基于标志位差异分析的整数溢出漏洞溢出点定位方法

黄克振[1,2] 连一峰[1] 陈 恺[1] 张颖君[1] 康 恺[1,2]
(中国科学院软件研究所 北京 100190)[1] (中国科学院研究生院 北京 100049)[2]

摘 要 近几年,整数溢出漏洞数量居高不下,危害性较大。目前,漏洞分析定位的方法仅在补丁自动生成或漏洞触发样本文件自动生成中有所涉及,且这些方法大多利用缓冲区溢出会覆盖其邻接内存数据的特点来进行定位分析,而整数溢出漏洞不具有直接覆盖重要数据的特点,所以现有的方法不能对其进行有效的定位分析。现阶段对整数溢出漏洞的分析大多依靠人工完成,效率较低。为了提高分析人员的工作效率,提出了一种结合动态污点分析技术进行EFLAGS标志位信息比对的方法,来将溢出点锁定在少量的地址中。在此基础上实现了一套整数溢出漏洞溢出点定位系统,并对提出的方法进行了验证。

关键词 漏洞定位,整数溢出漏洞,动态污点分析
中图法分类号 TP393.08 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2014.12.005

Locating Vulnerable Point for Integer Overflow Based on Flag Bits Differences

HUANG Ke-zhen[1,2] LIAN Yi-feng[1] CHEN Kai[1] ZHANG Ying-jun[1] KANG Kai[1,2]
(Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)[1]
(Graduate University of Chinese Academy of Sciences, Beijing 100049, China)[2]

Abstract In recent years, the number of integer overflow vulnerabilities is still high and they have great threat to security. However, in the previous study, methods of locating vulnerable code are only used when patches or vulnerabilities' proof of concept (POC) are automatically generated. Besides, when locating the vulnerable code, most of the previous methods tend to undermine buffer overflow that will cause its adjacent memory data to be overwritten. Integer overflow vulnerabilities, however, cannot directly overwrite important data, therefore, existing methods cannot locate integer overflow vulnerable code effectively. Currently, existing analysis of integer overflow vulnerabilities is inefficient and time-consuming as they are mostly conducted manually by manpower. In the present study, consequently, a novel method was proposed to locate vulnerable code of integer overflow. With view to enhance the efficiency on the part of analysts, this method combines dynamic taint analysis and EFLAGS register comparison so that it will decrease the number of instructions which can be used to locate the overflow point. On the basis of that, a system was further implemented and several experiments were conducted to verify our proposed method. The results show that our method is effective and efficient.

Keywords Vulnerabilities, Integer overflow, Dynamic taint analysis

随着信息技术的发展,如何保证网络和主机的安全性逐渐成为研究的重点。从攻击者的角度分析,攻入主机或网络内首先要发现可利用的漏洞,那么如何挖掘未知漏洞并利用漏洞(需要分析漏洞产生的机理)就成为了攻击者制胜的关键;而对安全研究人员而言,漏洞研究除了挖掘未知漏洞外,更重要的是分析导致漏洞发生的原因,并提出修复方法。近几年,涌现出了大量有关缓冲区溢出漏洞防范的研究成果,如地址随机化(ASLR)、数据执行保护(DEP)、栈区金丝雀保护(Stack Canaries)等[1]。随着上述研究成果的推广应用,缓冲区溢出漏洞的数量已呈下降趋势,但整数溢出漏洞的数量依然呈上升趋势[2]。整数溢出漏洞是指程序指令运算结果超出了变量类型所能表示的最大值而导致的溢出错误。整数溢出漏洞被攻击者利用将造成多种危害,如造成任意代码执行、触发缓冲区溢出、堆溢出等。

现有的研究工作缺乏对整数溢出漏洞溢出点定位的支持,漏洞的定位分析大多依赖人工分析:首先使用反汇编软件(如OllyDbg、IDA等)对可执行文件进行反汇编,然后结合漏洞触发样本文件进行动静态分析。整个分析过程需要解决以

