# 静态检测二进制代码上use-after-free漏洞

Josselin Feist · Laurent Mounier · Marie-Laure Potet

**摘要** 我们提出GUEB静态工具在反汇编代码上检测use-after-free漏洞。并已针对ProFTPD应用程序中的实际漏洞对该工具进行了评估。

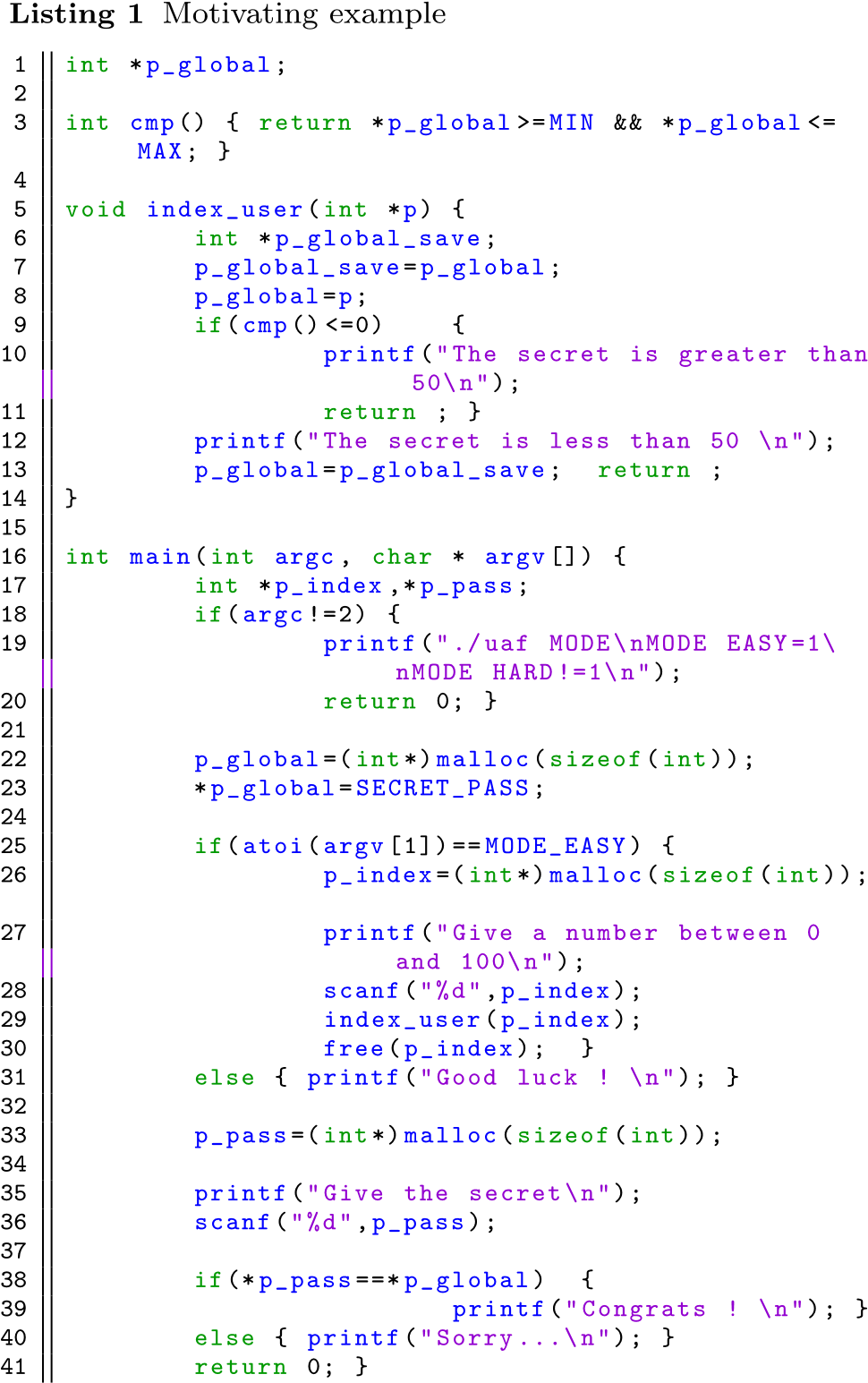
**1引言**

漏洞检测通常是从fuzzing测试的第一步开始，使用可能具有危险输入的应用程序进行测试。由fuzzing产生的跟踪通常使用一些启发式方法进行分类（例如，导致内存崩溃的跟踪）。然后，手动研究这些跟踪或类似跟踪的可利用性。为了获得我们感兴趣的跟踪路径，模糊器需要确定必须强调应用程序的哪些部分以及如何强调（即SQL注入，缓冲区溢出等）。为此，一种方法包括静态地识别易受攻击的模式（从语法模式，例如strcpy调用，到更复杂的模式，例如[13]）。

