# 基于数据流分析的UAF错误检查模型

## 摘要：

尽管有一些工具支持检测程序的“use-after-free”漏洞，但是由于它们的限制，它们的性能可能会降低。为了以相对较高的效率检测程序的UAF漏洞，我们提出了一种自动方法来检查应用程序中的UAF漏洞。首先，我们考虑并选择静态分析数据流的方法。然后，我们跟踪应用程序中变量的所有定义和用法。最后，介绍了等效变量和别名分析的技术。我们已在名为UAFChecker的工具中实施了本论文的方法，该工具可以自动检测使用后使用的漏洞，并通过一些实际案例研究进行了实验。

## 第一节 介绍

近年来，UAF漏洞的数量正在增加，导致了一系列严重危害，例如CVE-2010-0135，CVE-2010-0806和CVE-2010-1297。释放指向程序中内存的指针后，就会发生此漏洞的影响。由于未正确处理指针，因此如果再次引用释放的内存，可能会导致无法预料的情况。数据损坏是最简单的情况，而任何程序都可以运行则很严重[1]。

此漏洞是由编程语言的固有缺陷导致的[2]。例如，由于内存操作功能(例如C中的malloc)实际上是操作系统内存管理接口的包装，因此C编程语言未明确禁止类似UAF的情况。当它被释放时，该进程正在使用的内存页面通常不会被释放，并且标记为“ redistributed”。这种机制是UAF安全问题的原因。UAF在浏览器上更为常见。当某些对象以异步方式处理时，由于javascript的灵活性[3]，程序员无法考虑所有情况，并且不可避免地会出现一些问题。诸如此类的许多漏洞已在2010年发生。以IE浏览器上的CVE-2010-0249漏洞为例，攻击者制作了网页代码，并在CTreeNode发布之后，通过特定值修改了原始内存内容。然后结合堆喷射技术，攻击者填充了相应的内存区域，以使CElement :: Doc函数调用shellcode代码，并且攻击者可以获得浏览器进程的权限，可以远程执行任意代码[3]，[4]。

## 第二节 相关工作

发现和检测存储器操作的程序错误对于提高程序的鲁棒性和稳定性非常重要。目前广泛使用的内存检测方法一般可以分为两种。分别是是静态分析和动态分析方法[5]。通常基于静态分解过程提出静态分析方法，而动态分析方法是跟踪程序执行并搜索内存状态以进行动态程序分析。

研究人员基于这两种方法产生了一些工具。但是这些工具具有一定的适用性。Nicholas Nethercote通过使用valgrind [6]开发了一种名为memcheck的工具，该工具可以检查所有内存读取/写入操作以及malloc，new，free，delete等调用的拦截。它对于检测内存管理问题很有用，但不能处理别名问题。另外，该工具仅支持Linux平台。同样，Derek Bruening制作的基于DynamoRIO的Memory博士也存在相同的问题[7]，并且缺少对C ++语言的支持。可以说开源工具cppcheck [8]，它可以检测可能的内存错误，并提供简单的分析，例如分配和释放不匹配。但是cppcheck是用于源级别的静态分析方法。没有检测到二进制程序。因此，利用范围是有限的。至于funsniff，它是Immunity Debugger的一个插件，它不受源代码的限制。但是它依赖于调试器，受调试器的限制，并且只能以低效的方式自动检测动态实现中的UAF缺陷。另外，它也不是函数别名分析[9]。

针对上述情况，本文提出了基于使用-定义和定义-使用链的UAF漏洞检测方法，其主要思想是：使用静态单分配形式对数据流进行定义和进一步使用。然后找到所有指向相同内存地址的等效指针变量，并描述UAF漏洞模型。最后，对过程间和过程间的指令进行静态分析，以发现此类漏洞的存在。

## 第三节 定义-使用和使用-定义链模型

使用-定义链(UD链)是一种数据结构，由一个变量的使用U和该变量的所有定义D组成，D可以在没有任何其他中间定义的情况下达到该使用U。定义可以有多种形式，但通常是指将某些值分配给变量。

