计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2016.20150526
53(8):1829-1849,2016

内存地址泄漏分析与防御

傅建明[1,2,3] 刘秀文[1,2] 汤 毅[1,2] 李鹏伟[1,2]
[1](空间信息安全与可信计算教育部重点实验室(武汉大学) 武汉 430072)
[2](武汉大学计算机学院 武汉 430072)
[3](软件工程国家重点实验室(武汉大学) 武汉 430072)
(jmfu@whu.edu.cn)

Survey of Memory Address Leakage and Its Defense

Fu Jianming[1,2,3], Liu Xiuwen[1,2], Tang Yi[1,2], and Li Pengwei[1,2]
[1](*Key Laboratory of Aerospace Information Security and Trusted Computing* (*Wuhan University*), *Ministry of Education*, *Wuhan* 430072)
[2](*School of Computer Science*, *Wuhan University*, *Wuhan* 430072)
[3](*State Key Laboratory of Software Engineering* (*Wuhan University*), *Wuhan* 430072)

Abstract With memory address leakage, an attacker can bypass ALSR (address space layout randomization) mechanism, deploy ROP(return-oriented programming) chains to close the DEP (data execution prevention), and divert the program to execute Shellcode. With regard to memory address leakage, this paper gathers the related information of vulnerability instances, presents the classification of vulnerabilities resulting in memory address leakage based on the procedure of memory leakage. The paper analyzes all kinds of illegal operations of pointer or object which cause the operation of cross-border memory access, as well as side-channel information leakage. In the meantime, this paper divids the defense methods of memory address leakage into four categories according to the procedure of memory corruption attacks, including memory layout randomization, object border protection, object content protection, and the critical address information randomization. And these protections make memory layout vague, memory object unavailable, memory object unreadable and critical memory address untraceable. Finally, this paper points out that we need to provide support of memory layout randomization, fine-grained memory address randomization and object content protection in perspective of programming design, adapting the operating system to establish collaborative defense mechanism in order to build robust defense system in depth.

Key words advanced persistent threat (APT); memory corruption; memory address leakage; address space layout randomization (ASLR); border protection

摘 要 高级持续性威胁(advanced persistent threat, APT)攻击通常会利用内存地址泄漏绕过地址空间布局随机化(address space layout randomization, ASLR)、利用面向返回编程技术(return-oriented programming, ROP)绕过数据执行保护(data execution prevention, DEP).针对内存地址泄漏漏洞,以漏洞实例为样本,剖析了各种造成越界内存访问的指针或对象的非法操作,以及侧信道信息泄漏漏洞,

收稿日期:2015-06-15;修回日期:2015-10-19
基金项目:国家自然科学基金项目(61373168,61202387,61332019);高等学校博士学科点专项科研基金项目(20120141110002)
This work was supported by the National Natural Science Foundation of China (61373168,61202387,61332019) and the Research Fund for the Doctoral Program of Higher Education of China (20120141110002).

并基于造成内存泄漏的过程,给出了相应的漏洞分类.同时,从漏洞利用和攻击的过程出发,总结和归纳了内存布局随机化、内存越界读写保护、内存对象内容保护、内存对象地址随机化等对抗内存地址泄漏的防御方法,从而达到内存布局看不清、内存对象读不到、内存对象内容读不懂、关键内存地址猜不准的保护目的.最后,提出从程序设计角度提供对内存布局随机化、代码地址随机化、内存对象保护等的支持,同时与操作系统建立协作防御机制,从而构建纵深和立体的安全防御体系.

关键词 APT 攻击;内存损坏;内存地址泄漏;地址空间布局随机化;边界保护

中图法分类号 TP391

当访问的对象不同于期望的对象时,会发生内存损坏,例如对象指针越界、对象未初始化、指针指向的对象不存在等.自 20 世纪 70 年代以来,内存损坏一直是软件的重要漏洞之一,在 CERT 和 CVE 漏洞列表中占据前列,如栈溢出、堆溢出、格式化字符串漏洞、NULL 指针漏洞、整数溢出、释放后引用漏洞[1-2]等.内存损坏漏洞一般会导致拒绝服务攻击、信息泄漏、信息窜改、控制劫持等安全威胁,攻击者劫持控制流,使得软件转向攻击者设定的代码(*Shellcode*).攻击者植入的 *Shellcode* 一般存在于栈或者堆中.为了阻止 *Shellcode* 获得执行权,DEP(W^X)技术[3]禁止栈和堆中的数据执行.因此,代码重用技术,如 ret2lib[4],ROP[5]、面向跳转编程技术(jmp-oriented programming, JOP)[6]、面向调用编程技术(call-oriented programming, COP)[7]等,逐渐被用于 *Shellcode* 的编写,这些技术可以绕过 DEP.但这些技术的实施需要获取重用代码所在的内存地址.地址空间布局随机化(address space layout randomization, ASLR)可使得代码在内存中的地址随着模块加载而随机变化,从而降低 *Shellcode* 的执行成功率.因此 ASLR 在 Windows, Android,iOS 等操作系统中得到了部署.如果攻击者可以利用地址泄漏漏洞获得重用代码的内存地址,则可绕过 ASLR.例如,在 2014 年的 APT 攻击事件"雪人行动"中,攻击者利用 CVE-2014-0322 漏洞,修改 ActionScript 脚本中 *vector* 对象的长度,从而泄漏内存地址,绕过 ASLR 防护,最终成功实施了 APT 攻击[8].因此,内存地址泄漏成为实施内存损坏攻击的重要步骤.

APT 攻击可以分为内存地址泄漏、*Shellcode* 注入、控制流劫持、*Shellcode* 的执行和木马加载等步骤,在该攻击的每一步骤都可以实施攻击检测和防御,如 *Shellcode* 的动态检测[9]、控制流劫持的防御[10-11].文献[1]总结了内存损坏攻击的 4 种普遍类型,并列举了其对应的防御方法:

1) 代码损坏攻击,即利用越界指针(索引)、悬挂指针等内存漏洞达到覆写程序代码的目的,其防御方法是代码完整性(code integrity)[12]保护;

2) 控制流劫持攻击,即通过越界读写更改代码指针,使程序跳转到攻击者精心构造的 *Shellcode* 或 Gadget 中执行,其防御方法是代码指针完整性[10]保护、地址空间随机化(ASLR)、控制流完整性(code flow integrity, CFI)[11]保护;

3) 只面向数据的攻击,即恶意修改影响程序执行的关键数据,如分支变量等,来修改程序逻辑,其防御方法是数据完整性,数据流完整性(data flow integrity, DFI)[13];

4) 信息泄漏攻击,其防御方法数据空间随机化(data space randomization, DSR)[14]存在 2 进制代码级的兼容性问题.文献[2]按时间顺序总结了内存错误的发现利用过程,空间内存错误的根源在于通过指针解引用实现越界访问,时间内存错误的根源在于通过指针对已被释放对象再利用.文献[15]针对 C/C++编程语言中存在的代码注入攻击,阐述代码补丁、内存管理、动态污点跟踪、执行路径监控等保护机制.

本文聚焦在 APT 攻击的内存地址泄漏漏洞利用上,从实例出发,分析和总结非法操作引发的地址越界读写、侧信道攻击 2 类地址泄漏漏洞;同时,按照地址泄漏漏洞的攻击流程,从内存布局推测、内存越界读写、读取对象内容、关键地址信息获取 4 个阶段分析和讨论内存地址泄漏的检测和防御.本文的工作有利于较全面解释地址泄漏漏洞的本质,为降低内存信息泄漏带来的危害提供支持,从攻击源头阻止或者抑制 APT 攻击.

1 内存地址泄漏的基本模型

Shellcode 是攻击者向目标进程注入的攻击代码.一般情况下,*Shellcode* 尺寸约几百字节,其功能包括环境的探测、去 DEP、自解密、代码重定位、

获取系统 API 地址、下载并运行(或加载)可执行模块等. 另外,Metasploit[16] 嵌入了 *Shellcode* 的更多功能,如增加用户、DLL 注入、弹框、反向 shell 等. 本文讨论的 *Shellcode* 由各种重用代码的地址链构成,表 1 为一个简单的 *Shellcode*,该 *Shellcode* 略去了数据部分,仅由 kernel32. dll 中代码对应的地址构成,表示创建一个计算器的进程. 如果该 *Shellcode* 位于堆中,则需要增加修改 ESP 的指令(例如 mov esp,eax).

设 $Shellcode=\{a_1,a_2,\cdots,a_n\}$,其中 $a_i(1\leqslant i\leqslant n)$ 表示某重用代码的地址. 表 1 中的 *Shellcode* 表示为 {0x7c83ed85,0x7c9228cb,0x7c83ed85,0x7c96e68f,0x7c86114d,0x7c80e0bf,0x7c81caa2,0x7c9478e4},这里假设 kernel32. dll 的基址为 0x7c800000. 如果 kernel32. dll 加载时其加载的基址随机化为 0x77b70000,则表 1 的 *Shellcode* 无法执行,因为 0x7c83ed85 对应的指令不是"Ret".

Table 1 ROP-based *Shellcode*

表 1 基于 ROP 的 *Shellcode*

Gadgets Address	Corresponding Instructions
0x7c83ed85	ret
0x7c9228cb	pop edi # ret
0x7c83ed85	ret
0x7c96e68f	pop esi # ret
0x7c86114d	kernel32. WinExec
0x7c80e0bf	pop ebp # ret
0x7c81caa2	kernel32. ExitProcess
0x7c9478e4	pushad # ret
"calc. exe";	"calc. exe";

内存泄漏攻击假设在栈或者堆中有一个指针变量 vp,例如数据指针或者函数指针,vp 指向与 *Shellcode* 关联的模块中某个固定地址 Org_ma. Org_ma 表示没有随机化的内存地址值. 当该模块的加载地址随机化后,Org_ma 的值变为 $Aslr\_ma$.

内存地址泄漏是指攻击者利用漏洞探测到 vp 的值 $Aslr\_ma$. vp 随模块随机化加载前后的基址偏移量 $Image\_offset=Aslr\_ma-Org\_ma$. 当前广泛部署的 ASLR 机制只会对各可执行模块的加载基址进行随机化处理,模块内部的代码和数据结构的布局相对固定. 攻击者一旦窥知指针变量 vp 的值 $Aslr\_ma$,根据基址偏移量,可以计算出 vp 所在模块内的各指令序列的加载基址. 表 1 中的所有地址加上 $Image\_offset$ 后就可以正常执行. 因此,内存地址泄漏攻击就是泄漏指向模块空间的指针 vp 的值,$Value(vp)=Aslr\_ma$.

2 内存地址泄漏漏洞

内存地址泄漏的本质是读取特定地址的指针数据,然后利用该数据获得代码的内存地址. 图 1 从主动探测——地址越界读写、被动侦听——侧信道攻击 2 个角度介绍内存地址泄漏攻击. 地址越界读写不仅仅是内存空间上的地址越界访问,更包含逻辑上的越界读写,即在程序的正常执行流中,某个内存区域或者寄存器在程序逻辑的角度上是不应该被访问的或者是攻击者没有权限访问的. 侧信道攻击作为一种通过度量开销信息来间接窥探隐藏信息的攻击方式,与主流的攻击方式有很大不同. 我们无法把侧信道信息与攻击者完全隔离,攻击者也无法从开销信息里推测出完全准确的地址信息.