我们在这里提出一种静态方法，专门用于检测二进制代码中的UaF模式，自2008年以来，这种漏洞每年都在增加一倍[7]。UaF的特征是发生两个不同的事件：创建一个悬空指针，然后再访问该指针指向的内存[1]。因此，检测UaF需要分析较长的执行序列，这在处理大型应用程序时是一项艰巨的任务。因此，我们认为从可伸缩性的角度来看，静态分析可以提供良好的结果，找到使用纯动态方法很难检测到UaF模式。更准确地说，我们的目标是识别UaF中涉及的程序位置集，提供第一级漏洞检测步骤，并适用于大型（二进制）代码。 此外，在此步骤中收集的信息对于后续的可利用性分析（即，告诉以后是否可以以及如何重新分配或覆盖某些已释放的内存块）也可能有用。

**1.1一个激励人心的例子**

我们通过清单1给出的激励示例来说明我们的方法。主要思想是函数index\_user将函数参数p放入全局变量p\_global（第8行）中，并恢复p\_global的先前值（恰好在函数末尾第13行）。但是，当条件cmp在第9行失败时，index\_user不会还原p\_global的值。 在实际程序中会发现这种错误（即忘记恢复以前的指针值）（此UaF是从CVE2011-4130借来的）。因此，如果在第29行的调用中，此行为发生这种情况时，p\_global将指向与p\_index相同的内存区域，因此，在第30行释放后，p\_global将成为悬空指针。 然后，在第33行，malloc可以返回与p\_index指向的存储区相同的存储区。 在这种情况下，第38行的比较将始终为真。



**1.2我们的方法**

尽管更复杂，但是静态分析通过允许探索整个程序执行集合，提供了比动态分析更完整的结果。 例如，对于UaF，危险路径必须跟踪几个事件：首先，分配堆地址，其次，释放该地址，最后访问该地址指向的内存。 使用动态搜索，我们几乎没有机会找到满足所有这些要求的路径。 此外，静态分析与某些执行规范无关，例如有效地址。 例如，对于UaF，我们可以独立于调用的分配器来检测危险行为。

我们的贡献被称为GUEB，为检测use-after-free二进制漏洞生成图，它基于三个步骤。首先，我们使用专用的值分析（请参阅第2.2节），在考虑别名的情况下跟踪堆操作和地址传输。 其次，我们利用这些结果来静态识别UaF漏洞。 最后，我们为每个UaF提取子图，顺序描述悬空指针的创建，释放和使用位置（请参见第3.2节）。 图1给出了从清单1中提取的子图（在源代码级别显示）。 它标识块26（指针p的创建），块30（释放该指针）和块38（对p的引用）。

**1.3一些相关工作**

有几种工具可以在源代码级C代码中静态跟踪UaF（例如Polyspace或Frama-C [9]）。 这些工具主要用于安全性验证：不遵守某些约束的程序将被拒绝，例如C中未定义的行为。在漏洞分析中，我们恰恰对无法预测的行为感兴趣，这些行为可以被成功利用。 例如，对于UaF，我们不仅对悬空指针的使用感兴趣，还希望对它们的创建位置以及如何有效利用它们进行本地化。要获得有关内存布局的此类详细信息，需要在二进制级别分析代码，同时考虑到编译器优化和使用中的库。

但是，将静态分析提升为二进制代码非常具有挑战性[3]。 在学术方面，有几个开放的平台提供一些多用途的静态分析（程序切片，CFG构造，数据流分析等），以及定义明确的API，以访问中间代码表示形式。 例如，您可以提及REIL [10]，BAP [6]或Bincoa [5]。 但是它们都没有提供用于UaF检测的任何现成解决方案。