到稿日期:2013-12-19 返修日期:2014-01-29 本文受国家自然科学基金项目(61100226),北京市自然科学基金项目(4122085),"十二五"国家科技支撑计划-IT产品信息安全认证关键技术研究(2012BAK26B01),国家高技术研究发展计划(863)(SQ2013GX02D01211,2013AA01A214)资助。

黄克振 硕士,助理工程师,主要研究领域为网络与系统测评;连一峰 博士,研究员,主要研究领域为网络与系统测评;陈 恺 博士,助理研究员,主要研究领域为网络与系统测评;张颖君 博士,副研究员,主要研究领域为网络与系统测评;康 恺 硕士生,主要研究领域为网络与系统测评。

下3个问题：

1)如何识别汇编程序中存在的大量无害溢出？程序进行编译优化时，为了减少比较操作的次数，编译器和优化器会通过添加整数溢出的方法实现程序流程的正常跳转，这些添加的整数溢出不会造成程序崩溃或使程序进入错误状态，属于无害溢出。

2)如何分析得到数据的类型信息？漏洞分析时，数据的类型信息无法从二进制的程序文件中获知，只能通过人工分析获得，而人工判断每条语句是否发生溢出需借助EFLAGS标志位来进行判断，但无符号数和有符号数的溢出标志位变化是不同的，如无符号数加法，CF位的值由0变为1即表示发生了溢出；而有符号数加法则需要查看OF=1是否成立。在二进制文件中，数据类型信息的不确定性和不同数据类型的溢出判断方法不同将导致人工分析效率较低。

3)如何判断溢出点处的数据是否来自于输入源？程序发布后，漏洞的触发多由外部输入数据引起，因此溢出点处的数据是否来自于输入源是判定某个溢出点是否为有害溢出点的一个判定条件。因为对每个溢出点都需要进行上述判定，所以分析效率较低。

上述难点导致分析定位效率较低，因此如何提高对整数溢出漏洞溢出点进行定位分析的效率成为研究重点之一。本文提出一种通过比对EFLAGS标志位差异来定位整数溢出漏洞的方法：首先，将运行畸形文件(能够使漏洞效果重现的文件)收集的溢出模式(见定义1)信息同运行正常文件收集的溢出模式信息进行比对，过滤无害溢出(见定义2)；然后，结合动态污点分析技术，对EFLAGS标志位的差异比较结果进行二次筛选，将溢出点锁定在更小的指令范围内，分析人员利用这些参考点进行进一步的分析，以提高工作效率。

本文第1节介绍相关工作；第2节介绍EFLAGS标志位信息收集方法和动态污点分析方法；第3节介绍原型系统，并进行实验评估，对本文方法的可行性和有效性进行验证；最后总结全文。

1 相关工作

目前针对整数溢出漏洞的已有研究主要集中在未知漏洞的挖掘方面[3-5,12]，而针对漏洞分析定位的研究较少，主要在补丁自动生成或漏洞触发样本文件自动生成[7,14,16-18]中有所涉及。

在整数溢出漏洞挖掘方面，IntScope[3]系统将反汇编的代码首先转化为IntScope的中间语言形式，进而在中间语言形式的代码上使用符号执行的方法进行敏感数据流分析。该方法并不实际执行程序，而是使用符号执行来检测程序的重要路径。该系统的符号执行方法精确度较低，且该系统不能正确模拟内存块操作函数如memmove、memcpy等。RICB[4]根据部分整数溢出漏洞会导致缓冲区溢出的特点，通过定位缓冲区溢出点并检测缓冲区的溢出是否由整数溢出漏洞所引起来挖掘整数溢出漏洞。如果整数溢出漏洞导致出现程序崩溃或非缓冲区溢出的特征时，该方法就会失效。IntFinder[5]使用动静态结合的方法进行整数漏洞的挖掘，该系统使用EFLAGS标志位作为溢出的判断标志，同时使用动态污点分析技术挖掘漏洞。该系统在静态分析阶段使用反编译系统，反编译系统存在对一些函数不能进行反编译或某些函数不能正确反编译的局限性，导致该方法具有同样的局限性。

