Microsoft | Trustworthy Computing

漏洞分析、实际数据流分析和可视化

Jeong Wook oh
jeongoh@microsoft.com
Microsoft Malware Protection Center

漏洞分析、实际数据流分析和可视化

非官方中文译本·安天实验室 译注

文档信息			
原文名称	Vulnerability Analysis, Practical Data Flow and Analysis Visualization		
原文作者	Jeong Wook oh	原文发布日期	2012年3月23日
作者简介	Jeong Wook oh曾就职于微软，担任高级安全研究员；目前就职于惠普公司。 http://www.linkedin.com/in/ohjeongwook		
原文发布单位	微软公司		
原文出处	https://cansecwest.com/csw12/Vulnerability%20Analysis%20and%20Practical%20Data%20Flow%20Analysis%20%20Visualizationfinaledit.pdf		
译者	安天技术公益翻译组	校对者	安天技术公益翻译组

免责声明	• 本译文译者为安天实验室工程师，本文系出自个人兴趣在业余时间所译，本文原文来自互联网的公共方式，译者力图忠于所获得之电子版本进行翻译，但受翻译水平和技术水平所限，不能完全保证译文完全与原文含义一致，同时对所获得原文是否存在臆造、或者是否与其原始版本一致未进行可靠性验证和评价。 • 本译文对应原文所有观点亦不受本译文中任何打字、排版、印刷或翻译错误的影响。译者与安天实验室不对译文及原文中包含或引用的信息的真实性、准确性、可靠性、或完整性提供任何明示或暗示的保证。译者与安天实验室亦对原文和译文的任何内容不承担任何责任。翻译本文的行为不代表译者和安天实验室对原文立场持有任何立场和态度。 • 译者与安天实验室均与原作者与原始发布者没有联系，亦未获得相关的版权授权，鉴于译者及安天实验室出于学习参考之目的翻译本文，而无出版、发售译文等任何商业利益意图，因此亦不对任何可能因此导致的版权问题承担责任。 • 本文为安天内部参考文献，主要用于安天实验室内部进行外语和技术学习使用，亦向中国大陆境内的网络安全领域的研究人士进行有限分享。望尊重译者的劳动和意愿，不得以任何方式修改本译文。译者和安天实验室并未授权任何人士和第三方二次分享本译文，因此第三方对本译文的全部或者部分所做的分享、传播、报道、张贴行为，及所带来的后果与译者和安天实验室无关。本译文亦不得用于任何商业目的，基于上述问题产生的法律责任，译者与安天实验室一律不予承担。

漏洞分析—为什么？

找出漏洞的根本原因

- 找出用户提供的数据和漏洞之间的关系

方法论

- 动态分析
  - 调试器
    - 源代码或符号是正数
  - 动态二进制插桩
- 静态分析
  - 反汇编
  - 源代码审查
- 数据格式分析
  - 使用文件或协议规范解析分析对象
  - 分析目标数据的不一致

插桩和指令仿真

动态二进制插桩（DBI）

- 基本上，在指令、基本块、函数和内存访问操作时执行用户定义的操作。
- 回调模型
  - 使用DBI的方法之一为每个指令、基本块、函数和内存访问设置回调函数。
  - 安装
    - 引脚、dynamoRIO等
    - 专有安装
- 我们可以追踪一系列导致漏洞的指令

指令仿真

- 为什么二进制插桩不够？
  - 除了二进制插桩，为什么需要其他的指令仿真？
- 通常，二进制插桩提供回调，以保存指令、基本块、功能等信息 。
- 它们不提供评估指令执行副作用的方法
  - 例如：哪个寄存器被影响，哪个寄存器用于该指令等。

假设

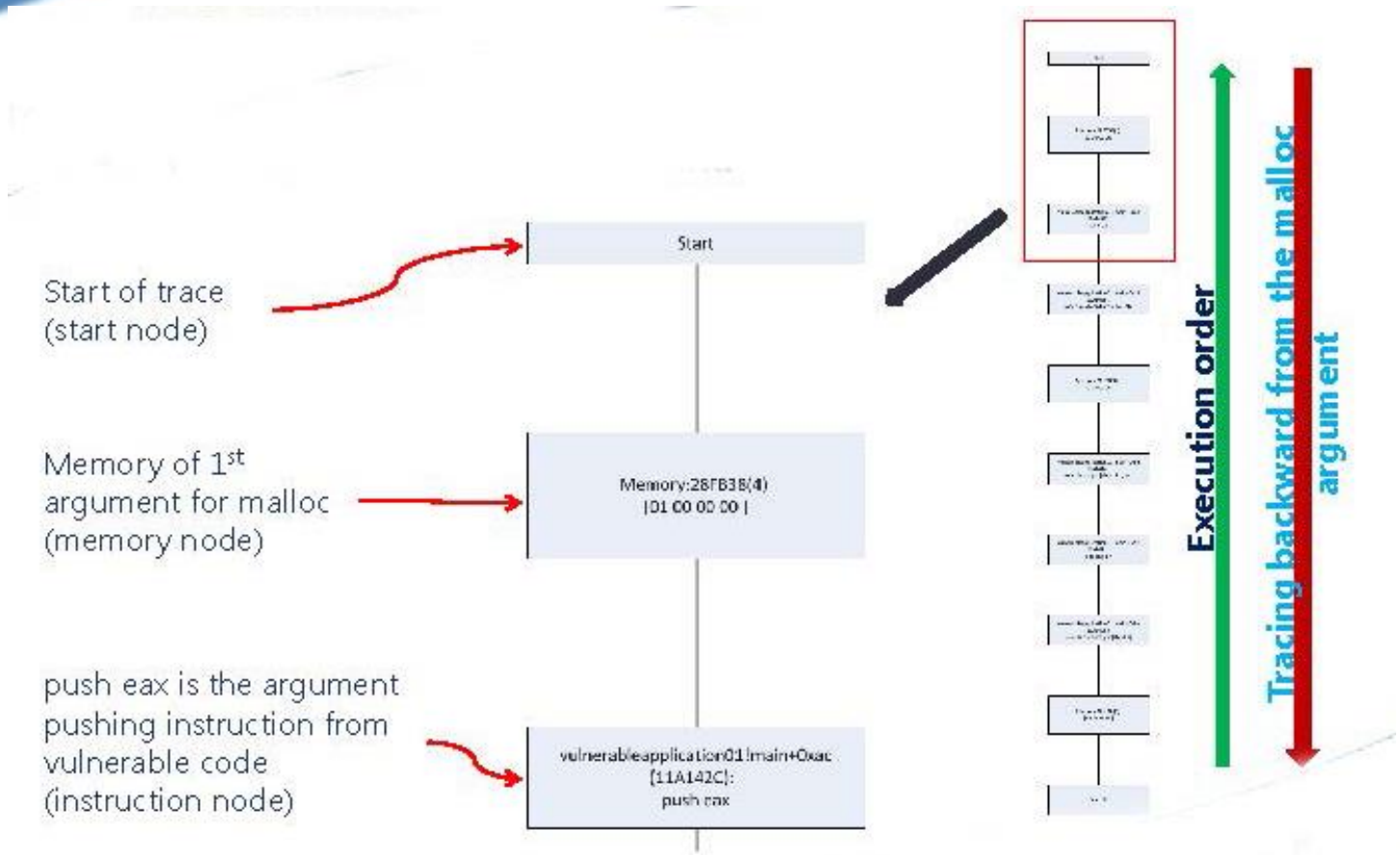
- 在令人兴奋的公共或私人工具，额外代码开发的情况下，下述假设是很可能的：
  - 有一个很好的插桩工具
    - 将程序执行日志保存到一个文件中，并在需要时获取所有运行信息（内存数据、注册表值）。
  - 有CPU指令仿真代码
  - 两个代码片段合作执行数据流分析
- 这些假设都是可能的，通过合理的努力是完全可以实现的。

