第45卷第7期 中国科学技术大学学报 Vol. 45, No. 7
2015年7月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Jul. 2015

文章编号:0253-2778(2015)07-0601-07

一种面向源代码的整数溢出缺陷静态检测方法

黄 晖,陆余良,刘林涛,赵 军

(解放军电子工程学院 安徽合肥 230037)

摘要:受限于不完备的函数调用图分析和路径可达性分析,当前静态整数溢出检测工具存在较为严重的误报情况.为解决这一问题,以源代码中外部输入可控的整数溢出缺陷的自动挖掘为目标,给出一种综合调用图分析、静态污点分析和静态符号执行的检测方法.提出一种域敏感的流敏感指针分析方法构建目标程序调用图的"高估计",应用静态污点-sink传播分析确定潜在的外部输入可控的整数溢出缺陷程序点,最后应用静态符号执行技术通过判定缺陷约束的可满足性对误报情况进行约减.实验验证了方法在实际整数溢出缺陷检测和误报情况约减方面的应用有效性.

关键词:整数溢出;域敏感流敏感指针分析;污点分析;静态符号执行

中图法分类号: TP311 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2015.07.006

引用格式: HUANG Hui, LU Yuliang, LIU Lintao, et al. A source code oriented static detection method for integer overflow defects[J]. Journal of University of Science and Technology of China, 2015,45(7): 601-607.

黄晖,陆余良,刘林涛,等. 一种面向源代码的整数溢出缺陷静态检测方法[J]. 中国科学技术大学学报,2015,45(7): 601-607.

A source code oriented static detection method for integer overflow defects

HUANG Hui, LU Yuliang, LIU Lintao, ZHAO Jun

(*Electronic Engineering Institution of PLA, AnHui* 230037, *China*)

Abstract: Limited by incomplete call graph analysis and path feasibility analysis, current static integer overflow defect detection methods generally return results with high false positives. To reduce this inefficiency, aiming at automatic exploration of the external input triggering integer overflow defects, a new source code oriented detection method was proposed combining call graph analysis, static taint analysis and static symbolic execution, in which a field-sensitive and flow-sensitive pointer analysis method was proposed for constructing an over-approximation of the target program's real call graph, with a static taint-sink propagation analysis carried out for calculating the potential external input reachable integer overflow defects, on which flow-sensitive static symbolic execution is conducted to reduce the false positives introduced by the detection system through justifying the satisfiability of the corresponding defect constraint. Experiments prove the effectiveness of the methodin real-world integer overflow defect detection and false alarm reduction.

Key words: integer overflow; field-sensitive flow-sensitive pointer analysis; taint analysis; static symbolic execution

收稿日期:2015-03-29;**修回日期**:2015-04-15

作者简介:黄晖,男,1987年生,博士生. 研究方向:信息安全、程序分析. E-mail:hhui_123@163.com

通讯作者:陆余良,博士/教授. E-mail:luyuliang@ah165.net

0 引言

整数溢出是指整数算术运算结果，由于超出目标机器上整数操作数的表示范围而无法得到正确表示的一种计算偏差情况[1]. 整数溢出并不会直接破坏系统安全属性，但是由整数溢出引发的其他安全漏洞（诸如缓冲区溢出）则会给系统带来严重威胁[2]. 当前基于动态分析的整数溢出检测的相关研究包括[3-5]，这些方法可以检测出程序运行过程中存在的整数溢出情况，但是采用的插桩技术可能降低目标系统的运行性能；同时，对于实际程序中存在的良构的整数溢出，方法可能会产生误报情况.

整数溢出缺陷的静态检测已有一些研究[6-8]，然而这些方法或受限于不完备的调用图分析导致丢失某些程序路径的分析，或由于未充分考虑目标程序点的路径可达性约束条件导致将实际不可能发生的情况判为缺陷状态，分析结果中误报情况较为严重.

