doi :10.3969/j.issn.1671-1122.2017.05.003

二进制程序整型溢出漏洞的自动验证方法

彭建山 [1,2], 奚琪 [1,2], 王清贤 [1,2]

（1. 解放军信息工程大学，河南郑州 450002；2. 数学工程与先进计算国家重点实验室，河南郑州 450002）

摘 要：整型溢出漏洞已成为威胁软件安全的第二大类漏洞，现有的整型溢出漏洞挖掘工具不支持自动验证漏洞，且现有的漏洞自动验证工具不支持整型溢出漏洞模式。因此，文章提出了一种二进制程序整型溢出漏洞的自动验证方法以填补这一空白。针对整型溢出漏洞中有价值的 IO2BO 漏洞，为避免程序在缓冲区溢出过程中发生 Crash 导致无法劫持控制流，通过污点分析建立可疑污点集合以缩小待分析污点范围，利用污点回溯技术追踪污点来源，通过符号执行收集内存读写操作的循环条件，控制循环次数以覆盖堆栈关键数据，最后通过约束求解生成新样本，将 IO2BO 漏洞的自动验证问题转化为传统缓冲区溢出漏洞的自动验证。实验证明该方法能够自动验证典型的 IO2BO 漏洞，生成能够劫持控制流并执行任意代码的新样本。

关键词：漏洞验证；符号执行；污点分析；整型溢出；缓冲区溢出

中图分类号：TP311　　**文献标识码：**A　　**文章编号：**1671-1122（2017）05-0014-08

中文引用格式：彭建山，奚琪，王清贤. 二进制程序整型溢出漏洞的自动验证方法 [J]. 信息网络安全，2017（5）：14-21.

英文引用格式：PENG Jianshan, XI Qi, WANG Qingxian. Automatic Exploitation of Integer Overflow Vulnerabilities in Binary Programs[J].Netinfo Security,2017(5):14-21.

Automatic Exploitation of Integer Overflow Vulnerabilities in Binary Programs

PENG Jianshan[1,2], XI Qi[1,2], WANG Qingxian[1,2]

(1. *PLA Information Engineering University*, *Zhengzhou Henan* 450002, *China*; 2. *State Key Laboratory of Mathematics Engineering and Advanced Computing*, *Zhengzhou Henan* 450002, *China*)

Abstract: Integer overflow vulnerabilities have become the second largest threat to software security. The existing tools for mining integer overflow vulnerability do not support automatic exploitation. Neither do the automatic exploitation tools support integer overflow vulnerability. To fill the gaps we proposes an automatic exploitation method of integer overflow vulnerabilities in binary programs. Aiming at the valuable IO2BO vulnerability of integer overflow, firstly trying to avoid crashing in the process of buffer overflow, which would make hijacking control-flow fail. Secondly building suspicious taint set to reduce the scope of taints. Thirdly collecting the loops condition of reading and writing memory by taint analysis and symbolic execution. Lastly overwriting the critical data in the stack and heap by controlling the number of loops and generating new samples for testing by solving constraint. The proposed method can transform the automatic exploitation of IO2BO vulnerability into that of traditional buffer overflow vulnerability. The test results show that this method work well for the typical IO2BO vulnerabilities and could generate new samples for hijacking the control-flow of testing programs.

Key words: vulnerability exploitation; symbolic execution; taint analysis; integer overflow; buffer overflow

收稿日期：2017-4-15

基金项目：河南省自然科学基金 [162300410187]

作者简介：彭建山（1979—），男，江西，副教授，博士研究生，主要研究方向为网络空间安全；奚琪（1978—），男，河南，副教授，博士，主要研究方向为网络空间安全；王清贤（1960—），男，河南，教授，硕士，主要研究方向为网络空间安全。

通信作者：彭建山 jxpjs@163.com

1 研究背景

整型溢出漏洞已成为威胁软件安全的第二大类漏洞[1,2]。许多工具实现了自动挖掘二进制程序整型溢出漏洞的功能：IntScope[3] 和 SmartFuzz[4] 采用 source-sink 模式，提取程序中从用户输入到整型溢出敏感点之间的所有执行路径，通过收集变量约束信息生成测试用例，判定路径上的整型操作能否导致整型溢出。IntFinde[5] 通过反汇编还原相对完整的类型信息，抽取可疑指令集进行整型溢出测试，降低了误报漏报率。DIODE[6] 产生满足完整性检查的输入以测试和触发整型溢出。

