计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2019.20190341
56(11): 2299-2314, 2019

有限资源条件下的软件漏洞自动挖掘与利用

黄桦烽[1,2] 王嘉捷[3] 杨 轶[1,2] 苏璞睿[1,2] 聂楚江[1,2] 辛 伟[3]
[1](中国科学院软件研究所可信计算与信息保障实验室 北京 100190)
[2](中国科学院大学计算机科学与技术学院 北京 100190)
[3](中国信息安全测评中心 北京 100085)
(huafeng@iscas.ac.cn)

Automatic Software Vulnerability Discovery and Exploit Under the Limited Resource Conditions

Huang Huafeng[1,2], Wang Jiajie[3], Yang Yi[1,2], Su Purui[1,2], Nie Chujiang[1,2], and Xin Wei[3]
[1](*Trusted Computing and Information Assurance Laboratory*, *Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100190)
[3](*China Information Technology Security Evaluation Center*, *Beijing* 100085)

Abstract Vulnerabilities are the core elements of system security and attack-defense confrontation. The automatic discovery, analysis and exploit of vulnerabilities has been a hot and difficult issue for a long time. The related researches mainly focus on fuzzing, propagate taint analysis and symbolic execution. On one hand, current solutions focus on different aspects of vulnerability discovery, analysis and exploit, which lack systematic researches and implementations. On the other hand, current solutions ignore the feasibility of limited resources under the realistic environment. Inside, the fuzzing is mainly based on large-scale server cluster system implementation, and the methods of propagate taint analysis and symbolic execution have high time and space complexity, which are prone to state explosion. Counter the problem of vulnerability automatic discovery and exploit under the limited resources, a program dynamic runtime Weak-Tainted model is established, then a complete solution for automatic vulnerability discovery, analysis and exploit is presented. The paper optimizes and enhances the ability of propagate taint analysis, and proposes a method for input solving based on output feature feedback, and any other analysis solutions under the limited resources to improve the ability and efficiency of vulnerability discovery, analysis and exploit. The paper designs and implements the vulnerability discovery and exploit automatic prototype system, which can concurrent 25 tasks for fuzzing, and propagate taint analysis and input solving with one server. The paper tests experiments on the samples of the 2018 BCTF competition, and the results show that the method of input solving in this paper is superior to ANGR for solving the atoi, hex and base64 encoding. The efficiency of vulnerability discovery is improved 45.7% higher than AFL, and 24 of the 50 samples can generate exploits automatically successfully. The advantages of Weak-Tainted vulnerability description model for vulnerability discovery and exploit are verified.

Key words vulnerability; fuzzing; taint propagate; symbolic execution; input solving; automatic exploit

收稿日期:2019-06-06;**修回日期**:2019-08-01

基金项目:**国家自然科学基金项目**(U1736209,U1636115,U1836117,U1836113,61572483)
This work was supported by the National Natural Science Foundation of China (U1736209, U1636115, U1836117, U1836113, 61572483).

通信作者:**杨轶**(yangyi@iscas.ac.cn)

摘 要 漏洞是系统安全与攻防对抗的核心要素,漏洞的自动发现、分析、利用是长期以来研究的热点和难点,现有研究主要集中在模糊测试、污点分析、符号执行等方面.当前研究一方面主要从漏洞的发现、分析和利用的不同环节提出了一系列解决方案,缺乏系统性的研究和实现;另一方面相关方法未考虑现实环境的有限资源条件,其中模糊测试主要基于大规模的服务器集群实施,污点分析和符号执行方法时间与空间复杂度高,且容易出现状态爆炸.针对有限资源条件下的漏洞自动挖掘与利用问题,建立了Weak-Tainted程序运行时漏洞模型,提出了一套面向漏洞自动挖掘、分析、利用的完整解决方案;提出了污点传播分析优化方法和基于输出特征反馈的输入求解方法等有限资源条件下的分析方案,提升了漏洞挖掘分析与利用生成能力;实现了漏洞自动挖掘和利用原型系统,单台服务器设备可并发运行25个漏洞挖掘与分析任务.对2018年BCTF比赛样本进行了实验对比测试,该输入求解方法在求解atoi,hex,base64编码的能力均优于ANGR,同等漏洞挖掘能力条件下效率比AFL提高45.7%,测试的50个样本中有24个能够自动生成利用代码,验证了Weak-Tainted漏洞描述模型用于漏洞自动挖掘和利用生成的优势.

关键词 漏洞;模糊测试;污点传播;符号执行;输入求解;漏洞自动利用

中图法分类号 TP311

软件漏洞是指存在于操作系统或者应用程序中,可导致攻击者在未授权情况下获取或破坏系统数据的安全缺陷.软件漏洞的成因主要包括软件开发人员疏忽及水平受限、编译器引擎存在缺陷、程序功能逻辑复杂、代码模块测试不充分等因素.鉴于软件的代码规模、功能组成、多线程并发、数据资源共享等高度复杂的机制,即使经过严格的代码审查与测试也难以消除软件中的安全漏洞.知名压缩软件Winrar被爆出的CVE-2018-20250等一系列漏洞已潜伏长达19年,通过该漏洞可以实现任意代码执行,实现信息数据窃取、系统破坏、敲诈勒索等恶意攻击,严重威胁到信息系统的安全.

软件漏洞是网络安全的主要威胁,国家计算机网络应急技术处理协调中心(CNCERT)联合国内多家单位建立了国家信息安全漏洞共享平台(China National Vulnerability Database, CNVD),建立软件安全漏洞统一收集验证、预警发布及应急处置体系.漏洞利用是系统入侵渗透的主要手段,如何提高软件漏洞的自动化挖掘与利用能力是攻防双方共同关注的焦点.在有限资源条件下实现软件漏洞的自动利用是未来的现实需求和技术发展方向,2014—2016年美国DARPA组织的CGC[1](Cyber Grand Challenge)比赛在该方向作了初步的尝试,并受到业界的广泛关注.

软件漏洞的自动挖掘与利用实际已有长久的研究历史.在软件漏洞挖掘方面,微软的Cui等人[2]提出了基于符号执行反馈的并行化Fuzzing方法;Zalewski等人[3]提出了AFL Fuzzing系统,实现了基于遗传算法的路径快速触发,有效提高了漏洞挖掘能力.在软件漏洞分析方面,加州伯克利的Song等人[4]提出了基于动态污点传播分析的漏洞分析平台BitBlaze,之后该团队又提出了比特粒度的DECAF[5]污点传播分析系统.漏洞利用方面代表工作包括AEG[6],APEG[7]漏洞利用自动生成系统等.近年来,人工智能方法被引入到漏洞挖掘研究中,代表性工作包括Xu等人[8]提出的Gemini,通过对代码进行深度学习建模的方式,检测程序中存在的漏洞函数.相关方法和系统有效提高了漏洞挖掘与利用能力,但仍存在较大的局限性.主要包括:1)缺少统一的模型和框架,漏洞挖掘、漏洞分析、漏洞利用环节的方法难以形成一体化的能力,各环节的结果难以互相支撑;2)资源消耗大,难以支撑实际需求,现有的漏洞挖掘采取并行集群,需要大量计算资源;3)漏洞分析和漏洞利用基于传统污点传播、符号执行,时间及空间复杂度高,分析效率低,常因计算资源不足失效;4)漏洞自动利用生成只支持控制流劫持等有限的类型,扩展类型支持工作量大.

CGC是实现漏洞挖掘、分析、利用完全自动化的一次尝试,参赛团队通过建立“自动攻击防御系统”,在无人干预的条件下,自动寻找程序漏洞、生成利用攻击敌方、以及部署补丁抵御敌方的攻击,目的在于探索完全无人干预条件下的网络入侵、网络防御方面的可行性方案,构建出一套具备自主攻击及自我防御的软件.该比赛是软件漏洞挖掘与自动化利用的里程碑,从系统实现的角度证明了智能化自动攻防的可行性.但是,CGC比赛中采用了裁剪后

的操作系统,程序的可执行文件格式等均进行了一定的修改,距离实际应用系统仍有较大的差异,难以应用于实际问题的分析.

漏洞自动挖掘与利用涉及多个环节,需要将各环节有效结合才能发挥作用,本文提出了基于程序运行时信息的 Weak-Tainted 漏洞描述模型,在该模型下优化污点传播方法和输入求解方法,基于 2 种优化方法指导模糊测试数据生成、漏洞自动分析及漏洞利用自动构造,构建了一套自动化的有限资源条件下的漏洞挖掘与利用框架,实现了轻量级的漏洞挖掘和利用,有效提高漏洞挖掘与利用的效率和准确性,并基于 Ubuntu 现实系统下的系列实验,验证了本文方法的有效性.

本文主要贡献和创新点有 4 个方面:

1) 提出了基于程序运行时信息的 Weak-Tainted 漏洞描述模型,通过脆弱性和污点属性对漏洞进行定义和描述,该模型支持漏洞挖掘、分析、利用过程中对漏洞进行统一描述,解决当前漏洞研究各环节的结果难以互相支撑的问题,形成了一体化的描述规范;

2) 提出了字节级标签的高效动态污点传播分析优化方法,基于多级页表存储污点结构属性信息,通过值校验消除传播误标记,解决了现有污点传播分析方法时间与空间复杂度高,难以应用于有限资源环境的问题,同时提高了污点传播准确度;

3) 提出了基于输出特征反馈的输入求解方法,基于输入点和输出点的关系统计分析预判进行输入求解和验证,解决了字符表映射运算的符号表达式难以构建导致符号执行求解失败的问题,降低了计算资源消耗,同时提高了输入求解的成功率,支撑漏洞挖掘与漏洞利用生成所需的输入求解能力;

4) 实现了漏洞自动挖掘与利用生成原型系统,支持漏洞描述模型规则的定义和扩展,针对真实系统中的漏洞程序,验证了字节级标签动态污点传播方法、基于输出特征反馈的输入求解方法、程序动态运行时 Weak-Tainted 漏洞描述模型的有效性.