类同于文献[9]，本文将整数溢出问题建模为一种特殊的 taint-sink 问题，关注外部输入可控、能够影响系统关键程序位置的整数溢出缺陷的自动发现过程，给出了一种面向源代码的整数溢出缺陷静态检测方法. 基于统一中间表示进行完备的静态调用图分析；然后进行污点正向、sink 信息逆向的双向污点传播分析，确定出能够受到外部输入影响的整数溢出缺陷程序点集合；最后运用静态符号执行技术对分析误报情况进行约减. 实验验证了该方法在实际整数溢出缺陷检测过程中的有效性，在分析误报情况约减方面具较好的应用效果.

1 整数溢出检测系统框架

基于 LLVM 编译系统[10]，本文的整数溢出检测系统如图 1 所示. 对于包含多个代码模块的 C、C++源程序，调用 LLVM 编译过程将各个模块单独编译成对应的字节码形式的中间表示. 基于这些模块形式的中间表示，静态调用图分析对程序中的函数指针进行分析，构建出目标程序的调用图；基于得到的全系统控制流图，进行静态污点、sink 传播分析，随后检测出受外部输入控制的、能够影响关键程序位置的潜在的整数溢出缺陷程序点；静态符号执行对这些程序点进行流敏感的路径可达性分析，通过剔除缺陷路径约束不可满足的程序点，实现系统误报情况的约减.

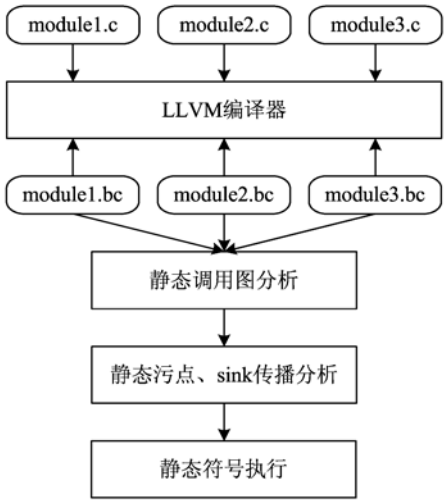

图 1 整数溢出检测系统框架

Fig. 1 Framework for integer overflow detection system

2 静态调用图分析

目标程序完整的控制流图，包括各函数的过程内控制流图、程序的函数调用图两部分信息. 前者可通过 LLVM 框架获取，但对于后者，由于缺乏程序中函数指针的使用分析，单纯运用 LLVM 框架内置的程序调用关系无法解析间接函数调用的情况，由此可能导致丢失对目标程序某些路径的安全分析.

基于上述问题，本文给出一种域敏感的流敏感可能别名分析算法[11]，考察目标程序中所有函数指针定值的可能指向函数集合，构建完整的过程调用图，为后端全程序污点-sink 传播分析的开展提供必要的控制流结构信息.

2.1 函数指针计算上下文

为构建函数指针计算上下文，首先为程序中任意函数指针类型定值 Def_i 建立对应的抽象机器位置 $\mathrm{LOCS}(\mathrm{Def}_i) \in absLocations$.

以 C 语言为例，函数指针定值在程序中一般表现为两种形式：单一定值实体和作为结构体定值中的某一成员域. 对于前者形式的定值 def_i，LOCS 维护唯一的映射关系指向其对应的抽象位置；后者在 LLVM 代码中对应于 GetElementPtr Ptr $\mathrm{field}_1 \cdots \mathrm{field}_n$ 形式的 LLVM 指令，一般情况下需要对基址指针操作数 Ptr 进行进一步的别名分析. 该过程可能导致：①分析引擎更复杂，影响分析的收敛时间；②无法计算出目标程序所有可能的函数指向情况，导致分析计算是不完备的.

针对上述问题，本文采用一种结构类型统一抽象的分析方法.以 $struct_i$ 结构中的函数指针成员域 $FuncPtr_j$ 为例，目标程序中任意对应于该域访问的程序定值对应的抽象机器位置，被统一映射到＜struct$i$，$FuncPtr_j$＞形式的“结构属性”$SA_{i,j}$ 对应的抽象机器位置中.