结果，就二进制代码而言，据我们所知，现有的UaF检测工具都基于动态方法。 例如，AddressSanitizer [14]需要使用特定的库来编译二进制文件。 Undangle [7]是专门用于UaF分析的工具：从给定的执行跟踪开始，该工具旨在识别在其中创建了悬空指针的程序点，以检测可能的漏洞的根源。 从这个意义上讲，它追求与我们相同的目标。 但是，这种方法需要首先识别导致UaF漏洞的路径，以检测每条汇编指令，这会减慢执行速度。

本文的组织方式如下：第2节介绍了内存模型和使用的值分析。第3节提出了我们建议的检测UaF模式和方法。 第4节总结了一些实验结果，并给出了当前原型的局限性以及未来工作的方向。

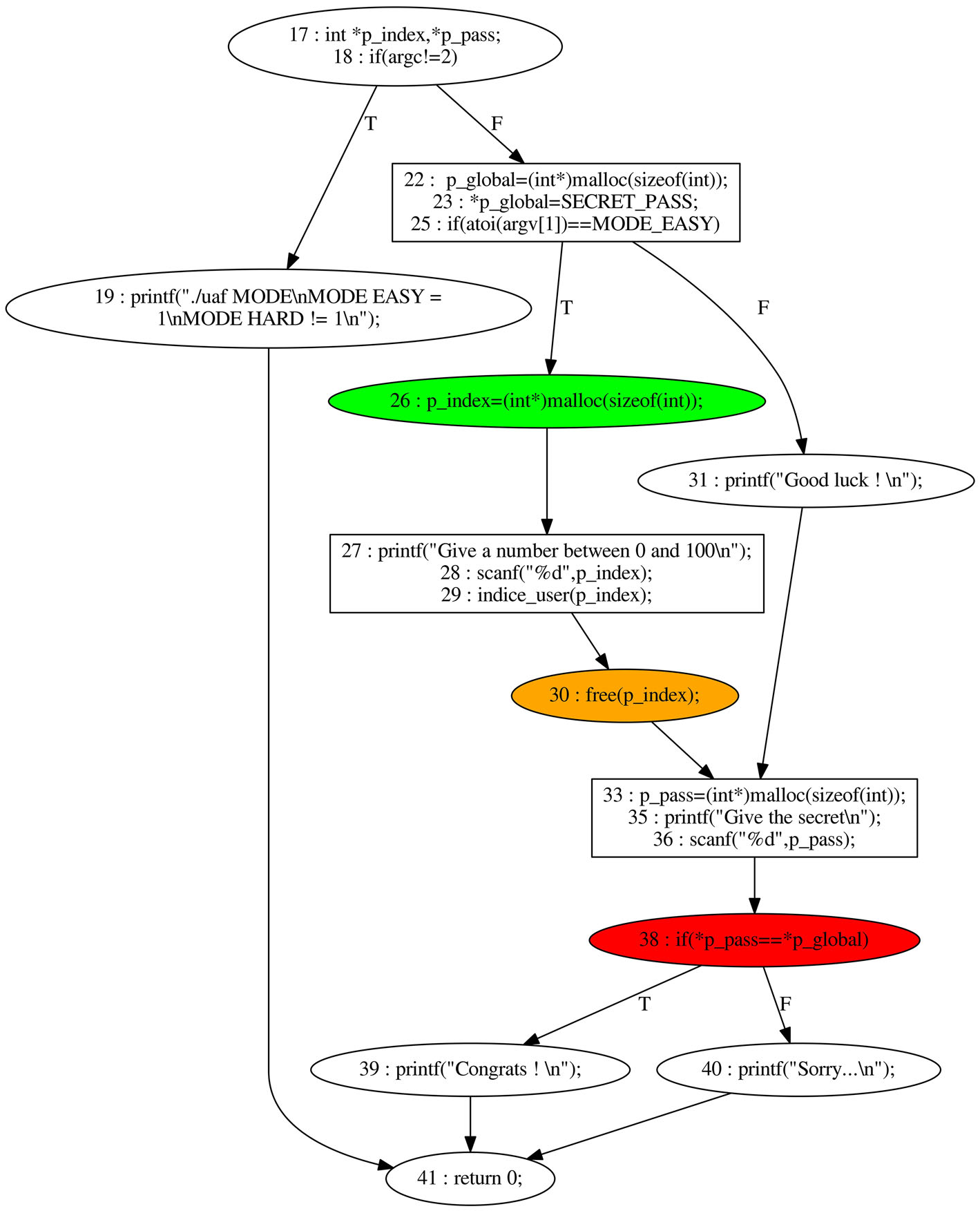


图1清单1中提取的子图

**2内存模式和值分析**

在本节中，我们详细介绍GUEB方法的三个步骤。 首先，我们解释堆栈和堆元素的表示方式。

**2.1抽象内存表示**

我们假设堆栈中的地址表示为相对于基址寄存器EBP的偏移量。由于过程内联分析是通过过程内联实现的，因此每个堆栈元素都由一对（EBP0，offset）表示，其中EBP0是初始值 EBP的值。例如，p\_index用（EBP0，−24）表示，p\_global\_save用（EBP0，−36）表示。全局变量具有由标识符表示的常量地址，此处为变量名（例如p\_global）。

关于堆，我们将HE定义为所有可能的堆元素的集合。 HE的元素是一对（base，size），其中base是分配标识符，size是分配大小。 这样的一对也被称为块。 PC是所有程序点的集合。 我们定义HA和HF，这两个函数分别关联每个点pc处所有当前已分配或已释放元素的集合（HA∈PC→P（HE），HF∈PC→P（HE），P（S）是S的幂集。 我们使用经典假设（在静态指针验证中），判定每次分配都提供一个新的内存块（尽管这种方法研究可利用性并不现实，但足以检测漏洞）。

表1 VSA结果

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**2.2值集合分析（VSA）**