1 相关工作

软件漏洞自动挖掘目前主要基于模糊测试方法,其基本思想是通过变异算法提供各种非预期输入,启发程序执行更多的代码模块和路径,监视目标程序在处理该输入是否出现异常的动态测试方法.经过多年的发展,模糊测试可分为四大类型,包括:1)测试用例随机生成,如 zzuf[9];2)基于目标静态数据结构信息生成测试用例,如基于已知文件格式和网络协议规范,代表性研究工作有 Peach,Sulley[10]等;3)基于遗传迭代的代码覆盖率反馈变异,代表工作有 honggfuzz[11]和 AFL[3]等;4)基于符号执行和求解生成用例,遍历程序执行路径,代表工作有 KLEE[12],SAGE[13]以及商用版本 SpringField 等.这些代表的模糊测试工作仅具有单独的漏洞挖掘能力,形成的数据难以有效支撑后续的漏洞分析与漏洞利用自动生成工作.

软件漏洞自动利用目前主要基于模糊测试技术和数据流分析方法求解生成使程序陷入可利用状态的输入数据,再根据该输入和状态自动求解生成漏洞利用代码.而可利用状态通常会伴随着内存访问越界、格式化字符串受恶意操作、系统调用接口参数受恶意操作等异常情况出现,基于这些异常信息进行利用构造,最终实现控制流劫持等方式的任意代码执行.漏洞自动利用的代表性工作有 2011 年 Avgerinos 等人[6]提出的 AEG,该工作使用了指令动态插桩技术和符号执行进行漏洞利用样本自动生成,验证了漏洞自动利用生成的可行性.但是,该方案依赖于程序源代码进行程序错误静态搜索,即可利用状态的搜索,不适用于非开源的二进制程序.

2012 年 Cha 等人[14]针对二进制程序提出了构建基于索引的内存优化模型,优化符号化内存的加载问题,提高符号执行的效率,形成漏洞自动挖掘与利用代码自动生成的方案 Mayhem.该方案在检测可利用状态方面,主要包括缓冲区溢出和格式化字符串 2 方面,针对指令指针符号化检测和构造控制流劫持的漏洞利用.但是 Mayhem 只对 30 个 Linux 系统调用和 12 个 Windows 库函数进行建模分析,无法高效处理复杂的程序,无法处理未建模系统调用函数引起的符号执行计算错误问题.

在漏洞利用样本多样化生成方面,2013 年 Wang 等人[15]提出了一套针对控制流劫持类漏洞的利用样本多样性自动生成方法 PolyAEG.该方法的基本思想是基于动态污点分析发现所有的程序控制流劫持点,然后基于等效指令替换构建不同的控制流转移模式,实现漏洞利用样本的多样性生成.PolyAEG 针对控制流劫持类漏洞实现了一套完整的漏洞利用样本多样化生成的自动化构造方案,测试验证了 8 个现实漏洞程序,其中针对单个控制流劫持漏洞最多生成了 4 724 个利用样本变种.该方案是一种较为有效的控制流劫持类漏洞自动利用生成

方法,但对于尚未触发控制流劫持点的 POC 样本难以生成有效的利用,而当前模糊测试生成的 POC 大多未触发控制流劫持.

在堆漏洞利用方面,He 等人[16]提出了基于堆溢出崩溃和溢出修复的堆漏洞可利用性评估方法 HCS IFTER,该方法首次提出了通过修复被破坏的内存,使得程序能够继续执行,并检测被破坏的内存数据后续使用情况,进而分析判定内存破坏点是否潜在利用条件,虽然该方法能够检测出触发崩溃样本本身是否潜在可利用性,但还不具备内存数据重新布局和构造的能力,即使检测出当前条件不可利用也不能得出该漏洞不可利用的结论.针对堆溢出内存数据的重新布局和构造,Wang 等人[17]在 2018 年提出了一种基于现有崩溃路径,结合导向性模糊测试的技术,触发堆漏洞程序更多可利用状态的方法 Revery.Wu 等人[18]在 2018 年提出了针对内核堆释放后使用漏洞的自动利用框架 FUZE,该框架利用内核模糊测试技术提供更多的内核崩溃上下文环境作为漏洞利用的依据,并利用符号执行在不同的上下文环境中去尝试利用目标漏洞.

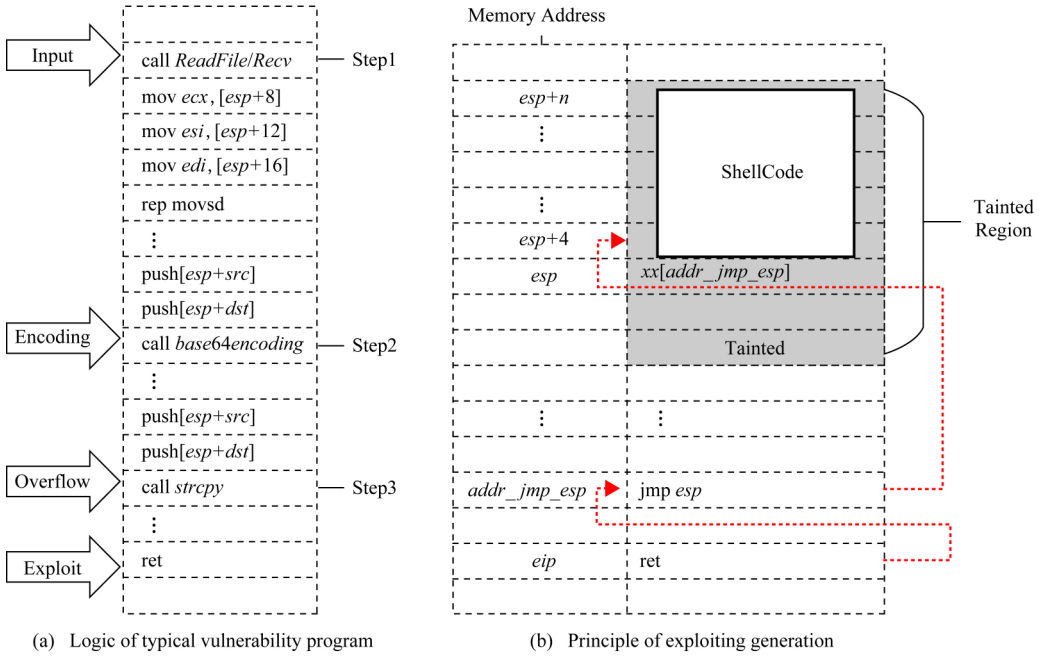
CGC 获奖团队之一的加州大学圣塔芭芭拉分校的 Shoshitaishvili 等人[19-20]提出了面向二进制软件攻防的基础分析平台 ANGR.该工作主要是将已有基础性软件分析方法进行模块化实现,形成一套功能完整的分析框架,包括:控制流图生成、动态分析执行、符号执行、逆向切片等内容.在 ANGR 基本框架的基础上,可以实现针对二进制程序的漏洞挖掘、分析与利用等具体应用场景,并进行自动化系统功能构建.ANGR 的作者将该系统应用到了 CGC 比赛中,将多种程序分析方法集成到同一个系统中,并实现了自动化,通过组合不同的分析技术进而构建面向二进制软件的自动攻防系统.但是 ANGR 系统在分析现实程序及静态链接的 BCTF 样本时却因为性能和兼容性等因素显得捉襟见肘,难以直接支撑实际工作.

与以上方法不同的是,本文提出了有限资源条件下基于动态模型的漏洞自动利用方案,在分析了不同类型漏洞利用方法的差异基础上,提取共性特征建立程序动态运行时的漏洞描述模型,基于漏洞的动态 Weak-Tainted 模型和优化的污点分析技术检测和判断漏洞类型,并提取利用约束条件,最后基于输入求解技术自动生成利用样本.

2 软件漏洞自动挖掘与利用问题分析

当前软件漏洞自动挖掘与利用的现实场景大多是在给定的二进制条件下,基于大规模分布式资源对目标软件进行模糊测试,对产生的程序异常点进行分析,筛选出存在控制流劫持等可利用条件的输入,对其进行数据流分析和内存布局分析,通过 ShellCode 布局和攻击链构造实现利用代码自动生成.

以典型的控制流劫持类漏洞为例,所针对的目标



(a) Logic of typical vulnerability program (b) Principle of exploiting generation

Fig. 1 Control flow hijacking vulnerability logic and exploit principle

图 1 控制流劫持漏洞逻辑与利用原理

和利用过程如图 1 所示.其中,图 1(a)表示一个典型的漏洞程序的基本逻辑,图 1(b)表示针对该程序的利用代码生成原理.图 1(a)的目标程序包括输入接口(图 1(a)中的 *ReadFile/Recv* 函数),处理输入数据的指令序列(图 1(a)中的阶段 1),通常情况下有数据的转移复制,甚至是编码与解码的运算(图 1(a)中的阶段 2),在对输入数据进行编码或解码运算后,程序执行至脆弱点(图 1(a)中的阶段 3).图 1(b)的漏洞利用代码生成主要分为控制流劫持检测、内存布局构造、约束条件求解等步骤:①基于模糊测试得到异常样本,所谓的异常样本在传统漏洞挖掘方法里大多是指造成程序崩溃的输入;②使用数据流分析检测异常点是否为可利用的控制流劫持;③分析控制流劫持过程发生时的 ShellCode 内存布局,设计转移到 ShellCode 的攻击链,如图 1(b)利用中间跳板指令 jmp *esp* 来触发栈空间 ShellCode 代码的执行;④基于 ShellCode 和攻击链部署条件提取约束,基于符号执行等方式求解生成利用代码.