定义值集分析的抽象上下文 FPLocCtx，完成抽象机器位置到对应函数指针值集的映射关系如下：

$$\begin{aligned}FPLocCtx=\{&<Loc_i,Val_i>|\\&Loc_i\in absLocations\\&Val_i\in 2^{\wedge}FuncSet\\&\}\end{aligned}$$

给定 LOCS、FPLocCtx，函数指针计算 FPCtx 上下文定义如下：

$$\begin{aligned}FPCtx=\{&<Def_i,Val_i>|\\&Def_i\in Defs,\\&type(Def_i)\ is\ FuncCPTR,\\&Val_i=FPLocCtx(LOCS(Def_i))\\&\}\end{aligned}$$

2.2　流敏感的指针别名分析

基于 2.1 节的指针计算上下文，分析引擎首先考察所有全局函数指针变量的初始化状况.对于有初始化过程的全局变量 $dv_{i,j}$，将对应的初始化定值添加到对应定值集合 $FPCtx(dv_{i,j})$ 中；然后为每种类型的 LLVM 指令定义计算操作语义，依托迭代算法框架求解目标程序中每个定值对应的函数指针集合.

负责完成对结构数据中特定成员域的访问的 GetElementPtr 指令的操作语义定义如下：

insn：GetElementPtr Ptr $field_1\cdots field_n$

SA(insn)＝
SACalculate(type(Ptr)，$field_1\cdots field_n$)
FPCtx(SA(insn))＝{ }

该型计算依据 GetElementPtr 指令操作数的类型信息，构造该指令定值的结构域属性，同时在全局上下文中建立相应的映射关系.

表征取值聚合的 PHI 指令的操作语义定义如下：

insn：PHI $Val_1\cdots Val_n$

```
forEach Val_i in PHI.Vals
  if (SA(Val_i)! =NULL) then
    vset=FPCtx(SA(Val_i))
  else
    vset=FPCtx(Val_i)
  FPCtx(insn)=FPCtx(insn)∪vset
 end
end
```

该型计算求解出 PHI 各操作数对应的指针值集，最后统一汇总到 PHI 指令定值对应的指针值集.Select、Cast 等指令的操作语义以类似的方式实现.

表征函数返回的 Re t 型指令操作语义定义如下：

insn：Re t op_1 in $Func_i$

```
if type(op1) is FuncPtr then
  FPCtx(Loc(Re t(Func_i)))=FPCtx(SA(op1))
```

通过该型计算，Re t 指令操作数的函数指针定值集合被传播到函数返回值对应的定值集合中.

LoadInst 型指令的操作语义定义如下：

insn：load Ptr_i

```
if type(Ptr1) is FuncPtr then
  if SA(Ptr1)! =NULL) then
    FPCtx(insn)=FPCtx(SAt(Ptr1))
  else
    FPCtx(insn)=FPCtx(Ptr1)
```

该型计算将 LoadInst 指令的指针操作数的指针值集传播到对应指令的定值状态.

StoreInst 型指令的操作语义定义如下：

insn：Store Ptr_1 op_1

```
if type(Ptr1) is FuncPtr then
  if SA(Ptr1)! =NULL) then
    FPCtx(SAt(Ptr1))FPCtx(op1)
  else
    FPCtx(Ptr1)=FPCtx(op1)
```

在该型指令的抽象分析中，考察 StoreInst 指令的指针操作数类型.依据其是否存在对应结构域属性确定指令源操作数的指针值集的传播方向.

对于过程调用的 CallInst 型指令，其计算操作语义定义如下：

insn：Call Callee-Val $arg_1\cdots arg_n$

```
forEach Func_i in FPCtx(Callee-Val) do
  forEach Call_arg_j in insn.call_arg s do
    if type(Call_arg_j) is FuncPtr then
      FPCtx(args(Func_i)_j))=FPCtx(Call_arg_j)
  end
 end
end
```

在该型指令的抽象分析中，首先由 Call 指令的

调用函数操作数的计算上下文确定该 Call 指令的所有调用函数. 随之对这些潜在的被调用函数,将 Call 指令对应的实际参数的对应函数指针值集状态传播到目标函数中对应的形式参数函数抽象状态.