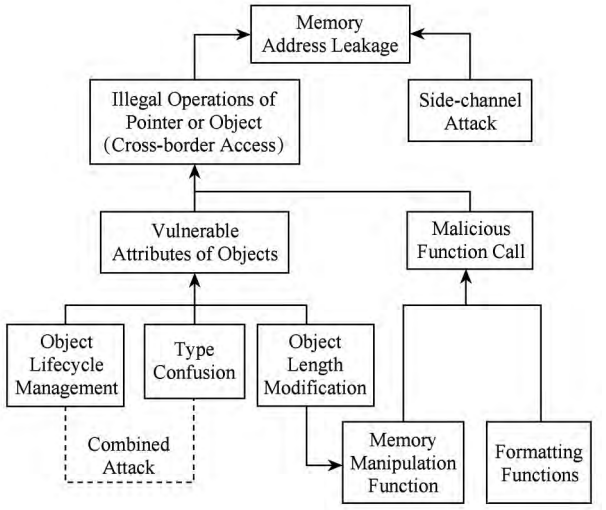

Fig. 1 Classification of memory address leakage vulnerabilities.

图 1 内存地址泄漏漏洞的分类

内存越界读写漏洞源于对指针或对象的非法操作. 根据非法操作能否使被攻击对象本身存在可被利用的脆弱点,把针对指针或对象的非法操作分为 2 类:对象的脆弱属性和函数的恶意利用. 对象的脆弱属性细分为:对象生命周期管理不规范、对象类型属性转换不安全、对象长度属性修改不合理,这类漏洞的直接后果是改变被攻击对象的某类脆弱属性,使攻击者可以利用该脆弱属性达到越界访问的目的;而函数的恶意利用细分为:内存操作函数的恶意利用、格式化函数的恶意利用,这类漏洞作用的被攻击对象本身不存在不合理的脆弱点,攻击者通过对这 2 类函数的恶意利用产生越界复制及内存敏感信息泄漏等后果.

2.1 对象的脆弱属性

对象的脆弱属性主要包括由释放后再引用(use after free, UAF)漏洞引起的对象生命周期管理不规范,由基本类型混淆、复杂类型混淆漏洞造成的对象类型属性转换不安全,由整数溢出漏洞、修改特定数据结构的长度引起的对象长度属性修改不合理这3类攻击方式.

2.1.1 对象生命周期管理

随着安全防御技术的发展和部署,传统的栈溢出和堆溢出漏洞的利用越来越困难.例如,StackGuard[17]和StackShield[18]增加了返回地址的保护,HeapGuard[19]增加了Heap完整性的保护,一般的栈溢出或堆溢出都会触发这些保护机制.UAF漏洞越来越受到攻击者的追逐,2011,2012,2013年分别有138,177,135个UAF漏洞被曝出[20].

C++对象的创建和销毁是对偶操作.对象创建后,会对对象中的成员变量赋值,同时,该对象会被其他指针变量引用.当对象销毁后,对象所占的内存被释放.但是引用该对象的指针可能没有被全部清空,或该对象的内容可能没有被清除,这样会产生3个问题:

1) 没有清空的指针称为悬挂指针[21],因为该指针指向的对象已经被释放,被释放的内存会被其他对象占用,则访问悬挂指针会产生不可预见的后果;

2) 如果该对象释放的空间被攻击者控制,则攻击者可以伪造原对象的函数指针,实现控制流劫持;

3) 如果该对象释放的空间被攻击者控制,攻击者可能会读取原对象的函数指针,会出现信息泄漏.若该函数指针指向模块地址,直接导致内存地址泄漏.

基于链表的堆块分配和释放机制遵循栈式管理,即对于相同大小的堆块,新分配的堆块可能是刚刚释放的相同大小的堆块.这样,当使用悬挂指针访问原堆块所在的空间时,该空间可能已被其他对象覆盖重用,甚至是攻击者精心构造的恶意数据.例如,漏洞CVE-2010-3971中,内存空间的重分配导致引用该旧空间的寄存器被重用.该漏洞在脚本语言中引用CSS文件,对象*CSharedStyleSheet*偏移0x0d8的位置处有一指针,该指针指向*CStyleSheet*元素的数组.当mshtml.dll解析到函数*CSharedStyleSheet*::*Notify*()时,会依次取出数组中的元素.当元素内偏移0x18处的值为1时,程序会调用函数*CStyleSheet*::*Notify*().当发现包含CSS文件且CSS文件含有import命令,程序会在数组中创建新元素(*CStyleSheet*).当元素个数大于当前数组大小时,程序会再分配*CSharedStyleSheet*数组内存空间.之前数组占用的内存空间被释放,程序继续执行并返回到函数*CSharedStyleSheet*::*Notify*()时,存储数组起始地址的寄存器edi仍然放着以前的内存空间的地址,导致了对数组之前所占用的内存空间的UAF漏洞.

CVE-2014-0322利用特定的函数释放空间,但该空间仍旧被引用.其中,攻击者构造一个恶意的网页,该网页包含一个"script"元素和"select"元素.当把"select"元素添加到"script"元素上时,函数*MSHTML*!*CElement*::*Var_appendChild*()被调用,*CMarkup*对象会被创建,随后onpropertychange事件被触发.该事件处理过程中,代码"*this.outerHTML* = *this.outerHTML*"会人为地释放*CMarkup*,而*CMarkup*对象仍被其他变量引用.如果攻击者构造好数据填充到释放的空间,则会造成严重的后果[22].

在释放对象没有被重用的情况下,也可以利用UAF漏洞完成恶意代码的执行[23].

2.1.2 对象类型属性转换

攻击者对被攻击对象的类型属性进行非法转换,达到内存越界访问的目的,即类型混淆漏洞.根据被攻击对象的不同,对象类型属性转换细分为基本类型混淆漏洞、复杂类型混淆漏洞.基本类型是指编译器规定的变量类型,而复杂类型则是由基本类型和复杂类型构建的数据结构.

1) 基本类型混淆漏洞

不同类型的指针所指对象类型不同,当不同类型的变量相互转换时,产生类型混淆漏洞.如果把指向短字节数据类型的指针强制转换为指向长字节数据类型的指针,则通过对该指针解引用访问到超过原指向对象边界范围的内存,从而产生函数或者数据的指针泄漏[1].例如,把char*转化为int*的实例时,int类型在内存中占用4 B,而char类型在内存中只占用1 B,当把指向char类型的指针强制转换为指向int类型的指针,并对该指针解引用时,编译器会误认为其指向对象为int类型,连续读取4B的内容,从而造成内存越界访问,产生地址泄漏.

文献[24]利用指针强制类型转换获得一个地址的整数值,可以利用该整数值获得内存地址的布局.攻击者首先创建一个object,并创建一个指向该对象的指针*thisObject*.接着通过强制类型转换int *address*=*(int*)*thisObject*;把指针*thisObject*指向

的对象的地址值拷贝到整形变量 *address* 中.然后,攻击者可以通过获取特定对象的起始地址来推断可执行模块的加载基址,因为某些可执行模块中的特定对象位置相对模块起始地址是固定的.

2) 复杂类型混淆漏洞

面向对象程序设计语言,如 C++和 Java 语言,可以设计各种复杂的数据结构和数据类型.这些数据类型的相互转换为数据访问提供了便利,但不恰当的转换会带来安全隐患,例如类型混淆(type confusion)[25].

C++提供的 4 种类型转换操作符的应用场合和潜在安全问题如表 2 所示:

Table 2 C++ Type Conversion

表 2 C++的 4 种类型转换

Casting Operators	Description	Potential Security Issues
const_cast	Cast away the "const-ness" or "volatile-ness" of a type.	A const object *d* can be pointed to by a non-const pointer *f*. *d* may be modified through this pointer.
static_cast	Converts between types using a combination of implicit and user-defined conversions. Convert a base class pointer to a derived class pointer, perform arithmetic conversions, etc.	The downcast of a base class pointer to a derived class pointer can be done statically. *Static_cast* relies on static (compile-time) type information and does not perform any run-time type checking.
dynamic_cast	Cast from a derived class pointer to a base class pointer, or cast a base class pointer to a derived class pointer. Each of these conversions may also be applied to references.	Conversions down the hierarchy from base to derived or across a class hierarchy rely on run-time type information.
reinterpret_cast	Provide reinterpretation of operand bit and perform conversions between two unrelated types. The result of the conversion is usually implementation dependent.	Misuse of *reinterpret_cast* operator leads to unsafe procedures easily. There is a strong platform dependence for the poor portability of this underlying conversion.

因为 *static_cast* 在作下行类型转换时,没有动态类型检查,会引发安全问题.同时,*static_cast* 可以实现把任何类型的表达式转换成 void 类型,这种转换可以绕过编译器对指针所指对象的类型检查,便于攻击者实施攻击.静态转换 *static_cast* 应用在 C++对象的虚表指针中会造成 vtable escape[26]错误.

图 2 为 vtable escape 的实例,其中父类 B 有 2 个虚函数,子类 D 有 3 个虚函数(其中继承父类 B 的 2 个虚函数).指向父类 B 的指针被强制转换(*static_cast*)为指向子类 D,但其所指向的内存地址范围并没有变化.当强制引用函数 $h(\cdot)$ 时,该函数指针并没有在基类对象的虚函数表中,从而造成内存越界操作.

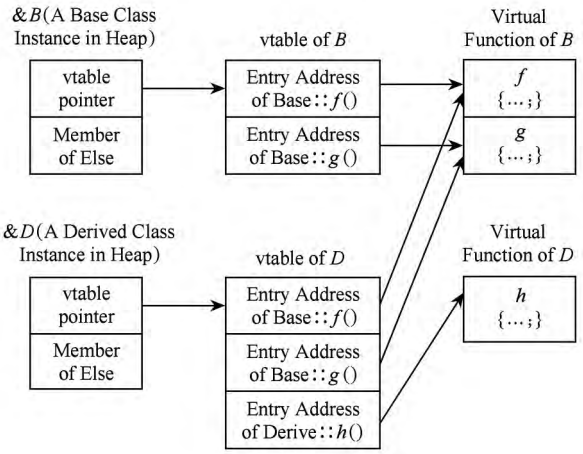

Fig. 2 A case of vtable escape.

图 2 vtable escape 实例

在 2013 年的 Pwn2Own 黑客大赛中,MWR Labs 研究人员利用 Webkit 内核的类型混淆漏洞 CVE-2013-2839 攻陷了 Chrome[25].Webkit 的内核 C++代码实现了把 HTML 的 DOM 节点转换为对应标签对象的结构,再通过这些对象去绘制网页.把 DOM 结构(如 input,img,able 等)转换为网页对应标签对象的类结构时,必须满足以下条件:1)拥有合理的标签名称(如 div,table,img 等);2)拥有合理的命名空间或者访问空间.只有同时拥有合理的 XML DOM 的 local 属性和合理的 XML DOM 的 namespace URI 属性,才可以被转换为 HTML 的对应标签对象的类结构.

漏洞 CVE-2013-2839 是类型转换漏洞在剪切和书签服务中的应用.Clipboard 是一个剪切和书签服务,其实现代码在处理可拖动图像的过程中,指定拖动图像被强制转换为 *HTMLImageElement* 对象,该过程中没有执行适当的检查,以确保它是一个 HTML 图像元素.其漏洞触发代码如例 1 所示:

例 1. Type confusion 1.

```
if (toElement(node)→hasLocalName
    (HTMLNames::imgTag))
clipboard→setDragImage(static_cast<
    HTMLImageElement*>(node));
```

从例1可以看出,在对DOM节点进行标签类型检查时,只检查了其标签名称,并没有检查其命名空间.这就会产生不合法的HTML图像元素(比如把SVG元素转换为HTML的对应图像的类结构).而*HTMLImageElement*对象的虚表占用的空间比*SVGElement*大得多,SVG元素相邻的存储器空间被当做*HTMLImageElement*虚表的一部分被占用.因此,类型混淆漏洞就造成了越界内存的访问.如果越界的内存区域保存有指向模块的指针,则该类漏洞会产生内存地址泄漏.