以上是当前漏洞自动挖掘与利用的典型过程,可总结为挖掘、分析、利用构造 3 个部分,而在前期的研究中我们注意到现有研究主要存在 3 个方面的难点与挑战:

1) 在漏洞自动挖掘方面,基于大规模计算实现并行化的模糊测试,Peach,Sulley 等 Fuzzing 工具通常会生成大量的冗余样本,缺乏分析数据的反馈,漏洞挖掘与利用缺乏统一描述模型,生成的数据难以相互支撑,造成漏洞挖掘与漏洞分析利用流程脱节;

2) 在漏洞自动分析方面,以 BitBlaze 和 Decaf 为代表的基于污点传播的分析方法为主,以 qemu 虚拟机全系统模拟为基础,其污点传播算法消耗大量计算资源,在面对复杂的大型目标程序时,其性能难以满足有限资源条件下的漏洞自动分析需求;

3) 在利用代码自动生成方面,大多数自动利用基于已有的漏洞知识构造自动利用专家系统,只能针对特定类型的漏洞,针对不同类型的漏洞需重新编写自动利用系统;另外,逻辑公式求解受到程序自身数据处理过程复杂性与路径约束条件规模的影响,求解准确性和效率堪忧;同时,当前的符号执行求解在应对 base64 等存在映射表运算的编解码情况难以求解,很难有效支撑漏洞利用代码的自动生成.

3 软件漏洞自动挖掘与利用方法

针对当前漏洞自动挖掘与利用存在的难点与挑战,本文提出了有限资源条件下的漏洞自动挖掘与利用方法,总体框架如图 2 所示.首先,针对不同类型漏洞在触发时的关键属性,对漏洞进行归类和建模,形成可被机器识别的、贯穿漏洞挖掘与利用全流程的程序运行时漏洞统一描述模型,该模型基于程序脆弱性特征和数据污点属性进行描述和定义,本文称之为 Weak-Tainted 漏洞描述模型;然后,基于动态污点传播分析方法,结合漏洞描述模型给出的描述,在模糊测试过程中提取分支路径对应输入,

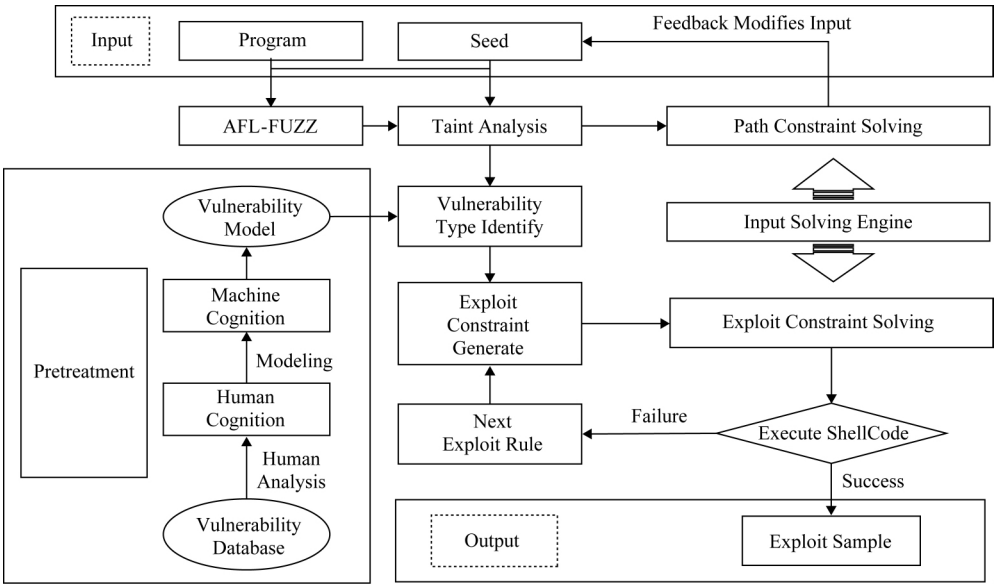


Fig. 2 Vulnerability automatic discovery and exploit based on vulnerability description model

图 2 基于漏洞描述模型的漏洞自动挖掘与利用方法

并求解执行分支另一条路径的输入约束条件，得到下一轮模糊测试的新种子，用于提高漏洞发现的能力和效率；之后，根据漏洞描述模型和动态污点分析判断模糊测试获得的漏洞是否可利用，在可利用条件下自动提取漏洞利用所依赖约束条件；最后由输入求解引擎进行利用约束条件的求解，生成漏洞利用样本.

本节从 Weak-Tainted 漏洞描述模型、基于页表标签值校验的动态污点分析方法、基于输出特征反馈的输入求解方法和原型系统设计与实现 4 个方面对本文提出的有限资源条件下的漏洞自动挖掘与利用方法展开介绍.

3.1 Weak-Tainted 漏洞描述模型

不同类型的漏洞，在运行过程中表现出来的特征具有差异性.控制流劫持类漏洞具有明显的指令特征，表现在控制流转移指令的目标位置受到输入数据的污染，或者转移目标位置的内容受到输入数据的污染；堆溢出漏洞表现在访问的内存空间超出了堆内存分配空间的范围；格式化字符串漏洞可能引发控制流劫持异常，但需要精心构造受输入控制的格式化参数，在触发控制流劫持异常触发之前，其具有自身的特征属性，具体表现在调用具有格式化操作函数 API 时，其格式化参数受到输入数据的污染控制；此外还有一些创建进程、管道调用命令行的 API 函数，如 *system*，*popen* 等，其关键参数受到输入污染控制也可触发任意代码执行的漏洞.

为了对多种不同类型的漏洞进行统一描述，便于机器自动识别和利用代码生成，同时也为了将漏洞挖掘、分析、利用各环节融合以相互支撑，本文提出了 Weak-Tainted 程序动态运行时漏洞描述模型，该模型包含脆弱点(weak)和污点属性 2 个部分.如图 3 所示，该模型定义了 2 个方面的内容：1)程序脆弱点集合，脆弱点集合包含 Weak-INS 和 Weak-PC 两种类型.Weak-INS 是可能引起漏洞的指令，包括触发控制流劫持的指令 call，ret，jmp 以及通过执行系统调用执行任意代码的指令 int 0x80，syscall，sysenter 等；Weak-PC 是指潜在漏洞的指令地址集合，主要通过静态分析敏感脆弱函数地址，比如 *memcpy*，*strcpy*，*scanf*，*system* 等容易产生漏洞的函数，获得模块信息和相对偏移地址，最后映射到动态运行时的函数指令地址，该地址通常是函数入口地址，当然也可以根据实际情况需求定义为函数中间某条指令的地址.2)检测点定义，需要给脆弱点集合的每个脆弱点定义检测点，检测其是否被输入污染(tainted)，每种脆弱类型都有其对应的存储单元敏感点，存储单元可以是内存或者寄存器，可基于常量地址、指令操作数、寄存器或操作数作为指针索引等进行定义，当脆弱点定义的敏感点受到输入数据污染，可初步判定为存在漏洞.

一个脆弱点可以定义多个检测点，这些检测点可以是满足其一的关系或者同时满足的关系，比如 call 指令的检测点，包括 3 种情况：1) call 指令 *op*0 操作数指向的内存为污点，这一类会导致指向的执行指令可以被输入改写；2) call 指令的 *op*0 操作数为污点，这一类会导致 call 指令的地址可以被输入改写，使程序跳转去执行预设的攻击代码；3) *op*0 的

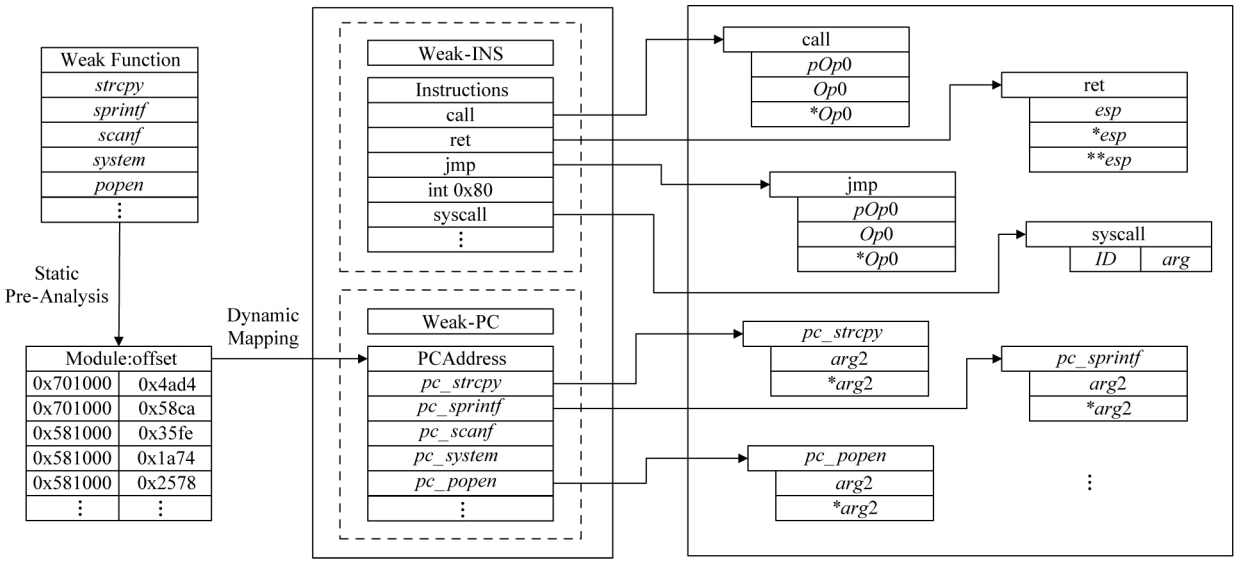


Fig. 3 Weak-Tainted vulnerability discription model during program dynamic runtime

图 3 Weak-Tainted 程序动态运行时漏洞描述模型

指针被污染,例如 call [*eax*+8]中的 *eax* 被污染,这种类型会导致获取跳转地址的位置可被改写,同时引起跳转地址的位置可被改写.类似的 jmp,ret 指令也存在 3 种类型的脆弱类型.

当前漏洞利用的自动构造依赖于特定的漏洞类型,本文分析收集了常见的控制流劫持、格式化字符串、系统调用后门类型的漏洞利用要素和步骤,基于 Weak-Tainted 程序动态运行时的漏洞描述模型,实现了相应的漏洞模型库及模型库的解析引擎,形成基于 Weak-Tainted 漏洞描述模型的漏洞利用样本自动生成框架.