由此,本文的静态调用图分析方法求解出的函数调用图,本质上是对目标程序正确调用图的一种"高估计"(over-approximation). 虽然计算结果可能包含某些目标程序运行时不可达的调用关系,但由于包含了所有可能存在的调用关系,使得后端的静态污点分析可以在控制流关系完备地计算上下文内对目标程序展开分析.

3 静态污点-sink 传播分析

整数溢出广泛存在于当前的应用程序中,但只有当溢出结果影响了程序的控制流、内存分配、内存访问等操作时,才会进一步引起其他安全问题.

关注于这样一种缺陷场景:将受外部输入影响的某些程序定值的计算结果用于内存分配函数的参数,导致访问分配小于预期配额的内存时,触发缓冲区溢出. 本文运用静态污点分析方法,为程序的外部输入标记污点属性,为可能受影响的内存分配函数(诸如 malloc,realloc 等)的关键参数标记 sink 属性. 通过进行污点属性正向、sink 属性逆向的双向污点分析,确定出能够影响目标程序内存安全状态的外部输入可控的缺陷程序点集合.

3.1 静态污点正向传播分析

污点正向传播分析由两部分构成:污点状态引入和污点传播分析. 对于前者,以 Linux 系统下应用程序文件输入污点引入为例,分析引擎将所有 read 函数调用点的实际参数 $\mathrm{arg}_{\mathrm{read},i}$ 对应的污点机器状态标记为 TAINT,由此在目标程序的全局上下文中引入外部输入相关的污点信息.

污点传播分析通过为每种类型的 LLVM 指令定义如下形式的计算操作语义,在第 2 节计算的完备程序控制流图上运用迭代收敛算法实现.

对于一般的三地址形式 LLVM 指令,其污点传播计算操作语义定义如下:

$$\frac{\mathrm{insn}:\ \mathrm{op}_3=\mathrm{op}_1\ \mathrm{BINOP}\ \mathrm{op}_2}{\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_3))=\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_3))\wedge_L(\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_3))\cup\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_2))}$$

经过该型计算,完成指令源操作数的污点状态向目标操作数的传播.

函数返回的 Re t 型指令的操作语义定义如下:

$$\frac{\mathrm{insn}:\ \mathrm{Re}\ t\ \mathrm{op}_1\ \mathrm{in}\ \mathrm{Func}_j}{\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_3))=\mathrm{TMap}(\mathrm{Loc}(\mathrm{Re}\ t(\mathrm{Func}_i)))=\mathrm{TMap}(\mathrm{Loc}(\mathrm{op}_1))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\mathrm{Re}\ t(\mathrm{Func}_i)))}$$

通过该型计算,Re t 指令操作数的污点状态被传播到对应函数的返回值中.

LoadInst 型指令的操作语义定义如下:

$$\frac{\mathrm{insn}:\ \mathrm{load}\ \mathrm{Ptr}_1}{\mathrm{TMap}(\mathrm{Loc}(\mathrm{insn}))=\mathrm{TMap}(\mathrm{Loc}(\mathrm{SA}(\mathrm{Ptr}_1)))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\mathrm{SA}(\mathrm{Ptr}_1)))}$$

StoreInst 型指令的操作语义定义如下:

$$\frac{\mathrm{insn}:\ \mathrm{store}\ \mathrm{Ptr}_1}{\mathrm{TMap}(\mathrm{Loc}(\mathrm{Ptr}_1))=\mathrm{TMap}(\mathrm{Loc}(\mathrm{SA}(\mathrm{Ptr}_1)))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\mathrm{SA}(\mathrm{Ptr}_1)))}$$