基本漏洞分析方法

基本案例：样本漏洞程序

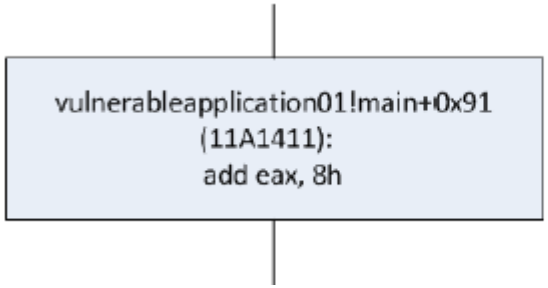
```
FILE_HEADER file_header;

DWORD NumberOfBytesRead;

ReadFile(
                hFile,
                &file_header,
                sizeof( FILE_HEADER ),
                &NumberOfBytesRead,
                NULL
);
if( NumberOfBytesRead == sizeof( FILE_HEADER ) )
{
                // Integer Overflow (file_header.size is DWORD)
                size_t length = sizeof( FILE_HEADER ) + file_header.size;

                // length is used for malloc argument
                char *dst = ( char * ) malloc( length );
                if( dst )
                {
                                memcpy( dst, &file_header, sizeof( FILE_HEADER ) );
                                ...

                }
}
```

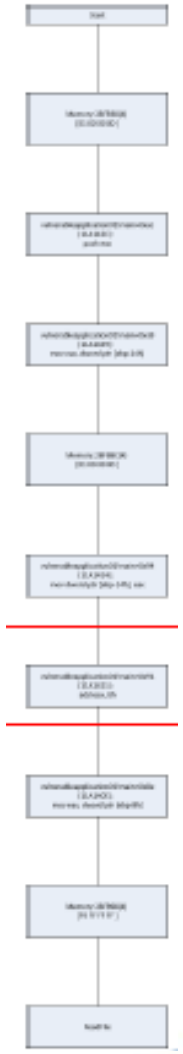
基本案例--追踪malloc参数



基本案例—整数溢出条件

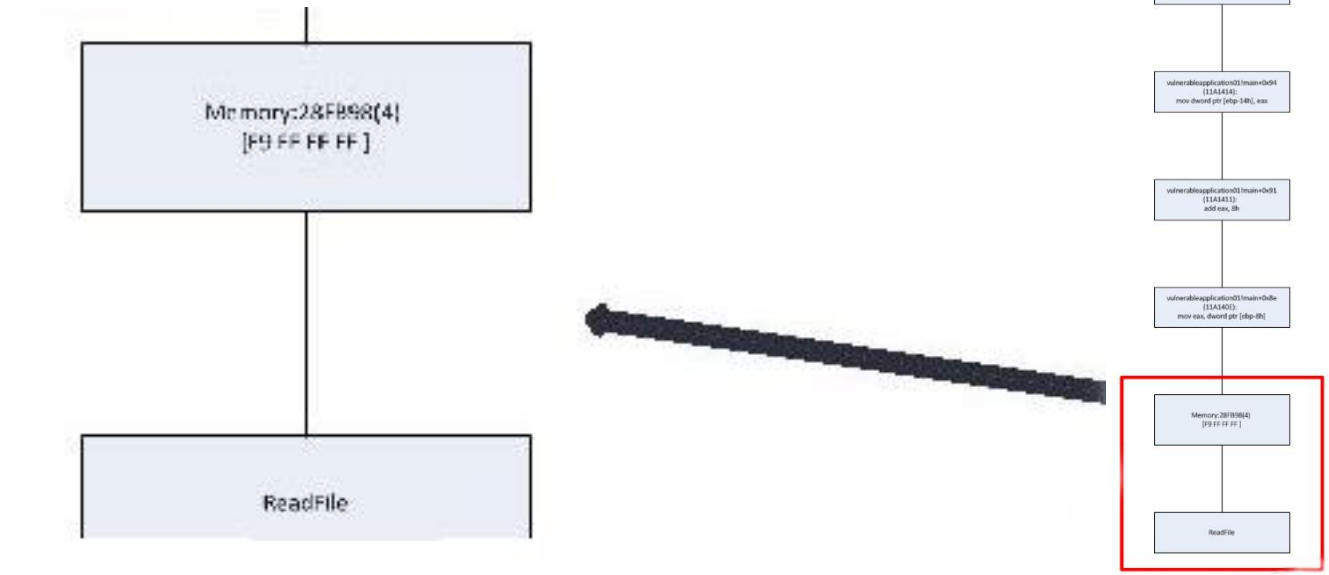
- 加8显示了整数溢出条件





基本案例—识别数据源

- 数据最终从来自用户提供的数据（ReadFile）
- 这意味着malloc的输入是用户可控的
- 对于基本案例，数据流分析是足够的，以确定该漏洞的特征。



控制流相关漏洞的案例

高级漏洞分析方法

污点分析vs漏洞分析

- 污点分析=数据流分析
  - 非常适合简单的漏洞，但在复杂情况下效果是有限的。
  - 例如，如果它是一个释放后使用（user-after-free）漏洞
    - 你可以追溯用于崩溃指令的数据的来源
    - 你可以进入被释放的内存
    - 你能确定这是一个未初始化的内存访问漏洞
    - 但是，数据流分析不会告诉你它是如何发生的。
- 需要更先进的漏洞分析方法
  - 如何能确定用户提供的数据会影响崩溃？
  - 用户数据不仅改变数据流，也可以改变控制流。