在漏洞描述模型库的表示方面,以常见的控制流劫持漏洞为例,call 指令的劫持与 ret 指令的劫持在利用构造的具体实现上存在差异,并且 call 指令的劫持还分为 call * Taint,call Taint 和 call [Taint] 三种情况,这些差异导致漏洞利用的构造步骤存在一定差异.本文分析总结了这类漏洞的利用构造步骤,在此基础上形成漏洞判断条件和利用要素描述模型库.系统实现的模型库对 call 指令的多种情况分别给出了不同的模型描述,其中 call 参数为寄存器的情况给出了 2 种模型,例如 call *eax*,包括 *eax* 为污点和 *eax* 指向的内存为污点 2 种状态;另外 call 参数为内存的情况给出了 3 种模型,例如 call [*eax*],包括 *eax* 为污点、[*eax*]为污点和[*eax*]指向的内存为污点的 3 种状态.针对 jmp 指令,与 call 指令的描述模型一致,针对 ret 指令,描述了 *esp* 为污点和[*esp*]为污点的 2 种状态.如图 4 所示,是针对 call 参数为内存情况的一种描述.

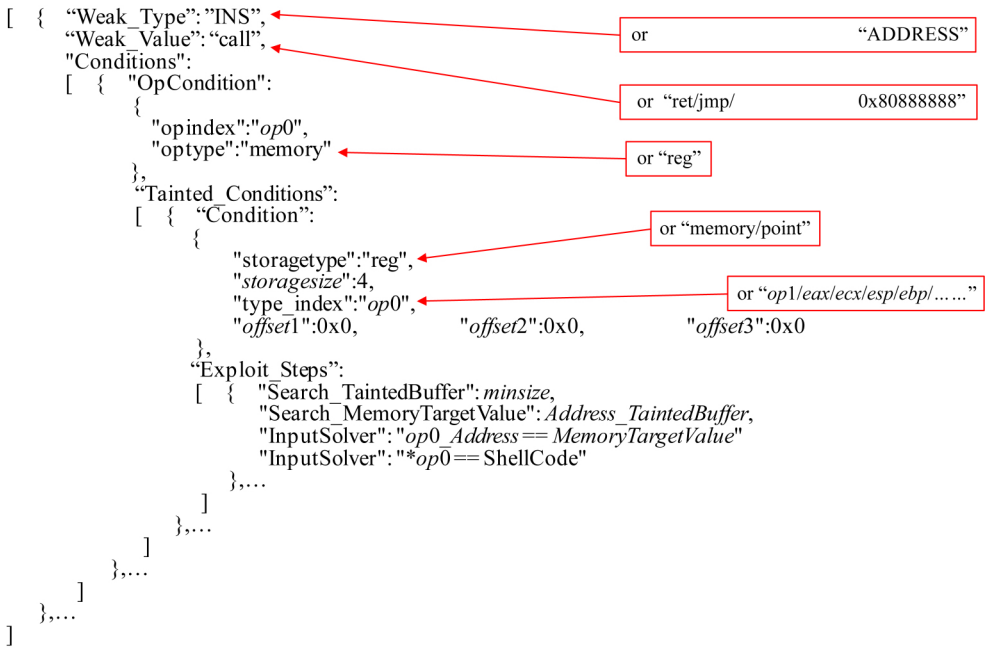


Fig. 4 Case of vulnerability description model

图 4 漏洞描述模型样例

图 4 的样例给出的 Weak_Type:INS 是通过特定敏感指令进行判别,Weak_Value:call 给出的敏感指令是 call,此外还有 jmp,ret 等指令.而本方案还支持通过指令地址进行判别的方式,表示方式为 Weak_Type:ADDRESS,如针对格式化字符串漏洞或者敏感系统调用 API,通过函数入口检测相应参数是否受到控制,函数的入口地址则是通过静态分析结合动态库基地址计算得到,表示方式为 Weak_Value:0x80888888.

程序执行到脆弱位置时需要检测相应条件再判断是否存在可利用点,不同的情形检测不同的参数,因此设计了图 4 所示的 OpCondition 域,用于区分检测指令(如 call)的参数是寄存器还是内存,针对不同情况需要检测不同的位置并使用不同的步骤进行构造,因此也需要分别给出不同的模式配置,针对无需区分的情况 OpCondition 项为空.

另外配置的污点检测 Tainted_Conditions 存储单元可能是指令参数或者函数参数,这些参数都可以通过寄存器、指令操作数及偏移,多次指针索引进行描述和表示,如图 4 中的 storagetype 域和 type_index 域.系统设计并实现的 storagetype 包括 reg,memory,point 三种类型,用于表示需要通过多少级

索引获取存储单元地址，type_index 包括 eax，ecx，esp，$op0$，$op1$ 等通用寄存器和指令操作数，作为存储单元的索引.图 4 的 reg 和 $op0$ 组合表示检测 call 指令的 $op0$ 本身是否被污染，如果是 memory 和 $op0$ 组合，表示检测 $op0$ 指向的内存是否被污染；如果是 reg 与 esp 组合，表示检测 esp 是否被污染；memory 与 esp 组合，表示检测[esp]是否被污染；point 与 esp 组合，表示检测*[esp]是否被污染.

系统实现了漏洞利用过程中所需步骤的子模块，包括但不限于：1)内存布局分析子模块，通过污点传播技术记录程序执行过程中存储单元污点状态，用于搜索内存中污点数据布局区域和对应的污点源关系，为 ShellCode 等注入代码提供存储空间；2)内存值搜索子模块，用于搜索内存中存储了特定数据的空间，返回相应的内存地址，在利用构造过程中提供跳板指令位置、函数指针、或者数据索引指针等特定数据结构位置；3)格式化字符串构造子模块，针对格式化字符串规则，构造任意地址的读或写的能力；4)约束表达式构建与求解子模块，基于模式库的约束模型，生成具体的利用构造约束条件，最后基于符号执行和输出特征反馈的求解方案进行利用样本生成.图 4 所示的 Exploit_Steps 则是通过调用内存布局分析子模块搜索出连续的污点内存区域，通过内存值搜索子模块搜索特定值的地址，通过约束表达式构建求解子模块实现其 2 条约束条件的构建与求解.

本文的系统基于上述的子模块，在漏洞描述模型库中添加了针对 call/jmp 指令各 5 种描述模型规则，ret 指令 2 种描述模型规则的控制流劫持漏洞的自动分析判定与利用生成模型.同时添加了针对各类格式化字符串的漏洞分析判定与利用生成规则，包括基于格式化字符的栈溢出和基于格式化字符串的任意地址写 2 种利用构造方式.另外还添加了针对 $popen/system$ 等敏感函数参数受污点控制的后门漏洞判定与利用构造模式.

本文通过构建 Weak-Tainted 的程序动态运行时漏洞描述模型，可配置漏洞描述模型的脆弱点和对应的敏感存储单元，结合动态污点分析技术，可以实现机器对漏洞的认知与识别.同时，本文提出的漏洞模型具有可扩展的能力，对于新型漏洞可基于该漏洞模型描述进行自定义扩展，提高了机器针对新型漏洞的自动识别与利用构造能力，免去了重新编写漏洞识别与利用代码生成算法的复杂工作.

3.2 基于页表标签值校验的动态污点分析方法

针对现有污点传播方法准确性和效率的缺陷问题，本文提出了基于页表标签的动态污点传播分析方法，通过优化污点状态记录方法、污点传播计算方法，实现低时间与空间复杂度的准确计算.在污点状态记录方面，我们解决了内存污点快速查询与记录、寄存器状态记录等问题；在污点传播计算方面，本文解决了因未监控内核指令导致的污点误标记导致的误报问题.

针对进程污点数据记录问题，本文借鉴操作系统内存管理的方法提出了基于多层页表的污点状态压缩记录方法.以 32 b 系统为例，32 b 系统进程的最大虚拟内存地址空间为 4 GB，但是进程实际用到

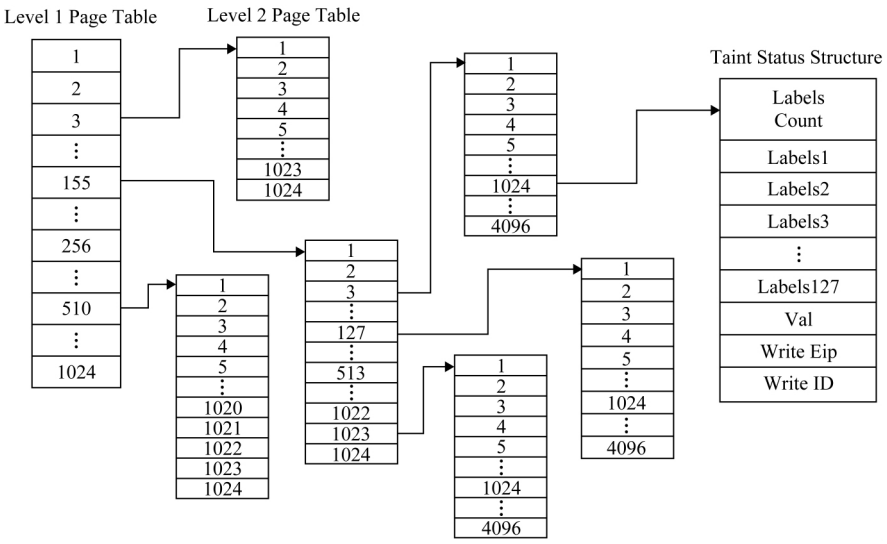


Fig. 5 The page table structure of taint status

图 5 污点存储页表结构

的内存空间远远不足 4 GB,通过 3 级页表索引的方式,如图 5 所示,未被操作或未被污染的内存区域在污点状态页表内可以用空表项表示,而污点内存则由对应的页表项指针指向污点标签记录结构.查询和更新指定内存地址的污点状态的时间复杂度为 $O(1)$,记录程序污点状态所需内存空间最多与目标程序内存空间呈线性关系,即空间复杂度 $O(n)$.