过程调用的 CallInst 型指令,其污点传播计算操作语义定义如下:

$$\frac{\mathrm{insn}:\ \mathrm{Call}\ \mathrm{Func}_i\ \mathrm{arg}_1\cdots\ \mathrm{arg}_n}{\begin{array}{l}\mathrm{TMap}(\mathrm{Loc}(\mathrm{arg}_1))=\\ \mathrm{TMap}(\mathrm{Loc}(\ \mathrm{arg}_1))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\ \mathrm{args}_{\mathrm{Func}_i,1}))\\ \cdots\\ \mathrm{TMap}(\mathrm{Loc}(\mathrm{arg}_n))=\\ \mathrm{TMap}(\mathrm{Loc}(\ \mathrm{arg}_n))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\ \mathrm{args}_{\mathrm{Func}_i,n}))\\ \mathrm{TMap}(\mathrm{Loc}(\mathrm{args}_{\mathrm{Func}_i,1}))=\\ \mathrm{TMap}(\mathrm{Loc}(\ \mathrm{args}_{\mathrm{Func}_i,1}))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\ \mathrm{arg}_1))\\ \cdots\\ \mathrm{TMap}(\mathrm{Loc}(\mathrm{args}_{\mathrm{Func}_i,n}))=\\ \mathrm{TMap}(\mathrm{Loc}(\ \mathrm{args}_{\mathrm{Func}_i,n}))\wedge_L\mathrm{TMap}(\mathrm{Loc}(\ \mathrm{insn}))\\ \mathrm{TMap}(\mathrm{Loc}(\ \mathrm{insn}))\wedge_L\mathrm{TMap}(\mathrm{Re}\ t_{\mathrm{Func}_i})\end{array}}$$

在该型指令的污点传播分析中,将完成如下工作:①被调用函数的形式参数污点状态向调用点处对应实际参数的传播;②调用点处实际参数污点状态向被调用函数对应形式参数的传播;③被调用函数的函数返回指令定值污点状态向调用点处 call 指令对应定值的传播. 其中,①和②两部分依赖迭代分析框架,构建形式参数和实际参数污点状态的相互反馈机制,最终达到收敛状态.

3.2 sink 信息逆向传播分析

本文定义 sink 信息为诸如内存分配函数 malloc 的 size 参数形式的"敏感函数的敏感操作数". 进行 sink 信息的反向传播,旨在确定出能够影响到这些关键程序位置的整数计算定值集合,进而明确潜在的整数溢出缺陷位置.

定义 sink 分析的值集为 SSet＝{⊥，SINK，NOTSINK，T}，程序任意定值 Loc_i 对应的 sink 机器状态 SMap(Loc_i)∈SSet. sink 信息传播框架如下.

算法 3.1　sinkModuleAnalysis 算法

```
1 算法 sinkModuleAnalysis(SMap,Module)
2   do
3     changed=false;
4     forEach Func in Module,functions do
5       if(Func is a SINKFuncion with SINKIndex i) then
6         changed|=sinkPropagate(SMap,arg(i),SINK);
7       end
8     end
9   while(changed);
10  return changed;
11 end
```

算法 3.2　sinkPropagate 算法

```
1 算法 sinkPropagate(SMap,DefV,sinkValue)
2     if SMap(DefV)=sinkValue then
3       return false;
4     end
5     SMap(DefV)=sinkValue;
6     changed=false;
7     if (DefV is a arg(Func,i) then
8       forEach CS in callSites(Func) do
9           changed | = sinkPropagate(SMap, CS, arg(i),
SINK);
10      end
11    else if((DefV is (insn: PHI(v1,…,vn)))|
12          (DefV is (insn: UOP(V)
13      ) then
14    forEach v in DefV.operands do
15          changed|=sinkPropagate(SMap,v,CS,arg(i));
16      end
17    end
18    return changed
19  end
```

算法 3.1 给出了静态 sink 信息逆向传播算法. 该算法对目标程序的所有 sink 函数，标记其敏感形式参数对应的 sink 机器位置为 SINK，然后迭代调用算法 3.2 进行 sink 信息的逆向传播(第 4～8 行). 传播过程具体如下.