漏洞CVE-2013-0912同样利用了类型混淆漏洞造成了越界内存访问.SVG是用XML定义的,用来描述二维矢量及矢量/栅格图形.其中,SVG文档中的任何元素一般是SVG元素,但其扩展性模块*ForeignObject*内可以嵌入非SVG元素.CVE-2013-0912的POC代码在创建的HTML页面中引用了SVG文件.而该SVG文件的扩展性模块中嵌入了一个无效元素,被Webkit解读为*HTMLUnknownElement*,即非SVG元素.通过把该非SVG元素转换成SVG元素,造成内存越界访问.其对应的触发漏洞的Webkit内核代码语句如例2所示:

例2. Type confusion 2.

```
if ( !m_contextElement)    return 0;
return static_cast<SVGElement*>
  (m_contextElement->treeScope()->
  getElementById(m_viewTargetString));
```

该SVG文件被Webkit视为DOM树,Webkit根据*viewTarget*属性加载SVG文档的view元素,相当于根据ID属性选择DOM树的节点.从例2代码可看出,Webkit内核将SVG文档中ID为*m_viewTargetString*的元素强制转换为SVG元素.CVE-2013-0912的POC代码通过堆喷射进行内存布局,*HTMLUnknownElement*被安插的位置紧挨着*HTMLDivElement*元素,当类型混淆漏洞被触发后,*HTMLUnknownElement*被视为SVG元素,SVG元素的大小为0x124B,远远大于*HTMLUnknownElement*的0x40B大小.攻击者可以通过访问*HTMLUnknownElement*,越界访问其相邻的*HTMLDivElement*元素的虚表指针.这就通过类型混淆漏洞泄漏了关键的内存地址信息.

2.1.3 对象长度属性修改

所有对象和数组都有长度约束,如堆块块首包含长度域、数组的索引下标.如果攻击者修改数据结构的长度约束,则可以读取给定地址的数据,达到内存地址泄漏的目的.该类漏洞包括整数溢出、修改特定数据结构的长度等.

2014年曝出的心脏出血漏洞(CVE-2014-0160)根源在于OpenSSL密码库处理心跳请求包时,对其长度属性值缺少真实性验证.TLS心跳包是在客户端和服务器间定时通知对方自己状态的一个请求包.TLS心跳请求包中包括有效载荷,*payload*是有效载荷的期望长度.服务器把有效载荷从请求包复制到响应包,忽略了对请求包大小的边界检查,即忽略了*payload*值的真实性.如果客户端的有效载荷达不到*payload*,接收端会把请求包存储位置之后的任意数据,也当成有效载荷装进响应包里,返回给发送方,从而造成服务器的敏感信息泄漏.

1) 整数溢出漏洞

整数分为无符号整数(unsigned int)和有符号整数(int).当2个整数相加或者相减时,可能出现上溢或者下溢,得到非期望的数据[1].或者,当把一个有符号的整数转换为无符号整数时,有可能到一个非期望的大整数[1].整数*a*(0xfffffff0)和整数*b*(0x12)相加导致数据上溢,得到一个比期望小很多的整数0x2,而程序仍按照期望的长度(0xfffffff0)访问数据,导致内存访问越界.如果越界区域可以覆盖一个部署好的函数或者数据指针,则发生内存地址泄漏.

CVE-2010-0010是一个整数溢出漏洞:当Apache服务器处理Web用户提交的分块数据的长度时,忽略了整数的符号问题,造成因整数溢出引发的堆溢出.Web用户以分块编码的方式向Apache服务器提交数据,数据的长度是一个有符号整数,服务器会为该数据分配一个缓冲区.在把数据拷贝到缓冲区之前,Apache会把用户提交的分块编码的长度与缓冲区长度进行比较,防止拷贝数据越界.攻击者可以把提交的数据长度设定为负值,绕过该安全检查.实际上攻击者提交的数据长度可以大于0x80000000B,造成缓冲区溢出,覆盖相邻位置的敏感信息(如数组索引),从而导致信息泄漏.

2) 修改特定数据结构的长度

为便于管理,堆中的数据块由数据的长度和数据内容2部分构成.如果攻击者能够覆盖数据的长度,则可以实现内存越界访问,引发信息泄漏.文献[2]给出了该类攻击,随后操作系统加强了对堆的管理,如引入heap cookie、元数据加密等安全机制,增加了攻击者篡改堆块数据长度的难度.

脚本语言,如 VBScript,JavaScript,ActionScript 等,为网页、文档、媒体等提供了丰富的交互性、灵活性和可扩展性. VBString 就是一种包含长度和数据内容的结构类型,CVE-2012-1876 漏洞就是攻击该数据结构的长度域,越界访问指向模块的指针. 该漏洞的原理在于可以实现一个任意地址的任意写漏洞. 例 3 为该漏洞的一个示例:

例 3. Arbitrary address written vulnerability.

```
〈table style="table-layout:fixed"〉
〈col id ="132" width="41" span="1"〉
   〈/col〉〈/table〉
〈script〉
function over_trigger() {
var obj_col = document.getElementById
  ("132");
obj_col.width="42765";
obj_col.span = 1000;
}
setTimeout("over_trigger();",15000);
〈/script〉
```

例 3 首先生成一个 *id* ="132"的 *CTablelayout* 对象,利用 *width* 和 *span* 的值改写对象的内存,其中 *width* 控制改写的值,*span* 控制改写的地址及改写值的写入次数. 随后执行函数 *over_trigger*(),由于没有把新引入的 *span* 值保存,则写入内存空间的大小不变,而写入次数变为 1 000,导致写入空间后紧邻的内存空间被改写. 攻击者控制好 *width* 和 *span* 的值,在写入空间后面部署 VBString,就可以改写 VBString 的长度域. 如果 *CButtonLayout* 对象被部署在 VBString 之后,因为 VBString 的访问空间变大,就可以读取 *CButtonLayout* 对象的虚表指针. 由于该指针指向 mshtml.dll 中的固定偏移量,攻击者可由该指针值推测 mashtml.dll 的加载基址.

CVE-2014-0322 在产生 UAF 漏洞后,结合 ActionScript 3.0 的 *vector* 属性,可以实现任意地址的数值加 1 的操作. *vector* 是一种经常使用的数据结构,包含一个 8 B 的头部,其中前 4 B 表示 *vector* 的长度字段 *size*. 头部后为存储数据的空间,数据元素个数由 *size* 决定,每个元素大小为 4 B. 如果 V_1 和 V_2 是连续存放的 *vector*,如(eax + 10h)指向 $V_1.size$,则执行 inc dword ptr[eax + 10h]后,$V_1.size$ 为0x03ff. 随后利用 ActionScript 的脚本直接修改 $V_2.size=V_1[0x03fe]=N$. N 为足够大的正整数,如 0x7fffffff,利用 $V_2[x]$就可以访问 V_2 后的任意地址空间.

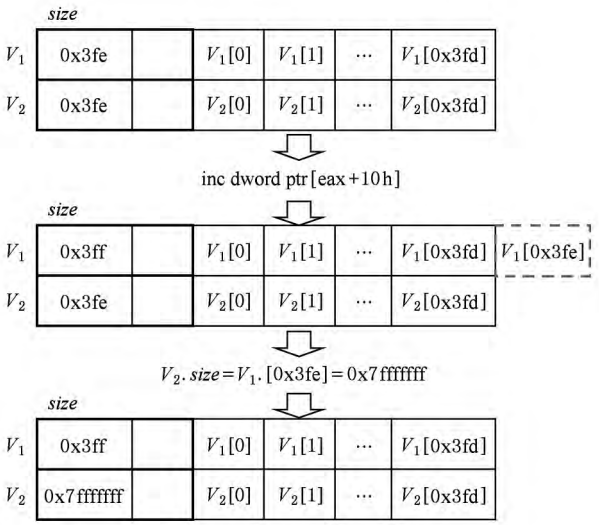

Fig. 3 The length field of *vector* modification in CVE-2014-0322.

图 3 利用 CVE-2014-0322 修改 *vector* 长度

包含长度属性的数据结构还有 ArrayBuffer, Array, Int8Array, Uint8Array, Int16Array, Uint16Array, Int32Array, Uint32Array, DataView 等. 攻击者通过修改这些特定数据结构的长度字段实现任意地址的读写.

2.2 函数的恶意利用

对于不存在敏感属性被攻击者利用的被攻击对象,恶意利用内存操作函数(*sprintf*, *strcpy*, *strncpy*, *strncat*, *memcpy*, *memset* 等)会导致越界复制数据;恶意利用格式化函数(*fprintf*, *printf*, *sprintf*, *snprintf*, *vfprintf*)则会偷窥栈内敏感信息,最终引发函数或者数据指针的泄漏[27].

2.2.1 内存操作函数的恶意利用

文献[28]给出了一个覆盖邻接内存变量,产生信息泄漏或数据改写的实例. 该实例类似于数组赋值中常见的 off-by-one 漏洞. *strncpy*(·)只按照给定长度拷贝的字符串,拷贝后不会自动添加 NULL. 该实例中,客户端要与服务器端建立起联系,必须先提交其用户名和密码. 服务器把客户端发来的用户名存储在数组 *buff* 中,用 *strncpy*(·)将其拷贝到 *user*[100]中. 如果数组 *buff* 中的字符串大于等于 100 B,*user*[100]中不存在 NULL 结束符. 但在调用函数 *snprintf*(·)的过程中,程序会从数组 *user* 的源地址开始以指定格式复制字符串到数组 *buff* 中,直到遇到 NULL 结束符为止. 这样会泄漏

security cookie、前栈帧 ebp、返回地址等堆栈敏感信息.

CVE-2007-3410 漏洞就是利用函数 *strncpy* 造成栈缓冲区溢出,导致了任意指令的执行.该漏洞存在于 RealPlayer/HelixPlayer 播放器的墙壁时钟(wallclock),该功能在处理 hh:mm:ss:ff 时间格式时,忽略了 *strncpy*(*buf*, *pos*, *len*)代码中 *buf* 缓冲区的常数长度.攻击者对该漏洞造成的栈溢出加以利用,会泄漏栈内的敏感信息.

2.2.2 格式化函数的恶意利用(format string)

常见的格式化函数有 *fprintf*, *printf*, *sprintf*, *snprintf*, *vfprintf*, *vprintf*, *vsprintf*, *vsnprintf* 等.这些格式化函数可以把变量内容按照指定的格式输出到给定的缓冲区.如果攻击者可以控制格式化函数中的输出格式、输入的地址、输出的缓冲区,则会造成严重的格式化漏洞.格式化字符串漏洞就是指攻击者可以控制输出格式,即在调用格式化的输出函数时,没有指定要输出的字符串的输出格式.这样,函数就会在要逐字打印的字符串中寻找特殊的格式字符.

如果要输出打印的字符包含"%n",则能写任意值到攻击者选定的内存中.因为格式规定符"%n"把前面已经打印的字符长度写入其对应的实参变量中.该攻击利用巧妙的布局,可以实现修改程序的返回地址,进而取得程序的控制权.该输出格式在目前的编译器中被缺省关闭[29].

如果要输出的字符串为"%x%x%x%x%x%x%x%x"或"%p%p%p%p%p%p%p%p"[30],但输出函数并没有指定其输出的的格式规定符.攻击者可以利用这点窥知堆栈中的内容并以 16 进制的形式输出.文献[31]给出了利用格式化字符串漏洞来偷窥程序内容,甚至关键函数的返回地址的方法.

2.3 侧信道攻击

侧信道攻击利用侧信道信息(如时间开销、空间开销或者功耗开销)推测出对攻击者有用的信息,如加密的密钥、安全保护措施或者内存地址信息等.