在补丁自动生成和漏洞触发样本文件自动生成方面，AEG[16]第一次实现了从漏洞挖掘到漏洞触发样本文件(POC)的自动生成。该系统对漏洞的定位在源码级别上实现，尚不能单纯地依靠二进制代码实现，而且AEG主要针对基于栈的缓冲区溢出漏洞(stack-based buffer overflows)和格式化字符串漏洞(format string vulnerabilities)，尚不支持基于堆的缓冲区溢出漏洞(heap-based overflows vulnerabilities)和整数溢出漏洞(integer overflows vulnerabilities)。APEG[17]通过比对未打补丁的漏洞软件和修复后的软件来定位漏洞点，当软件的修复较多或者漏洞模块被重写时，二进制代码的比对效果将大大降低。当补丁不存在时，该方法则无效。ATAM[18]方法使用基于属性的污点分析技术确定漏洞点，并对漏洞点进行修复。该方法对漏洞的定位目前仅限于缓冲区溢出，对整数溢出的定位尚不支持。

在程序执行路径比对方面，Noah M. Johnson[19]等人提出使用程序切片的方法对不同的输入进行程序执行路径的对比，分析执行路径的分歧点，若使用该方法定位整数漏洞，那么输入文件的执行路径要严格的相似。

与已有研究成果相比，本文侧重于整数溢出漏洞溢出点的定位分析，通过研究分析提取整数溢出漏洞溢出点处的特点，并与动态污点分析方法相结合，提出了基于标志位差异分析的整数溢出漏洞溢出点定位方法，从而为漏洞分析人员提供了参考依据。

2 基于标志位比对的动态分析方法

基于标志位比对的动态分析方法，首先是在程序实际运行过程中收集标志位信息，并进行动态污点分析；然后对收集的标志位信息进行比对排除。本文2.1节将介绍标志位信息的收集存储方法；2.2节将介绍本文使用的动态污点分析方法；2.3节将介绍比对排除算法。

2.1 EFLAGS标志位信息收集存储方法

定义1(溢出模式) 溢出模式表示算数指令发生溢出后，EFLAGS标志位中CF、OF、SF和ZF位的变化情况。它用OM=〈InsAddress，oldCF，newCF，oldOF，newOF，oldSF，newSF，oldZF，newZF〉表示，其中oldXX表示地址为InsAddress的指令执行前XX位的数值(0或1)，newXX则表示指令执行后的值。简记为OM。

定义2(正常溢出模式) 正常溢出模式表示不会造成程序崩溃或进入错误状态的溢出模式(或称为无害溢出模式)，用OM_Normal表示，OM_Normal=OM_All-OM_Abnormal，其中OM_All表示程序收集到的所有OM的集合，OM_Abnormal表示整数溢出漏洞溢出点处的OM集合。

定义3(畸形溢出模式) 畸形溢出模式表示整数溢出漏洞溢出点处产生的溢出模式(或称为有害溢出模式)，用OM_Abnormal表示。

定义4(集合机制) 利用集合内不允许有重复元素的特点进行数据存储的机制，用Set表示，$Set=\{a_1,a_2,a_3,\cdots,a_n\}$，其中$a_1\neq a_2\neq a_3\neq\cdots\neq a_n$。

EFLAGS标志位信息的收集主要针对易发生整数溢出的ADD、SUB、MUL、IMUL等指令，在指令前后通过插装方法来读取EFLAGS寄存器信息，并将指令执行前后EFLAGS

