文件触发类二进制程序漏洞挖掘技术

笔记本： 论文
创建时间： 2019/11/1 8:11 **更新时间：** 2019/11/1 9:25
作者： Mrs.Lee

书名	文件触发类二进制程序漏洞挖掘技术	作者	王连赢
出版社	北邮	阅读日期	2019.11.1

主要内容集中在前三章（主要关注脆弱性定位的内容）

摘要

本文研究一种多维Fuzzing技术，首先，提出了一种基于静态分析的二进制代码脆弱性评估方法来定位程序的脆弱点。其次，提出了一种基于运行时监控的方法来建立程序脆弱点与文件中数据元素之间的映射关系。最后，针对每一个程序脆弱点进行多维 Fuzzing测试。另外，本文设计实现了一种基于异步编程模型的分布式文件Fuzzing系统，通过并行运行的方式使得Fuzzing系统在单位时间内可以测试更多的样本文件。开发了原型系统DMFFuzzer。

在本文中，结合传统源代码安全审计的评判指标及二进制代码的自身特点，我们选取了三个指标作为对二进制代码安全性评定的标准，这三个指标的分析评估也就是静态分析主要的研究内容，它们分别是；复杂循环分析，危险函数调用分析，函数圈复杂度分析。

下面分别对各项研究内容进行具体阐释。

绪论

研究现状

常用的二进制程序漏洞挖掘技术包括：手工挖掘技术，二进制比对技术，fuzzing技术。

手工挖掘技术研究人员通过静态分析对二进制代码进行评估，通过调试跟踪观察程序实际运行状态。

二进制比对技术（补丁比对技术），Halvar Flake于2004年提出了基于结构化比对的补丁比对算法[1]，同年，Tobb Sabin 提出图形化比对算法作为补充。2005年，Thomas Dullien提出了基于图的结构化比对方法，以图的方式进行基本块比对[2]。2008年，David Brumley等人研究了基于补丁比对的exploit自动化生成技术[3]，以实际应用证明了补丁比对的重要研究价值。

2005年，iDefense发布了IDA插件IDACompare[4]，它是一个IDAPro的二进制插件，可以基于IDA反汇编结果进行比对；2006年，eEye发布了开源的eEye Binary Diffing Suite（EBDS）套件[5]，它包含了批量补丁分析工具BinaryDiffing Starter（BDS）和二进制补丁比对工具Darungrim[6]；2007年，Sabre发布了另外一款IDA插件形式的补丁比对工具Bindiff2[7]。

Fuzzing技术，本文将Fuzzing技术分为传统Fuzzing和智能Fuzzing, 传统Fuzzing技术按照变异的策略不同可以分为随机Fuzzing和结构Fuzzing，随机Fuzzing即在对输入进行变异时采用随机的方法，而结构Fuzzing则考虑不同文件的文件结构信息，通过文件模板的方法对文件中的有价值字段进行Fuzzing。智能Fuzzin是污点，符号执行与Fuzzing的结合。本文提出的多维Fuzzing属于智能Fuzzing.

上面所述的所有Fuzzing技术在对输入进行变异时，只能对一个输入元素进行变异，而多维Fuzzing是指可以对多个输入元素同时进行变异，因为同一段程序逻辑可能同时受多个输入元素影响，这种情况下如果只对输入元素进行单维的变异就很难达到触发程序潜在脆弱点的效果。

多维Fuzzing的基本实施步骤可以分为三步：（1）通过静态分析确定目标程序中的脆弱点；（2）确定输入输出之间的映射关系；（3）同时变异多维输入，监控程序异常。前人有很多工作在其中一点或多点进行了研究。

研究点

（1）基于静态分析的二进制代码脆弱性评估技术，提出三个评价指标：复杂循环分析、危险函数调用、函数复杂度分析。

（2）基于运行时监控的有向Fuzzing技术本文使用一种基于运行时监控的高效有向Fuzzing技术，它是一种动态的行时分析方法，使用调试器挂载目标程序打开样本文件，在静态分析模块给的脆弱点处下断点，观察在对样本文件进行变异时脆弱点相关参数的变化情从而建立文件中的基本数据元素与程序代码片段的映射关系。针对每一个弱点，同时变异影响它的所有数据单元。

（3）分布式文件格式Fuzzing框架研究本文研究一种分布式文件格式Fuzzing框架，采用Client/Server架构。任意多个客户端可以和服务器端通信。服务器端负责作业管理、变异策略生成等任务，原始种子测试文件放在服务器端，客户端通过网络通信从服务器端取种子文件和变异策

略，按照服务器端给定的变异策略进行变异。客户端产生变异文件，在调试器监视下对目标程序进行测试。若程序产生了崩溃异常，客户端端将当前的测试文件和程序崩溃信息发送给服务器端。

文件格式漏洞及其挖掘相关技术

具体要介绍的文件格式漏洞挖掘方法包括：二进制代码比对技术，文件格式Fuzzing技术，智能灰盒测试技术。

补丁比对技术的一般做法是比对打补丁前后的程序文件进行比对分析，通补丁比对定位出有差异的函数，再经过进一步的人工分析，可以确定出补丁件对程序执行逻辑上的修改，从而推测漏洞位置及成因，辅助漏洞挖掘。