然而，以上工具都没有漏洞自动验证的功能。漏洞验证是漏洞研究中必不可少的一个环节，通过对漏洞挖掘得到的 Crash 样本进行可利用性分析和危害等级验证，可以评估漏洞危害程度,及时修补漏洞。由于漏洞在成因、机理、环境等方面存在复杂性，当前的漏洞验证还是以人工方式为主。

随着污点分析[7]、符号执行[8]等技术在程序分析领域的广泛应用，为了与自动化漏洞挖掘相适应，人们尝试将漏洞验证也进行自动化。针对二进制程序，BRUMLEY 等人首次提出了基于二进制补丁比较的漏洞自动验证方法 APEG[9]，但该方法只能自动验证已公布了补丁的漏洞。HEELAN 提出了一整套基于控制流劫持的漏洞自动验证方法 AXGEN[10]，通过动态插桩和污点分析技术将软件漏洞分为不可利用、直接利用和间接利用 3 种类型，通过符号执行技术构造 EIP 跳转地址和 ShellCode 等关键输入数据，能够自动验证基于栈溢出的覆盖返回地址和函数指针的漏洞，但不能分析多路径情况，也不能解决在控制流劫持点之前程序提前崩溃的问题。Mayhem[11] 结合了在线式符号执行和离线式符号执行技术的优点，较好地缓解了路径爆炸和资源消耗等问题，但其过于依赖对软件输入接口和库函数的建模而无法高效处理大型程序。CRAX[12] 能够对 Fuzz 产生的异常样本进行自动快速的危害验证，同样使用了符号执行技术，重点解决了控制流劫持点（EIP 被符号化）定位、可注入 ShellCode 的连续符号化内存探测等问题，适用于 Microsoft Office 等大型软件的漏洞验证。

以上漏洞自动验证工具主要针对堆栈溢出、格式化字符串溢出等漏洞类型，没有专门处理整型溢出漏洞。

综上所述，现有的整型溢出漏洞挖掘工具不支持漏洞自动验证功能，同时现有的漏洞自动验证工具不能够验证整型溢出漏洞。针对这一现状，本文提出了一种基于污点分析和符号执行技术的二进制程序整型溢出漏洞的自动验证方法，针对整型溢出漏洞的特点，计算循环约束条件，通过符号执行收集路径约束，通过约束求解器生成新的输入样本，将问题转换为传统缓冲区溢出漏洞的自动验证，弥补了现有漏洞自动验证工具的不足。

2 整型溢出简介

2.1 整型溢出原理

整型是高级编程语言中常见的数据类型，包括多种字节长度的定义，如单字节、双字节、四字节以及八字节等，存在无符号和有符号两种符号定义类型，被广泛用于算术运算、内存地址、整数表达等。计算机系统中主要存在以下容易导致整型溢出的问题[13]。

1）宽度溢出。尝试向一个整型变量存入一个大于其表达范围的整型值时，只保留整型值的低位部分。

2）运算溢出。没有考虑加法、乘法等运算产生的进位，运算结果的高位部分被截断。

3）符号溢出。错误地将有符号整型变量视为无符号值，或将无符号整型变量视为有符号值。

以上问题将导致程序错误地使用整型值，从而在后续的逻辑中出现难以预估的错误。在二进制程序中检测整型溢出较为困难，主要原因是编译过程消除了整型变量的类型定义，仅通过汇编代码难以准确还原变量的表达范围和符号信息，现有工具一般通过类型推断（Type Inferrence）[14,15]、别名分析（Alias Analysis）[16] 等技术来还原变量信息，准确率较低。

2.2 整型溢出危害

整型溢出漏洞是一类特殊的漏洞，单纯的整型溢出不会导致程序发生执行错误或劫持程序控制流，攻击者需要同时利用其他漏洞才能造成危害。如果整型溢出产生的畸形整型值被用于之后的缓冲区操作，则可能发生缓冲区溢出，称之为整型溢出导致的缓冲区溢出（Integer Overflow to Buffer Overflow，IO2BO）[17]。以下代码存在典型的 IO2BO 漏洞。

```
void CopyIntArray(int*a,DWORD len ){
```

```
    int*b=(int*)malloc(len*sizeof(int));
    for(DWORD i=0;i<len;++)
      i[b]=i[a];
}
```