数据流分析

- 从崩溃点（Ground 0）
  - 追踪崩溃的原因
- 可以用该方法分析更多漏洞

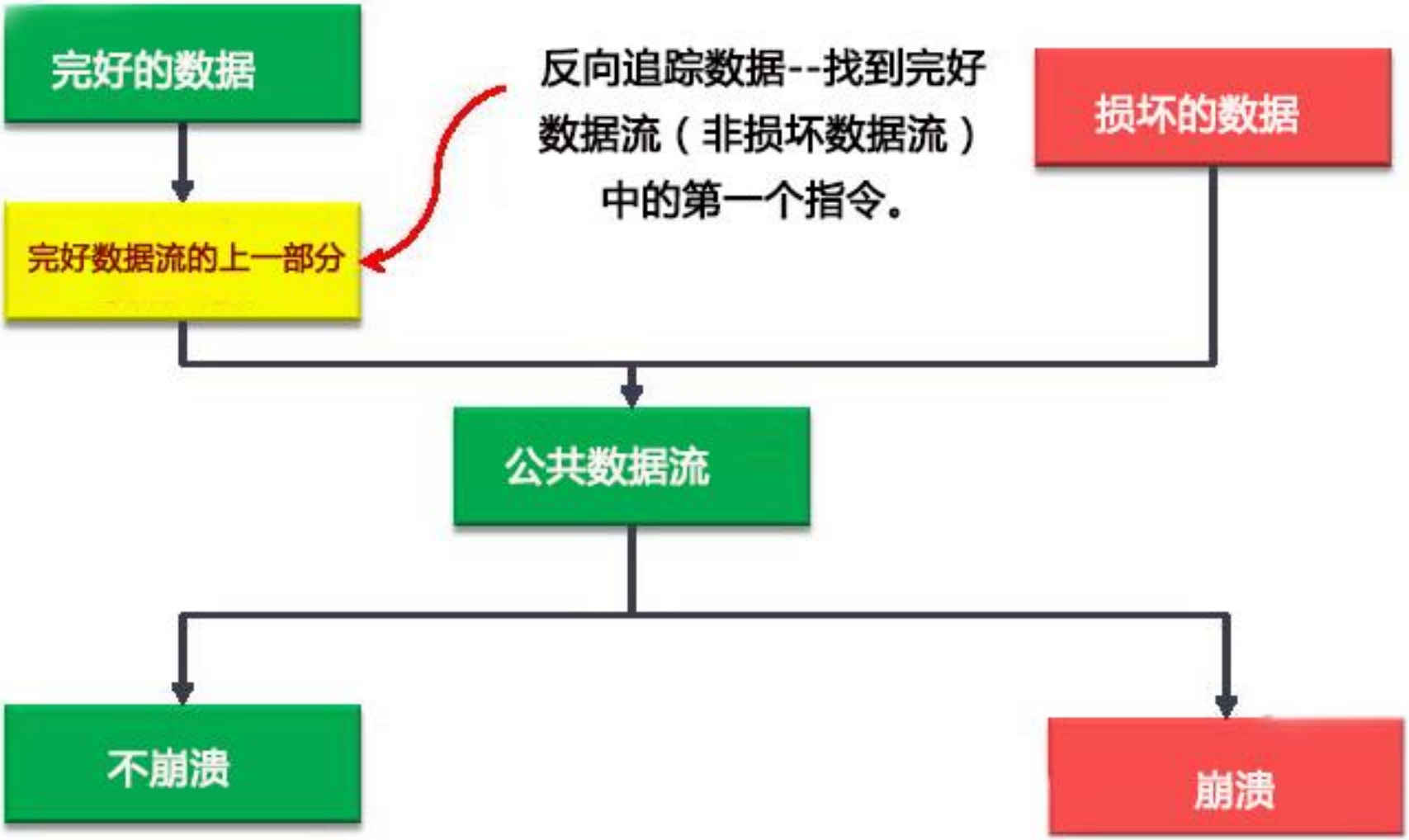
数据流分析的限制

- 有些漏洞无法通过简单的数据流分析进行追踪。
- 这些漏洞涉及基于用户提供数据的控制流变化。
  - 我们需要找到引起漏洞的控制流

数据流差分分析

- 比较数据流
  - 获得同一个漏洞代码位置的正常文件
  - 从正常文件和崩溃文件对比数据流
  - 分析该数据流差异
  - 重点分析负责数据流分流的代码部分

数据流差分分析


完好的数据
完好数据流的上一部分
反向追踪数据--找到完好数据流（非损坏数据流）中的第一个指令。
损坏的数据
公共数据流
不崩溃
崩溃

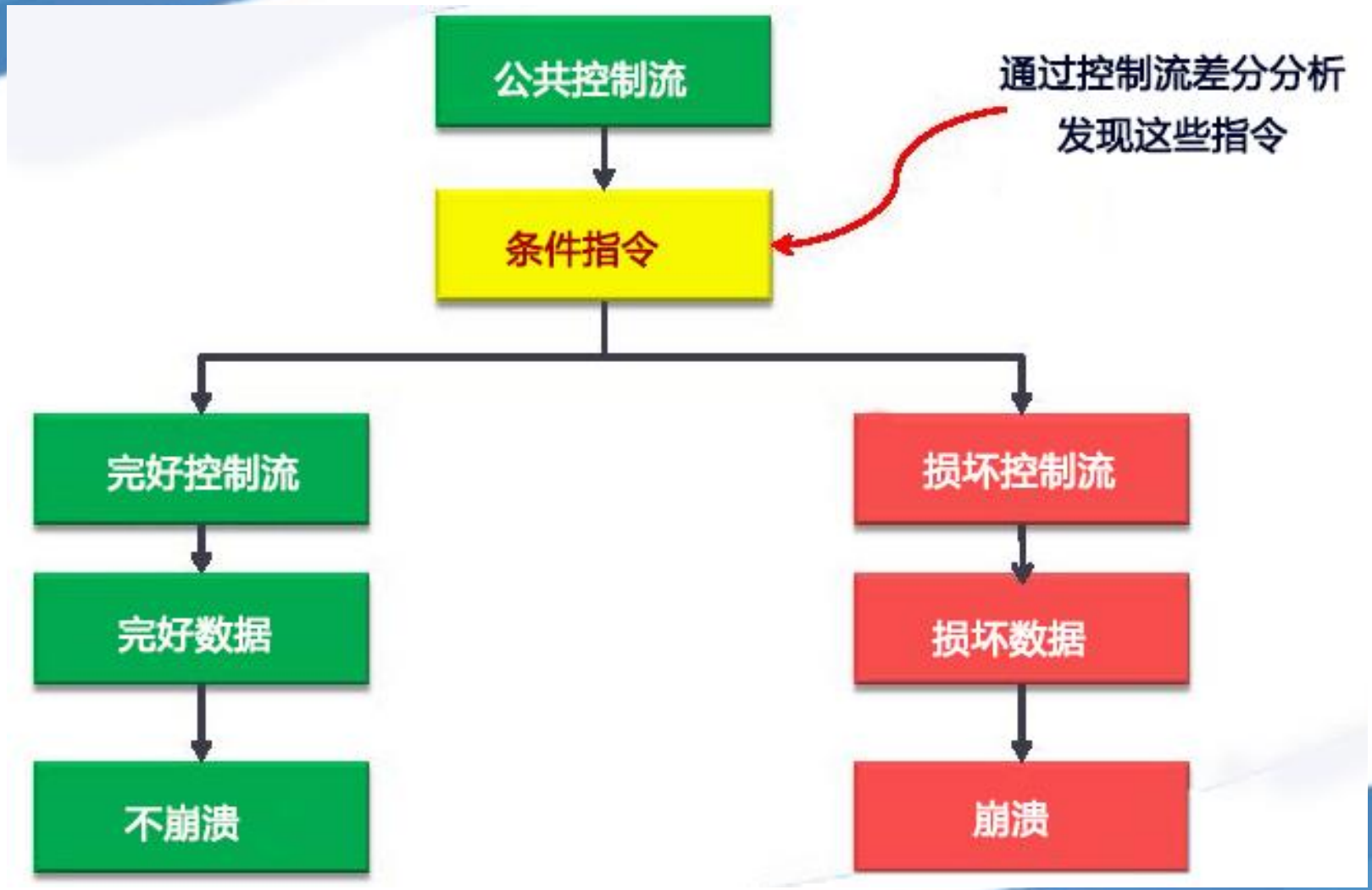
获得正常文件

- 知道崩溃点
  - 在崩溃点进行代码覆盖测试（基本模块或功能层面）
- 有样本文件
  - 你必须具备大量受影响格式的文件（在这种情况下，是指从各地获得的PDF文件）。我们称它们为模板文件。
- 使用漏洞软件运行样本文件，对漏洞点进行代码覆盖测试
  - 发现有漏洞点的样本
  - 这些样品用于数据流差分分析

控制流差分分析

- 确定控制流差分的关键条件
  - 对用于数据流差分分析的好/坏repro文件执行控制流差分分析
  - 推断建立控制流差分的条件
    - 采用数据流分析回溯到用户所提供的数据