针对寄存器状态记录问题,32 b 系统的寄存器包括通用寄存器、标志寄存器、浮点寄存器等,考虑到通常情况下,进程的低地址 0x0000～0x1000 位置不会被使用,将 CPU 寄存器映射至页表中的低地址空间,并且为每一个线程维护了一套寄存器的污点状态信息.

污点标签结构记录了存储单元的污点源数量、污点源标签组、存储单元值、对应指令地址和序号等信息.污点源标签采用偏移量和字节数的压缩格式表示:$\{[offset1,\ size1],[offset2,\ size2],\cdots,[offsetn,\ sizen]\}$,标签之间通过求并集进行污点源的合并,标签求并集可将重叠的连续字节进行压缩合并,为了使标签合并高效进行,从初始化的单标签开始,合并标签按照升序进行排列,可以保障之后的多标签之间的合并只需按归并排序的策略依次合并,时间复杂度可控制在 $O(n)$,其中 n 为标签组的数量,且在实现当中限定了标签组最多支持 127 组,由于标签组是压缩存储,127 组在实际案例测试过程中是足够用的,如果超出该数量多出的标签会被丢弃,但能保证消耗的空间复杂度不超过 $O(n)$.

在污点传播计算方面,本文解决了基于指令关系分析的字节粒度污点传播计算和因未监控内核指令引起的污点过标记误报问题.字节粒度污点传播计算为每个字节单独计算和维护污点状态信息,通过反汇编解析指令,提取指令操作数之间的关系,为了达到字节级粒度的精度,操作数间的关系并不再是简单的一对一或者一对多关系,而是复杂的多对多关系,本文用如下表达式模型表示一条指令派生出的操作数关系组,其中每个符号表示一个字节:

$$\begin{cases} y_1=f(x_1,x_2,\cdots,x_{n_1}),\\ y_2=f(x_1,x_2,\cdots,x_{n_2}),\\ \vdots\\ y_k=f(x_1,x_2,\cdots,x_{n_k}). \end{cases}$$

根据关系组,更新目标操作字节 y_i 的污点状态,污点状态的更新首先是查询目标操作字节对应的所有源操作字节 $x_1,x_2,\cdots,x_{n_i}$ 的污点状态,并计算这些污点标签的并集作为目标操作字节 y_i 的污点标签,然后更新 y_i 的污点标签,同时记录该字节的值以备后续使用.

通过分析发现,用户态的内存在释放回收后大多并未清除数据,并且其中有一部分被标记为污点,而当重新分配之后如果由内核模块对其初始化无法在用户态检测到存储单元被漂白的操作,因此导致了后续的传播误报问题.而本文不去监控内核指令是由于内核态指令执行占比大,初步测定是用户态指令数量的 3 倍,会导致性能的极度下降.针对上述问题,本文提出的动态污点传播方法带有值校验检测的能力,污点传播计算过程中,如果查询源操作字节的污点状态是污点,需要校验此时该字节的值是否与标记该字节为污点时的值是否一致,如果不一致将其漂白,从而解决了因未监控内核指令引起的污点过标记误报问题.另外,通过大量实验(包括 office word,notepad++,kmplayer 等常用软件)发现用户态污点数据经过内核,再传播到用户态的情况较为少见,在针对应用程序漏洞挖掘与利用构造这一应用场景下可以忽略不计,因此无需去监控内核态的指令.

3.3 基于输出特征反馈的输入求解方法

输入求解是为了获得满足某种条件的输入,比如执行某一条程序分支,或者使得某个时刻内存或寄存器的值满足某个约束条件.目前符号执行是理论可行的解决方案,但在实际应用中却面临计算关系难以用算术表达式或者逻辑表达式进行表示、符号表达式太多、约束条件太复杂等挑战.比如 Base64 编解码、AES 加解密算法,其运算过程都存在基于映射表进行字符替换的过程,从输入到输出难以构造对应算术表达式,也就难以进行求解.

针对符号执行的缺陷,本文提出了基于输出特征反馈的输入求解方法,用于解决映射表运算等导致符号执行无法求解的问题.首先通过污点分析建立输入与目标输出之间存在关系的字节集合,此处的污点传播支持指针为污点的传播,因此污点状态不会因为映射表丢失,并且在传播过程中为指针标签增加了指针属性以便识别指针类型的污点数据.目标输出字节集表示为 $T=\{t_1,t_2,\cdots,t_k\}$,输入字节集表示为 $X=\{x_1,x_2,\cdots,x_n\}$,建立输入与输出字节级的污点关系,用如下函数模型表示它们之间的关系,其中函数 g 是一个未知的黑盒函数.求解的目标是找到一个输入集合 $X=\{x_1,x_2,\cdots,x_n\}$ 使得输出集合 T 刚好为 $T=\{t_1,t_2,\cdots,t_k\}$.

$$\begin{cases} t_1 = g_1(x_1, \cdots, x_{n_1}), \\ t_2 = g_2(x_1, x_2, \cdots, x_{n_2}), \\ \vdots \\ t_k = g_k(x_1, x_2, x_3, \cdots, x_{n_k}). \end{cases}$$

本文提出的输出特征反馈的输入求解基本思想是对输入字节逐个进行求解.图 6 给出了约束关系的样例,*Conditions*1 中的 0x0,4 表示待求解目标偏移位置 0 的连续 4 B,相关的输入字节为偏移 0x1d 的连续 4 B;*Conditions*3 中的 0x8,4 表示待求解目标偏移位置 8 的连续 4 B,相关的输入字节为偏移 0x1d 的连续 12 B,以此类推.本例中不难看出,*Conditions*1 的输入字节是其他 *Condition* 输入字节的子集,这种情况应当优先求解 *Conditions*1.目标字节集在各个 *Conditions* 中没有交集,而输入字节集中是可能存在交集的.

```
Conditions1: 0x0,4: 0x1d,4@
Conditions2: 0x4,4: 0x1d,8@
Conditions3: 0x8,4: 0x1d,12@
Conditions4: 0xc,4: 0x1d,16@
Conditions5: 0x10,4: 0x1d,20@
⋮
```

Fig. 6 The demo of constraint relations

图 6 约束关系样例

本文通过算法 1 实现条件求解顺序的排序.首先对约束条件输入字节进行排序,决定字节的求解顺序,如算法 1 所示,按照关联输入字节数少的约束优先原则进行排序,如果 $InputSet_1 \in InputSet_2$ 优先求解 $InputSet_1$ 对应的约束条件,再得到 $InputSet_1$ 之后对于约束中的部分变量将设置为常量,然后再求解剩余的输入集合$\{InputSet_2 - Inputset_1\}$.

算法 1. 条件求解顺序的排序算法.

输入:多个待求解的约束集合中的输入集 $M = \{Set_1, Set_2, Set_3, \cdots, Set_n\}$;

输出:约束求解顺序.

① $OrderSets = \{\}$
② While M != $OrderSets$
③ $MinSet$ = one of $M - OrderSets$;
④ For each set in $M - OrderSets$
⑤ If $set \in MinSet$
⑥ $MinSet = set$;
⑦ End If
⑧ End For
⑨ $OrderSets$ insert $MinSet$;
⑩ End While
⑪ Output $OrderSets$ by Order.

从算法 1 可知,如果这些约束的输入条件没有子集关系,它们的求解顺序不会进行调整,将按照原有约束条件的顺序进行求解.

针对单个约束,$t_i == f(x_1, x_2, \cdots, x_{n_i})$,使用算法 2 进行处理,算法 2 通过反复修改 t_i 对应的关系输入字节集$\{x_1, x_2, \cdots, x_{n_i}\}$,动态运行监控对应的 t_i 值变化,采用控制变量法,控制单个输入变量的变化,基于遗传迭代和贪心算法思想,保留最接近目标值结果的一组输入进行下一轮的变量修改.多次反馈迭代调整改变输入,直到输出的 t_i 满足求解条件或者求解失败.这种方法能够有效应对大多数编解码算法.

算法 2. 单约束条件输出反馈求解算法.

输入:目标 t_i 所需满足的目标值 t_i',有数据关联的输入字节变量集合的一组值$(x_1, x_2, \cdots, x_{n_k})$,其中 x_i 的每个变量均为单字节;

输出:满足$(t_i = f(x_1', x_2', \cdots, x_{n_k}')) == t_i'$ 的约束输入$(x_1', x_2', \cdots, x_{n_k}')$.

① $BestInput = Init(x_1, x_2, \cdots, x_{n_k})$ /* 初始输入为约束求解前的一个未满足的输入 */
② $BestT_i = f(x_1, x_2, \cdots, x_{n_k})$ /* 所获得的离目标值最接近的结果 */
③ While ($BestT_i$!= t_i')
④ For x_i In $BestInput(x_1, x_2, \cdots, x_{n_k})$
⑤ $x_{i_1} = x_i + 1$;
⑥ $x_{i_2} = x_i - 1$;
⑦ $T_{i_1} = f(x_1, x_2, \cdots, x_{i_1}, \cdots, x_{n_k})$;
⑧ $T_{i_2} = f(x_1, x_2, \cdots, x_{i_2}, \cdots, x_{n_k})$;
⑨ If $|T_{i_1} - t_i'| < |BestT_i - t_i'|$
⑩ $BestT_i = T_{i_1}$;
⑪ $BestInput = (x_1, x_2, \cdots, x_{i_1}, \cdots, x_{n_k})$;
⑫ End If
⑬ If $|T_{i_2} - t_i'| < |BestT_i - t_i'|$
⑭ $BestT_i = T_{i_2}$;
⑮ $BestInput = (x_1, x_2, \cdots, x_{i_2}, \cdots, x_{n_k})$;
⑯ End If
⑰ End For
⑱ End While
⑲ Output $BestInput(x_1, x_2, \cdots, x_{n_k})$.