UD链的对应项是“定义－使用”链(DU链)，该链由变量的定义D和所有使用U组成，该定义D无需任何其他中间定义即可达到。

### A.模型的相关定义

为了描述变量，必须在程序语句中首先声明一些协议和集合。

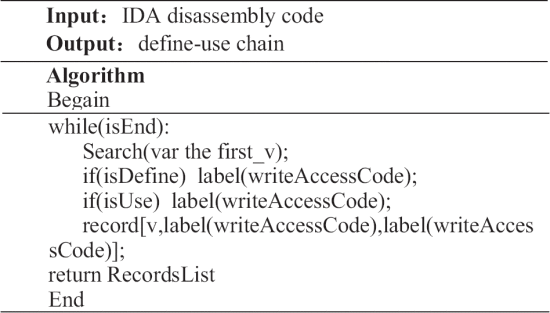
* 1. 语句的集合可以使用 s (i)表示。在式子中，变量i在区间内被限制为整数 [ 1 ，n ]。 n 用于计算基本块中的语句数。
  2. 变量以斜体标记。例如变量的声明*u / v*可以通过使用他们的大写字母来区分*U/ V*或使用 *s (0)*。
  3. 假设程序中的每个变量都有定义的上下文。

通过上述约束，可以定义位于赋值左侧的变量，例如 *v*。如果语句记作*s (j)*， *s (j )* 可以是 *v*的定义，每个变量至少一个这样的定义。如果一个变量*v*被使用，例如*s (j )*，则变量 *v*位于 *s (j )* 的右边。还有一个语句*s (j )*，并且 *i< j*，那么必须有一个变量 *v*并在 [*i，j*]中使用，如 *s (j )*。

### B.定义使用链和使用定义链模型

由于执行了程序语句的顺序，假设语句*s(j)*是定义语句。当*i <j*和*k≥  j*时，如果定义在某处如*s(k)* 有用，则对*j*有效。在语句*i*中，*i*的有效定义集合定义为*A(i)*，可以将有效定义编号为 | *A(i )*|。对于*k <i*，语句*s (i )*中的定义也已在语句*s (k )*中定义 ，则先前的定义将同样适用*i*的定义。

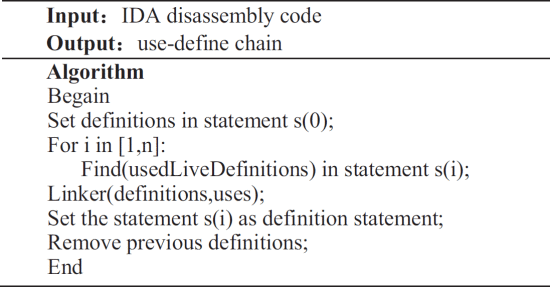
可以通过某些特定步骤[10]获得实施过程中的变量，假设要处理的变量是*v*，寻找其DU链的算法如图1所示：



**图1** 定义使用链算法

通过以上算法，结合读访问和写访问，可以完成一个变量。

UD链的构建方法如图2所示：



**图2**使用定义链算法

使用此算法，可以完成两件事：

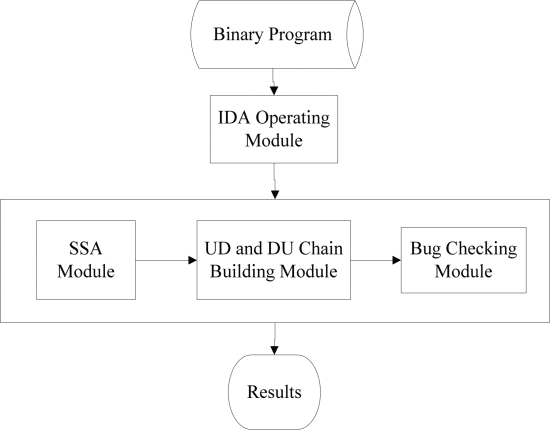
1. 在变量使用和定义上创建有向无环图(DAG)。DAG指定赋值语句之间的数据依赖性以及部分顺序(因此语句之间具有并行性)
2. 当声明 *s (i)*到达时，存在活动变量赋值列表。例如，如果只有一个赋值是活跃的，则可以使用恒定传播。

## 第四节 UAF漏洞检测模型

在本节中，我们将继续讨论有关处理UD和DU链的问题，这些链指向等待释放的内存变量指针。然后找到所有此类等效变量。另外，在每个路径中，它分别记录彼此等效的变量。之后，它将分析程序指令并为UAF漏洞建模。