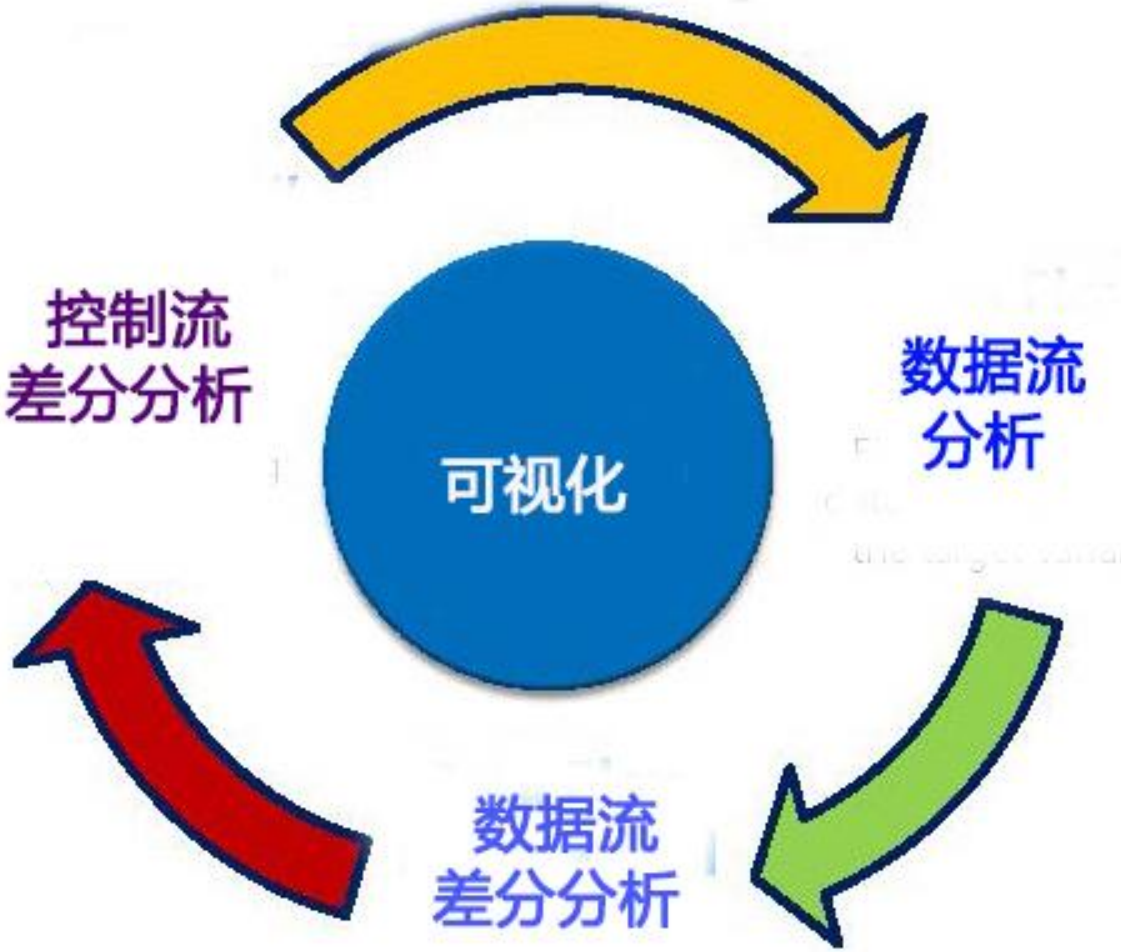
控制流差分分析



可视化

- 可视化有时会令事情变得更容易理解
- 当我们有数据/控制流数据的长链时，可视化可以大大帮助研究人员进行分析

高级漏洞分析—方法示例



示例

示例

- CVE-2411-2462（Adobe Reader）
  - 用于针对美国政府和军方公司的高针对性攻击。
  - 问题是U3D流处理代码
- 为什么呢？
  - 未初始化的内存访问
    - 不是一个非常典型的内存损坏问题
    - 调试比较棘手
    - 污点追踪只是告诉你一些数据来自未初始化或已释放的内存区域。
  - 公共漏洞已经出现，但除了提供给Adobe MAPP合作伙伴的漏洞分析，还没有其他任何漏洞分析。
    - 我们不披露任何新的漏洞
    - 这里披露的所有信息用于研究目的，不以任何方式帮助新漏洞的开发。

CVE-2411-2462（Adobe Reader）

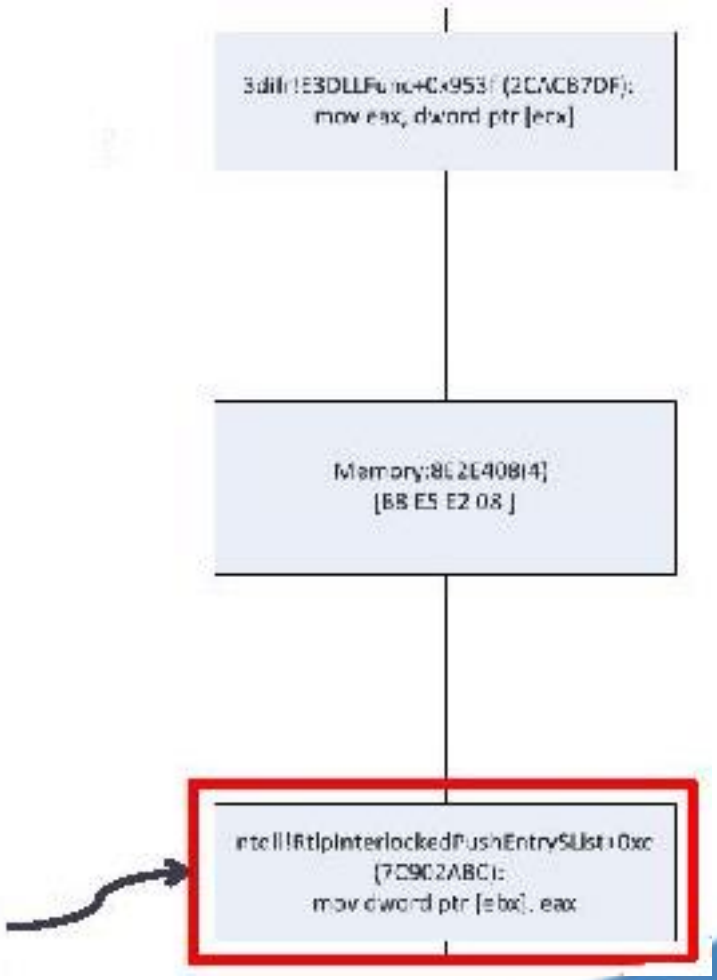
- 挑战
  - 被释放的内存来自哪里？
    - 我们能否发现导致漏洞的代码片段？
  - 崩溃是否与输入文件中的值有关？
    - 我们能否导致漏洞的文件位置？
  - 什么数值范围将会导致漏洞？
    - 我们能否找出漏洞发生的条件？