如果将用户输入的 *len* 设置为 0x80000001，则缓冲区 b 实际申请的大小为 4 字节，而之后的循环内存操作次数仍然为 0x80000001 次，显然很快会发生内存读写错误。

有实质危害的整型溢出漏洞大部分是 IO2BO 类型，也是本文的研究对象。

3 问题引入和目标设定

3.1 问题引入

现有漏洞挖掘工具能够发现大量 Crash 样本，其中包含了各类漏洞类型。对于缓冲区溢出类型程序 Crash 的原因典型情况如下所示。

1）EIP 指向无效地址或非法指令，一般是缓冲区溢出已经完成，覆盖了返回地址或函数指针等数据导致控制流被劫持。

2）读指令或写指令的内存地址错误，且该指令不处于 *memcpy*、*strcpy* 等内存块操作函数或内存操作循环结构中，一般是缓冲区溢出已经完成，关键数据被覆盖导致程序之后的流程发生了错误。

3）读指令或写指令的内存地址错误，且该指令处于 *memcpy*、*strcpy* 等内存块操作函数内或内存操作循环结构中，一般是缓冲区溢出正在发生，但覆盖范围太大超过了内存区块边界导致错误。

以上 3 种情况既可能是单纯的缓冲区溢出漏洞，也可能是 IO2BO 漏洞，各种情况缓冲区溢出漏洞类型如表 1 所示。

对于情况 1），AXGEN[10]、CRAX[12] 等工具已经可以很好地对此类情况实现漏洞自动验证。情况 2)与情况 1)类似，但在控制流劫持之前就发生了错误，CRAX 工具通过收集条件约束、构造缓冲区数据来验证漏洞。情况 3）一般是由内存操作边界值过大造成的，如果是单纯缓冲区溢出漏洞，现有工具通过构造合适的输入长度来避免边界值过大问题，但对于表 1 所示的 IO2BO 漏洞，要达到劫持控制流的目的则需要构造合适的 *len* 变量值以避免发生内存读写错误。现有工具并不能做到这一点。

表 1 现有工具能够自动验证的缓冲区溢出漏洞类型

	程序 Crash 点	单纯缓冲区溢出	IO2BO
情况1）	缓冲区溢出已发生 且控制流已被劫持	✓	✓
情况2）	缓冲区溢出已发生 且控制流还未被劫持	✓	✓
情况3）	缓冲区溢出正在发生	✓	✖

3.2 目标设定

本文实现一种整型溢出漏洞的自动验证方法，达到以下目标：

1）针对二进制程序的 IO2BO 漏洞进行分析。

2）尽可能避免 IO2BO 漏洞中因缓冲区溢出覆盖范围过大导致的内存读写错误。

3）通过缓冲区溢出能够覆盖关键数据，使程序达到控制流劫持状态。

4）将 IO2BO 漏洞的自动验证问题转换为缓冲区溢出漏洞的自动验证问题，因为后者已被现有工具较好地解决。

4 整型溢出漏洞的自动验证过程

本文基于开源工具 BAP[18] 开发实现了一套二进制整型溢出漏洞的自动验证系统 IntExp，主要完成将 IO2BO 漏洞验证转换为缓冲区溢出漏洞验证的工作，之后借助 CRAX 工具自动验证缓冲区溢出漏洞。

4.1 应用场景和工作流程

IntExp 系统的应用场景如下：通过漏洞挖掘工具对某个二进制程序进行漏洞挖掘，发现了 Crash 样本，以此作为 IntExp 系统的输入，输出是以下三者之一：1）非整型溢出漏洞，不处理；2）不可验证的漏洞，不处理；3）输出能够导致程序发生控制流劫持的样本。

因此 IntExp 系统的实现包含 3 个主要步骤：1）判断是否 IO2BO 漏洞；2）是 IO2BO 漏洞，判断能否交给 CRAX 处理；3）对 CRAX 不能处理的，如表 1 所示的情况，尝试修改 Crash 样本，将问题转换为对缓冲区溢出漏洞的验证，再交给 CRAX 处理。IntExp 工作流程如图 1 所示，后续章节详细描述以上步骤。