### A.模型框架

本文基于对UD和DU链的研究，设计并实现了一个测试框架，该框架通过不正确的指针主模块框架来支持可执行二进制指针漏洞，如图3所示：



**图3**模型的框架

在上面的系统中，首先测试将二进制数据作为输入并由IDA分析。然后，它使用SSA中间表示来处理IDA的结果。下一步是在UD和DU链建立模块中为已处理代码中的变量设置UD链和DU链。然后将结果放入漏洞检测模块，最后将输出保存为指令形状和问题的地址。

### B.搜索等效变量

UAF漏洞是由释放内存中的不正确操作引起的。因此，首先要处理的是指向内存的指针。检查所有指向内存某处的变量，以查看它们是否正确释放。如果没有，您可以相信使用这些变量将产生错误。为了获得这一排等效变量，本文通过逆向参照系点的执行路径确定变量的使用和计算值。记录所有等效变量，并用等效变量递归处理其中一个变量。具体算法如图4所示

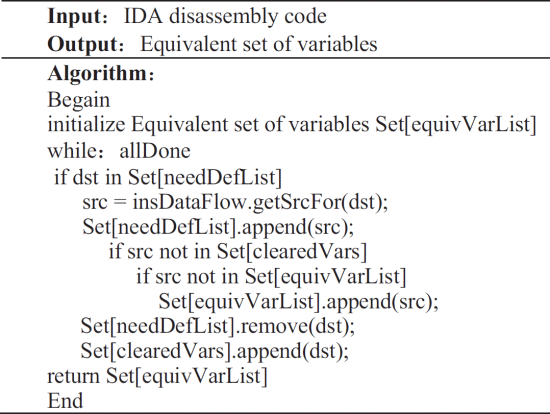


图4  搜索等效变量的算法

### C.程序内分析

对于一个函数，因为它使用一些在其自己函数中定义的变量，和某些在该函数中未定义的变量，例如全局变量。在本文中，检测中包括free函数在内的存在此类问题；因此，对过程内分析至关重要。

在此算法中，首先应开始使用API​​调用来查找函数或函数别名。要查找UAF漏洞，需要找到free函数调用或其别名函数调用。如果函数记录为二元数组，则可以触发此类漏洞(f，n )，这里 f是有问题的函数， n将要释放或关闭。n也可以是其他这类操作的指针变量。此外，其别名可以记录为(g，m )，其中g是调用函数，如果 x充当自变量m。所以在(f，n )，f 是调用函数，如果 x 作为自变量 n。通过别名分析，结合以前的方法，可以找到错误的过程。

首先，在函数f内使用先前介绍的等效变量算法来建立关于变量n的等效变量集，可以假定为E。考虑g，如果上面的段落中已描述了集合内部建立的自变量，则应在以后进行分析，则应对此进行分析，然后可以定义为(g，m ) 是(f，n )的别名。m是变量的序列号，过程之间的错误可以由二元组 (g，m )找到。在此过程中，主要困难是变量的跟踪函数。本文讨论使用IDA的处理机制的方法，例如形式标识为arg\_X，自动重命名内存地址等。

### D. UAF漏洞建模

UAF错误的原因主要是由程序引起的，它使用已释放的内存指针来计算新的内存地址，从而产生不可预测的结果。因此，为了检测这种错误，主要的危险是检查上述操作。

在实践中进行测试时，系统会向其提供指令inst，内存地址或寄存器x以及已释放指针S的集合。如果要将x加入到S集合，则仅在满足下面条件的情况下：

x∈inst\_dsts∧(y∈inst\_srcs[x]∣y∈S)≠∅

其中，dsts是一个指令的目的操作数的集合。srcs[x]是影响目的操作数的源操作数的集合。

如果要从集合S中删除x，则仅当且仅当：

x∈inst\_dsts∧(y∈inst\_srcs[x]∣y∈S)=∅

通过上面的描述，本文选择的方法是：在每条指令中，我们遍历用于计算指针的所有基址寄存器，以确定它们是否已在释放的指针的集合中。具体算法如图5所示：

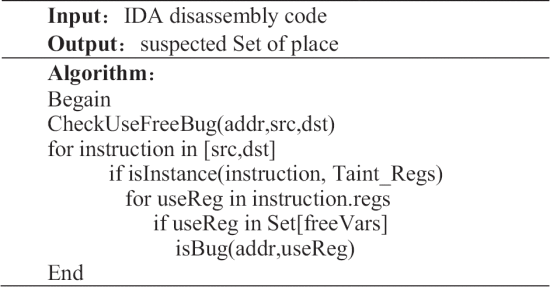


图5 错误检查算法

在查找UAF错误的算法中，Taint\_Regs表示至少一个用于寄存器的操作以建立内存地址结构。Src和dst对象表示程序指令中的源操作数和目标操作数。FreeVars是一组等待释放的等效变量。useReg是正在使用的寄存器。addr是程序执行此类危险操作的地址。