3.4 原型系统设计与实现

本文实现了漏洞自动挖掘与利用原型系统

AOTA,该系统框架流程如图 7 所示,包含污点分析、输入求解 2 个基本引擎和一个漏洞描述模型,基于漏洞描述模型辅助实现漏洞挖掘、分析、利用全流程系统.漏洞挖掘主要基于模糊测试基础方法,结合漏洞描述模型及其污点分析和输入求解技术,反馈获得额外的变异样本辅助漏洞挖掘,提高漏洞挖掘的效率;漏洞分析目的是分析漏洞类型,判定漏洞是否为可利用,主要基于污点分析引擎和漏洞描述模型来完成;漏洞利用自动生成是针对可利用的漏洞及样本,结合分析获得的约束条件、关联的输入字节,及其对应的漏洞模型和输入求解引擎实现利用样本的自动生成.AOTA 系统的污点分析引擎基于 QEMU 的指令插桩实现,基于 udis86 反汇编结果编写传播规则,污点源是通过监控获取外部输入的系统调用标记,每个字节采用不同标签进行区分.输入求解引擎所需的数据采集基于 Python 框架,迭代调用 QEMU 引擎监测求解点的输出结果,从而进行遗传迭代反馈预测求解,约束关系式是在漏洞分析过程结合漏洞模型描述提取的,求解引擎除了使用本文提出的优化方法,还结合了符号执行求解引擎 Z3.漏洞描述模型以 JSON 格式数据形成漏洞模型配置文件,模型库包含了 call/jmp/ret 等控制流劫持漏洞,格式化字符串漏洞,system/popen 等敏感命令行执行后门漏洞;可由系统导入进行配置;模型的解析主要用来根据配置执行对应的检测操作和求解操作,在 QEMU 动态运行目标程序过程中进行;另外,模型库内置了 2 个不同版本的执行 shell 的 ShellCode.

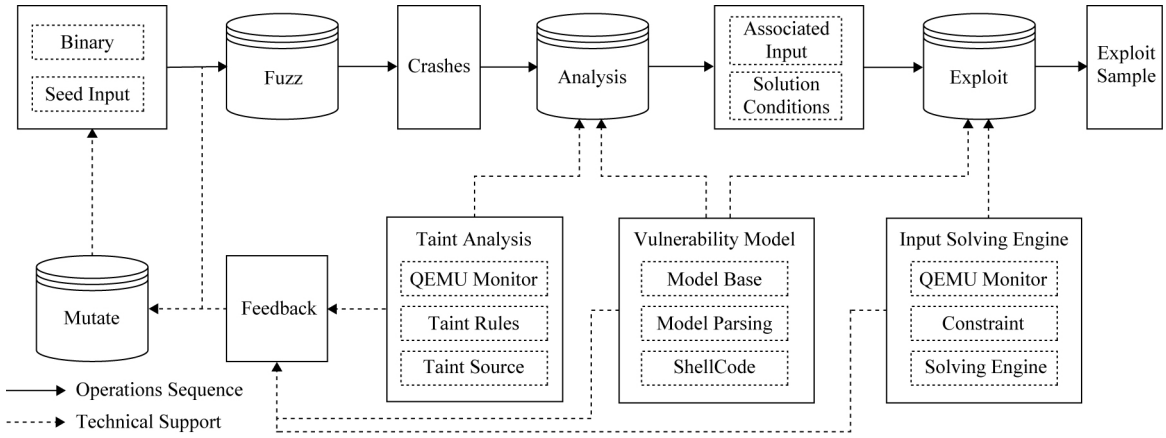


Fig. 7 Framework of prototype system

图 7 原型系统框架流程

4 实验验证与结果分析

参照国际同行选择实验测试对象的情况,如 Driller[20] 以 2016 年 CGC 自动攻防比赛的样本作为测试用例,本文以 2018 年 DEFCON 和百度安全联合举办的 DEFCON China 大会上的 BCTF 比赛样本[21] 作为测试集,其中共有 50 个含有漏洞的 Linux 二进制程序,测试漏洞自动挖掘与利用能力.本文的测试环境均为 Intel Xeon® CPU E5-2630 v3 @ 2.4 GHz × 16,内存 64 GB,硬盘 2 T 的联想 RD650 服务器,操作系统 Ubuntu 16.04.5 64 b.

本文的原型系统也直接应用于 2018 年 BCTF 比赛,在 99 s 内完成第 1 个漏洞的自动挖掘和利用,在 2 min48 s 内完成了 4 个样本的自动利用生成,而人类战队完成第 1 个漏洞利用花了 17 min,与国内外知名 CTF 战队同台竞技,获得了机器人自动攻防单项排名第 1 的成绩.

4.1 污点传播功能测试

本文实现的污点传播模块包括污点源标记、污点传播计算、污点异常检测 3 个部分.污点源标记通过监测外部输入实现,当检测到外部设备、文件或网络数据读入内存时,将对应的内存字节标记为污点,并使用对应的输入字节偏移量为标签进行标记.污点传播计算通过代码插桩提取目标程序执行的指令,针对 Linux 的目标程序,实验采用 qemu 的用户态模式进行代码插桩,然后反汇编指令,解析指令语义更新内存和寄存器的污点状态,记录对应污点源标签.污点异常检测根据配置的漏洞模型规则获得检测点和检测内容,查询对应内存或寄存器操作数是否为污染状态,如果是污染状态取出污点源.

实验从测试集选取了 12 个栈溢出和格式化

字符串漏洞样本，使用对应的 POC 样本单独测试 AOTA 系统的污点传播模块，实验均能基于漏洞模型通过污点分析检测到漏洞点，并且未发现误报和漏报.实验表明：本文改进后的污点传播方法其准确度能够满足漏洞分析与判定的需求.

性能方面，测试的 12 个样本中，污点分析消耗的时间最多的是 357 ms，最少的是 19 ms，平均值为 174.25 ms.消耗的内存最多的为 25.73 MB，最少的为 12.6 MB，平均消耗 22.2 MB.另外，实验统计了 1 000 组指令数量与污点分析时间消耗关系数据，如图 8 所示，横轴是指令数量，纵轴是消耗的毫秒时间.由于程序开始执行时自身需消耗较多的时间用于 IO 加载，对污点分析速度的测算影响较大，实验筛除指令数量少于 20 000 条的测算数据，对剩下的速度数据求平均值，得到的污点传播速度为每秒 57. 34 万条指令.

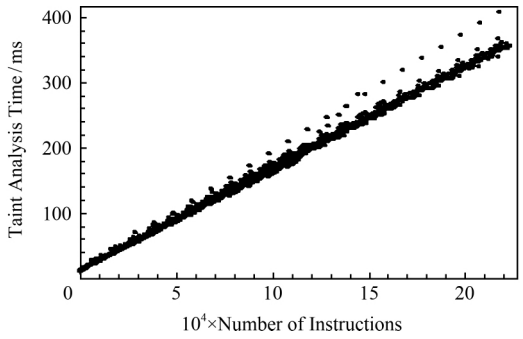


Fig. 8 Velocity curve of taint analysis

图 8 污点分析速度曲线

4.2 基于输出特征反馈的输入求解测试

本文提出的基于输出特征反馈的输入求解方法在原型系统 AOTA 中进行了实现，与 ANGR 中的符号执行模块进行了实验对比，测试输入数据分别经过 atoi 转换、base64 编码、hex 编码处理的求解能力，从求解能力和求解效率的角度进行评估.通过学习 ANGR 符号执行的用例，发现其主要应用在路径求解，并且其约束条件是用输出含有指定字符进行描述.为了适应 ANGR 的应用场景，实验构造了符合 ANGR 应用场景的测试用例，由标准输入经过编码转换，然后将结果与预设值匹配，匹配成功则使用标准输出打印出 Success 字段.

实验分别使用 ANGR 和 AOTA 对 atoi，base64，hex 编码进行求解，求解成功与否如表 1 所示.其中对于 atoi 的转化均能求解成功；ANGR 对 base64 编码的求解失败，但 AOTA 能够求解成功，这是因为 base64 中有映射表转换的操作，ANGR 的符号执行无法支持该转换的求解，导致求解出来的结果是错误的；对于 hex 编码的求解 ANGR 只能求解少量字符，当字符串长度不少于 3 个时，内存消耗超过 64 GB 求解失败，而 AOTA 的求解不受长度限制.总体而言，AOTA 的求解能力要优于 ANGR.

Table 1 Results of Input Solving Ability

表 1 输入求解能力结果

Tool	atoi	base64	hex
ANGR	Success	Failure	Partial Success
AOTA	Success	Success	Success

实验对比分析了 AOTA 和 ANGR 分别求解 atoi 和 hex 编码的资源消耗.图 9(a)是 ANGR 和 AOTA 求解 atoi 转换的时间消耗对比，横轴是字符个数，纵轴是消耗时间的秒数.ANGR 仅在字符数少于 3 时有求解速度上的优势，当字符数达到 6 以上时求解时间维持在 15 s 左右，而 AOTA 的求解时间稳定在 10 s 左右.图 9(b)是 ANGR 和 AOTA 求解 atoi 转换的内存消耗对比，横轴是字符个数，纵轴是内存消耗，单位 MB.ANGR 在求解 6 字符以内的 atoi 时内存消耗从 130 MB 随字符数增多而增加，之后稳定在 145 MB 左右，而 AOTA 一直维持在 60 MB 左右的内存消耗.

