data-oriented exploit [C]// Proceedings of 24th USENIX Security Symposium, 2015: 177 – 192.
[13] Luo L N, Zeng Q, Cao C, et al. System service call-oriented symbolic execution of android framework with applications to vulnerability discovery and exploit generation [C]// Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017: 225 – 238.
[14] Chipounov V, Kuznetsov V, Candea G. S2E: a platform for in-vivo multi-path analysis of software systems [J]. ACM SIGPLAN Notices, 2011, 47(4): 265 – 278.
[15] Chipounov V, Kuznetsov V, Candea G. The S2E platform: design, implementation, and applications [J]. ACM Transactions on Computer Systems, 2012, 30(1): 1 – 49.
[16] 彭建山, 丁大钊, 王清贤. 结合容错攻击与内存区域统计的 ASLR 绕过方法[J]. 计算机工程与应用, 2019(2): 72 – 78.
PENG Jianshan, DING Dazhao, WANG Qingxian. An ASLR bypassing method combining crash-resistance and memory range statistics[J]. Computer Engineering and Applications, 2019(2): 72 – 78. (in Chinese)
[17] Gawlik R, Kollenda B, Koppe P, et al. Enabling client-side crash-resistance to overcome diversification and information hiding[C]// Proceedings of Network and Distributed System Security Symposium, 2016.
[18] Shankar U, Talwar K, Foster J S, et al. Detecting format string vulnerabilities with type qualifiers[C]// Proceedings of the 10th Conference on USENIX Security Symposium, 2001.