值分析步骤的目标是静态发现哪个程序点分配或释放哪个堆元素。 然后，必须跟踪地址传输以及分配大小。 因此，除了HA和HF这两个函数外，我们的值分析还为每台pc生成了一个抽象环境AbsEnv。 AbsEnv将每个地址包含一个可能的值集，每个地址对应于块或分配大小。

表1显示了分析清单1时获得的一些结果。在两个第一次分配中，第22和26行中，创建了两个新的块。 在调用index\_user的过程中，将存储，修改和恢复p\_index的值（第7、8和13行）。 由于index\_user中有两个分支，因此在此调用行29的末尾，p\_index的可能值为chunk0或chunk1。 因此，在第30行，当chunk1释放时，p\_index变为悬空指针。 在第33行，将新分配分配给p\_pass，在第38行，将p\_pass的值与p\_global的值进行比较。

我们的VSA分析是[4]等现有分析的简化版本。 确实，我们专注于通常不需要复杂的数值计算的信息：在内存位置之间查找别名，检索堆元素访问以及计算分配大小。 此外，似乎大多数已知的UaF漏洞对特定的循环迭代都不敏感。 结果，我们的分析被实现为应用程序的控制流图（CFG）的正向遍历，其中循环最多展开一次，表示一次内存抽象（同一位置的抽象）表示同一循环的多次执行 ）。 而且，在近似的情况下，使用所谓的弱更新假设[4]来实现内存修改。

我们在下面给出与malloc和free调用相关的传递函数。 对于malloc调用，ad表示包含块大小的参数，r表示返回值。 对于free调用，ad表示参数，其中包含要释放的指针。 f←{x→e}表示与f相同的函数，只是在点x处的值为e。

**定义1** 与malloc相关的传递函数(pc, H A, HF, AbsEnv,ad,id\_max) *=(HA′,HF′,r, id\_max′)*

where:

*r* = (*baseid*\_*max*,*size*(*AbsEn*v(*ad*)))

*HF ′= HF*

*HA ′*= *HA* ← {*pc* 

*id*\_  and *size*(*s*) = v if *s* = {v} and *size*(*s*) = *Any* otherwise[[1]](#footnote-1).