数据流分析—崩溃情况

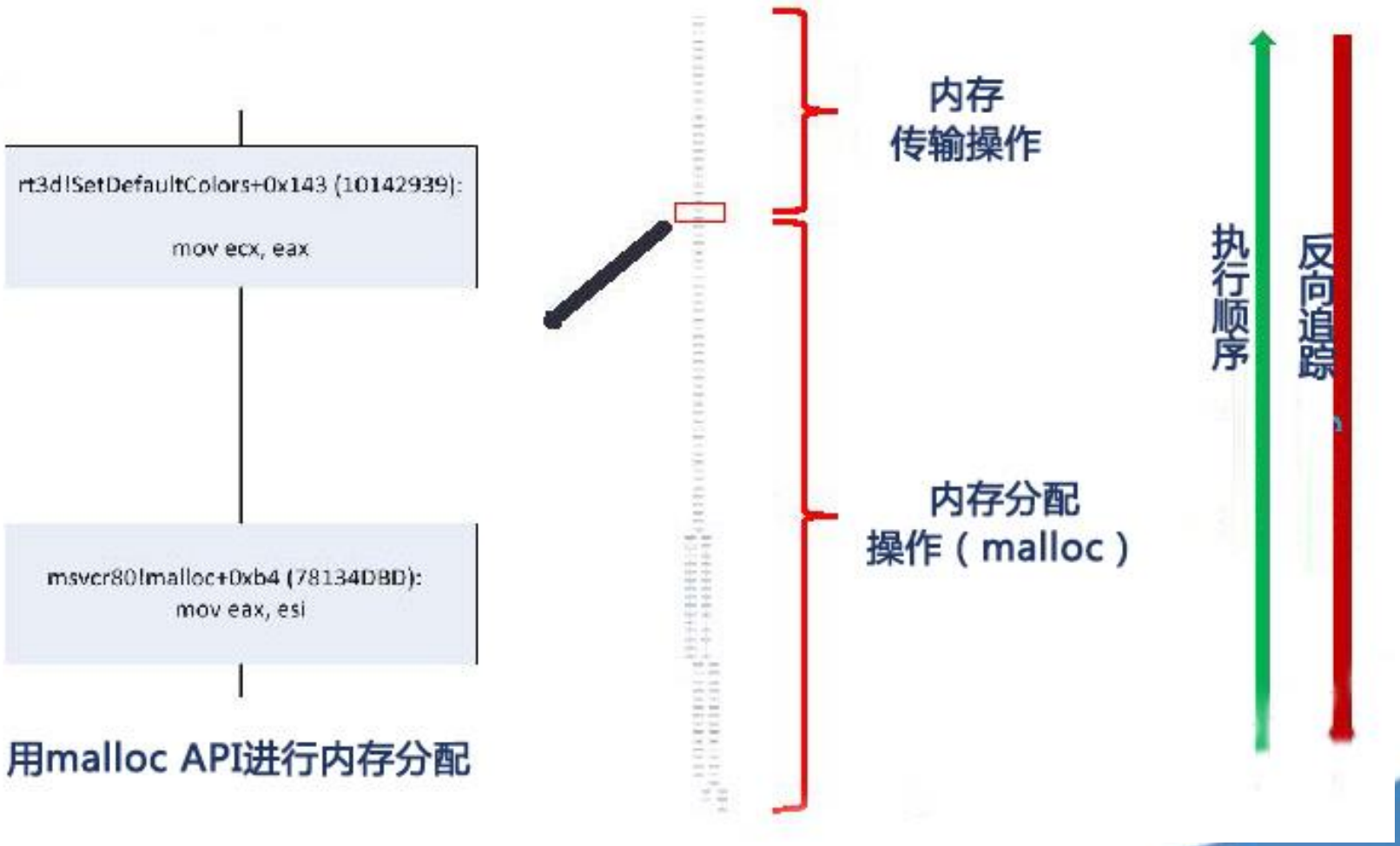
数据流分析—崩溃情况

- 该指针的值直接来自ntdll堆管理函数。
- 此漏洞是一个释放后使用或未初始化的内存访问漏洞。

如果您看一看调用堆栈，这是堆开放的API（ntdll!RtlFreeHeap）的一部分，最终从msvcr80!free API调用。



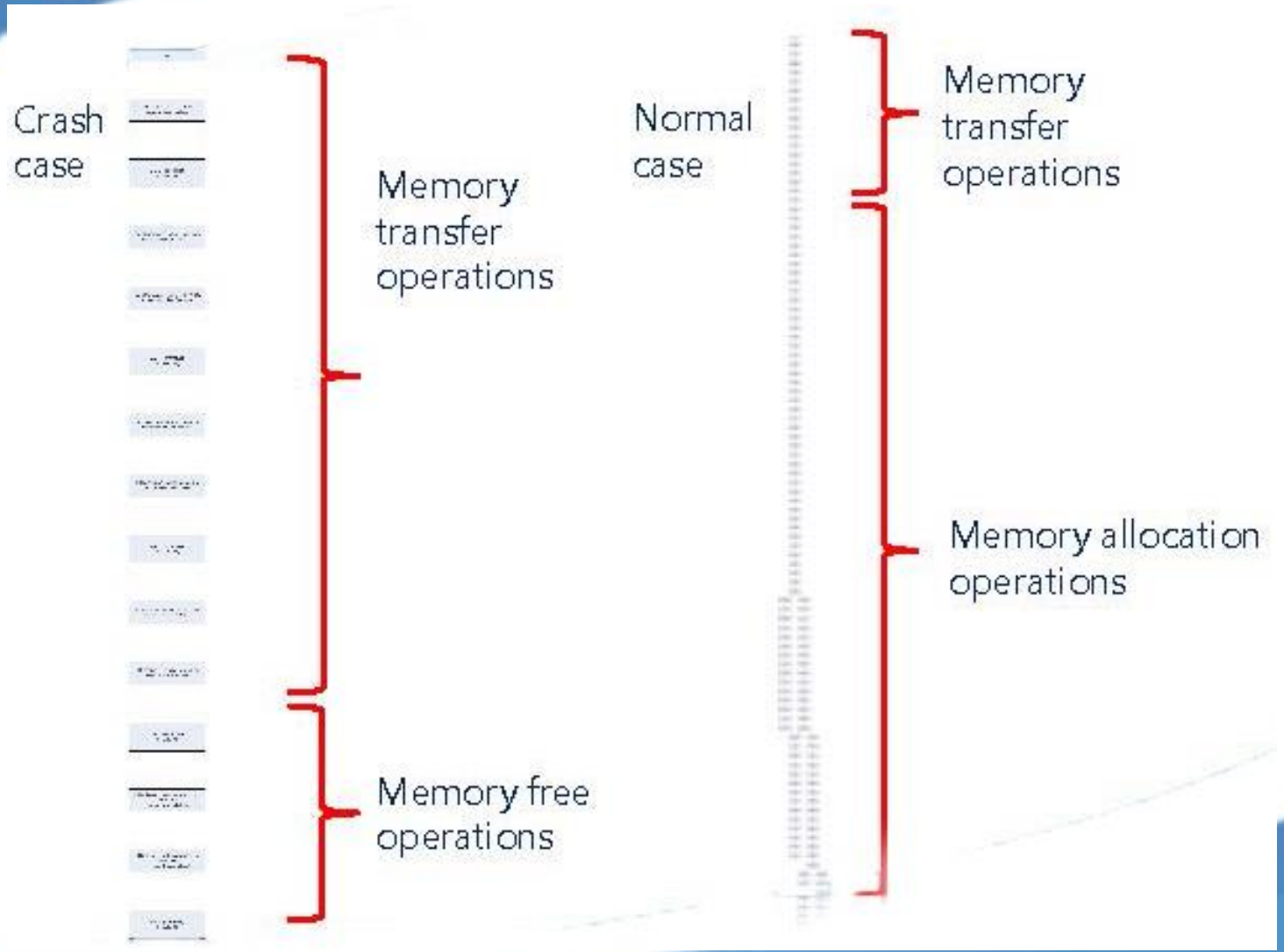
数据流图--正常情况


rt3dlSetDefaultColors+0x143 (10142939):
mov ecx, eax
msvcr80!malloc+0xb4 (78134DBD):
mov eax, esi
用malloc API进行内存分配
内存
传输操作
内存分配
操作（malloc）
执行顺序
反向追踪