## 第五节 实验

本文实现了一个简单的IDA的Python语言验证工具UAFChecker。该工具在Windows系统上运行。测试硬件环境：Intel Pentium(R)4处理器3.00GHz，内存2G。这些测试主要针对开源软件或开发包，这些软件或开发包具有use-after-free的漏洞历史记录，例如Webkit，Clam Antivirus和libxml2。然后测试软件，看看是否可以在现有程序中发现漏洞。结果证明了该算法的正确性。表1是具体的实验数据。

表I.测试案例的数据

| Application | Comparative Aspects | | | |
| --- | --- | --- | --- | --- |
| free calls | alias | bugs | false positive |
| Webkit | 9035 | 0 | 5 | 1 |
| libxml2 | 1621 | 8 | 4 | 1 |
| Clam Antivirus | 1412 | 9 | 3 | 0 |

使用此工具测试Webkit后，它总共报告了五个错误。然后，我们手动分析这些错误，发现了四个UAF缺陷。图6显示有缺陷的部分。它是webkit核心功能的一部分。Webkit中的此功能可以用作内存管理系统的一部分。释放渲染目标时，它将调用此函数。从上面的代码中我们可以看到，ptr +0位置将被刚刚释放的地址填充。当您调用已释放的对象函数时，该函数地址实际上指向先前释放的属性对象的位置。如果可以控制此属性的值，则eip是可控的，从而达到执行任意代码的目的。实际上，在大多数情况下，可以通过自定义属性来控制此属性，而EIP是可以控制的[11]。

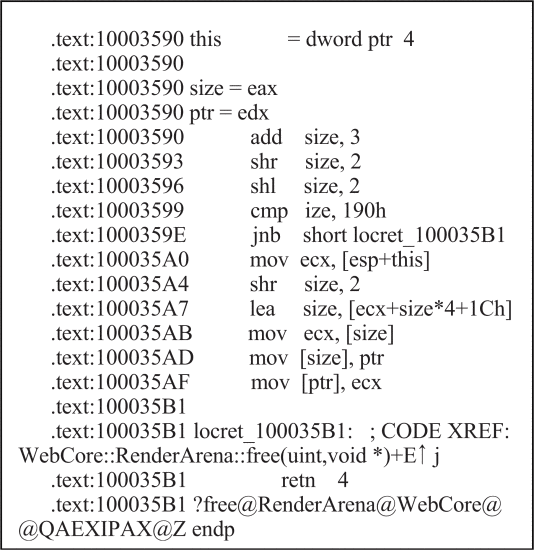


图6 use-after-free漏洞示例

在Clam Antivirus和libxml2的实验中，发现通过free的别名函数调用会发生一些错误。例如cl\_engine\_free()。使用函数别名算法，可以正确找到UAF错误。在Clam Antivirus中，通过别名分析发现了三个此类错误。这些位置是：contrib / clamdtop / clamdtop.c中的cli\_ac\_free()；在libclamav / mpool.c的cli\_mp\_hex2ui()中的free；在libclamav / readdb.c中的phishing\_ donee()。由于直接调用free而导致两个UAF libxml2错误。另外两个是由free别名引起的。这些错误的具体原因与CVE-2009-2414，CVE-2009-2416有关。比较表1中的结果。本文中描述的方法证明了可能的UAF缺陷。也有少量误报结果。

此外，我们研究了类似的检测工具cppcheck，Immunity调试器中的插件funsniff和Memor博士。表2是特定情况下的比较。

表II与其他工具的比较

| Application | Comparative Aspects | | | |
| --- | --- | --- | --- | --- |
| Alias Support | False Positive and Negative | Checking Type | Platform |
| UAFChecker | yes | 12% | static | windows, linux |
| cppcheck | no | 13% | static | windows, linux |
| funsniff | no | 28% | dynamic | windows |
| memcheck | no | – | dynamic | linux |
| Dr. Memory | no | 15% | dynamic | windows, linux |