**定义2** 与free相关的传递函数*f free*(*pc*, *H A*, *HF*, *AbsEn*v,

where:

*HF*= *HF*←{*pc* → (*HF*(*pc*) ∪ (*AbsEn*v(*ad*) ∩ *HE*))}

*HA*= *HA*←{*pc* → (*HA*(*pc*) \ (*AbsEn*v(*ad*) ∩ *HE*))}

之所以使用（AbsEnv（ad）∩HE），是因为AbsEnv（ad）可能引用了一些不在堆中的元素。

**3 UaF检测和子图提取**

我们首先说明上一节中描述的VSA如何用于识别UaF模式，然后展示如何提取程序片段（作为CFG的一部分）以充分表征每个UaF。

**3.1 UAF检测**

UaF对应于使用悬挂指针。 根据VSA的结果，我们定义了AccessHeap（pc），该函数返回在pc上访问的HE的所有元素（AccessHeap：PC→P（HE））。 在我们的实现中，我们使用REIL [10]作为中间表示。在REIL中，只有两条指令专用于内存访问：

– LDM ad„ reg，用于将Mem（ad）的内容加载到寄存器reg中  
 – STM reg„ ad，用于将寄存器reg的内容存储到Mem（ad）中

因此，我们定义AccessHeap如下：  
 AccessHeap（LDM ad ,, reg）= AbsEnv（ad）∩HE。

AccessHeap（ST M reg ,, ad）= AbsEnv（ad）∩HE

最后，所有可能的UaF漏洞的setUaf Set定义为：

**定义3** use after free 描述

Uaf Set = {（（pc，chunk）| *chunk*∈（AccessHeap（pc）∩HF（pc））}

在表1的示例中，我们有：AccessHeap（38）= {chunk0，chunk1，chunk2}

然后Uaf Set = {（38，chunk1）}，这意味着chunk1在第38行悬空并取消引用。

**3.2 use after free子图**

GUEB的最后一步是从初始代码中提取一个子图，其中包含给定UaF中涉及的所有指令。 令pred为返回pc的所有前任（在CFG中）的函数，并pred \* 的传递闭包。 还让point\_alloc（resp.point\_free）是一个函数，该函数将给定块分配给该块的pc集合与给定块相关联（resp.freed）。（，）在Uaf Set中，我们通过选择以下程序点（从开始）来切片初始CFG：

1.所有程序点位于引起UaF的点，和释放（以橙色表示）的点之间，即

pred \*（）∩succ \*（point\_free（））

2.所有程序点都位于释放和分配的点之间（绿色），即

pred \*（point\_free（））∩succ ∗（point\_alloc（））

3.位于分配chunkuaf的点与程序的入口点（蓝色点）之间的所有程序点，即 pred \*（point\_alloc（））

图2显示了为我们的示例提取的子图的二进制级别。 此子图对于研究是否可以利用UaF以及如何进行研究非常有用：例如，在释放和取消引用UaF的点之间是否发生了新分配，如本例中的第33行。