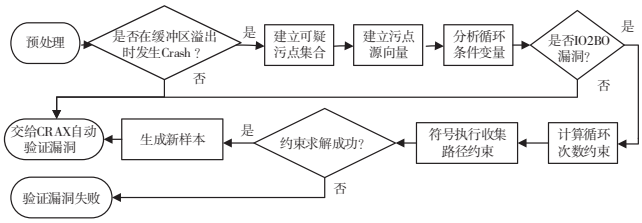

图 1 IntExp 工作流程图

4.2 预处理

该步骤判断 Crash 点是否处在缓冲区溢出正在发生的时刻，本文采用了两种方法。

1）查看出错指令是否是读写内存指令，并且发生在内存块操作函数的内部，包括 *memcpy*、*strcpy*、*strncpy*、*strcat* 等函数。需要注意的是，如果程序不是以外部调用而是内联编译的方式使用这些函数，IntExp 借助 IDA[18] 等反编译工具的分析结果来辅助定位函数。

2）查看出错指令是否是读写内存指令，并且处于循环操作中。IntExp 搜索出错指令的前后若干条指令，判断是否存在循环结构，例如，

mov [*esi*], *eax*

cmp ecx, *edx*

jb lastaddress

指令组合形成了循环结构。另外也将 *rep mov* 等特殊的单个内存块读写指令视为循环结构。

如果以上条件都不满足，则直接将 Crash 样本交给 CRAX 处理。

4.3 建立可疑污点集合

在进一步筛选 Crash 样本是 IO2BO 还是单纯缓冲区溢出之前，IntExp 使用污点分析技术，从程序的输入点开始记录污点传播路径，并建立可疑污点集合（Suspicious Taint Set，STS），以方便后续的污点回溯。

图 2 展示了 IntExp 记录的从程序输入点到 Crash 点的污点传播路径，其中内存读写循环已经被展开。

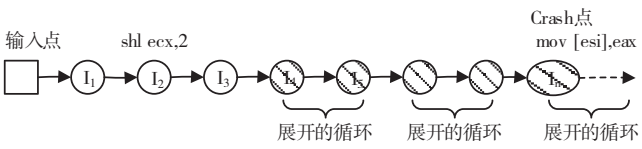

图 2 污点传播路径示意图

为适用于大型程序，IntExp 的污点记录方式与 BAP[18]、BitBlaze[20] 等工具相同：对混合污点，只记录污点是否被污染，而不记录其污点来源。例如，污点 *eax* 来源于样本中的字节偏移 1，*ebx* 来源于偏移 2，分别标记为 *eax*[*u*32,1]，*ebx*[*u*32,2]。如果有 *add eax*,*ebx* 指令，则 *eax* 被标记为 *eax*[*u*32,–1]，即混合污点。这种记录方法可以加快分析速度，减小内存消耗，但也存在无法获得混合污点来源的问题。本文在 4.4 节通过污点回溯以消除该问题。

为建立 STS，IntExp 首先从 Crash 点往前识别出污点传播路径上被展开的循环结构，并将结构中疑似与循环条件相关的污点都放入 STS 中，例如循环中出现的 *cmp ecx*,*edx*;*jb* 指令组合，如果 *ecx* 是污点，则将该指令在传播路径上的序号和 *ecx* 标记存入 STS。虽然理论上可以通过循环边界来准确获得循环条件，但考虑到存在非循环相关的跳转指令和嵌套循环等复杂情况，本文将循环结构中的所有可疑污点都存入 STS，算法 1 展示了识别循环结构和建立 STS 的过程。

算法 1

```
Input: taint record {I1, I2,... In},In is crash instruction
Output: STS
RecognizeLoop(In)
{
  loop_set = {};
  for i = n to 1
    if not Ii ∈ loop_set then
      loop_set = loop_set ∪ Ii
    else
      loop_set_next = {Ii–1,Ii–2,⋯,Ii–(n–i)};
      if loop_set == loop_set_next then
        return loop_set
      else
        loop_set = loop_set ∪ Ii
      end if
    end if
  end if
}
STS = {};
loop_set = RecognizeLoop(In);
foreach Ii ∈ loop_set do
```

```
if isCompareInstruction(Ii) then
    if isJmpInstruction(Ii+1, Ii+2,Ii+3) then
        STS = STS ∪ Ii
    end if
  end if
end for
```

4.4 通过污点回溯建立污点源向量

如 4.3 节所述，IntExp 没有记录混合污点的具体来源，这对一般的符号执行工具而言并不是问题，因为它们的正向符号执行技术会记录每一个符号变量的计算过程。混合污点记录加正向符号执行技术虽然能保证较高的计算准确率，但如果输入是大样本则会给符号执行带来很大的性能开销，不适用于大型程序。