寄存器的变化情况以 OM 的方式存储；对程序在执行过程中数据类型未知的问题，本文采取如下方法：只要某种运算满足有符号数或无符号数二者之一的溢出条件就对其进行记录，因为无符号数的溢出判断为 CF 和 ZF 的组合，有符号数的溢出判断为 SF^OF 与 ZF 的组合[20]，所以在溢出模式的定义中囊括了 CF、OF、SF、ZF 位，不再需要对数据类型做进一步判断。

针对出现在循环中的溢出模式，本文采用集合机制存储溢出模式来剔除重复的溢出模式，以减少不必要的信息量，如〈0x7c9425aa,0,1,0,0,0,0,0,0〉模式处于某个循环中，那么这种溢出模式会出现多次，集合机制能有效地减少程序中循环带来的大量无害溢出模式。

2.2 动态污点分析方法

动态污点分析主要是在程序运行过程中记录对输入数据进行操作的指令，即污点数据（输入数据）传播的过程。动态污点分析需要分析跟踪的起始点（污点数据传入点）、传播的过程（主要为传播规则）和跟踪的结束点（污点数据消亡点）。

首先，污点数据多从外部可控输入源进入，例如文件、网络数据包、命令行参数等，且常由特定的 API 函数（如 ReadFile、read、fread 等）引入，本文将从文件读取数据作为污点跟踪的起始点，对输入的数据进行标记，为每个内存单元设计相应的影子内存[10,13]，影子内存采用类似页表的数据结构，来保证影子内存不会占用太大的空间，并能快速进行访问。针对寄存器中的污点数据，本文采用类似于影子内存的方法设置影子寄存器，以数组的方式进行存储。

其次，污点的传播分析[11,15]针对每类指令进行，主要有数据传送指令、逻辑操作指令、二元运算指令等。针对需要运行在 Ring0 级别的指令如 sysenter/sysexit 等，本文不做跟进分析，而是通过拦截包含该指令的函数来解决。对于污点数据在寄存器中的传播，需要考虑同一个寄存器在不同形式访问时的情况。表 1 给出了 EAX 寄存器的污点传播规则，其中 $T[x]=0$ 表示未受污染状态，$T[x]=1$ 表示污染状态。其他寄存器的传播规则与 EAX 相同。

表 1 寄存器传播规则

寄存器	状态值
AL	T[AL]=T[AL]+T[AH]+T[AX]+T[EAX]
AH	T[AH]=T[AH]+T[AX]+T[EAX]
AX	T[AX]=T[AL]+T[AH]+T[AX]+T[EAX]
EAX	T[AX]=T[AL]+T[AH]+T[AX]+T[EAX]

最后，跟踪结束点。主要由 xor eax,eax、sub eax,eax、mov eax,0、(rep) stos n 等指令将变量所在内存或寄存器置零或者常数，使污点消亡。

2.3 比对排除

本文提出的方法主要涉及两次比对排除，第一次比对排除是运行畸形文件与正常文件收集的溢出模式信息的比对，目的在于排除重复的溢出模式和大量的共有无害溢出模式；第二次比对排除是将第一次比对结果（指令地址有序）和动态分析结果求交集，目的是排除不受污点数据影响的溢出模式，具体算法如算法 1 所示。利用算法 1 进行比对的方法能够有效地绕过程序执行路径上的循环、递归等重复次数不定带来的难点，比文献[19]使用切片技术在程序执行期间进行比对的方法要简单易行，因为文献[19]比对要求两次执行路径要严格相似，而本文对执行路径无任何要求，仅需要畸形文件和任意正常文件即可。

算法 1（溢出模式（OM）排除算法）

```
Input:
动态污点分析文件文件 dyTaintFile.out
畸形文件溢出模式记录文件 abnormFile.out
正常文件溢出模式记录文件 normFile.out
Output:
result.out 文件，记录整数溢出参考点
Algorithm: delNoTaintOM(dyTaintFile.out, abnormFile.out, normFile.out)
1. 集合机制处理文件 normFile.out
2. 将处理后的 normFile 存入红黑树结构 b-redTree 中
3. 集合机制处理文件 abnormFile.out
4. for each OM in abnormFile
5.     if OM not in b-redTree
6.         record OM 的指令地址至 firstCompare.out 文件内
7. 集合机制处理文件 dyTaintFile.out
8. 定义 bitmap 含 0x7fffffff bits
9. for i=0 to 0x7fffffff
10.            bitmap[i]=0
11.        for each insAddr in dyTaintFile.out
12.            bitmap[insAddr]=1
13.        for each insAddr in firstCom.out
14.            if bitmap[insAddr]==1
15.            output insAddr to result.out 文件
```