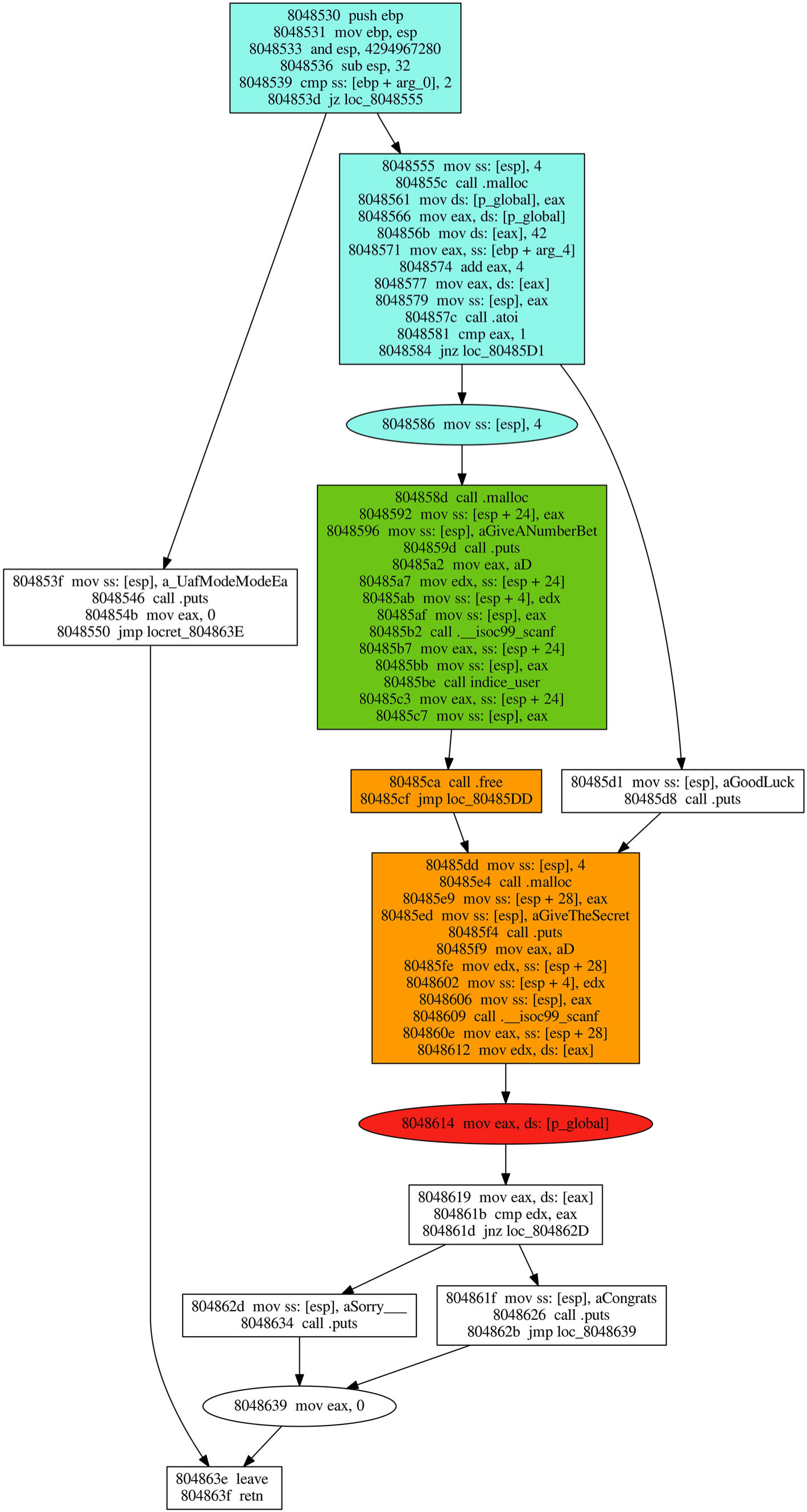


图2

**4结论**

首先我们给出一些实验结果，然后讨论当前原型的一些局限性以及未来工作的方向。

**4.1实验结果**

我们的方法已在Jython开发的原型工具GUEB中实现，该工具使用IDApro转换汇编代码中的二进制文件。然后，使用BinNavi将此汇编代码转换为中间表示REIL [10]。最后，我们使用BinNavi的专用API Monoreil，该API可以轻松地在控制流程图上实现静态分析。GUEBwase评估了真实漏洞，CVE 2011-4130出现在ProFTPD中（有关此CVE的详细说明，请参见[15]）。这种形式的UaFi类似于清单1：错误对应的路径，其中指针未正确还原。从静态分析的角度来看，此案例研究引入了一些困难：代码多，复杂结构，局部和全局变量等。GUEB是原型，它相对较慢。为了加快分析速度，我们手动选择了要分析的10个功能的子集。尽管如此，该实验还是非常有意义的：我们在i7-2670QM处理器上，在30分钟内处理了大约2200个节点的CFG。我们确定没有任何误报的UaF，并且提取的子图是一小片（包含460个节点）。

**4.2 VSA实施的局限性和改进**

为了补充在执行过程中检测内存错误的动态工具[12,14]，我们特意选择使用静态方法，以解决自由检测后更好的使用范围。特别是，我们针对小型程序（或程序的一部分），这些程序应具有较高的安全性要求。这种静态方法可以用来对程序的一部分进行深度分析，这要归功于轻量级分析（包括动态检测）。但是，我们的VSA实际上并不完整，可以通过几种方式进行改进。首先，可以加强值分析步骤：当前循环最多扩展一次，这种近似值不足可能导致假阴性（缺少某些别名），并且分配大小不正确（检测步骤中未真正使用）。在源级别和二进制级别都将分配考虑在内是一个悬而未决的问题[2]。尽管如此，此处采用的解决方案包括折叠所有在给定分配位点分配的节点（在[2]中称为“分配位点抽象”）。Use after free在使用方面进行良好的权衡。特别是，我们不会错过（通常）仅取决于迭代次数的UaF。从实践的角度来看，我们还需要形式化和开发更有效的过程间分析。在当前的实现中，我们使用的是幼稚的内联技术，这种技术无法很好地扩展。最后，为了与REIL框架兼容，GUEB用Jython编写，这很慢。选择一种更有效的语言（例如Caml或C / C ++）将大大加快我们的实施速度。