IntExp 使用混合污点记录加污点回溯技术，追踪 STS 中的每一个污点的具体来源，即在样本中的偏移，为每一个污点建立污点源向量，方便分析计算与循环条件相关的变量，达到两个目的：1）判断是否为 IO2BO 漏洞；2）方便挑选与整型溢出漏洞相关的污点，缩小待分析污点的范围，减少符号执行开销。

污点回溯是一种反向的污点传播分析：对一个给定的污点，分析指令的语义，翻转数据的传播方向。例如 *mov eax,ebx* 指令，如果 *ebx* 被标记为污点 *ebx*[*u*32,–1]，则观察 *eax* 的污点标记是否包含具体来源，否则继续回溯 *eax* 的污点来源。

污点源向量（Taint Source Vector，TSV）包含了污点的准确来源，定义如下：

$$STV(I_i)=(Source\{t_1,[t_2]\cdots[t_m]\}\rightarrow Dest\{t_1,[t_2]\cdots[t_m]\}),I_i\in STS_i$$

其中 t 是文件偏移量，污点源 *Source* 变量可能包含多个污点 t_1，t_2 等，传播到目的变量 *Dest* 形成混合污点。

污点回溯和建立 TSV 的过程如算法 2 所示。

算法 2

```
Input: Ii in STS
Output: TSV(Ii)
SearchPreviousSource(Ii)
{
  foreach src in Ii .src_set
    if src.taintflag > 0 then
      TSV(Ii) = TSV(Ii)∪({src.taintflag} → {src.taintflag})
    else
      for j = i–1 to 1
        if Ij.dest == src and not Ij.dest.taintflag == 0 then
          SearchPreviousSource(Ij)
          break
        end if
      end for
      if Ij.dest.taintflag > 0 then
        TSV(Ii) = TSV(Ii)∪({Ij.dest.taintflag} → {Ij.dest.taintflag})
        end if
    end if
  end for
}
```

4.5 分析循环条件变量

STS 包含了疑似循环条件的污点，需要通过识别循环边界准确定位其中的循环条件变量，以方便计算循环条件。考虑到嵌套循环的情况，如图 3 所示。

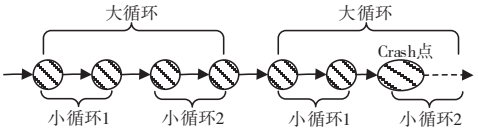

图 3 嵌套循环示意图

需要对每一层循环都分析出循环条件变量，才能精确计算内存操作边界值，控制缓冲区溢出范围。因此本步骤使用算法 1 中的 RecognizeLoop 函数先识别 STS 中的里层循环，再依次向外识别嵌套循环，最后将边界上的比较和跳转指令组合视为循环条件变量的计算依据，将其中用于比较的变量视为循环条件变量，记为 LC_i，如 *cmp ecx,edx* 指令中的 *ecx* 和 *edx*,*cmp ecx*,10 指令中的 *ecx*。

4.6 判断 IO2BO 漏洞

判断是否 IO2BO 漏洞的关键有两点。

1）在 STS 中定位导致缓冲区溢出的循环条件变量。通过 4.5 节可以发现 LC_i 与 Crash 点相关，即 LC_i 就是导致缓冲区溢出的循环条件变量。

2）该变量在循环之前是否与整型溢出相关。整型溢出漏洞至少包含了 3 种不同的类型：运算溢出、宽度溢出和符号溢出，每个类型都具有明显的指令特征，需要分析

LC_i 在 STS 中是否与这些指令相关。例如，图 3 中的 *ecx* 是 LC_i，在循环之前经过了 *shr ecx*,2 指令，是典型的运算溢出。表 2 示例了部分与整型溢出相关的指令。

表 2 整型溢出相关指令列表

整型溢出类型	相关的指令示例
运算溢出	*add, sub, imul, shl, mov/lea(with *, +)*
宽度溢出	*movzx, movsx*
符号溢出	*jg, jge, jl, ja, jb*