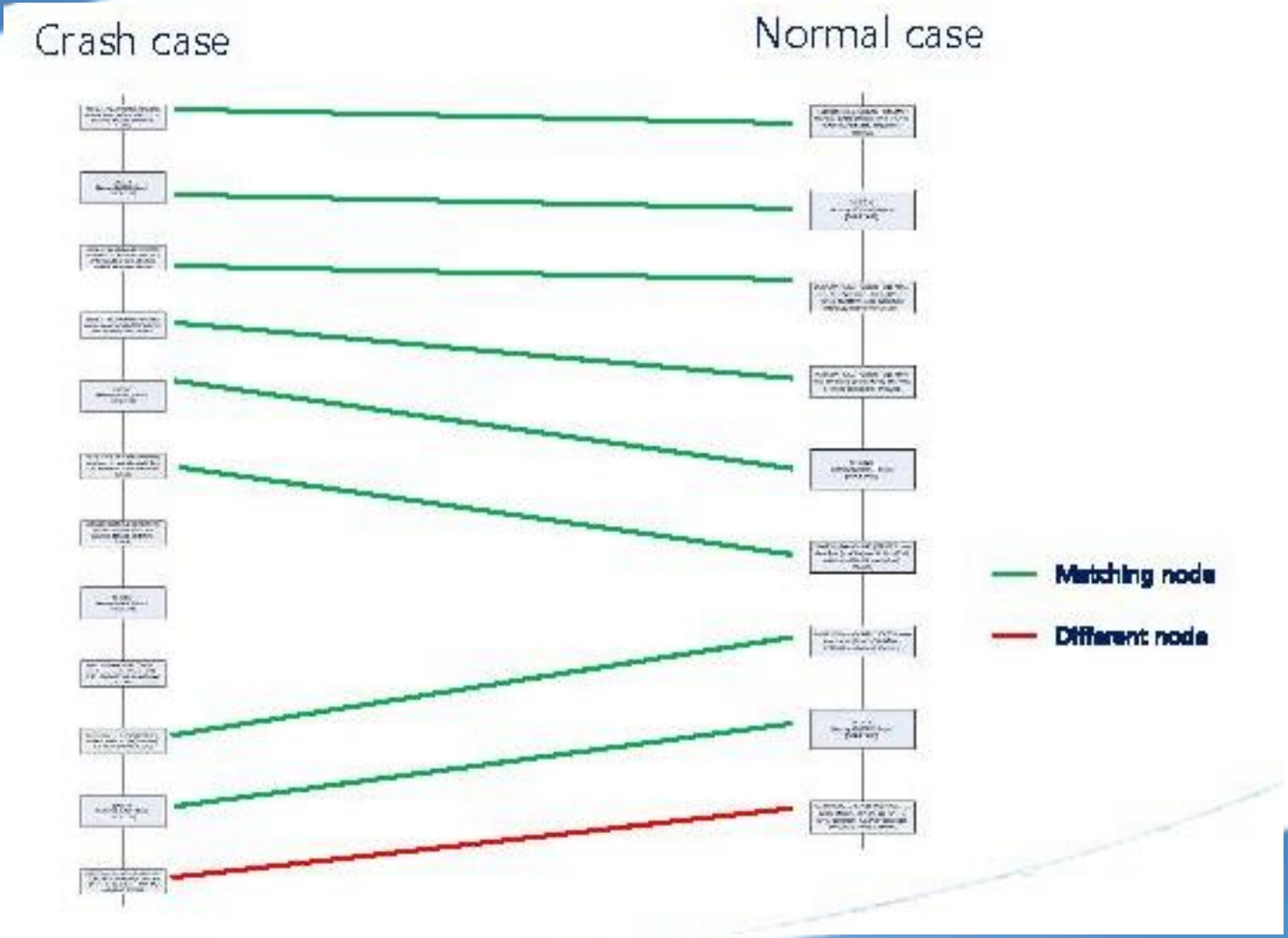
数据流图--正常情况

- 我们可以得出结论，正常指针应指向使用malloc API分配的内存区域。
- 从崩溃痕迹来看，指针指向开放的API释放的区域。

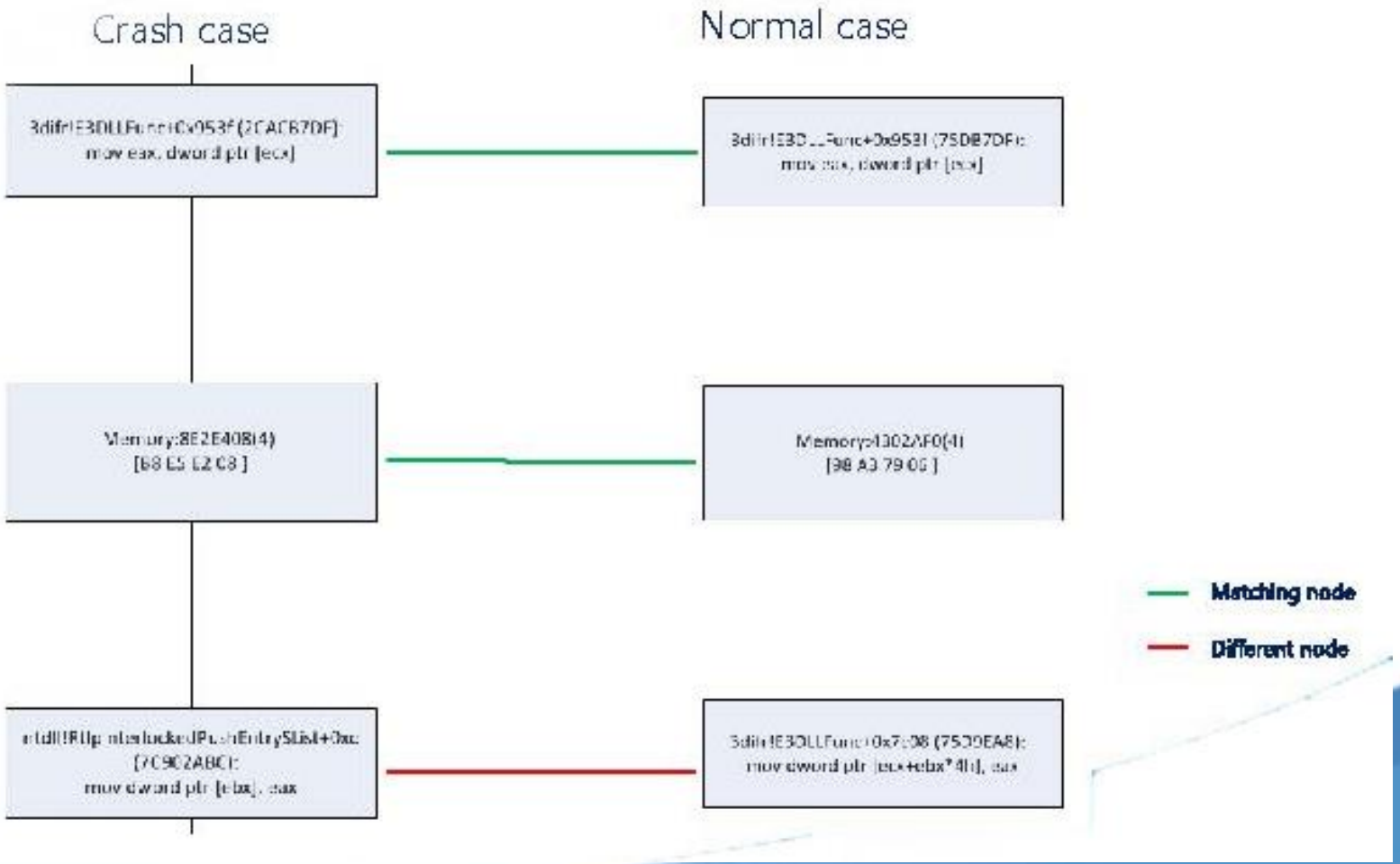
数据流差分分析


Crash case
Memory transfer operations
Memory free operations
Normal case
Memory transfer operations
Memory allocation operations

数据流差分分析



数据流差分分析


Crash case
Normal case
mov eax, dword ptr [ecx]
mov eax, dword ptr [ecx]
[68 E5 E2 C8]
[98 A3 79 0C]
mov dword ptr [ebx], eax
Matching node
Different node

数据流差分分析

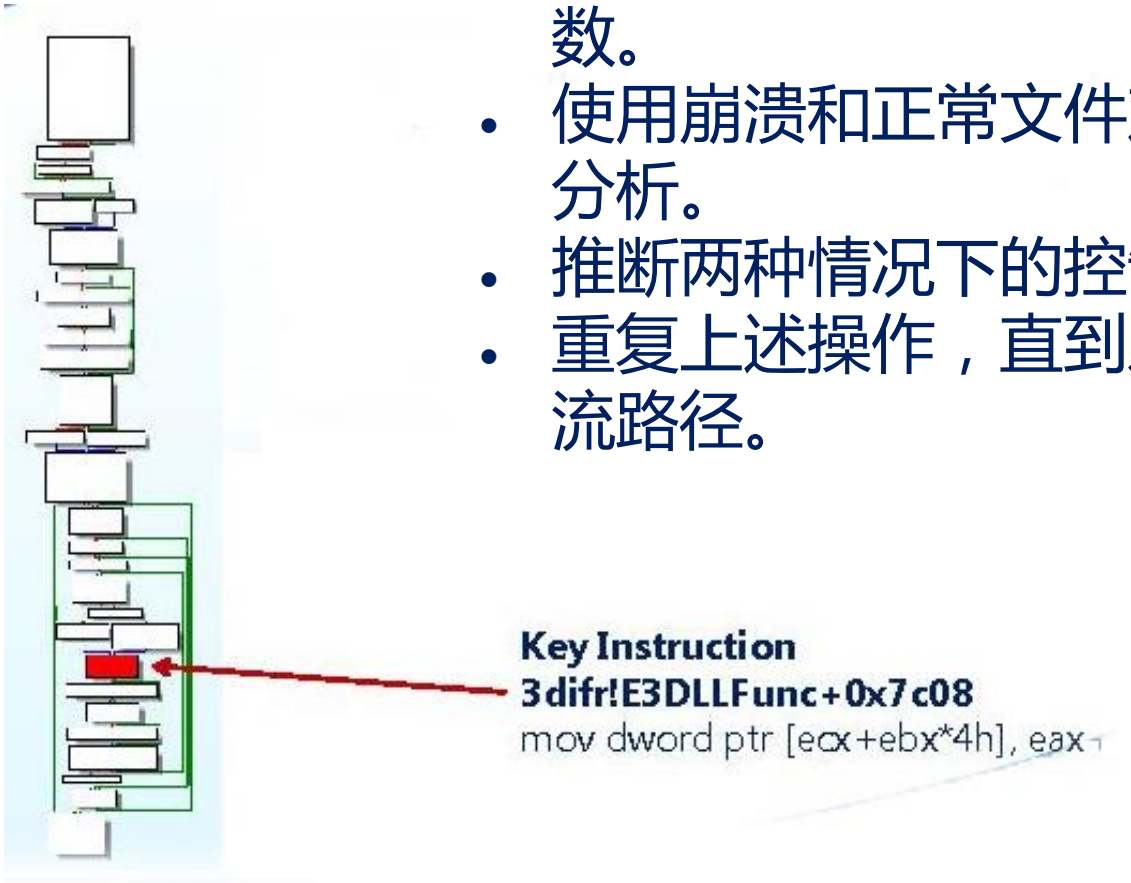
	崩溃情况	正常情况
地址	ntdll!RtlpInterlocked PushEntrySL ist+0xc	3difr!E3DLLFunc+0x7c08
指令	mov dword ptr [ebx], eax	**mov dword ptr [ecx+ebx*4h], eax (Key Instruction)**
分析	Ntdll!RtlpInterlocked PushEntrySL ist 从开放的API调用。 因此，ebx是堆管理数据结构的一部分，这意味着崩溃是由释放后使用类的漏洞导致的。	[ecx+ebx*4h] 是一个数组的地址，该数组的下界是ecx，索引是ebx。 Eax来自一个malloc API。 我们需要搞清楚为什么这个指令没有被攻击，以及围绕这个关键指令的最接近攻击。