对于任意定值 DefV，考察其定值构成形式. 若为 Func 函数的某个调用位置的实际参数，即将 Func 对应的形式参数 sink 状态进行标记；若 DefV 是由定值聚合型 PHI 指令表征或其定值来源于一元计算的计算操作数，即将该定值对应源操作数的 sink 状态进行标记(算法 3.2 第 11～17 行).

综合 3.1 节论述的污点正向传播分析和本节论述的 sink 信息逆向传播分析，对于任意程序点 insn，若其定值 Loc(insn)在污点机器状态 TMap 和 sink 机器状态 SMap 中都有对应的映射关系，即判定该程序点为一个可能发生整数溢出的缺陷程序点.

4　流敏感的静态符号执行

第 3 节进行的外部输入可控的整数溢出程序点计算存在较为严重的缺陷误报可能性，一个重要原因在于其在计算过程中仅仅关注目标程序点的控制流图可达情况，并未充分考虑绑定在控制流上的程序数据流约束，缺乏对目标程序点路径可达谓词的相关分析.

为了对静态分析的误报情况进行约减，基于计算出的缺陷程序点及其缺陷操作数，本文构建反向的静态符号执行引擎，关注两类约束的收集计算：目标缺陷谓词约束和程序控制流图上蕴含的目标基本块控制流约束，通过判定缺陷程序点的目标函数内局部路径可达性，实现误报情况的部分缓解.

目标基本块控制流约束收集，旨在确定该基本块的控制流可达路径约束. 本文以一种流敏感的方法，从目标基本块反向求解其控制流约束.

对于程序中任意基本块 bb∈BBs(Func)，本文定义其控制流可达路径约束如下：

$$
\begin{aligned}
&\mathrm{CFGCons}(bb)=\\
&\{\vee \mathrm{EdgeCons}(bb_i,bb)\\
&\ |bb_i\in \mathrm{prev\_nodes}(bb)\\
&\ \text{and bb is not bb.parentFunc.ENTRY}\\
&\}\cup\\
&\{\mathrm{true}\,|\,bb \text{ is } bb.\mathrm{parentFunc.ENTRY}\}
\end{aligned}
$$

目标基本块控制流约束 CFGCons(bb)的收集，旨在完成收集从目标函数入口位置到目标基本块的可达路径约束. 特别地，定义任意函数 $Func_i$ 的入口基本块对应的控制流约束 CFGCons($Func_i$.ENTRY). 对于一般的基本块 bb，其控制流约束由其所有入口边中蕴含的边控制流约束 EdgeCons(bb_i,bb)的析取构成(bb_i 为的 bb 一个前驱节点).

边控制流约束 EdgeCons(src Bb,dst Bb)定义形式如下：

$$
\begin{aligned}
&\mathrm{EdgeCons}(\text{src Bb},\text{dst Bb})=\\
&\mathrm{CFGCons}(\text{src Bb})\wedge
\end{aligned}
$$

Edge Pr edicate(src Bb,dst Bb)

DefUseCons tra int(src Bb,dst Bb)

该型约束由 3 类约束的合取构成:源基本块的控制流约束、边本身蕴含的谓词约束和源基本块出口点的所有定义定值与目标基本块所有引用定值的定值合并约束.该约束定义如下:

$$\begin{aligned}&\text{DefUseCons tra int(src Bb,dst Bb)}=\\&\{\text{Def}(dv_i)==\text{Use}(dv_j)\\&\ |\text{Loc}(dv_i)==\text{Loc}(dv_j),\\&\quad dv_i\in\text{DV(src Bb)},\\&\quad dv_j\in\text{DV(dst Bb)},\\&\quad \text{Def}(dv_i)\in\text{Defs(src Bb)},\\&\quad \text{Use}(dv_j)\in\text{Uses(dst Bb)}\\&\}\end{aligned}$$

Defs(src Bb)和 Uses(dst Bb)集合的计算一般基于经典的到达定值算法实现.在 LLVM 的 SSA 形式的中间语言表示中,已经隐式地为程序所有定值建立了对应的 Def-Use、Use-Def 关系链表,实际分析中基于该 U-D 链表完成 DefUseConstraint(Defs(src Bb),Uses(dst Bb))约束的计算.