内核空间 ASLR 能有效阻碍对操作系统内核或驱动程序的本地攻击.文献[32]通过针对内存管理系统的通用的侧信道攻击方法,突破了 ASLR 保护机制的限制,推断出程序的地址空间布局.文献[32]中列举了 3 种方法:高速缓存探查(cache probing)、检测发生 2 次页错误的时间差(double page fault, DPF)、预加载地址转换缓存(address translation cache preloading),这些突破内核 ASLR 的方法能定位内核空间到页的粒度.一个用户模式的应用程序不能直接访问内核空间,每次在用户模式下尝试访问内核空间内存时会导致一个访问冲突.以 DPF 为例,从用户模式访问内存页 p,会导致页错误并传递到进程的异常处理.如果 p 被加载过,即 p 是已经分配物理内存的页面,第 1 次从用户模式访问 p 时,旁路转换缓冲(translation lookaside buffer, TLB)会缓存该页,因此在第 2 次访问页面 p 时,页错误的传递时间会相对较短.如果 p 没有被加载过,则不会被 TLB 缓存,2 次页错误的传递时间无明显差别,所以可以根据页错误的传递时间估计该页面是否加载过.DPF 方法对内核空间的页面分配和驱动程序的映射给出一个大致的估计,可以定位到页面的范围,但准确定位一个驱动程序的基址还是比较困难的.

模块级地址随机化如 ASLR、指令级地址随机化(instruction location randomization, ILR)[33]、基本指令块(连续的指令序列)地址随机化(self-transforming instruction relocation, STIR)[34],在代码中随机加入 NOP 指令模糊 Gadget 定位如 librando[35],这些代码多样化技术混淆了执行代码的内存布局,但文献[36]提出的故障分析攻击及计时侧信道攻击在利用内存损坏攻击越界覆写后,能识别多样化代码中的部分 NOP 指令,Gadget 甚至函数的布局.故障分析攻击比如攻击者覆写数据指针索引,并指向代码段,如果指针解引用值为 0x00,攻击者则收到报错信息.借此定位字节 0x00,而 0x00 一般分布在代码段的常量中,甚至可作为定位 Gadget 或者函数的明显特征.同样改写数据指针索引,在计时攻击中,攻击者通过构造不同输入,来猜测其指向的栈中关键地址值.如下列分支:

```
1) if (ptr[index] % input==0)
2)     i=i×2;
3) else
4)     i=i+2.
```

分支 2)明显比分支 4)执行时间长,借此推断输入值与修改后的指针解引用值(如返回地址)的关系.但故障分析攻击及计时攻击泄漏的信息很大程度上依赖于攻击者构造值与其收到回应之间映射关系的唯一性,且可利用内存损坏漏洞造成的指针覆写须直接影响控制流.

3 内存地址泄漏的防御

内存损坏攻击主要包含 4 个步骤:

1) 内存布局推测. 不管是注入代码攻击,还是ROP,JOP,COP等代码重用攻击,攻击者成功攻击的前提条件是能够推测出被攻击对象的内存布局. 如果攻击者无法预知被攻击对象(如函数指针、虚表指针、函数返回地址、可重用指令序列等)的相关地址信息,则无法达到劫持啊程序控制流,执行恶意代码的目的.

2) 内存越界读写. 在攻击者事先获知被攻击对象的内存地址信息的情况下,攻击者可以控制的进程对象(如与外部输入相关的数组等)与被攻击对象在内存布局上还存在一定差距,攻击者无法直接访问这些被攻击对象的信息,只能利用第2节的漏洞达到越界读写的目的.

3) 读取对象内容. 如果攻击者达到越界读写甚至访问整个进程内存空间的目的,防御方则需要对被攻击对象的内容进行加密. 这样,攻击者越界访问到的敏感信息只是加密后的信息,无法获取真实值.

4) 关键地址信息获取. 在攻击者已经成功获取泄漏点——被攻击对象的真实信息(可执行模块中的函数指针、数据指针、虚表指针)的情况下,执行恶意代码还需要攻击者获取其他关键的地址信息,比如可重用代码的实际地址.

防御方在攻击者已经成功获取切入点的情况下,仍然可以通过设置代码页不可读权限阻止大规模的读代码行为,或者通过细粒度的地址空间随机化技术妨碍其窥知攻击成功所需的其他敏感信息,大大降低内存损坏攻击的成功率. 如果攻击者突破了以上4层防护,已获取实施攻击所需的地址信息,构造攻击所重用的Gadget,实施代码重用攻击.

如图4所示,针对内存损坏攻击的4个步骤,主要有如下防御方法:1)对被攻击对象的相关内存布局进行随机化处理,增大攻击者猜测被攻击对象

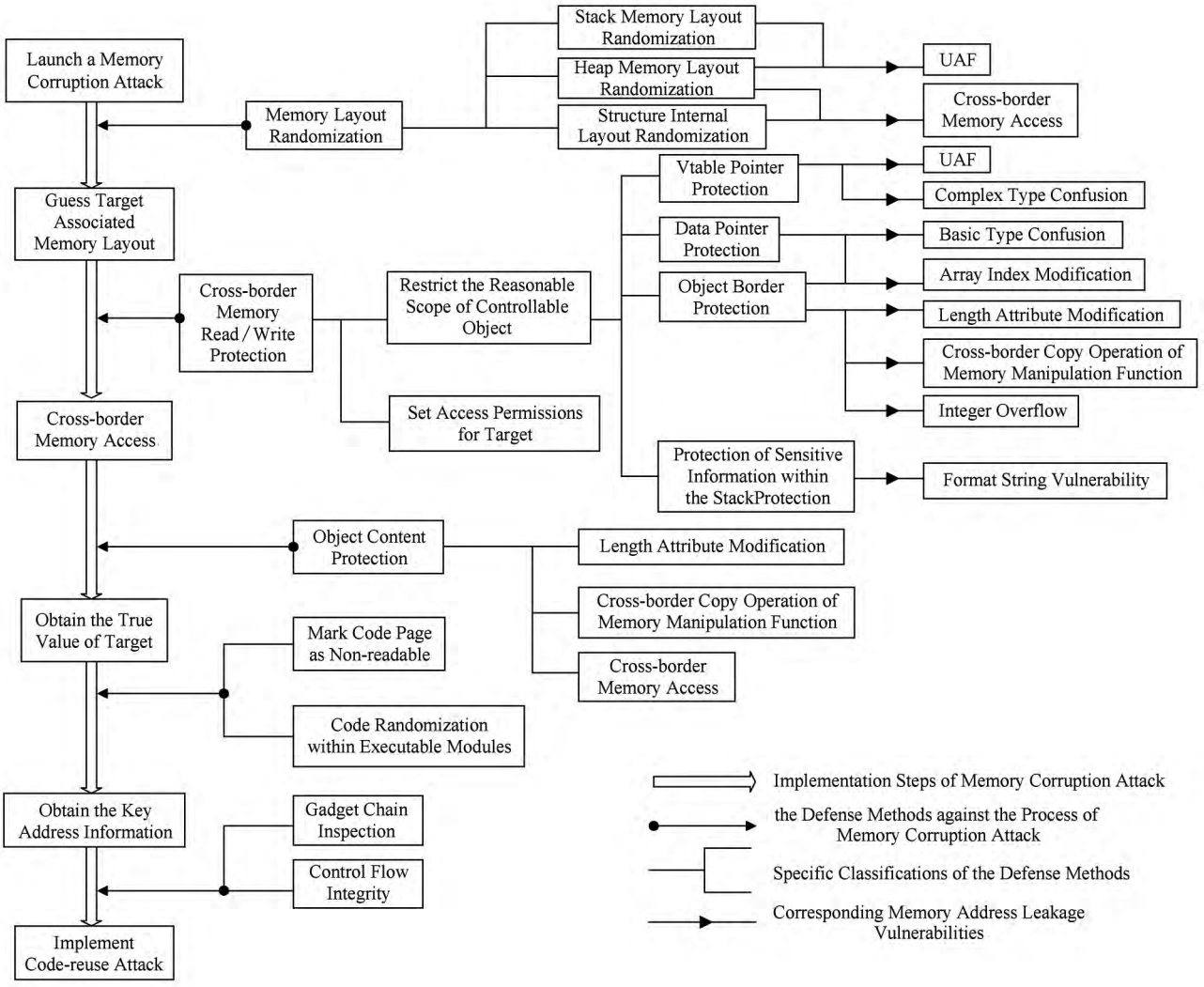

Fig. 4 Steps and defensive approach of memory corruption attack.

图4 内存损坏攻击的步骤及防御方法

地址信息的难度;2)对被攻击对象提供边界保护,防止攻击者越界访问敏感信息;3)对被攻击对象内容的真实值进行加密,混淆攻击者获取的敏感信息;4)设置代码页不可读属性或部署细粒度的地址空间随机化技术,降低切入点泄漏带来的影响.即使面对潜在的代码重用攻击,可以从检测 Gadget 链,控制流完整性 2 个方面及时检测甚至阻止攻击.

本节的防御方法为内存敏感信息(如进程对象的内存基址、指令序列的内存基址等)、虚表指针、数据指针、代码指针、由指针(索引)访问的对象、栈内的敏感信息等被攻击对象提供保护.

3.1 内存布局随机化

攻击者成功实施攻击的前提是猜测出被攻击对象的内存布局,比如函数指针或函数返回地址在栈内存的布局是遵循一定规则排列的,攻击者可以根据先验知识,利用与被攻击对象相邻的数组或缓冲区,间接修改函数指针或返回地址,成功实施攻击.如果在栈内存、堆内存、静态缓冲区中的进程对象分布不可预知,则攻击者无法获知被攻击对象的相关内存布局,也无法成功发起攻击.本节分别从栈内存、堆内存的内存布局随机化和结构体内部布局随机化混淆内存布局.

3.1.1 栈内存布局随机化

攻击者利用基本栈的缓冲区溢出打破栈内对象边界,越界读取敏感信息,甚至篡改控制数据.而传统的调用栈规则为栈帧及栈内对象的分布提供了可预测性,使栈内对象 UAF 及未初始化读取错误成为可能.

文献[37]提出的基于栈的内存错误保护——StackArmor——有效混淆了栈内存布局. StackArmor 提供的随机栈帧分配器为每个线程调用栈维持多个连续的、由不被映射的 Page Guard 页环绕的物理栈帧,并在虚拟地址空间中为逻辑栈帧预定对应的映射项,而映射项对应的物理栈帧序列随机.这样,逻辑上相邻的栈帧其物理分布不可预测.同时,该分配器在设置新栈帧时,会把刚释放的栈帧映射项与其他任意一个映射项交换,提高了栈帧分配选择的随机性. StackArmor 通过 2 进制级插桩调用该分配器为静态分析时筛选出的栈帧及缓冲区提供保护.涉及到局部变量指针计算的函数栈帧,是潜在的被攻击对象,程序在该函数调用点为其随机分配并切换到新栈帧,并在调用结束后返回到原始堆栈继续执行.作为攻击者的可控栈内对象,如果某个缓冲区的所有引用不用作访问其他内存对象,则可将其独立放置到随机分配的新栈帧中.被隔离的函数栈帧和缓冲区使越界读写完全失效. StackArmor 结合了静态分析和插桩的策略,在利用新栈帧隔离可控对象及被攻击对象的基础上,随机化处理新栈帧的分配布局,攻击者无法通过栈内可控对象推测被攻击对象的布局.

3.1.2 堆内存布局随机化

基于链表的堆块分配和释放机制是遵循栈式管理的,攻击者可以利用该特性明确内存布局,这为通过越界内存访问获取敏感地址信息提供了很大便利.

在 CVE-2012-1876 漏洞利用中,攻击者首先通过脚本申请 16 个 Button 对象,每个 Button 对象大小为 0x100,同时设置每个 Button 对象的 *ClassName*, *Title*(长度均为 0x100 的字符串对象).接着释放掉 16 个 *ClassName* 占用的 0x100 大小的空间,这样将会在刚申请的内存空间上留下 16 个"小孔".然后,攻击者申请 0x100 大小的 Spanstructure,希望 Spanstructure 落在 16 个小孔中的一个.继续释放 Button 对象的 *Title*,同时大量申请长度为 0x100 的 VBString,将刚释放掉的堆空间填充.最后,利用该漏洞修改 VBString 的长度,实现对给定地址的读操作,达到内存信息(地址)泄漏的目的.