由于比对排除会涉及近百万条的数据量（下文的实验数据可以看出），因此本算法需要着重考虑时间效率。考虑到查询次数和存储的问题，本文算法中引入了红黑树（程序第 2 行）和位图（程序第 8 行）两种数据结构进行处理，总体上使算法的时间复杂度为 $O(n)$，$n=\max\{$dyTaintFile.out, abnormFile.out, normFile.out$\}$ 中指令的条数。

3 系统实现及实验评估

3.1 系统实现

根据上文的方法，本文设计并实现了原型系统。如图 1 所示，原型系统主要包含 4 个模块：EFLAGS 标志位信息收集模块、动态污点分析模块、共有正常溢出模式排除模块、未污染的正常溢出模式排除模块。其中 EFLAGS 标志位信息收集模块和动态污点分析模块主要基于 Pin 平台实现，通过插桩对二进制指令进行检测和分析，在 Pin 平台内部的实现中还采用了内联、寄存器重分配、指令调度等技术来保障程序执行的效率[6]。

首先，在 Pin 平台上分别运行畸形文件（abnormFile）和任意正常文件（NormFile），同时调用 EFLAGS 信息收集模块来收集溢出模式信息，并将收集的信息存入相应的文件（abnormFile.out 和 normFile.out）中；其次，将 abnormFile.out 和 normFile.out 两个文件输入至共有正常溢出模式排除模块，通过模块内的集合机制处理和剔除正常溢出模式运算，完成第一次比对排除，排除共有的正常溢出模式信息，并将结果输出至 firstCompare.out 文件内；然后，在 Pin 平台上再次运行畸形文件（abnormFile），同时调用动态污点分析模块，收集受污点数据污染的指令地址，并存入 dyTaintFile.out 中；最后，将 firstCompare.out 和 dyTaintFile.out 文件输入未污染

的正常溢出模式排除模块，通过模块内部的集合机制预处理和求交集运算，计算出参考结果并输出至 result. out 文件内。

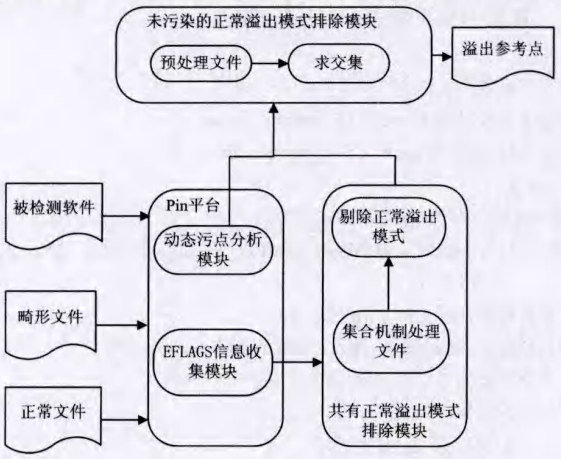

图 1　原型系统结构图

3.2　实验评估

本文所有的实验验证均在 Intel(R) 4 CPU、2GB 内存、OS 为 Microsoft Windows XP SP3 的机器上进行。首先测试简单的示例查看效果；然后针对实际的漏洞进行测试；最后进行系统性能分析。