控制流差分分析

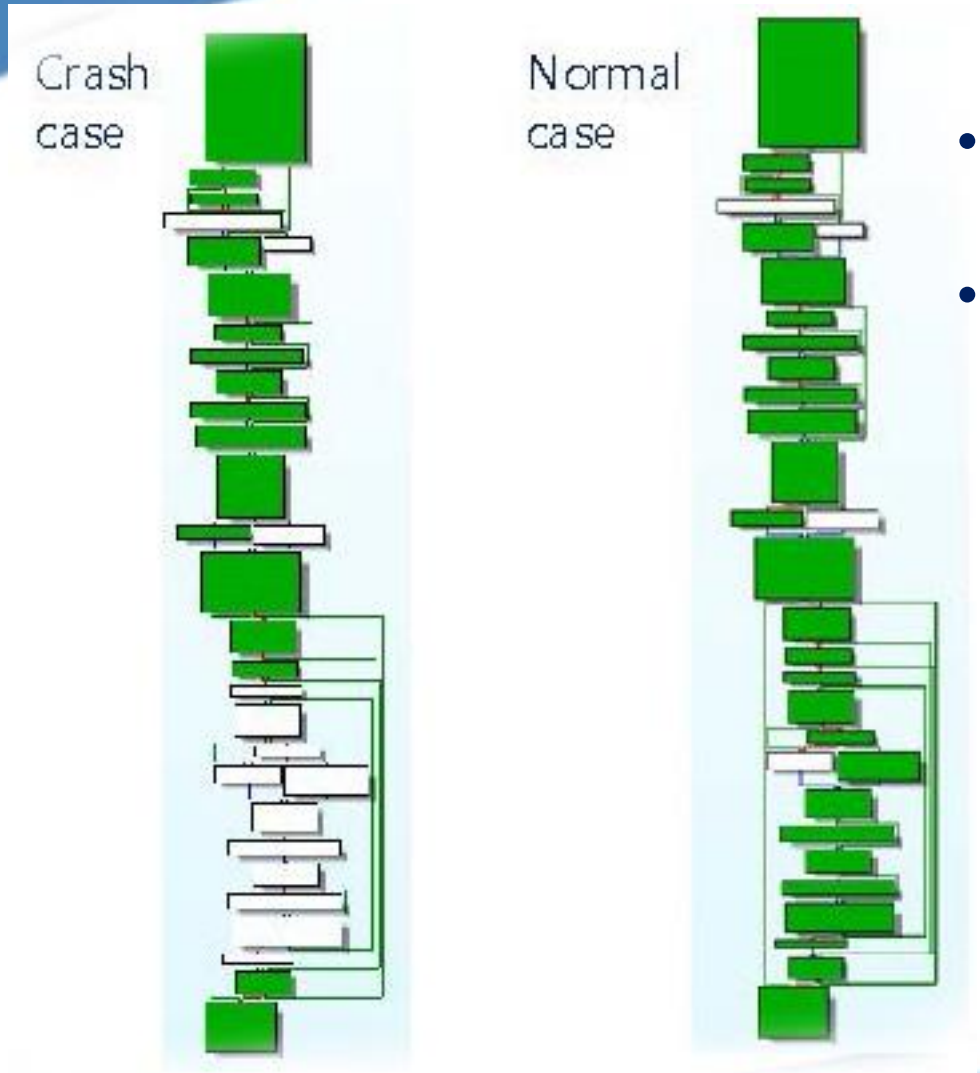
- 现在我们有一个创建数据流差异的关键指令。
- 我们可以使用这个关键指令，执行崩溃与正常文件之间的控制差分分析。
- 此分析的目的是找到正在创建控制流差分的另一个关键指令。

控制流差分分析--从关键指令开始

- 识别具有关键指令（影响数据流）的函数。
- 使用崩溃和正常文件对函数执行控制流分析。
- 推断两种情况下的控制流差异。
- 重复上述操作，直到发现任何公共控制流路径。