在遵循栈式管理的内存分配机制下,攻击者利用 UAF 及 2 次释放漏洞精确控制被攻击对象相邻的内存布局.针对 UAF 漏洞根源——悬挂指针解引用, Undangle[38] 和 FreeSentry[39] 在程序运行时插入动态检查,起到一定程度的防御作用. Undangle[38] 利用动态污点分析工具跟踪程序在指定输入下的执行轨迹,其核心 Early Detection 技术从执行轨迹中识别悬挂指针的创建点,跟踪悬挂指针在执行流中的传播,直到再次使用该悬挂指针,从而及时检测出 UAF 及 2 次释放漏洞. FreeSentry 对指针创建及指针指向对象修改操作插入自定义的函数 *regptr*(),记录指针地址,引用对象的边界等信息,当捕获到释放内存函数时,将引用对应内存块的所有悬挂指针指向无效内存地址. FreeSentry 可扩展为边界检查.而 DANGNULL[40] 则在 LLVM 的中间代码级进行静态插桩,识别指针分配操作并插入轨迹函数,在运行时建立指针和内存对象之间的引用关系,并追踪堆内存分配、释放函数,及时发现悬挂指针并使其失效,从根本上阻止悬挂指针解引用,但废弃不被解引用的良性悬挂指针会带来额外开销.

如果增加相同大小的堆块在内存分布的随机性,突破栈式管理中内存分配释放的特性,那么攻击者部署2个邻接对象,精确对象的内存布局的难度就会大大增加.文献[41]中提出了一种基于虚拟簇的堆块分配机制.每次分配大小为 A 的对象时,从虚拟簇中大小为 A 的堆块和2倍大小为 A 的堆块中分配一块.因此,该机制扩大堆块待分配的空间和打乱分配顺序,以此提高堆块分配的随机性,降低释放后被分配的概率.这样,攻击者越界访问的对象并不可控.DieHard[42]是随机化的内存管理器,应用程序对应的堆空间大小是其所需最大空间的 M 倍($M\geqslant 2$).程序所请求堆块对应的堆区域(相同大小的堆块区域)只允许至多填满 $1/M$ 的空间,且刚刚释放的堆块与同堆区域内的空闲堆块有相同概率被下次分配操作选中.DieHarder[43]的分散页布局参考了OpenBSD的内存分配机制,利用系统调用 *mmap*() 实现进程对象到内存之间的分散页映射,即内存页面不连续.分散页之间的空隙是未被映射的内存,该内存作为Page Guard预防内存的越界访问.

上述内存分配的随机化增加了内存布局的不确定性.当然,该随机化需要修改内存的分配机制,增加了空闲的内存空间,同时也增加了内存分配的开销.

3.1.3 结构体内部布局随机化

结构体是攻击者劫持程序执行流的主要利用目标.在缓冲区溢出漏洞中,攻击者通过越界覆写栈中与用户输入相关的数组,修改函数的返回地址.在包含有函数指针的结构体中,攻击者控制与函数指针相邻的外部输入变量,篡改函数指针.

文献[44]中提出的基于封装结构的随机化方法在一定程度上缓解了上述攻击过程.这种随机化方法利用了编译器GCC,在源程序中的函数、结构体以及类这些数据结构的抽象语法树(AST)生成以后,随机更改其内部数据成员的链表串接顺序.随机化策略是:1)在靠近被攻击对象(函数返回地址及函数指针)的相对低地址处布置"哨兵"(Guard区域);2)在与用户输入相关的数组之间布置"哨兵";3)分别随机化处理与用户输入相关的数组、与输入无关的数组、其他变量(包括函数指针)这3类数据结构的内存排列.这样,通过在运行时检查Guard区域是否被覆写,及时检测出缓冲区溢出.同时,这种结构体内部随机化增加了猜测被攻击对象内存布局的难度.

自动结构体随机化[45]对占有相同大小内存空间的域实现内存布局随机化.这里的域包括C/C++程序中出现的各种数据类型,最常用的是int,char,long等基本类型及结构体.文献[45]的自动随机化算法扫描包含数据变量的结构体,构建邻接表,其纵向链节点保存不同类型(占内存空间不同)的域,横向链节点保存相同类型(占内存空间相同)的域,遍历每个横向链表,对其域节点的串接顺序进行随机化处理.

这2种细粒度的随机化方法部署在函数内部、结构体、类内部,使攻击者无法预知结构体域排列的特点,降低攻击者越界篡改被攻击对象的成功率.

本节内存布局随机化方法的对比分析如表3所示:

Table 3 Method Comparison about Memory Layout Randomization

表3 内存布局随机化的方法比较

Detection/Prevention Method	Memory Address Leakage	Protected Object	Characteristics	Limitations
StackArmor[37]	UAF Uninitialized Read	Target Associated Memory Layout within the Stack	Break traditional rules of stack frame organization.	Prone to be affected by the stack pointer aliasing information to recognize any uses of pointers into stack objects.
Undangle[38]	UAF Double Free	Target Associated Memory Layout	Capture lifecycle information of dangling pointer, including creation, propagation, use and destruction .	Coverage depends on the input; False Negatives.
FreeSentry[39]	UAF Cross-border Memory Access	Target Associated Memory Layout	Link object boundaries with reference pointers.	Provide protection for stack objects with extra overhead for their short lifecycle.
DANGNULL[40]	UAF Double Free	Target Associated Memory Layout	Nullify all dangling pointers in runtime in combination with static instrumention.	Need the support of source code; Modules Limitation.
Nested Virtual Clusters[41]	UAF Cross-border Memory Access	Target Associated Memory Layout	Improve by 25% compared with existing techniques for the randomization of larger blocks.	Difficult to deploy for its difference from default memory allocation in mainstream OS.

Continued (Table 3)

Detection/Prevention Method	Memory Address Leakage	Protected Object	Characteristics	Limitations
DieHard[42] DieHarder[43]	UAF Cross-border Memory Access	Base Address of Randomized Object	Reduce the probability of cross-border accessing sensitive information to $1-(1-\frac{1}{M})^O$ (O is the number of overwritten objects)	Compatibility issues with the mainstream memory allocator. Redundant heap space.
Randomizing the Encapsulated Structure[44]	Cross-border Memory Access	Target Associated Memory Layout	High proportion of reorderring the structure(function, struct, class) (above 90%).	Need the support of source code and GCC.
Automatic Data Structure Randomization[45]	Cross-border Memory Access	Target Associated Memory Layout	The unexpected data structure field sequence for initialized data structure randomization.	Need the support of source code and GCC .

3.2 内存越界读写保护

攻击者得到被攻击对象的内存布局后,会试图获取被攻击对象的内容.例如,栈内存信息有其固定分布特征,攻击者可以预知函数返回地址和可控变量的内存排列,但却无法直接打印出函数返回地址值,只能利用格式化字符串攻击等获得字符串后面的敏感信息.这里的越界读写不仅指内存空间的越界,也包括逻辑上的越界访问,如已释放的空间里的信息不应该被访问,但越界获取释放对象的遗留信息造成 UAF 漏洞.内存越界读写需要可控对象作为跳板,获得被攻击对象的访问权限.这里分别从界定可控对象访问范围、设置被攻击对象不可读取权限 2 个方面分析内存越界读写的防御方法.我们选取了虚表指针、数据指针、由指针(索引)访问的对象、栈内的敏感信息 4 类典型的攻击者可控对象,讨论其合理的访问范围,并分析当针对 4 类可控对象的越界保护失效后,如何设置被攻击对象(代码指针、虚表指针等指向关键代码的地址信息)的访问权限,才能使越界访问无效.

3.2.1 界定可控对象的合理访问范围

1) 虚表指针保护

UAF 漏洞和类型混淆漏洞会涉及到虚表指针,对虚表指针的保护可以抑制攻击者进行内存越界读写,从而阻止内存地址的泄漏.

对于 UAF 漏洞,对象 A 被释放后,其占用的地址空间被其他对象填充.因为 C++对象的前 4 B 是虚表指针,攻击者把占用对象的前 4 B 填充为精心构造的地址信息 $Addr$(如与被攻击对象关联的固定指针变量 vp 的地址).攻击者对悬挂指针解引用并调用对象 A 中的虚函数时,程序转向越界访问 $Addr$ 处的内容,造成关键地址信息泄漏.如果 $Addr$ 处的变量是数据指针,则发生了指针类型冲突(把虚表指针视为代码指针).为此,PointerScope[46] 为 X86 指令系统设计了指令操作数类型限制规则,如 MOV 等数据传输指令的 2 个操作数的类型相同,MUL,DIV 指令的操作数类型是整数,JMP,JZ 等控制流相关指令的操作数是控制指针,这种类型界定和类型传播为类型推理算法提供了依据,把指针误用检测转化为类型冲突检测.

对于类型混淆漏洞,攻击者可通过解引用虚表指针,引起父类和子类虚函数的混淆调用,最终造成内存越界访问,如图 2 中的 vtable escape 错误.这类由虚表指针引起的内存越界访问,直接后果是错误的虚函数调用.对虚函数调用的检查,也能及时检测虚表指针是否发生错误解引用.SAFEDISPATCH[47] 提供 2 种级别的 C++虚函数调用检查:针对对象实例的方法实现、针对对象的虚表.虚函数的作用是因对象继承而采用不同的策略实现函数名相同的不同方法.在编译时分析调用虚函数的对象,确定其指针或引用的静态类型,静态类型必须声明为该对象的基类指针或者引用.在运行时,C++的动态调度机制确定调用虚函数的对象实际类型,是本身基类类型还是其子类类型.虚函数的调用保护包括静态类层次分析(class hierarchy analysis,CHA)和运行时检查 2 个阶段.CHA 分析需要编译器对源代码进行静态分析,确定虚函数合理的调用点范围(基类和子类的虚函数),并以 ValidM 表格的形式表示.在运行时,把获得的虚函数指针与 ValidM 中规定的方法实现对比,把虚函数的调用点控制在 ValidM 规定的合理范围内.

2) 数据指针保护

指针是 C 或 C++重要的数据结构,为程序的灵活性和可扩展性提供了支持,攻击者往往劫持函数返回地址、虚函数地址、或者其他指针,达到控制

劫持或者数据劫持的目的.基本类型混淆漏洞以及修改指针索引造成的越界访问,都离不开对指针的非法操作.

SoftBound[48]通过对 C/C++语言中的数据指针进行边界检查来保证内存访问的安全性,可以保护内存、寄存器、函数返回值中的指针.针对存储在寄存器的指针值,生成对应的基址 base 标识和指针范围 bound 标识.对于每一次指针的内存访问,SoftBound 在其前面插入 check 函数,来保证无指针越界访问.针对存储在内存的指针,SoftBound 会利用查找表,在指针地址与元数据表(包括指针对应的 base 标识和 bound 标识)之间建立映射,在取指针值时,SoftBound 会插入查表指令,来获取该指针对应的 base 和 bound 值;在存入指针值时,SoftBound 更新查找表里的相应指针项.

类型化内存模型[49]把 C 语言环境中的字节内存模型抽象为由字节域和类型域组成的内存块,其中字节域即常规的有序数据内容,类型域则是其对应的类型集合.这种内存安全模型的核心准则是,保证按照数据类型的写入类型 mt 的兼容类型 mt' 进行读取,而所占字节数量相等的 2 种数据类型则视为兼容类型.基于该安全准则,在内存读写操作中,该模型会先根据读写内容的数据类型判断目的内存地址范围是否合法.

3) 对象边界保护

内存操作函数的恶意利用破坏了原有的对象边界,整数溢出漏洞恶意修改了访问对象的索引,本节讨论这些攻击的检测与防御.