如果不是 IO2BO 漏洞，说明是单纯的缓冲区溢出，直接将 Crash 样本交给 CRAX 处理。

4.7 计算循环次数约束

获得循环次数的目的是为了精确控制缓冲区溢出的覆盖范围，实际包含了两个要解决的问题：

1）每次循环覆盖的字节数 *OverwriteBytesPerLoop*；

2）需要覆盖的字节数 *OverwriteBytesNeeded*。

为简单起见，IntExp 通过试探法获得 *OverwriteBytesPerLoop*。在已经获得大小循环边界的条件下，在循环开始前通过动态插桩技术向可能待写入的内存区域填入连续的随机字节，例如 *0xfd*，在循环结束时计算被改写内存的大小，即是每次循环覆盖的字节数。为避免误判，再用随机字节试探一次。

为获得 *OverwriteBytesNeeded* 需要区分栈溢出和堆溢出两种缓冲区溢出情况。1）验证栈溢出漏洞，一般需要覆盖到返回地址、保存在局部变量中的函数指针、SEH 链表等关键数据，可以通过 IDA 等反汇编工具获得这些数据在内存中的位置。2）而验证堆溢出漏洞较为复杂，简单起见，IntExp 在不触发内存访问错误的条件下尽可能覆盖到当前堆之后的其他堆，需要借助一些可主动识别内存的可访问性的系统函数试探可覆盖堆内存的范围，例如 Windows 下的 API 函数 *IsBadWritePtr* 等。

循环次数 *LoopNumber* 的计算公式为：

$$LoopNumber = \left\lfloor \frac{OverwriteBytesNeeded}{OverwriteBytesPerLoop} \right\rfloor \cdots\cdots\cdots\cdots (1)$$

4.8 通过符号执行生成新样本

IntExp 的最终目的是通过修改原 Crash 样本的内容，将整型溢出漏洞转换为可控制的缓冲区溢出漏洞，IntExp 的输出是修改后的新样本。

BAP 工具通过 Pin[21] 记录程序的执行轨迹，将执行轨迹转换为中间语言（Intermediate Language，IL）后进行符号执行。IntExp 基于 BAP 的实现方式，通过符号执行生成新样本的具体步骤如下：

1）以 Crash 样本作为输入运行程序，记录执行轨迹；

2）建立 STS，通过污点回溯建立 TSV；

3）分析得到循环条件变量 LC_i，计算循环次数 *LoopNumber*；

4）将执行轨迹转换为 IL，标记其中的 LC_i 的位置；

5）对 IL 进行符号执行，收集与 LC_i 相关的约束条件，包括路径分支约束和循环次数约束。由于循环已经被展开，所以这里只记录最里层循环的路径分支约束。生成约束公式如下：

$$Path_{LC_1} \wedge Path_{LC_2} \wedge \cdots \wedge Path_{LC_n} \cdots\cdots\cdots\cdots (2)$$

其中，*n=LoopNumber*

6）将公式（2）交给 STP 求解器求解，如果求解失败说明无法生成新样本，否则将求解结果与 TSV 相匹配，修改原始 Crash 样本为新样本。

7）最后将新样本交给 CRAX 自动验证漏洞。

5 实验评估

5.1 功能测试

本实验测试 IntExp 自动验证整型溢出漏洞的功能。编写了 3 个被测程序，每个程序包含了一种类型的整型溢出漏洞，并提供了各个程序的初始 Crash 样本[22]。被测程序的编译环境为 Visual Studio 2012，编程语言为 C++。

实验原理如下：Crash 样本能导致程序崩溃，但由于输入长度过大导致 Crash 点发生在内存块操作函数如 *memcpy* 或内存操作循环结构中，不能直接用 CRAX 进行自动验证。IntExp 收集和计算保证内存操作不会超过可读写范围的约束条件，求解后修改了 Crash 样本的输入长度并生成新样本，从而将 IO2BO 漏洞转换为缓冲区溢出漏洞并交给 CRAX 验证。

表 3 中“关键语句”一栏是被测程序与漏洞相关的主要代码，其中加粗部分是漏洞成因；“Crash 样本”一栏表示了导致被测程序 Crash 的关键输入值；“Crash 点”一栏