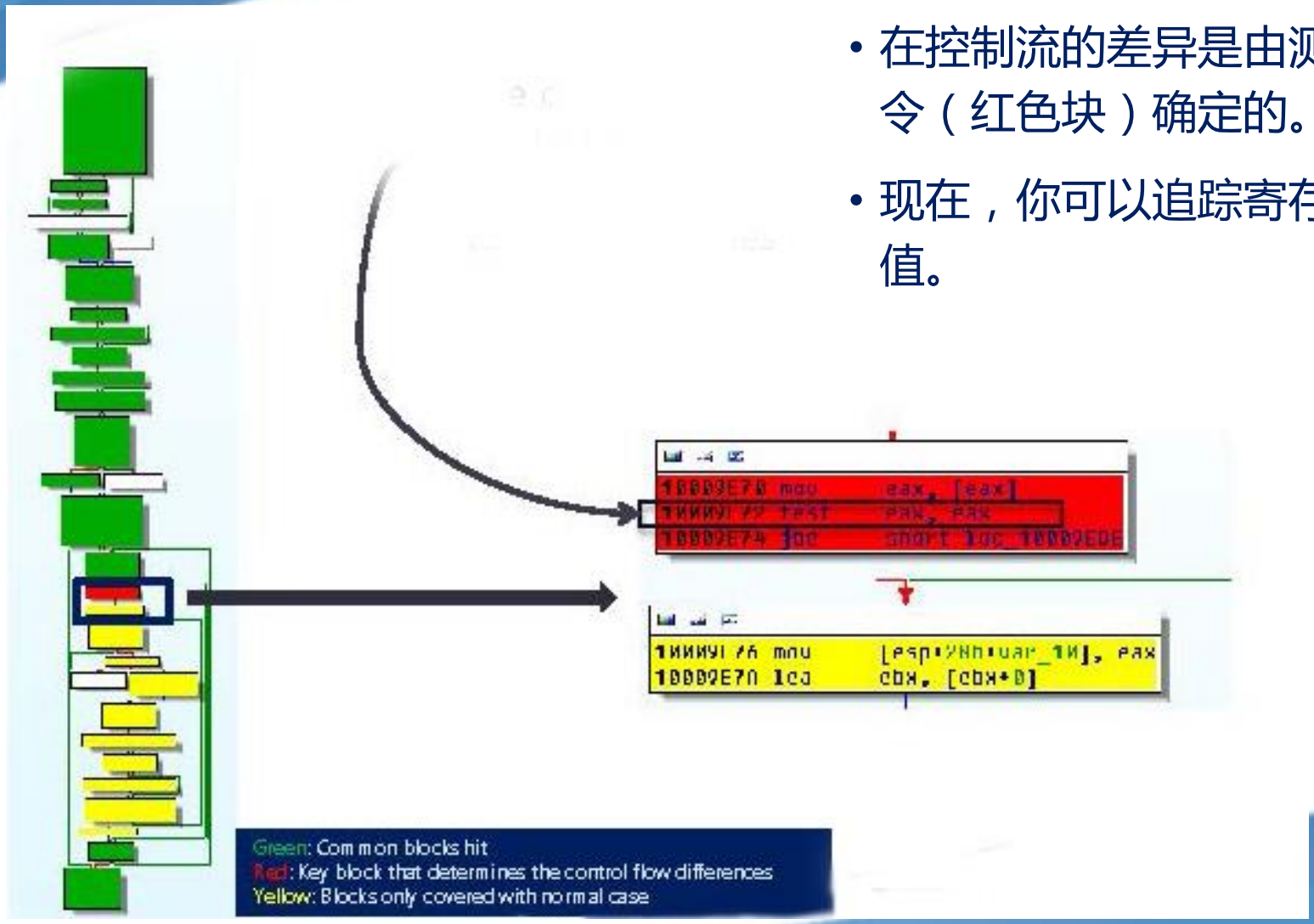


控制差分分析--找出共同指令



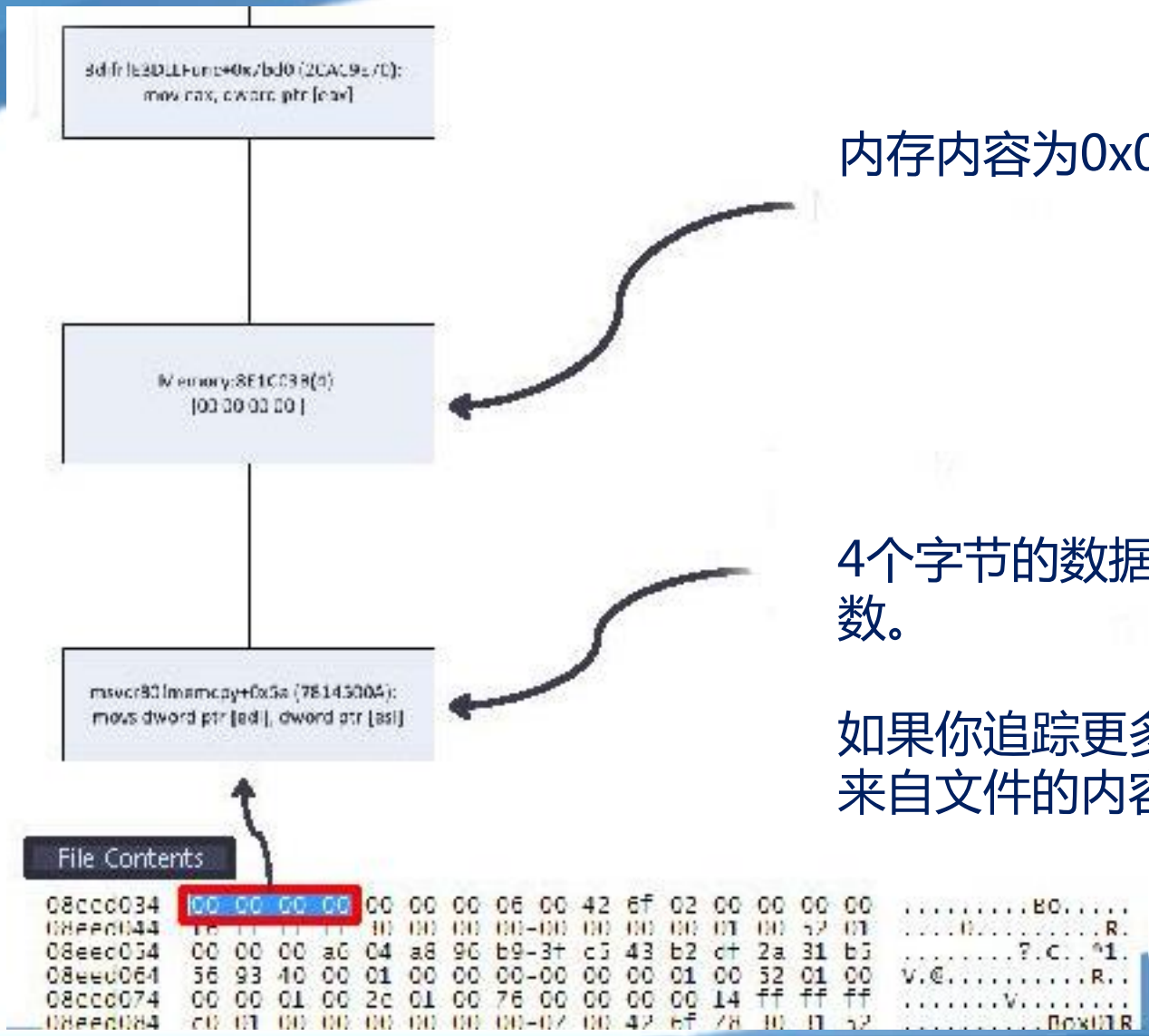
- 我们能够发现函数本身内的公共控制流。
- 如果没有在同一函数中发现，则需要追踪，直到发现公共控制流。

控制差分分析—差异



- 在控制流的差异是由测试指令（红色块）确定的。
- 现在，你可以追踪寄存器eax值。

另一个数据流分析--崩溃情况



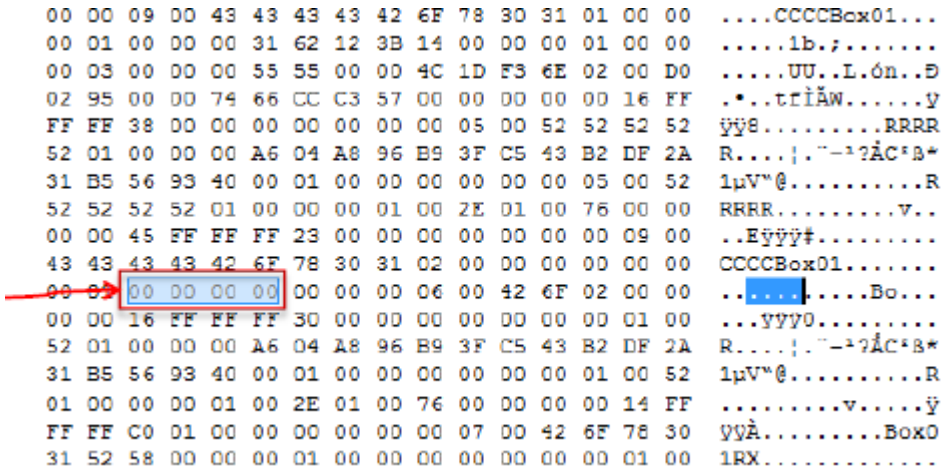
内存内容为0x00000000

4个字节的数据来自memcpy函数。

如果你追踪更多，就可以看到它来自文件的内容。

进一步分析

- 现在，我们可以找出导致该漏洞的数据的确切位置。



- 执行进一步分析
  - 阅读说明文档
  - 编写解析器
  - 对二进制文件中的解析代码执行静态和动态分析

本文的策略总结


控制流
差分分析
找到使数据流出现
差异的关键指令
数据流分析
找到传输给目标
变量的数据源
可视化
数据流
差分分析
发现使数据流中出现差异
的第一个节点

结论

- 本PPT是二进制插桩和指令仿真的例子。
- 详细的策略和方法应根据每种情况使用。
- 一些复杂情况下可以使用这个技术和应用解决。
- 将数据流/控制流与正常情况进行对比，能够使漏洞更加明显。
- 除了传统的数据流分析，我们建议使用数据流差分分析及控制流差分分析的方法来帮助分析漏洞。