SoftBound 从指针保护的角度阻止了内存的越界访问.同理,保护用指针访问的对象也可以阻止内存的越界访问.WIT(write integrity testing)[50]对中间语言代码进行静态的指针分析和写安全性分析.指针分析就是寻找每个指针的指向对象的集合(包括局部变量、全局变量及动态分配的内存对象),即 PTS(points-to set).每个指针及其对应的 PTS 安全级别相同(以颜色标记安全级别).写安全性分析就是确定每条指令的安全级别.写指令与其目标内存地址颜色一致,间接调用指令与其对应的目标函数颜色一致.但是,WIT 并不能阻止越界读操作,因为读指令被默认为安全级别.而越界读操作会引发内存关键信息的泄漏,如内存地址泄漏.

WIT 把对象边界与其对应的语义(同一指针指向的 PTS,写指令的目标内存地址,间接调用指令的目标函数)联系起来.Baggy Bounds Checking[51]根据对象类型不同,在程序执行的不同阶段记录对象边界信息.在每次为对象分配内存时,Baggy Bounds Checking 会多分配若干字节,使分配内存大小满足 2^n.它利用内存对象边界表[52]记录对象边界信息,根据 $malloc(\cdot)$,$free(\cdot)$等函数更新堆块的边界信息,在调用函数和函数返回时更新栈内局部变量的边界信息,在程序启动时更新全局变量的边界信息.

对内存操作函数的恶意利用会破坏原有的对象边界.基于规则的代码补丁[53]对 C/C++中可能造成缓冲区溢出的代码段进行修改并重新编译,在不改变程序逻辑基础上,很大程度上阻止内存函数越界复制等漏洞.其中补丁规则比如用函数 $strncpy$ 替代函数 $strcpy$,对索引值进行取模运算限制访问范围,对指针运算结果添加必要检查,对内存输入函数添加长度参数等等.在目标缓冲区的末尾添加"\0"字符,有效防御因缺少 NULL 终结符造成的越界操作.类型化内存模型[49]用四元组表示每个内存块的状态,其中一个元素 S 表示内存块与其大小的映射.该模型在处理内存操作函数时,会先检查目标地址所在内存块的合理范围,同时把源地址的字节域与类型域一起拷贝到目标地址,能在很大程度上阻止攻击者利用这类函数进行超长拷贝.

整数溢出漏洞会恶意修改访问对象的索引,造成内存越界访问.RICH(run-time integer checking)[54]设计了类型安全规则:R-UNSAFE 和 R-BINOPZ.R-UNSAFE 对 *down-cast* 的变量做边界检查,溢出时报警.R-BINOPZ 对整数运算的结果做边界检查,防止其造成上溢或下溢.基于信息流策略的整数漏洞插装和验证[55]在 Dening 建立的信息流安全模型[56]的基础上,对源代码转换而来的中间表示代码抽取信息流,针对其中的整型变量,制定相应的安全操作约束.针对用户可控、可能触发潜在漏洞的的整数操作,如果不满足安全约束,如整数相加的运算结果值是否在目标整数类型的取值范围内、偏移量的取值是否为非负值等等,则需要插入相应的动态验证代码,保证程序在实时运行时及时检测出整数溢出漏洞.

与上述基于源代码的代码补丁不同,IntScope[57]是在符号执行 2 进制程序的过程中检测是否发生整数溢出漏洞.IntScope 总结出整数溢出漏洞的 3 个关键点:①不可信的来源.发生整数溢出的运算操作数都来源于不可靠的输入源,如网络通信消息,输入

文件,命令行选项等等.诸如 *read*, *fread*, *recv* 等函数从不可靠的输入源里读取数据,这里定义为 taint source;②溢出整数的使用.当溢出的整数在程序的敏感点被使用时,可能会引发其他的漏洞.比如内存分配函数(*malloc*, *alloca*)把溢出整数作为 *size* 参数,会使分配的内存不够用,并最终导致缓冲区溢出.溢出整数作为访问内存的数组索引或者指针偏移,会导致任意内存位置读写.这些敏感点命名为 sinks;③不完善的过滤检查.根据这些特点,IntScope 识别出触发整数溢出漏洞的执行流,在每个 sinks 点强制加入边界检查,判断是否发生整数溢出漏洞.

CVE-2010-0010 的引发原因是整数的符号混淆问题,而文献[53]把有符号的索引变量值转换为一个无符号数,阻止了这类整数溢出漏洞的发生.针对整数漏洞的符号错误,防御方需要保证攻击者的可控数据,如内存操作函数的长度参数,数组索引这些整型变量,满足如下 2 种情况之一[58]:①该变量作为无符号整数,通过上界检查;②该变量作为有符号整数,需保证其为非负值并通过上界检查.

4) 栈内敏感信息保护

栈内的敏感信息比如函数的返回地址,是攻击者控制程序执行流的重要数据.而造成栈内信息泄漏的格式化字符串漏洞,其根源在于 *printf*(·)系列函数的参数中有"%"标识的形参和实参的数目不一致.针对该类漏洞,FormatGuard[59]比较传递给函数 *printf*(·)的变量实参的个数与格式规定符(形参)的个数,如果实参的个数小于格式化字符串中有"%"标识的形参个数,则 FormatGuard 判定其为格式化字符串攻击.但是如果攻击者使用的实参和形参个数相符,则会产生漏报[58].

基于信息流策略的污点传播分析及动态验证[60]对源代码转换得到的中间表示进行静态信息流分析,识别来自于网络、文件或用户输入的污染数据,引入污染数据的安全操作被归纳为污点传播的检查点[61],在中间代码的检查点插入动态验证代码.该方法可应用于格式化输出漏洞检测中,如在执行函数 *printf*(·)之前,检查其传入的实参变量里面是否存在特定的格式字符,如果存在则报错.

3.2.2 为被攻击对象设置访问权限

3.2.1 节中 4 个方面从指针类型、函数调用、内存空间 3 个角度为可控对象界定合理的访问范围,使其不能作为越界读写的跳板.而 Readactor[62]在界定范围打破后,仍能使代码指针、虚表指针等指向代码区域的数值不被越界访问.Readactor 为分布于堆、栈的代码指针、虚表指针提供保护,添加专门的 Trampoline 代码区域,用于存储目标为函数入口地址、返回地址、虚表地址的直接分支指令,代码指针则转换为 Trampoline 指针.同时,Readactor 利用 Intel 扩展页表特性设置所有代码页为只执行属性.这种间接跳转的保护方法使攻击者无法读取 Trampoline 区域中真正的敏感地址,越界读写失效.而且 Readactor 基于编译器实现,隔离了代码和数据,防止因正常程序流读代码造成误报.

本节针对 4 类可控对象的"边界"保护(包括合理的指针类型、虚函数调用点范围、内存空间边界)进行分析,并大胆假设在"边界"保护失效后,通过隐藏受保护对象并设置有限的访问权限,达到直接有效的防御效果.界定可控对象的合理访问范围是在通往被攻击对象的必经之路上设置障碍,而为被攻击对象本身设置不可逾越的访问权限则从根本上保障其不被泄漏.其中涉及到的内存越界读写保护方法的对比分析如表 4 所示:

Table 4 Method Comparison between Cross-border Memory Access Protection

表 4 内存越界读写的保护方法比较

Detection/Prevention Method	Memory Address Leakage	Protected Object	Characteristics	Limitations
PointerScope[46]	UAF Cross-border Memory Access	Data Pointer Code Pointer	Avoid false negatives/positives induced by direct/indirect data dependency in dynamic taint propagation.	Difficult to infer variable type accurately at binary level. Generated detailed execution trace.
SAFEDISPATCH[47]	Type Confusion UAF vtable escape	vtable Pointer Virtual function calls	Ensure that target addresses of virtual function calls is in the reasonable scope of call sites.	Need source code and C++ compiler modification. Modules Limitation.
SoftBound[48]	Type Confusion Array Index Modification	Data Pointer Objects Accessed through Pointers	Detect the behavior of space security violation timely.	Need source code. Detect memory read/write operation through pointers with a high-performance cost. (67%)

Continued (Table 4)

Detection/Prevention Method	Memory Address Leakage	Protected Object	Characteristics	Limitations
Typed Memory Module[49]	Array Index Modification Cross-border Copy	Data Pointer Objects Accessed through Pointers	Reserve byte semantics of C language. Give a strict description of memory operations semantics.	Lack of practical application. Many non-quantifiable factors in the actual operation.
WIT[50]	Length Attribute Modification Cross-border Copy	Objects Accessed through Pointers (Index)	Provide protection from attacks against write integrity violation.	Need specific compiler. Limited effectiveness affected by the accuracy of static pointer analysis .
Babby Bounds Checking[51]	Type Confusion Array Index Modification	Objects Accessed through Pointers (Index)	Reduce the time cost to bounds lookups and bounds checks for the size of allocated memory regions is always 2^n.	Unable to detect temporary space security violation, cross-border access within data structure.
Buffer overflow patching: rule-based approach[53]	Cross-border Copy Array Index Modification	Target as Memory Manipulation Function Parameters	Prevent cross-border access caused by memory manipulation function calls, array index modification, pointer arithmetic, etc.	Unable to prevent cross-border access caused by pointer re-initialize, user-defined function calls.
RICH[54]	Integer Overflow	Objects Accessed through Pointers (Index)	Check for arithmetic overflow and under-flow in runtime. Make static type checking for *down-cast*.	False negatives for implicit conversion caused by integer accessed through different types of pointer.
Statically Detect and Run-Time Check[55]	Integer Overflow	Objects Accessed through Pointers (Index)	Reduce false negatives via extracted inter-compilation unit tainted information flow. Improve precision via memory address reference information flow.	Modular programming. False negatives for multistage referenced memory address of tainted information.
IntScope[57]	Integer Overflow	Objects Accessed through Pointers (Index)	Verify 20 vulnerabilities of 26 detected unknown integer overflow vulnerabilities.	False positives for lack of prior information. The inaccuracy of simulation execution of some memory manipulation functions.
FormatGuard[59]	Format String Vulnerability	Sensitive Information within the Stack	Unable to provide protection when program calls other libraries except for Glibc.	False negatives for quantitative comparison of formal/actual parameter of formatting function.
Taint Propagation Analysis and Dynamic-Verification[60]	Format String Vulnerability	Sensitive Information within the Stack	Provide effective detection of the predefined type of vulnerabilities in source code.	Inaccurate static analysis. Produce false positives if actual parameters of formatting function contains formatting character.
Readactor[62]	Cross-border Memory Access	Code Pointer vtable Pointer	Hide the target within data page into the code page with the help of hardware features.	Need source code ,specific compiler. Limited effectiveness of compile-time fine-grained randomization.

3.3 被攻击对象的内容保护

如果被攻击对象内容是特定的指针值,则需要通过加密、掩码等方法保护该内容. PointGuard[63]利用编译器把源代码转换为中间表示形式,例如优化 AST 抽象语法树,利用 *Mask* 在数据指针,代码指针被保存前对其进行加密,在指针使用前解密. 其中,*Mask* 源于 Linux 内核包含的随机数发生器,在进程启动时动态生成,确保 *Mask* 足够安全. PointGuard 不能保护寄存器中的指针,以及动态代码生成的指针,对于不通过指针访问的对象越界访问没有作用.