(1)整数溢出实例分析

整数溢出示例如下：

```
1. int main()
2. {
3. string filename;
4. cin>>filename;
5. ifstream infile;
6. infile. open(filename. c_str(),ifstream::in);
7. if(! infile)
8. {
9.      cout<<"文件未打开"<<endl;
10.     exit(0);
11. }
12. int size=0;
13. while(! infile. eof())
14. {
15. char str;
16.     if(infile. read(&str,1))
17.     {
18.         int tmp=str-48;
19.         size=size * 10+tmp;
20.     }
21. }
22. cout<<size<<endl;
23. size * =500;
24. cout<<size<<endl;
25. return 0;
26. }
```

显然在程序的第 23 行，size * =500，当 size 达到一定的值后会发生溢出。在 VS2005 下的 Debug 模式下，编译链接程序进行测试。如图 2 所示，测试结果定位于地址为 0x40116e 的指令，使用 OD 软件对 exe 文件进行反汇编，查看 0x0040116E 处的指令，对立即数 0x1F4(=500)做乘法运算，与 23 行指令相同。初步证明定位系统的可行性。

```
00A0115C   |. E8 5F070000    CALL Integer0.std::basic_ostream<char,std::char_tr>
00A01161   |. 8BC8           MOV ECX,EAX
00A01163   |. E8 E0060000    CALL Integer0.std::basic_ostream<char,std::char_tr>
00A01168   |. 8B85 2CFFFFFF  MOV EAX,DWORD PTR SS:[EBP-D4]
00A0116E   |. 69C0 F4010000  IMUL EAX,EAX,1F4
00A01174   |. 8985 2CFFFFFF  MOV DWORD PTR SS:[EBP-D4],EAX
00A0117A   |. 68 A0134000    PUSH Integer0.std::endl
00A0117F   |. 8B85 2CFFFFFF  MOV EAX,DWORD PTR SS:[EBP-D4]
00A01185   |. 50             PUSH EAX
00A01186   |. B9 708A4700    MOV ECX,OFFSET Integer0.std::cout
00A0118B   |. E8 30070000    CALL Integer0.std::basic_ostream<char,std::char_tr>
```

图 2　OD 反汇编结果

使用原型系统进行实际整数漏洞测试，实验结果如表 2 所列。表中列出了上文的整数溢出示例程序、Microsoft Paint Integer Overflow Vulnerability(DoS) MS10-005(CVE-2010-0028)漏洞、RealPlayer FLV Parsing Integer Overflow(CVE-2010-3000)漏洞、Adobe Acrobat Font Parsing IntegerOverflow(CVE-2010-2862)漏洞的实验效果，其中第一列畸形文件溢出模式数是通过运行畸形文件收集到的溢出模式数；第二列是通过运行任意正常文件收集的溢出模式数；第三列的比对排除主要是对第一列的溢出模式集合和第二列的溢出模式集合求差集所得的集合元素数；第四列主要是通过运行畸形文件并进行污点分析获得的受污染指令条数；最后求交集后指令条数是第二次比对排除的结果，是将第一次的比对排除结果中的指令集合同污点分析获得的指令集合求交集得到的指令数(亦是系统最终给出的溢出点参考结果)。

为了验证本系统不需要畸形文件和正常文件执行路径严格相似的特性，本文对 Microsoft Paint 软件进行了两次实验：两次实验使用的畸形文件相同，而正常文件不同(如表 2 中的第 3 行和第 4 行所示)。第 3 行 Microsoft Paint(1)使用的正常文件与畸形文件很相似(图形内容相近)，理论上执行路径应该相仿；而第 4 行 Microsoft Paint 使用的正常文件与畸形文件差别很大，但是正常文件(人脸图像)比畸形文件(类似于一个矩形图片)在图形内容上要复杂得多，从实验结果来看，第二次的测试结果稍好，这一点可能与第二次打开正常文件时执行路径的覆盖率较高有关，基于这一点考虑，可以将畸形文件与多个正常文件进行比对，然后对每次的结果再次进行交集运算，以进一步减少信息量。