5 实验分析

为衡量方法的应用有效性,本文首先对 Linux 平台下 libpng-1.5.13、tiff-3.8.2、cups-1.3.0 等应用程序进行整数溢出缺陷的存在性检测实验.这些程序都包含有对应 CVE 编号的公开漏洞.实验结果如表 1 所示.由表 1 可见,对于上述应用程序,运用本文的检测算法,成功地发现了其中存在的已公开的程序漏洞;同时,对于某些目标程序,运用本文方法识别出部分无对应 CVE 编号的程序缺陷并报警.对于这些报警的程序缺陷,经手工分析其操作数约束关系,确认其中的某些报警确实为外部输入可控可溢出缺陷.

表 1 检测出的部分程序缺陷

Tab.1 Program defects detected by our method

目标程序	缺陷函数	缺陷代码位置	溢出类型	CVE 编号
libpng-1.5.13	png_set_unknown_chunks	Pngset.c:1036	UMUL	CVE-2013-7353
libpng-1.5.13	Main	wpng.c:707	UMUL	无
ibpng-1.5.13	png_text_compress	pngwutil.c:519	UMUL	无
libpng-1.5.13	png_set_filter	pngwrite.c:1117	SADD	无
libpng-1.5.13	png_set_sPLT	Pngset.c:999	UMUL	无
tiff-3.8.2	cvt_whole_image	tiff2rgba.c:338	UMUL	CVE-2009-2347
tiff-3.8.2	Tiffcvt	rgb2ygbcr.c:284	UMUL	CVE-2009-2347
tiff-3.8.2	gtStripSeparate	tif_getimage.c:874	SMUL	无
tiff-3.8.2	TIFFBuildOverviews	tif_overview.c:789	SMUL	无
cups-1.3.0	cupsImageReadPNG	image-png.c: 165,175,177	UMUL	CVE-2008-1722
cups-1.3.0	_cupsImageReadBMP	image-bmp.c:182	UMUL	无
cups-1.3.0	cups_raster_read	raster.c:763	UMUL	无
swftools-0.9.1	getPNG	png.c:503	UADD	CVE-2010-1516
swftools-0.9.1	jpeg_load	jpeg.c:356,369	UMUL	CVE-2010-1516
xine-lib-1.1.12	real_parse_mdpr	demux_real.c:287	UMUL	CVE-2008-5238

对于表 1 中给出的某些已公开的整数溢出漏洞,其补丁版本程序往往完成了对该漏洞的修复.对于表 1 中列出的已公开程序漏洞,在确定运用本文方法可以发现的基础上,对其补丁版本程序进行检测实验分析,结果如表 2 所示.

由表 2 可见,本文方法能够较为准确地识别出补丁程序中所作的整数溢出检测修正.这是由静态符号执行对目标程序点的控制流可达路径约束的计算所保证的.由于仅仅关注于目标程序点所在函数内部的局部约束,在目标程序对整数溢出的完备过滤判定发生在目标函数外部时,分析引擎依旧会产生误报情况(CVE-2010-1516 对 jpeg 代码的分析即对应于该情况).下一步工作中,对更完备的过程间符号执行计算进行研究是解决该问题的一个要点.

表 2　已公开整数溢出漏洞的补丁测试

Tab. 2　Experiments on patched version versus unpatched version for unveiled integer overflow vulnerabilities

CVE 编号	缺陷位置	缺陷程序版本	补丁程序版本	补丁是否报警
CVE-2013-7353	pngset. c:1036	libpng-1. 5. 13	Libpng-1. 6. 0	否
CVE-2009-2347	tiff2rgba. c:338	tiff-3. 8. 2	tiff-3. 9. 4	否
CVE-2009-2347	rgb2ygbcr. c:284	tiff-3. 8. 2	tiff-3. 9. 4	否
CVE-2008-1722	image-png. c:165	cups-1. 3. 0	cups-1. 3. 8	否
CVE-2008-1722	image-png. c:175	cups-1. 3. 0	cups-1. 3. 8	否
CVE-2008-5238	demux_real. c:287	xine-lib-1. 1. 12	xine-lib-1. 1. 19-2	否
CVE-2010-1516	png. c:503	swftools-0. 9. 1	swftools-0. 9. 2	否
CVE-2010-1516	jpeg. c:356,369	swftools-0. 9. 1	swftools-0. 9. 2	是