数据随机化(data randomization/data space randomization, DR/DSR)[14,64]把保护内容从指针推广到一般的数据内容,且提供对数据内容的读写保护,优于 WIT[50]的写保护. 文献[64]对程序中由指针(或索引)访问的对象(如数组 char $A[i]$)进行划分,如果不同指令是对同一个对象进行读/写操作,这些指令的操作数被划定为同一等价类. 对于不通过指针访问的对象(如整形变量 i),则单独将其划分为一个等价类. 数据随机化的关键是针对不同的等价类,分配不同的 *Mask* 值. 对于写指令,用对应的 *Mask* 值对要写入内存的内容进行加密;对于读指令,用对应的 *Mask* 值对从内存中读出的值进行解密. 因此,即使攻击者可以读出邻接对象值,攻击者也无法获取该对象的真实值,因为该对象值是

用不同的 *Mask* 加密的. DR/DSR 对被攻击对象的数据内容提供保护,不管对该对象的访问是否通过指针,但面临等价类划分不准确、暴力破解 *Mask* 值等限制.

3.4 关键地址信息保护

假设攻击者已经成功获取泄漏点的地址信息,但是成功发起攻击还需要攻击者获取其他关键地址信息. XnR[65]从代码页的访问属性角度对关键地址信息提供保护. 代码指针在代码页中的相互引用会引发大规模的内存泄漏,攻击者利用某个代码页的泄漏地址通过直接分支指令识别其他代码页. XnR 采用滑动窗口机制仅把正在运行的代码页设为 present,当程序执行到属性为 non-present 的内存页时,引发页访问错误. 内存管理单元(memory management unit, MMU)的页错误处理机制经过修改后,会分析造成缺页的 3 种内存操作:1)正常取指;2)从数据区域加载数据;3)从代码区域读数据. 第 3 种情况视为非法读代码,应中断程序执行. 这种对可执行代码的禁止读操作属性,把泄漏的地址信息限制在正在运行的代码页中,无法提供足够的 Gadgets 来组织 ROP 链. 但对于良性程序流存在读代码的情况,XnR 有误报.

细粒度的地址随机化技术从可执行模块的内部布局角度对关键地址信息提供保护. 在现在部署的地址空间随机化机制下,可执行模块的内部布局相对固定,攻击者可以借由泄漏点窥知整个模块内部的内存布局,如图 1 所示的 $Aslr\_ma$,因为 $Aslr\_ma$ 在模块中相对偏移 offset 固定,攻击者通过 $Base\_ma=Aslr\_ma-offset$ 得到该模块随机化处理后的加载基址,并由此推断表 1 中各重用代码的起始地址. 细粒度的地址随机化技术可以随机化模块内代码地址,从而阻止攻击者获取 *Shellcode* 装配必须的关键地址信息. 以表 1 中的 *Shellcode* 为例,假设 kernel32. dll 基址为 0x7c800000,以模块内相对偏移地址表示该 *Shellcode*,$\{a_1, a_2, \cdots, a_n\}$ 表示为 {0x3ed85, 0x1228cb, 0x3ed85, 0x16e68f, 0x6114d, 0x0e0bf, 0x1caa2, 0x1478e4}. 而在细粒度的地址随机化技术保护下,重用代码的相对偏移被随机化处理,上述 $a_i(1\leqslant i\leqslant n)$ 的取值对应的指令序列不一定是攻击者所需要的 Gadget.

1997 年,Forrest 等人[66]指出可以利用内存布局随机化来缓解缓冲区溢出攻击,例如在栈帧里面填充随机数量的字节,对变量、栈帧的内存位置进行随机化调整等. 该随机化增加了定位局部变量和返回地址的难度,可以抑制传统的 *Shellcode* 攻击,但无法防御 ROP 攻击. 2004 年,ASLR 已经被应用到 Linux 和 OpenBSD 系统[67]. PaX 通过对可执行模块、映射模块和堆栈区域进行随机化部署,其中可执行模块包括程序的可执行代码、初始化及还未初始化的数据;映射区域包括堆、动态链接库、线程栈及共享内存. 文献[67]采用页面加载基址的随机化实现页内函数簇的基址随机化,但是攻击者仍然可以通过全局偏移表(global offset table, GOT)来定位本模块内部或外部函数的实际地址. 随后,ASLR 被逐渐引入到 Windows[68],iOS, Android 等操作系统中.

传统的 ASLR 仅实现模块、Stack、Heap 等首地址的随机化. 文献[69]把加载基址随机化扩展到函数、基本指令块(连续的指令序列)[34]甚至每条指令[33],实现细粒度 ASLR. 这样,即使某个数据结构或函数的地址被攻击者得知,也无法推断出同一模块下的其他数据结构和函数的地址.

STIR 由地址映射算法实现指令序列加载基址的重定位. 通过修改目标程序的导入表,例如把 kernel32. dll 的对应项替换为自定义的动态链接库,在程序加载时会应用该动态链接库中的地址随机化映射算法,实现内存的分配,从而实现了大规模的 X86 代码中大量指令块的随机化,仅带来 1.6% 的运行时间开销. 但 STIR 对于运行时加载及动态生成的代码不提供保护.

ILR 实现了指令位置随机化,分为静态分析和实时保护 2 个阶段. 静态分析的目的是生成覆写规则,该规则包括指令地址定义和重定向定义. 其中指令地址定义规定了可执行模块内部偏移地址与指令的对应关系,如:39bc cmp eax, #24,其中 0x39bc 为地址;重定向定义实现了上一条指令到下一条指令的执行流转换,如 39bd→d27e 表示地址 0x39bc 处的上一条指令执行完毕,处理器会到 0x39bd 处取下一条指令,而下一条指令随机化处理后的偏移地址为 0xd27e. ILR 需要在特定平台进程级虚拟机(per-process virtual machine, PVM)实现. PVM 在取指时,会检查 PC 里的地址及其对应的指令,根据覆写规则决定指令执行还是停机. ILR 对于指令起始地址,函数调用的返回地址,间接分支跳转的目标地址的随机化程度很高且定位准确.

细粒度 ASLR 的实现方法会引起高额的 CPU 开销和内存分配开销,难以部署在主流的操作系统中.

除此以外,把关键地址信息存储在段寄存器,因为攻击者在用户空间不可访问其中的内容,该方法很好地保护了关键的地址信息[70].

在现有保护机制下,这些基于不同攻击阶段的防御方法,在不同程度上阻碍攻击者准确获取内存地址信息,使攻击者缺少足够的信息去成功实施攻击.但现有的防御方案都存在一定局限性,上述的防御方法也因实现难度大、性能限制、开销较大等原因并没有得到广泛部署.层出不穷的 APT 攻击方式,使攻击者成功绕过前面 4 层保护成为可能.这里从检测 Gadget 链和控制流完整性 2 个角度分析代码重用攻击的实时监测.

及时检测出用于攻击的 Gadget 链,面临着检测特定 Gadget 类型,需要源代码或自定义的编译器的支持等性能限制,带来了 2 进制重写、时间开销大等不可避免的代价.DROP[71]检测以 RET 指令结束的短指令序列;Ropdefender[72]利用隔离的影子栈检测返回地址是否被修改,两者都针对 Ret-based Gadget 进行动态检测,打破了 2 进制程序的完整性,带来的时间开销不容忽视.G-free[73]通过消除程序中的 Gadget 达到抑制 ROP 攻击的目的,但需要事先获取源代码.ROPecker[74]以分支指令作为切入点,利用 LBR(最近分支记录)寄存器中的信息,识别执行流中已经执行过的 Gadget 链,其离线分析识别了 Ret-based Gadget,Jmp-based Gadget,Stack Pivoting 等指令序列类型,预测接下来的执行流中潜在的 Gadget 链,防御更为全面.仅在运行时模拟执行目标地址难以确认的 Jmp-based Gadget,相对于对所有影响控制流的指令做插桩处理,节约了很多时间.

代码重用攻击必然造成控制流的恶意篡改.CPM[10]对代码指针进行掩码操作,以阻止控制流的非法跳转.PointerScope[46]通过检测指针类型冲突来实现对代码指针的保护,检测控制流劫持.

综上所述,我们从内存布局的随机化、内存越界读写保护、被攻击对象的内容保护、关键内容地址的随机化 4 个方面分析和总结了内存地址泄漏攻击和相应防御方法,并简短分析了代码重用攻击的防御检测.

4 总结与展望

内存损坏攻击一直是黑客和安全专家关注的焦点,也是 APT 攻击进行信息泄漏和控制劫持的基础。APT 攻击从依赖于单个漏洞的有效利用,逐步发展到多个漏洞的综合利用.本文着眼于内存地址泄漏漏洞的分析与防御,结合漏洞实例,剖析了针对指针或对象的非法操作:利用被攻击对象本身的脆弱属性造成的内存越界读写,如 UAF 漏洞、类型混淆漏洞、整数溢出漏洞、直接修改特定数据结构的长度属性漏洞,总结了内存操作函数越界漏洞、格式化字符串漏洞以及侧信道攻击漏洞.同时,从内存损坏攻击过程看,总结和归纳了内存布局随机化、内存越界读写保护、内存对象内容保护、内存对象地址随机化等对抗内存地址泄漏的攻击,从而达到内存布局看不清、内存对象读不到、内存对象内容读不懂、关键内存地址猜不准的保护目的.

针对 APT 攻击,不仅需要从软件的全生命周期通盘考虑程序的安全性,如微软的安全开发生命周期(security development of lifycycle, SDL)从需求、设计、编码、测试、运行和维护等考虑安全性,而且需要确保程序运行环境的安全性,包括程序调用的公共库安全确保、内核库的安全确保.换言之,程序的安全性确保与操作系统的安全确保或者第三方的安全确保联动.例如,内存地址泄漏的防御需要操作系统公共库支持内存布局(包括内存页和内存对象)的非连续性和大小的非一致性、代码加载地址的细粒度随机化;内存对象资源的访问控制和内存对象的加密混淆同样需要从程序本身发展到与操作系统的协同防御.同时,在内存地址泄漏的防御上,还需要考虑纵深防御思想,即从最外层的内存布局随机化、内存越界保护到内存对象内容保护,最后一层是内存对象地址的保护.这样,才能做到立体防御、防范于未然.

另外,不同方面的安全防御方法均存在不容忽视的代价.被攻击对象内容加密混淆会带来额外的时间开销,需要协调与其他程序的共享访问;内存布局随机化需要对现在主流操作系统的内存分配机制做很大改动,且会消耗过多的内存空间;WIT,Baggy Bounds Checking 等保护被攻击对象边界的方法则需要源代码,并在特定编译器平台下做静态分析或代码补丁,其有效性依赖于静态指针分析[75]的准确性;细粒度的代码基址随机化需要准确的静态反汇编,时间开销大,其执行环境限定在特定平台,如 PVM.

未来的研究工作有:

1) 从防御措施的部署角度,设计兼备安全性、实用性的内存布局随机化方案,以及高效的细粒度

的地址空间随机化方案;推进与主流操作系统兼容的,开销易于接受、便于实现的防护策略,推进其广泛应用.

2) 从程序设计角度,设计便于标记代码地址随机化、内存对象资源的访问控制和内存对象的加密混淆的程序设计语言.

3) 从2进制分析角度,针对X86指令系统损失变量类型信息的特点,设计准确识别2进制程序中数据结构的方法,在2进制层还原高级程序语言的语义信息.

4) 从源代码分析角度,设计全面有效的静态分析方法,克服程序运行所需的动态链接库因没有被编译进可执行文件而缺少相关保护这种模块化限制;针对程序分析的辅助信息,如函数调用图、控制流图,设计准确高效的信息收集方案.

5) 从漏洞挖掘角度,结合控制流分析、数据流分析、切片分析、代码插桩等技术实现灵活的、功能丰富的漏洞挖掘平台;同时在防护能力不断增强的情况下,可考虑漏洞样例的自动生成并提取有针对性的防御思路.

6) 从主被动融合式的防御体系角度,在被动的局面下实现主动防御. 基于先验知识的被动防护不能避免或缓解未知漏洞. 随机化等主动防御方案致力于扰乱或阻断未知漏洞的攻击链,性能开销不容忽视. 深度组合运用拟态安全防御体系,构成主被动融合式防御体系,增大内外部攻击难度.