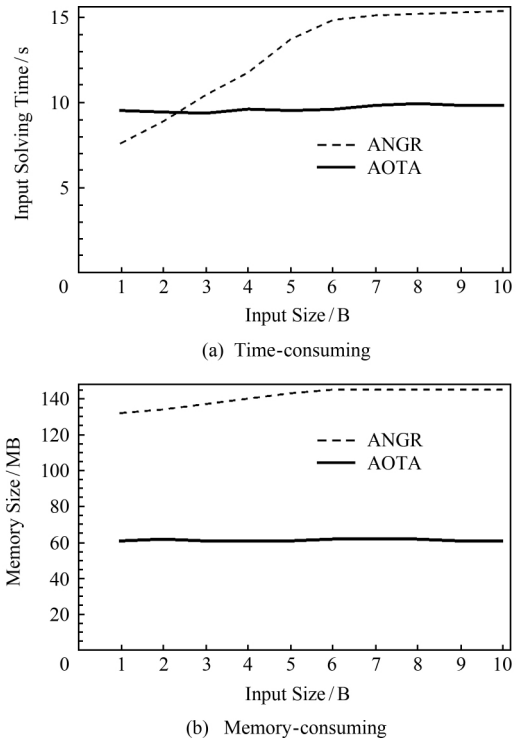


Fig. 9 Comparison of time and memory consuming to solve atoi

图 9 求解 atoi 时间与内存消耗对比图

另外,求解出的 atoi 输入存在非预期解,这是由于计算过程中存在整数溢出导致,如求解 atoi(x)值为 87654321,ANGR 给出的结果是字符串"8677588913",而并非字符串"87654321",此处预设的字符长度并非求解出来的真实长度,求解提前到达了有解情况的最大输入长度,这也导致了 ANGR 后续求解时间和内存消耗不再随着预设的字符数增加而增加,而是保持在稳定的范围内.而 AOTA 的求解更多的时间消耗在输入输出关系的建立和收集,另外 atoi 的输出目标都是一个整型数,恒定为4 个字符,关系建立受输入长度的影响较小,因此求解时间较为稳定.

在求解 hex 编码能力方面,ANGR 仅在字符数少于 3 时能够求解成功,单字符所需的求解时间 10 s 左右,内存消耗 400 MB 以上;2 字符所需时间将近 300 s,内存消耗 8 GB 以上;当字符数达到 3 及以上时所需内存超过 64 GB 求解失败.而 AOTA 的求解时间随着字符数在 1～10 之间的增加,时间消耗在 20～100 s 的区间内递增,所消耗内存维持在 60 MB 左右,并且均能求解成功.总体而言,AOTA 的求解性能优于 ANGR.

与 ANGR 相比,本文的求解方法建立在优化的基于页表标签的动态污点分析,通过页表的多级索引提高了检索计算的效率,通过值校验的方式消除了污点数据的误标记扩散,避免了污点爆炸引起的内存额外消耗,通过污点相关性筛选出需要求解的输入字节,降低了直接使用符号执行求解的复杂度,一定程度消除了传统符号执行求解中的路径爆炸问题.实验也表明,在针对 atoi,base64,hex 编码的反向求解方面,本文的求解能力要优于 ANGR,其中对 atoi 的求解效率提升了 33%左右,而 hex 和 base64 编码 ANGR 无法成功求解.

4.3 漏洞自动挖掘能力测试

本文提出的漏洞描述模型与污点分析、输入求解结合,优化了程序漏洞的自动挖掘,在 AOTA 原型系统中进行了实现,与 AFL-Fuzz 进行了实验对比.实验以 BCTF2018 的 50 个样本作为样例测试程序,使用相同的种子文件,分成 2 组,每组 25 个测试用例,同时对 50 个测试用例程序进行了 24 h 的漏洞挖掘测试.使用 4 台配置型号相同的 RD650 服务器并行测试,配置说明见本节开头,其中 2 台服务器测试 AFL 的漏洞挖掘能力,另外 2 台服务器测试 AOTA 系统的漏洞挖掘能力.测试每隔 10 min 统计 1 次 AOTA 和 AFL 挖掘出漏洞的样本数,24 h 的统计结果曲线如图 10 所示.

实验表明:相同种子输入、时间和测试用例,

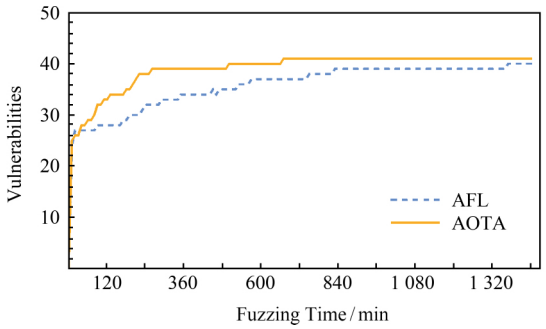


Fig. 10 Comparison of vulnerabilities sequence diagram between AOTA and AFL

图 10 挖掘漏洞样本数时序对比图

AOTA 能挖掘出 41 个样本的漏洞,而 AFL 只挖掘出了 39 个样本的漏洞,并且这 39 个样本的漏洞包含在 AOTA 的 41 个样本范围内,同时对比这 39 个样本的挖掘时间发现,AOTA 比 AFL 先挖掘出漏洞的样本有 12 个,AFL 比 AOTA 先挖掘出漏洞的也有 9 个,其中 4 个是由于样本有大量的模型约束需要求解,消耗了 30～60 min 的时间用于约束求解,另外 5 个在误差允许的范围之内,但是 AOTA 挖掘出这 39 个样本的平均时间为 65.72 min,而 AFL 挖掘出这 39 个样本的平均时间为 121.03 min,总体上 AOTA 挖掘漏洞的能力与 AFL 相当,挖掘出漏洞的平均时间缩短了 55 min,平均效率提升 45.7%.

AOTA 的模糊测试优化功能为 18 个测试样本生成了新的测试用例,将这些用例作为 AOTA 新增的种子集合,与 AFL 在相同硬件配置环境下进行模糊测试路径覆盖对比测试,得到的路径覆盖数量如图 11 所示,其中 AOTA 的平均覆盖路径数量为 1 000,而 AFL 的平均覆盖路径数量为 992,相比之下,AOTA 的覆盖路径数量比 AFL 增加了 0.8%.

4.4 漏洞利用自动生成测试

本文实现的 AOTA 系统基于 Weak-Tainted 漏洞模型的漏洞识别和漏洞利用要素提取,并基于输出特征反馈对输入进行求解,实现漏洞利用自动生成,并以 BCTF2018 的 50 个样本作为测试用例进行了实验验证,实验统计结果如表 2 所示.

实验结果表明:本文提出的自动利用生成方法在 AOTA 系统中得到了实现和应用,在 50 个测试用例中成功发现了 41 个用例的漏洞,比例达到 82%,其中 26 个判定为可利用,比例达到 52%,并进行了自动利用生成的尝试,成功生成利用的有 24 个用例,比例达到 48%.这些成功生成利用的漏洞中,包括栈溢出、格式化字符串和后门系统调用多种类型.

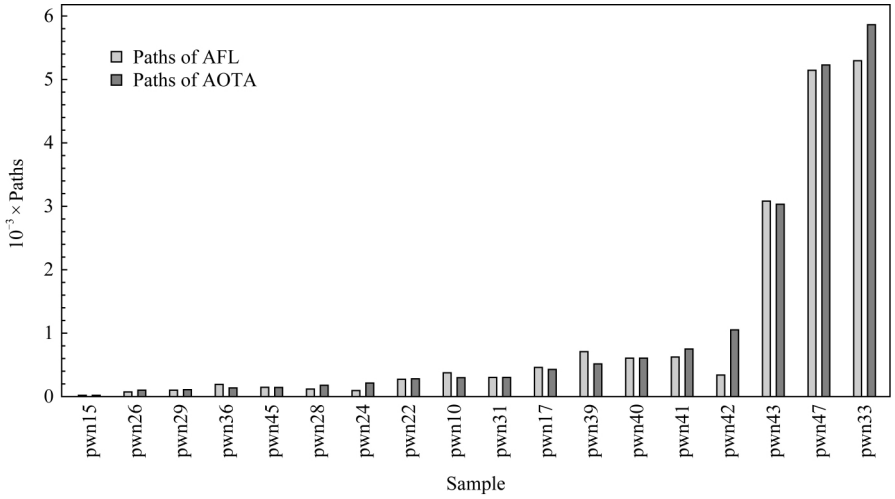


Fig. 11 Comparison of paths coverage between AOTA and AFL

图 11 样本覆盖路径数量对比图

Table 2 Result of Vulnerability Automatic Discovery and Exploit

表 2 漏洞自动挖掘与利用生成结果

Statistic Categories	Sample Total	Sample Percent/%
Vulnerability	41	82
Assess Exploit	26	52
Success Exploit	24	48

同时我们研究分析测试了 ANGR 的自动利用生成插件 REX 和 AEG，发现 ANGR 提供的测试用例是动态链接 libc 等动态库，主模块代码量较少，其本身进行了局部优化；对于 BCTF 这类静态链接 libc 等链接库，程序代码量较大（500 KB 左右）的样例，ANGR 的插件无法成功进行漏洞自动利用生成，并且针对单个测试用例内存消耗高达 30 GB 以上.

为了更充分地对比，本文重新选取了 10 个含原代码的测试用例，以动态链接库的方式编译出符合 ANGR 的 AEG 插件测试条件的二进制程序，同时用本文的漏洞自动利用生成模块进行对比测试，实验结果如表 3 所示.

表 3 第 1 个样本为 ANGR 项目中的测试样本，两者均能成功自动生成利用，其余 9 个样本为收集的 BCTF 比赛测试样本，ANGE 的 AEG 插件只生成了一个 exploit 样本，但是测试运行时只是产生了段错误，并未能成功利用，分析原因是该漏洞是栈溢出，该 AEG 功能未考虑栈地址随机性.同时，实验中 AEG 每项失败的利用生成测试时间均超过 12 h，内存消耗达到 30 GB 以上，单台设备每次只能运行 2 个任务.而本文的 AOTA 系统在这些测试用例中，成功生成并利用成功的样本数有 6 个，且能够在单台设备并发 10 个任务，在性能和能力方面具备明显的优势.

Table 3 Exploit Ability Comparison Between ANGR AEG and AOTA

表 3 ANGR AEG 与 AOTA 漏洞自动利用对比

Sample	Vulnerability	ANGR AEG	AOTA
demo_bin	overflow to func	Success	Success
pwn02	stack overflow	Failure	Success
pwn13	integer overflow	Failure	Success
pwn20	heap overflow	Failure	Failure
pwn23	stack overflow	Failure	Failure
pwn34	format string	Failure	Failure
pwn45	stack overflow	Failure	Failure
pwn48	stack overflow	Generate	Success
pwn49	buffer overflow	Failure	Success
pwn50	buffer overflow	Failure	Success

5 讨 论

本文提出的有限资源条件下基于程序动态运行特征 Weak-Tainted 模型的漏洞自动挖掘与利用方法具备快速发现漏洞和生成可利用样本的能力，支持漏洞类型的自定义扩展描述，扩充了适用范围.而 PolyAEG 与 Maythem 等相关工作只能对控制流劫持等特定类型的漏洞自动生成利用，本文提出的漏洞模型支持对多种类型的漏洞自动生成利用，通过漏洞描述的配置实现多类型漏洞支持的扩展.