表 2　实验结果数据

程序名称	畸形文件溢出模式数	正常文件溢出模式数	第一次对比排除结果数	受污染指令条数	求交集后指令条数
整数溢出示例	1628	1596	5	142	1
CVE-2010-0028	1842329	1903377	90	1555260	60
CVE-2010-0028	1842329	2049954	80	1555260	54
CVE-2010-3000	3249565	3820474	931	21879563	82
CVE-2010-2862	716884	1560899	376	2385960	122

从表 2 可知，定位系统给出的参考指令数在 100 条左右，实际上这些指令中属于程序地址空间的仅占很少一部分，其他属于系统 DLL(如 kernel32. dll、ntdll. dll 等)的指令范围。虽然属于系统 DLL 的指令可以排除，但这些参考指令对一些漏洞的分析亦具有一定的参考意义，如在 Winamp 5.551 MAKI Paring Integer Overflow(CVE-2009-1831)漏洞中，由于使用符号扩展，导致将－1 作为移动字节数传入 memmove 中，而在 memmove 函数的内部实现中会对移动的字节数进

行 shr 操作，对传入的－1 进行 shr 操作后就会出现溢出，在此漏洞的分析中就可以参考这些指令的信息。对 CVE-2009-1831 漏洞的分析，除使用对传入函数的参数进行拦截验证的方法外，本文方法亦是可行的。

(2)性能分析

系统的主要时间消耗在动态污点分析模块和 EFLAGS 信息收集模块，这两个模块均基于 Pin 平台实现。表 3 显示了 Pin 平台上运行的时间开销，其中第二栏为无任何检测时 Pin 平台的运行时间，主要包括 Pin 平台下 Symbols 的初始化和手动打开畸形文件的时间；动态污点检测的平均时间增长 4.9 倍，EFLAGS 信息收集的平均时间增长 2.7 倍；动态污点分析模块的时间开销主要在污点分析和信息记录中，而 EFLAGS 信息收集模块主要在信息记录时消耗时间，因为需要实时向文件中记录信息，从而导致文件写入操作频繁，带来时间开销。

表 3　时间开销对比(畸形文件测试数据)

软件名	无任何检测时的 Pin 平台运行时间(s)	动态污点检测时间(s)	EFLAGS 信息收集时间(s)
整数溢出示例	5	14	12
Microsoft Paint	44	175	171
RealPlayer	167	795	420
Adobe Reader	77	622	161
平均开销倍数		4.9x	2.7x

除了在检测时带来部分额外时间开销外，本系统的另一个不足之处是对一些不会影响标志位变化的整数溢出不具有检测作用，如图 3 所示，VLC 1.1.11 libavcodec_plugin.dll 中的一个漏洞，在 0x03F42C25 处会将 poc 文件中偏移量为 0x4、0x5 的数 0x0096 存入 EDX，然后减去 0xF6E 作为地址存入 EAX 中，这个溢出过程就不会影响到标志位，以后系统的改进应该将该问题考虑在内。

```
03F42C17  893424          MOV DWORD PTR SS:[ESP],ESI
          E8 E14AFFFF     CALL libavcod.03F37700
03F42C1F  89C2            MOV EDX,EAX
03F42C21  66:8943 18      MOV WORD PTR DS:[EBX+18],AX
03F42C25  8D82 92F0FFFF   LEA EAX,DWORD PTR DS:[EDX-F6E]
03F42C2B  66:83F8 28      CMP AX,28
03F42C2F  0F87 E7050000   JA libavcod.03F4331C
03F42C35  66:81FA 8B0F    CMP DX,0F8B
```

图 3　OD 反汇编代码

结束语　挖掘和分析漏洞是信息安全研究的重要课题，现有的一些研究工作在挖掘整数溢出漏洞方面取得了很大的进展，但对如何高效地分析整数溢出漏洞这一问题，尚未提出较好的方法。本文首先分别收集畸形文件和正常文件运行时的溢出模式，然后进行比对以排除大量无害的溢出模式，最后结合动态污点分析技术，排除未受污点数据影响的无害溢出模式，将整数溢出点锁定在一定的指令地址范围内，提高了整数溢出漏洞分析人员的工作效率。