表 3 IntExp 对整型溢出漏洞的验证结果

整型溢出漏洞类型	关键语句	Crash 样本	Crash 点	约束条件	新样本	CRAX 验证结果
运算溢出	**WORD len = inputlen*sizeof(int);** int* myarray = (int*)malloc(len); for(i=0;i<inputlen/4;i++) myarray[i] = inputarray[i];	inputlen=0x4001	mov [edx+ecx*4],eax	inputlen*4>=0x10000 && inputlen <= NextHeapSize + (inputlen*4)&0xFFFF	inputlen=0x4ff2	✓
宽度溢出	**BYTE len = strlen(input);** char* buf = (char*)malloc(256); **if (len < 256)** strcpy(buf, input);	input="AAA…A" (20000 个)	strcpy	strlen(input) >= 256 && strlen(input) <= NextHeapSize + 256	input="AAA…A" (19008 个)	✓
符号溢出	short inputlen; char buf[0x7ffff]; short size = sizeof(buf); if(inputlen <= size) **memcpy(buf,inputbuf,inputlen);**	inputlen=0xffff	memcpy	inputlen>=0x8000 && inputlen <= RetAddressInStack - buf	inputlen=0x8023	✓

表示了被测程序 Crash 指令发生的位置；“约束条件”一栏表示了 IntExp 生成新样本时需要满足的约束条件；“新样本”一栏表示了新样本中的关键输入值，注意到与“Crash 点”一栏中的输入值有明显的区别；最后一栏表明 IntExp 产生的新样本都能被 CRAX 工具处理以控制流劫持，达到了自动验证以上整型溢出漏洞的目的。

5.2 实例测试

为测试 IntExp 对真实漏洞的验证能力，本文选用了 5 个真实的 IO2BO 漏洞实例作为测试集，分别代表了较为常见的整型溢出漏洞效果，并为每个 CVE 漏洞准备了一个初始 Crash 样本作为 IntExp 系统的输入。为了与 4.1 节所述的应用场景相匹配，即 Crash 样本是通过漏洞挖掘得到的，准备的 Crash 样本会导致程序 Crash 发生在缓冲区溢出时，而不是发生在诸如 *call* 指令等控制流劫持位置。

测试结果如表 4 所示。“缓冲区溢出位置”一栏表示了程序 Crash 点发生的位置，都在 *memcpy*、*strncpy* 等内存操作函数中。

表 4 IO2BO 漏洞实例测试结果

CVE 编号	程序	缓冲区溢出位置	IntExp 处理结果
2007-5601	RealPlayer	memcpy	✓
2011-0097	MS Excel	memcpy	✓
2011-3026	libpng	memcpy	×
2012-0711	db2dasrrm	strncpy	✓
2013-3906	MS Word	memcpy	✓

实验结果表明 IntExp 能够处理其中大部分的 IO2BO 漏洞，但不能对 CVE-2011-3026 漏洞产生有效的输出，原因是该漏洞本身是一个无法劫持控制流的拒绝服务漏洞，漏洞的成因是错误地将有符号数用于 *memcpy* 函数的第三个参数，而该参数被定义为无符号数，产生了符号溢出。但是该漏洞会导致大范围（最大 4 GB）的缓冲区溢出，对于此类漏洞，IntExp 无法通过控制循环次数来控制溢出覆盖范围，所以只能触发访问无效内存的错误，而不能产生用于执行任意代码的缓冲区溢出。并且理论上来说，其他方法也不能利用此类漏洞劫持控制流并执行任意代码。因此 IntExp 的处理结果是符合预期的。

6 结束语

本文针对 IO2BO 漏洞特点，提出了一种二进制程序整型溢出漏洞的自动验证方法：通过建立可疑污点集合缩小待分析污点范围，通过分析循环边界获得循环条件，通过符号执行收集循环次数约束，通过求解器产生新样本以形成传统缓冲区溢出。实现的 IntExp 系统通过功能测试和实例测试，能够较好地解决整型溢出漏洞的自动验证问题。（责编 程斌）

参考文献：

[1] 高川，严寒冰，贾子骁 . 基于特征的网络漏洞态势感知方法研究 [J]. 信息网络安全，2016（12）：28-33.

[2]CHRISTEY S, MARTIN R A. Vulnerability Type Distributions in CVE[EB/OL]. http://cve.mitre.org/docs/vunl-trends/vuln-trends.pdf, 2007-3-16/2017-4-5.

[3]WANG T L, WEI T, LIN Z Q, et al. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution[EB/OL]. https://www.researchgate.net/publication/221655385_IntScope_Automatically_Detecting_Integer_Overflow_Vulnerability_in_X86_Binary_Using_Symbolic_Execution, 2009-2-11/2017-4-5.

[4]MOLNAR D, LI X C, WAGNER D A. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs[C]// USENIX SSYM'09 Proceedings of the 18th Conference on USENIX Security Symposium, August 10-14, 2009, Montreal, Canada. Berkeley：USENIX，2009:67-82.