本文优化了污点传播算法和输入求解方法,国内对输入求解的优化研究[22]更侧重在漏洞挖掘方面提高路径覆盖率.当前的漏洞自动利用生成方案PolyAEG,Maythem,ANGR等均使用了污点传播和符号约束求解,PolyAEG在硬件虚拟化平台QEMU基础上进行指令插桩,构建指令级的数据传播流图iTPG和全局污点状态记录GTSR,其更侧重于漏洞利用样本的多样性生成,本文优化后的污点传播方法在内存消耗和效率上均优于它,同时PolyAEG未考虑因内核指令未监控导致的污点数据过标记问题,会导致控制流劫持的误报,消耗更多的资源用于无效的利用生成求解.同样,Maythem也具有同样的问题,其更侧重于符号执行求解的优化,未能消除内核未监控导致的误报问题.

另外,测试中我们发现部分的漏洞自动利用生成失败,其原因主要包括3种情况:1)输入无法满足特定字符,如通过*scanf*获取的输入字符串不可能有"\r","\n"以及空格等字符,对应的ASCII码值为0x0d和0x0a,而当求解的条件要求输入的字符为0x0d或0x0a时就会导致求解失败;2)修改后的输入改变了程序的原有逻辑,导致程序无法执行到我们设定的求解点,无法获得用于分析输入输出关系的更多数据,比如程序检测输入的数据必须全部为数字否则退出,当改变输入为非数字时就会导致程序执行不到预设的输出点;3)输入输出关系复杂度太高,如产生雪崩效应的加密算法在搜集了大量的输入输出关系之后仍然无法求解.

针对求解失败产生的原因,本节讨论可能的解决思路.针对原因1,本文的思路是修改等效的求解条件,在漏洞触发前题下,其利用输入的约束并非唯一的,比如栈溢出中所需要的跳板指令jmp *esp*可以用等效的call *esp*代替,并且在程序代码块中存在该指令的位置可能不止一处,我们在选取跳板指令时尽量避免导致输入为空格、回车等字符的约束.另外ShellCode可以进行变形,使用等效指令替换,以此规避一些导致无解的约束条件.针对原因2,我们在监测到输入导致无法执行到输出点后,只需舍弃该变异的输入,导致的结果仅仅是消耗更多的计算资源用于收集输入输出关系,但仍能继续求解.针对原因3,目前的研究尚无可行的解决方案,传统的符号执行方案也可能因为复杂度太高求解失败,毕竟对于高强度的RSA等公钥加密、SHA256等HASH算法也并非符号执行能够解决的问题,但毕竟这类问题很少出现在漏洞利用需要突破的范畴,需要将输入数据经过复杂加解密运算后作为ShellCode或者跳板指令地址的场景并不常见,但经过编码转换的情况则较为常见.

本文提出的Weak-Tainted程序动态运行时漏洞描述模型适用于控制流劫持类漏洞、格式化字符串漏洞、脆弱函数造成的缓冲区溢出漏洞、命令行执行API调用(疑似后门)漏洞等多种漏洞类型的漏洞识别和利用自动生成,但目前对UAF和Double Free仅能给出漏洞判定模型,尚未能给出适当的描述用于此类型漏洞的利用生成方案,这将作为后续的研究点.

6 总 结

本文介绍了一种有限资源条件下基于动态模型的漏洞自动利用方法,基于污点分析和输入求解优化模糊测试的数据生成,并建立了可贯穿应用于漏洞挖掘、分析、利用全流程的Weak-Tainted程序动态运行时漏洞描述模型,然后结合基于标签的污点传播方法用于漏洞判定和利用约束条件提取,最后基于输出特征反馈的输入求解方法进行利用样本自动生成.本文实现了原型系统,通过对2018年BCTF比赛样本进行了实验测试,在与AFL具备同等漏洞挖掘能力的前提下,平均效率提高45.7%,样本分析内存消耗维持在100 MB以内,为并行化提供了有利条件,在测试的50个样本中有24个能够自动生成利用代码,验证了有限资源条件下Weak-Tainted程序动态运行时漏洞描述模型用于漏洞自动挖掘与利用生成的有效性.

参 考 文 献

[1] Song J, Alves-Foss J. The DARPA cyber grand challenge: A competitor's perspective [J]. IEEE Security & Privacy, 2015, 13(6): 72-76

[2] Cui W, Peinado M, Cha S K, et al. Retracer: Triaging crashes by reverse execution from partial memory dumps [C] //Proc of 2016 IEEE/ACM 38th Int Conf on Software Engineering. Piscataway, NJ: IEEE, 2016: 820-831

[3] Zalewski M. American fuzzy lop [CP/OL]. 2013 [2019-06-01]. http://lcamtuf.coredump.cx/afl

[4] Song D, Brumley D, Yin H, et al. BitBlaze: A new approach to computer security via binary analysis [C] //Proc of the 4th Int Conf on Information Systems Security. Berlin: Springer, 2008: 1-25

[5] Henderson A, Prakash A, Yan L K, et al. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform [C] //Proc of the 2014 Int Symp on Software Testing and Analysis. New York: ACM, 2014: 248-258

[6] Avgerinos T, Sang K C, Hao B L T, et al. AEG: Automatic exploit generation [C/OL] //Proc of Network and Distributed System Security Symp (NDSS) 2011. Reston, VA, USA: The Internet Society, 2011 [2018-09-25]. https://softsec.kaist.ac.kr/~sangkilc/papers/avgerinos-ndss11.pdf

[7] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and implications [C] //Proc of 2008 IEEE Symp on Security and Privacy (SP 2008). Piscataway, NJ: IEEE, 2008: 143-157

[8] Xu Xiaojun, Liu Chang, Feng Qian, et al. Neural network-based graph embedding for cross-platform binary code similarity detection [C] //Proc of the 2017 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2017: 363-376

[9] Hocevar S. zzuf—multi-purpose fuzzer [CP/OL]. 2011 [2019-03-01]. http://caca.zoy.org/ wiki/zzuf

[10] Devarajan G. Unraveling SCADA protocols: Using sulley fuzzer [R/OL]. USA: DefCon 15 Hacking Conf, 2007 [2019-03-01]. https://www.defcon.org/ images/defcon-15/dc15-presentations/dc-15-devarajan.pdf

[11] Swiecki R. Honggfuzz[CP/OL]. 2016 [2019-03-01]. http://code.google.com/p/honggfuzz

[12] Cadar C, Dunbar D, Engler D R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs [C] //Proc of the 8th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2008: 209-224

[13] Godefroid P, Levin M Y, Molnar D. SAGE: Whitebox fuzzing for security testing [J]. Communications of the ACM, 2012, 55(3): 40-44

[14] Cha S K, Avgerinos T, Rebert A, et al. Unleashing mayhem on binary code [C] //Proc of 2012 IEEE Symp on Security and Privacy. Piscataway, NJ: IEEE, 2012: 380-394

[15] Wang Minghua, Su Purui, Li Qi, et al. Automatic polymorphic exploit generation for software vulnerabilities [C] //Proc of the 19th Int Conf on Security and Privacy in Communication Systems. Berlin: Springer, 2013: 216-233

[16] He Liang, Cai Yan, Hu Hong, et al. Automatically assessing crashes from heap overflows [C] //Proc of the 32nd IEEE/ACM Int Conf on Automated Software Engineering. Piscataway, NJ: IEEE Press, 2017: 274-279

[17] Wang Yan, Zhang Chao, Xiang Xiaobo, et al. Revery: From proof-of-concept to exploitable [C] //Proc of the 2018 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 1914-1927

[18] Wu Wei, Chen Yueqi, Xu Jun, et al. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities [C] //Proc of the 27th USENIX Security Symp (USENIX Security'18). Berkeley, CA: USENIX Association, 2018: 781-797

[19] Shoshitaishvili Y, Wang R, Salls C, et al. Sok: (state of) the art of war: Offensive techniques in binary analysis [C] // Proc of 2016 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2016: 138-157

[20] Stephens N, Grosen J, Salls C, et al. Driller: Augmenting fuzzing through selective symbolic execution [C/OL] //Proc of Network and Distributed System Security Symp (NDSS) 2016. Reston, VA, USA: The Internet Society. 2016 [2018-12-16]. http://dx.doi.org/10.14722/ndss.2016.23368

[21] Wang Yihang. Attack-Defense-Challenges [CP/OL]. 2018 [2019-03-01]. https://github.com/WangYihang/Attack-With-Defense-Challenges/tree/master/2018-Defcon-China-Final/ai

[22] Qin Xiaojun, Zhou Lin, Chen Zuoning, et al. Software vulnerable trace's solving algorithm based on lazy symbolic execution [J]. Chinese Journal of Computers, 2015, 38(11): 2290-2300 (in Chinese)
(秦晓军, 周林, 陈左宁, 等. 基于懒符号执行的软件脆弱性路径求解算法[J]. 计算机学报, 2015, 38(11): 2290-2300)

Huang Huafeng, born in 1988. PhD candidate, engineer. His main research interests include vulnerability discovery and automatic exploit generation.

Wang Jiajie, born in 1982. PhD, associate professor. His main research interests include cyber security, mobile security, vulnerability analysis.

Yang Yi, born in 1982. PhD, associate professor. His main research interests include computer system security and vulnerability discovery.

Su Purui, born in 1976. PhD, professor. His main research interests include computer system security and vulnerability discovery.

Nie Chujiang, born in 1983. PhD, senior engineer. His main research interests include industrial system security and vulnerability discovery.

Xin Wei, born in 1981. PhD, associate professor. His main research interests include cryptography and vulnerability analysis.