6　结论

本文给出了一种综合静态控制流分析、静态污点分析和静态符号执行等技术的面向源代码的整数溢出缺陷静态检测方法，即将源程序转换到统一的中间语言表现形式，运用静态控制流分析为程序中各函数调用位置确定对应的调用函数集合，构造程序调用图的“高估计”(over-approximation). 基于该调用图进行污点正向、sink 信息逆向的双向污点传播，定位出能够受到外部输入影响的缺陷程序点. 同时，为了约减误报情况，还运用静态符号执行计算各程序点的数据流可达情况. 实验结果表明，本文提出的方法在整数溢出缺陷发现过程是有效的.

基于当前已有实现，拟在以下方面进行拓展研究，包括：结合更加完备的指针别名算法，提升静态调用图分析的计算精度；扩展静态符号执行的分析范畴，加强误报情况的约减计算；将本文工作与动态程序分析技术相结合，形成完整的静态检测-动态测试用例生成工具链，自动化检测目标程序中存在的整数溢出缺陷并构造对应的验证输入实例(proof of concept)等.

参考文献(References)

[1]Dietz W, Li P, Regehr J, et al. Understanding integer overflow in C/C++[C]// Proceedings of the 34th International Conference on Software Engineering. Zurich, Switzerland: IEEE Press, 2012: 760-770.

[2]Zhang S R, Xu L, Xu B W. Methods of integer overflow detection to avoid buffer overflow[J]. Journal of Southeast University(English Edition), 2009, 25(2): 219-223.

[3]CUI Baojiang, LIANG Xiaobing, ZHAO Bing, et al. Detecting integer overflow vulnerabilities in binary executables based on target filtering and dynamic taint tracing[J]. Chinese Journal of Electronics, 2014, 23(2): 348-352.

[4]Long S B, F, Sidiroglou-Douskos S, Kim D, et al. Sound input filter generation for integer overflow errors sound input filter generation for integer overflow errors[J]. ACM SIGPLAN Notice, 2014, 49(1): 439-452.

[5]Pomonis M, Petsios T, Jee K, et al. IntFlow: Improving the accuracy of arithmetic error detection using information flow tracking[C]// Proceedings of the 30th Annual Computer Security Applications Conference. New Orleans USA: ACM Press, 2014: 416-425.

[6]Moy Y, Bjørner N, Sielaff D. Modular bug-finding for integer overflows in the large sound, efficient, bit-precise static analysis[R]. Technical Report, MSR-TR-2009-57, 2009.

[7]Brumley D, Chiueh T, Johnson R. RICH: Automatically protecting against integer-based vulnerabilities[EB/OL]. http://www. cs. berkeley. edu/~ dawnsong/papers/efficient _ detection _ integer-based_attacks. pdf.

[8]Ashcraft K, Engler D. Using programmer-written compiler extensions to catch security holes[C]// Proceedings of IEEE Symposium on Security and Privacy. San Jose, USA: ACM Press, 2002: 143-159.

[9]Wang T L, Wei T, Lin Z Q, et al. IntScope: Automatically Detecting integer overflow vulnerability in X86 binary using symbolic execution[C]// Proceedings of the 16th Network and Distributed System Security Symposium. San Diego, USA: IEEE Press, 2009:1-14.

[10]Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]// Proceedings of the International Symposium on Code Generation and Optimization. Palo Alto, USA: IEEE Press, 2004: 75-86.[11]Muchnick S S. 高级编译器设计与实现[M]. 赵克佳，沈志宇，译. 北京：机械工业出版社，2005.