**4.3展望**

旨在精确分析可利用性的方法通常基于符号推理或共和论，如[8,11]中对缓冲区溢出可利用性的描述。 我们在此追求的目标是建立可利用后使用的输入。 因此，下一步在于表征无用后使用的可利用性，即修改和控制悬空存储器的内容的可能性。 为此，我们必须确定执行路径，其中在“空闲”和“使用”操作之间进行了一些新的分配，以及如何从用户输入中重写此分配的内存。 此步骤需要一个相当细粒度的堆模型，以模拟分配器行为（包括潜在的重新分配）。

**参考**

1. Afek, J., Sharabani, A.: Dangling pointer: pointer. Smashing thepointer for fun and profit. Black Hat USA (2007)
2. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocatedstorage static analysis. In: Yi, K. (ed.) SAS ’06: static analysis symposium, volume 4134 of LNCS, pp. 221–239. Springer, Berlin (2006)
3. Balakrishnan, G., Reps, T.: Wysinwyx: what you see is not whatyou execute. ACM Trans. Program. Lang. Syst. **32**(6), 23:1–23:84 (2010)
4. Balakrishnan, G., Reps, T.W.: Analyzing memory accesses in x86 executables. In: Duesterwald E (edi) CC, volume 2985 of LNCS, pp. 5–23. Springer, Berlin (2004)
5. Bardin,S.,Herrmann,P.,Leroux,J.,Ly,O.,Tabary,R.,Vincent,A.: The bincoa framework for binary code analysis. In: Proceedings of CAV’11, pp. 165–170. Springer, Berlin (2011)
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: Bap: a binaryanalysis platform. In: Proceedings of the 23rd International Conference on Computer Aided Verification. CAV’11, pp. 463–469. Springer, Heidelberg (2011)
7. Caballero, J., Grieco, G., Marron, M., Nappa, A.: Undangle: earlydetectionofdanglingpointersinuse-after-freeanddouble-freevulnerabilities. In: Heimdahl, M.P.E., Su, Z. (eds.) ISSTA, pp. 133– 143. ACM (2012)
8. Cha,S.K.,Avgerinos,T.,Rebert,A.,Brumley,D.:Unleashingmayhem on binary code. In: IEEE Symp. S&P, pp. 380–394 (2012)
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J.,Yakobowski, B.: Frama-c—a software analysis perspective. In: SEFM, pp. 233–247 (2012)
10. Dullien, Thomas, Porst, Sebastian: Reil: A platform-independentintermediate representation of disassembled code for static code analysis. CanSecWest (2009)
11. Heelan,S.:Automaticgenerationofcontrol flow hijackingexploitsfor software vulnerabilities. Master’s thesis, University of Oxford, Computing Laboratory (2009)
12. Nethercote, N., Seward, J.: Valgrind: a program supervision framework. Electr. Notes Theor. Comput. Sci. **89**, 44–66 (2003)
13. Rawat, S., Mounier, L.: Finding buffer overflow inducing loopsin binary executables. In: Proceedings of the Sixth International Conference on Software Security and Reliability, SERE 2012, pp. 177–186. IEEE (2012)
14. Serebryany,K.,Bruening,D.,Potapenko,A.,Vyukov,D.:Addresssanitizer: a fast address sanity checker. In: USENIX ATC 2012 (2012)
15. Vupen. Technical analysis of proftpd response pool use-afterfree (cve-2011-4130). [http://www.vupen.com/blog/20120110.](http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php)

[Technical\_Analysis\_of\_ProFTPD\_Remote\_Use\_after\_free\_ CVE-2011-4130\_Part\_I.php](http://www.vupen.com/blog/20120110.Technical_Analysis_of_ProFTPD_Remote_Use_after_free_CVE-2011-4130_Part_I.php)

1. [↑](#footnote-ref-1)