补丁比对的一般原理，或者说一般实现流程，可以总结成图 2-5。

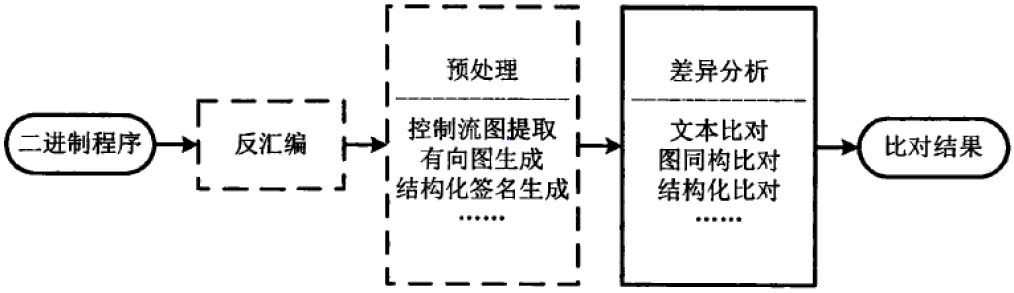

图 2-5 二进制程序比对一般原理

（1）基于文本的比对。基于文本的比对是最简单的比对方式，其比对的对象分为两种：二进制文件、反汇编代码。其中，针对二进制文件的文本比对对两个二进制文件（补丁前后的程序文件）逐字节进行比对，能够全面的检测出程序中任何细小的变化，但是完全不考虑程序的逻辑信息，输出结果范围大，漏洞定位精确度差，误报率很高，因而实用性较差。

针对反汇编代码的文本比对是将二进制程序先经过反汇编，然后对汇编代码进行文本比对，这种比对方法的结果信息中包含一定的程序逻辑信息，但是对程序的变动十分敏感，比如编译选项变化时，虽然程序逻辑相同，但通过这种文本比对方法得出的结果就是不同的，因而也有很大的局限性。

（2）基于图同构的比对。为了克服基于文本比对无法提供二进制程序语义相关信息的情况，基于图同构的二进制比对方法得以提出。基于图同构的比对技术依托于图论知识，首先对可执行程序的控制流图进行抽象，将二进制程序转化为一个有向图，即将二进制比对问题转化为图论中的图同构问题，然后根据图论的知识进行解决。

（3）基于结构化的比对。在基于图同构的比对技术中，其核心是基于指令语义级的相似性，它的优点在于不会漏掉非结构化的差异，如：

分配缓冲区的大小变化等。但它也有一些缺点，如：受编译器优化的影响大，全局结果受局部结果的影响大等。为了解决这些问题，基于结构化的比对技术得以提出。结构化比对技术区别于指令比对技术，其注重点是可执行文件逻辑结构上的变化，而不是某一条反汇编指令的改变，自然就避免了基于图同构方法的缺点。

基于多维Fuzzing的文件格式漏洞挖掘技术（主要关注脆弱性评估）

现有的比较成熟的文件格式Fuzzing工具，如Peach、FileFuzz、FOE（Failure Observation Engine）等，在对测试样本集进行变异的时候，都是变异单一的元素或字段，这种方法我们称之为单维Fuzzing技术，它确实是行之有效的，也有很多成功的经验。不过，在一些复杂代码逻辑的情况下，单维Fuzzing就无法满足更深层次的漏洞挖掘需求，此时就需要多维Fuzzing 技术。多维Fuzzing是指同时对多维输入元素进行变异，具体到文件格式上就是对多个文件字段同时进行变异，它会覆盖到一些单维Fuzzing无法覆盖到的程序漏洞。

1970年，Allen和Cocke就提出了控制图的可归约性问题[24]，这一问题也是程序循环结构分析研究的理论基础，因此可以认为对循环结构分析的研究是从那个时候开始的。1997年，Havlak在Tarjan[25]的基础上提出了对常规控制流图的近似线性时间的算法[26]。两年后Ramalingam 提出了对Havlak算法的一个改进算法[27]，Ramalingam的算法中着重解决了在复杂情况下算法的适用问题，该算法在最差情况下能够接近线性时间复杂度。2007年，韦韬等人提出了一种在二进制可执行程序中识别循环的新方法[28]，这种方法不需要复杂的数据结构，只需要对控制流图进行一次深度优先搜索就可以准确的处理不可归约的控制流图，比Havlak的算法快2-5倍。[29]

自然循环[30]满足两个条件：

（1）有唯一的入口结点。

（2）存在一条指向循环头的回边。

本文所使用的具体的循环结构识别算法如下：

算法功能：基于IDA反汇编结果在汇编代码层面进行循环结构分析。算法输出：每个函数中的循环个数，每个循环的起止地址，每个循环中包含的基本块数。算法关键步骤：

（1）对当前函数进行预处理，添加基本块节点。基本块节点边界有如下几种情况：1.末尾时retn；2.有跳转；3.下一条语句远端交叉引用。

（2）在所有节点中循环遍历，寻找存在循环结构的节点对。对任意两个节点对，假设其中一个为master节点，另一个为slave 节点，判断master节点和slave节点之间是否存在循环，判定条件为：slave节点有一条

回边指向 master 节点，并且从master 节点出发存在一条通往 slave节点的通路。

（a）判定slave节点是否有指向master节点的回边可以直接通过IDA的交叉引用函数进行判定。

（b）判定master 节点到slave节点是否存在通路是本算法中比较复杂的地方，这里给出IDAPython编写的算法描述，算法的相关信息如下：

```
def have_path_to(src, dst):
    BBL_ea = src
    i = 0
    while BBL_ea != BADADDR:
        BBL_ea_end = slide_to_BBL_end(BBL_ea)
        fcref = Rfirst0(BBL_ea_end)
        next_BBL = slide_to_next_BBL(BBL_ea)
        if next_BBL == 0:
            return 0
        else:
            #进入该分支的次数等于循环中所含 node 的数目
            i = i + 1
            if BBL_ea == BADADDR or dst == BADADDR:
                return 0
            if fcref == dst or\
                (next_BBL == dst and GetMnem(BBL_ea) != 'retn'):
                if src != dst:
                    i = i + 1
                gl.loop_node_num[dst] = i
                return 1
            BBL_ea = next_BBL
```