从上表的数据可以看出，UAFChecker的误报率约为12％，而静态分析工具cppcheck的性能更接近，比基于动态仪器的Dr. Memory更好，比funsniff检测功能要好得多。从表2中可以看出，基于IDA的UAFChecker具有良好的跨平台。总体而言，所提出的方法和模型具有良好的实用性。

但是由于该算法没有路径敏感方法，并且可能存在一些意外情况，因此无法达到某些路径。因此它将在某个时候产生误报测试[12]，[13]。

## 第六节 结论

本文实现了一种“use-after-free”漏洞自动检测工具，可以发现UAF漏洞并自动记录错误的地址和处理环境。现有的一些漏洞已得到验证，并取得了良好的效果。

但是，由于在以下领域中提到的方法局限性，未来需要进一步改进：首先，由于约束求解[14]，[15]和其他特殊操作尚未用于过程路径执行，因此漏报和误报是由于路径未到达引起的。因此，为了解决该问题，需要引入进一步的方法来引入约束求解器以减少未到达的路径。第二个是，由于提出了改进标准的UD和DU算法的建议，因此没有对具有复杂路径的数据流进行深入分析[16]。漏报很难确定。最后，二进制文件中的别名跟踪和分析比源代码要困难得多。本文简要运用IDA分析处理结果，只能处理相对简单的目标。需要更深入的别名分析。

## 致谢

这项工作得到了中国国家高技术研究发展计划(“ 863”计划)的部分支持(授权号2008aa01z420)。

## 引用

[1] G. Balakrishnan and T. Reps, “Analyzing memory accesses in x86 executables,” Lecture Notes in Computer Science. Compiler Construction. Heidelberg: Springer Berlin, 2004, pp. 5–23.

[2] M. Costa, M. Castro, L Zhou, L Zhang, and M. Peinado, “Bouncer:securing software by blocking bad input,” 21st ACM SIGOPS symposium on Operating systems principles, Stevenson, Washington,USA, 2007. New York, NY, USA: ACM Press, 2007, pp. 117-130.

[3] J. McDonald, C. Valasek. “Practical Windows XPSP3/2003 Heap Exploitation,” Black Hat USA. Jul, 2009.

[4] HD Moore. “Metasploit3: Exploit Intelligence and Automation,” Microsoft Blue Hat 3, 2006.

[5] Dawson Engler and Madanlal Musuvathi, “Static Analysis versus Software Model Checking for Bug Finding,” VMCAI 2004,LNCS 2937,2004, pp. 191-210.

[6] Nicholas Nethercote. “Dynamic Binary Analysis and Instrumentation.PhD Dissertation,” University of Cambridge, November 2004.

[7] derek bruening. “Memory Debugger for Windows and Linux.” http://dynamorio.org/drmemory.html, 2010.

[8] http://sourceforge.net/apps/mediawiki/cppcheck/index.php?title=Main\_Page.

[9] J. Newsome and D. Song, “Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,” Proceedings of the 12th Annual Network and Distributed System Security Symposium, San Diego, California, 2005. The Internet Society, 2005.

[10] Kian Salem,Übung zu Software Engineering. <http://www.wi.unimuenster.de/pi/lehre/ws0910/se/uebungen/Uebung> 6-Vorstellung.pdf, 2010.

[11] Alexander Sotirov. “Heap Feng Shui in JavaScript,” Black Hat Europe. 2007.

[12] P. Boonstoppel, C. Cadar, and D. Engler, “RWset: attacking path explosion in constraint-based test generation,” 14th International Conference, TACAS, Budapest, Hungary, 2008. New York: SpringerVerlag, 2008, pp. 351-366.

[13] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, “Fitness-guided path exploration in dynamic symbolic execution,” Microsoft: MSRTR-2008-123, 2008.

[14] L. de Moura and N. Bjørner. “Z3: An efficient SMT solver,” In Intl.Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337340.

[15] Robert Brummayer and Armin Biere. “Boolector:An effcient SMT Solver for Bit-vectors and Arrays” 2009, pp. 1174–177

[16] R. Chugh, J. W. Voung, R. Jhala, and S. Lerner. “Dataflow analysis for concurrent programs using datarace detection,” In Conf. on Programming Language Design and Implementation(PLDI), 2008, pp. 316–326.