实验结果表明，整数溢出漏洞的分析定位可以达到较好的效果，但是仍具有一定的局限性。如何扩展系统使其具有自动分析其他漏洞类型的能力将成为以后工作的重点。

参 考 文 献

[1] Silberman P, Johnson R. A comparison of buffer overflow prevention implementations and weaknesses [R]. IDEFENSE, August, 2004

[2] Christery S, Martin R A. Vulnerability type distributions in CVE [R]. Mitre report, May 2007

[3] Wang T, Wei T, Lin Z, et al. Intscope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution[OL]. www.isoc.com/isoc/conferences/ndss/09/pdf/17.pdf

[4] Wang Y, Gu D, Xu J, et al. RICB: Integer overflow vulnerability dynamic analysis via buffer overflow [M]// Forensics in Telecommunications, Information, and Multimedia, 2011: 99-109

[5] Chen P, Han H, Wang Y, et al. intfinder: automatically detecting integer bugs in x86 binary program [M]// Information and Communications Security, 2009: 336-345

[6] Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation; proceedings of the ACM SIGPLAN Notices[C]// F, 2005. ACM, 2005: 190-200

[7] YanivMiron aka Lament. Microsoft Patch Analysis. Confidence 2010[OL]. http://www.intelligentexploit.com/articres/Microsoft-Patch-Analysis.pdf

[8] CVE stack overflow[OL]. http://www.cve.mitre.org/cgi-bin/cvekey.cgi? keyword=stack%20overflow

[9] CVE integer overflow [OL]. http://www.cve.mitre.org/cgi-bin/cvekey.cgi? keyword=integer%20overflow

[10] Newsome J, Song D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software [OL]. citeseerx.ist.psu.edu/viewdoc/summary? doi=10.1.1.62.8372

[11] 陈恺，苏璞睿，冯登国. 基于延后策略的动态多路径分析方法[J]. 计算机学报，2010，33(3)：493-503

[12] Wang Y, Ruan D, Tang Z, et al. RICF: Dynamic Analysis of Integer Arithmetic Overflow Vulnerability via Finite State Machine [J]. Journal of Computational Information Systems, 2010, 6(6): 1933-41

[13] Chen K, Feng D G, Su P R, et al. Black-box testing based on colorful taint analysis [J]. Science China Information sciences, 2012, 55(1): 171-83

[14] Jeongwook O H. ExploitSpotting: Locating Vulnerabilities Out of Vender Pathes Automatically[OL]. http://www.blackhat.com/htrul/bh-us-10/bh-us-to-briefings.html

[15] Chen Kai, Lian Yi-feng, Zhang Ying-jun. AutoDunt: Dynamic Latency Dependence Analysis for Accurate Detection of Zero Day Vulnerabilities[C]// ICICS. 2010: 367-382

[16] Avgerinos T, Cha S K, Hao B L T, et al. AEG: Automatic exploit generation[J]. Communications of the ACM, 2011, 57(2): 74-84

[17] Brumley D, Poosankam P, Song D, et al. Automatic patch-based exploit generation is possible: Techniques and implications[C]// 2008 SP 2008 IEEE Symposium on Proceedings of the Security and Privacy. IEEE, 2008

[18] Chen K, Lian Y, Zhang Y. Automatically generating patch in binary programs using attribute-based taint analysis [M]. Information and Communications Security, 2010, 367-82

[19] Johnson N M, Caballero J, Chen K Z, et al. Differential slicing: Identifying causal execution differences for security applications [C]// 2011 IEEE Symposium on Proceedings of the Security and Privacy (SP). IEEE, 2011: 347-362

[20] Bryant R, David Richard O H. Computer systems: a programmer's perspective [M]. Prentice Hall, 2003