[5]CHEN P, HAN H, WANG Y, et al. IntFinder: Automatically Detecting Integer Bugs in x86 Binary Program[C]// Springer. ICICS'09 Proceedings of the 11th International Conference on Information and Communications Security, December 14-17, 2009, Beijing, China. Berlin, Heidelberg:Springer, 2009:336-345.

[6]STELIOS S D , ERIC L, NATHAN R, et al. Targeted Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement[J]. ACM Sigplan Notices, 2015, 50(4):473-486.

[7]JAMES N, DAWN S. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software[C]// Network and Distributed System Security Symposium, February 3-4, 2005, San Diego, California, USA. DBLP, 2005. Beijing:Chinese Journal of Engineering Mathematics , 2012 :720-724.

[8]GODEFROID P, LEVIN M Y, MOLNAR D A. Automated Whitebox Fuzz Testing[EB/OL]. https://www.researchgate.net/publication/221655409_Automated_Whitebox_Fuzz_Testing, 2017-4-5.

[9]BRUMLEY D, POOSANKAM P, SONG D, et al. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications[C]// IEEE. Security and Privacy, May 18-21, 2008, Oakland, California, USA. New York:IEEE, 2008:143-157.

[10]HEELAN S, KROENING D. Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities[EB/OL]. https://www.mendeley.com/research-papers/automatic-generation-control-flow-hijacking-exploits-software-vulnerabilities，2017-4-5.

[11]SANG K C, AVGERINOS T, REBERT A, et al. Unleashing Mayhem on Binary Code[C]// IEEE. Security and Privacy, May 20-23, 2012, San Francisco, California, USA. New York:IEEE, 2012:380-394.

[12]HUANG S K, HUANG M H, HUANG P Y, et al. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations[C]// IEEE. Sixth International Conference on Software Security and Reliability, June 20-22, 2012, Washington, D.C., USA. New York:IEEE, 2012:78-87.

[13]AHMAD D. The Rising Threat of Vulnerabilities Due to Integer Errors[C]// IEEE. Security and Privacy, May 11-14, 2003, Oakland, California, USA. New York:IEEE, 2003:77-82.

[14]LEE J, AVGERINOS T, BRUMLEY D. TIE: Principled Reverse Engineering of Types in Binary Programs[EB/OL]. https://www.researchgate.net/publication/221655364_TIE_Principled_Reverse_Engineering_of_Types_in_Binary_Programs, 2017-4-5.

[15]LIN Z, ZHANG X, XU D. Automatic Reverse Engineering of Data Structures from Binary Execution[EB/OL].https://www.researchgate.net/publication/221655362_Automatic_Reverse_Engineering_of_Data_Structures_from_Binary_Execution, 2017-4-5.

[16]DEBRARY S, MUTH R, WEIPPERT M. Alias Analysis of Executable Code[C]// ACM. POPL '98 Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, January 19 - 21, 1998, San Diego, California, USA. New York:ACM, 1998:12-24.

[17]ZHANG C, WANG T, WEI T, et al. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time[C]// Springer.European Symposium on Research in Computer Security, September 20-22, 2010, Athens, Greece. Berlin, Heidelberg:Springer, 2010:71-86.

[18]BRUMLEY D, JAGER I, AVGERINOS T, et al. BAP: A Binary Analysis Platform[C]// Springer.International Conference on Computer Aided Verification, July 14-20, 2011, Snowbird, Utah, USA. Berlin, Heidelberg:Springer, 2011:463-469.

[19]KAMINSKY D, FERGUSON J, LARSEN J, et al. Reverse Engineering Code with IDA Pro[J]. Journal of the Royal Society for the Promotion of Health, 2008, 123(4):210-215.

[20]SONG D, BRUMLEY D, YIN H, et al. BitBlaze: A New Approach to Computer Security via Binary Analysis[C]// Springer .Information Systems Security, December 16-20, 2008, Hyderabad, India. Berlin, Heidelberg:Springer, 2008:1-25.

[21]Pin - A Dynamic Binary Instrumentation Tool[EB/OL]. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool, 2015-1-30/2017-4-5.

[22] 陈颖聪，陈广清，陈智明，等 . 面向智能电网 SDN 的二进制代码分析漏洞扫描方法研究 [J]. 信息网络安全，2016（7）：35-39.