参 考 文 献

[1] Szekeres L, Payer M, Wei T, et al. SoK: Eternal war in memory [C] //Proc of Security and Privacy (SP) 2013. Piscataway, NJ: IEEE, 2013: 48-62

[2] van der Veen V, Cavallaro L, Bos H. Memory errors: The past, the present, and the future [G] //Research in Attacks, Intrusions, and Defenses. Berlin: Springer, 2012: 86-106

[3] Andersen S, Abella V. Data execution prevention: Changes to functionality in Microsoft Windows XP service pack 2, part 3: Memory protection technologies [EB/OL]. 2004 [2014-08-08]. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

[4] Wikipedia. Return-to-libc attack [EB/OL]. 2010 [2014-06-06]. http://en.wikipedia.org/wiki/Return-to-libc_attack

[5] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86) [C] //Proc of the 14th ACM Conf on Computer and Communications Security. New York: ACM, 2007: 552-561

[6] Bletsch T, Jiang X, Freeh V W, et al. Jump-oriented programming: A new class of code-reuse attack [C] //Proc of the 6th ACM Symp on Information, Computer and Communications Security. New York: ACM, 2011: 30-40

[7] Göktas E, Athanasopoulos E, Bos H, et al. Out of control: Overcoming control-flow integrity [C] //Proc of SP'2014. Piscataway, NJ: IEEE, 2014: 575-589

[8] Shuai Z. The detection and defense about APT attack [EB/OL]. 2014 [2014-06-16]. http://finance.ifeng.com/a/20140616/12544462_0.shtml

[9] Liang Yu, Fu Jianming, Peng Guojun, et al. S-Tracker: Attribution of shellcode exploiting stack [J]. Journal of Huazhong University of Science and Technology: Natural Science Edition, 2014, 42(11): 39-46 (in Chinese)
(梁玉, 傅建明, 彭国军, 等. S-Tracker: 基于栈异常的shellcode检测方法[J]. 华中科技大学学报: 自然科学版, 2014, 42(11): 39-46)

[10] Philippaerts P, Younan Y, Muylle S, et al. CPM: Masking code pointers to prevent code injection attacks [J]. ACM Trans on Information and System Security (TISSEC), 2013, 16(1): No.1

[11] Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity [C] //Proc of the 12th ACM Conf on Computer and Communications Security. New York: ACM, 2005: 340-353

[12] Milenković M, Milenković A, Jovanov E. Hardware support for code integrity in embedded processors [C] //Proc of the 6th Int Conf on Compilers, Architectures and Synthesis for Embedded Systems. New York: ACM, 2005: 55-65

[13] Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity [C] //Proc of the 7th Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2006: 147-160

[14] Bhatkar S, Sekar R. Data space randomization [G] // Detection of Intrusions and Malware, and Vulnerability Assessment. Berlin: Springer, 2008: 1-22

[15] Younan Y, Joosen W, Piessens F. Runtime countermeasures for code injection attacks against C and C++ programs [J]. ACM Computing Surveys, 2012, 44(3): 17-51

[16] Metasploit L L C. The metasploit framework [EB/OL]. 2007 [2012-03-15]. http://www.metasploit.com/

[17] Cowan C, Pu C, Maier D, et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks [C] //Proc of the 7th USENIX Security Symp. Berkeley, CA: USENIX Association, 1998: 63-78

[18] Richarte G. Four different tricks to bypass stackshield and stackguard protection [EB/OL]. 2002 [2013-07-16]. http://downloads.securityfocus.com/library/StackGuard.pdf

[19] Harbaugh S, Wavering B. HeapGuard, eliminating garbage collection in real-time Ada systems [C] //Proc of NAECON' 1991. Piscataway, NJ: IEEE, 1991: 704-708

[20] Özkan S. CVE details: The ultimate security vulnerability datasource—Vulnerabilities by type [EB/OL]. 2013 [2014-04-13]. http://www.cvedetails.com/

[21] Afek J, Sharabani A. Dangling Pointer: Smashing the Pointer for Fun and Profit [M/OL]. Waltham: Watchfire, 2007 [2014-03-25]. https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

[22] Kotov V. Dissecting the newest IE10 0-day exploit (CVE-2014-0322) [EB/OL]. 2014 [2014-02-14]. http://article.pchome.net/content-1696545.html

[23] Akritidis P. Cling: A memory allocator to mitigate dangling pointers [C] //Proc of the 19th USENIX Security Symposium. Berkeley, CA: USENIX Association, 2010: 177-192

[24] Wang T, Lu K, Lu L, et al. Jekyll on iOS: When benign apps become evil [C] //Proc of the 22nd USENIX Security Symp. Berkeley, CA: USENIX Association, 2013: 559-572

[25] MWR Labs. Mwr labs pwn2own 2013 write-up—webkit exploit [EB/OL]. 2013 [2013-04-19]. http://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up—webkit-exploit/

[26] Dewey D, Giffin J. Static detection of C++ vtable escape vulnerabilities in binary code [C] //Proc of NDSS'2012. Reston: Internet Society, 2012 [2014-06-05]. http://www.internetsociety.org/static-detection-c-vtable-escape-vulnerabilities-binary-code

[27] Pincus J, Baker B. Beyond stack smashing: Recent advances in exploiting buffer overruns [J]. IEEE Security & Privacy, 2004, 2(4): 20-27

[28] Strackx R, Younan Y, Philippaerts P, et al. Breaking the memory secrecy assumption [C] //Proc of the 2nd European Workshop on System Security. New York: ACM, 2009: 1-8

[29] Randolph N, Gardner D, Anderson C, et al. Professional Visual Studio 2010 [M]. New York: John Wiley & Sons, 2010

[30] Wang Qing, Zhang Donghui, Zhou Hao, et al. Zero Day Security: Software Vulnerability Analysis [M]. 2nd ed. Beijing: Publishing House of Electronics Industry, 2011: 244-245 (in Chinse)
(王清, 张东辉, 周浩, 等. 0day安全: 软件漏洞分析技术[M]. 2版. 北京: 电子工业出版社, 2011: 244-245)

[31] Liu L, Han J, Gao D, et al. Launching return-oriented programming attacks against randomized relocatable executables [C] //Proc of TrustCom'2011. Piscataway, NJ: IEEE, 2011: 37-44

[32] Hund R, Willems C, Holz T. Practical timing side channel attacks against kernel space ASLR [C] //Proc of SP'2013. Piscataway, NJ: IEEE, 2013: 191-205

[33] Hiser J, Nguyen-Tuong A, Co M, et al. ILR: Where'd my gadgets go? [C] //Proc of SP'2012. Piscataway, NJ: IEEE, 2012: 571-585

[34] Wartell R, Mohan V, Hamlen K W, et al. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code [C] //Proc of the 2012 ACM Conf on Computer and Communications Security. New York: ACM, 2012: 157-168

[35] Homescu A, Brunthaler S, Larsen P, et al. Librando: Transparent code randomization for just-in-time compilers [C] //Proc of the 2013 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2013: 993-1004

[36] Seibert J, Okhravi H, Söderström E. Information leaks without memory disclosures: Remote side channel attacks on diversified code [C] //Proc of the 2014 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2014: 54-65

[37] Chen X, Slowinska A, Andriesse D, et al. StackArmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries [C] //Proc of NDSS'2015. Reston: Internet Society, 2015 [2015-12-06]. http://www.internetsociety.org/doc/stackarmor-comprehensive-protection-stack-based-memory-error-vulnerabilities-binaries

[38] Caballero J, Grieco G, Marron M, et al. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities [C] //Proc of ISSAT'2012. New York: ACM, 2012: 133-143

[39] Younan Y. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers [C] //Proc of NDSS'12. Reston: Internet Society, 2015 [2015-12-15]. http://www.internetsociety.org/doc/freesentry-protecting-against-use-after-free-vulnerabili-ties-due-dangling-pointers

[40] Lee B, Song C, Jang Y, et al. Preventing use-after-free with dangling pointers nullification [C] //Proc of NDSS'2015. Reston: Internet Society, 2015 [2015-12-24]. http://www.internetsociety.org/doc/ preventing-use-after-free-dangling-pointers-nullification

[41] Tey C M, Gao D. Defending against heap overflow by using randomization in nested virtual clusters [M] //Information and Communications Security. Berlin: Springer, 2013: 1-16

[42] Berger E D, Zorn B G. DieHard: Probabilistic memory safety for unsafe languages [J]. ACM SIGPLAN Notices, 2006, 41(6): 158-168

[43] Novark G, Berger E D. DieHarder: Securing the heap [C] //Proc of the 17th ACM Conf on Computer and Communications Security. New York: ACM, 2010: 573-584

[44] Chen Ping, Xing Xiao, Xin Zhi, et al. Protecting programs based on randomizing the encapsulated structure [J]. Journal of Computer Research and Development, 2012, 48(12): 2227-2234 (in Chinese)
(陈平, 邢骁, 辛知, 等. 基于封装结构随机化的程序保护方法[J]. 计算机研究与发展, 2012, 48(12): 2227-2234)

[45] Xin Zhi, Chen Huiyu, Han Gao, et al. Kernel rootkit defense based on automatic data structure randomization[J], Chinese Journal of Computers, 2014, 37(5): 1100-1110 (in Chinese)

(辛知, 陈惠宇, 韩浩, 等. 基于结构体随机化的内核Rootkit防御技术[J]. 计算机学报, 2014, 37(5): 1100-1110)

[46] Zhang M, Prakash A, Li X, et al. Identifying and analyzing pointer misuses for sophisticated memory-corruption exploit diagnosis [C] //Proc of NDSS'2012. Reston: Internet Society, 2012 [2015-08-20]. http://www.comp.nus.edu.sg/~liangzk/papers/ndss12.pdf

[47] Jang D, Tatlock Z, Lerner S. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks [C] //Proc of NDSS'2014. Reston: Internet Society, 2014 [2014-10-22]. http://cseweb.ucsd.edu/~lerner/papers/ndss14.pdf

[48] Nagarakatte S, Zhao J, Martin M M K, et al. SoftBound: Highly compatible and complete spatial memory safety for c [J]. ACM SIGPLAN Notices, 2009, 44(6): 245-258

[49] He Yanxiang, Wu Wei, Chen Yong, et al. A kind of safe typed memory model for C-like languages [J]. Journal of Computer Research and Development, 2013, 49(11): 2440-2449 (in Chinese)

(何炎祥, 吴伟, 陈勇, 等. 一种用于类C语言环境的安全的类型化内存模型[J]. 计算机研究与发展, 2013, 49(11): 2440-2449)

[50] Akritidis P, Cadar C, Raiciu C, et al. Preventing memory error exploits with WIT [C] //Proc of SP'2012. Piscataway, NJ: IEEE, 2008: 263-277

[51] Akritidis P, Costa M, Castro M, et al. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors [C] //Proc of the 18th USENIX Security Symp. Berkeley, CA: USENIX Association, 2009: 51-66

[52] Jones R W M, Kelly P H J. Backwards-compatible bounds checking for arrays and pointers in C programs [C] //Proc of AADEBUG'97. Berlin: Springer, 1997: 13-26

[53] Shahriar H, Haddad H M, Vaidya I. Buffer overflow patching for C and C++ programs: Rule-based approach [J]. ACM SIGAPP Applied Computing Review, 2013, 13(2): 8-19

[54] Brumley D, Chiueh T, Johnson R, et al. Rich: Automatically protecting against integer-based vulnerabilities [C] //Proc of NDSS'07. Reston: Internet Society, 2007 [2015-09-24]. http://www.ecsl.cs.sunysb.edu/tr/TR207.pdf

[55] Sun Hao, Li Huipeng, Zeng Qingkai. Statically detect and run-time check integer-based vulnerabilities with information flow [J]. Journal of Software, 2013, 24(12): 2767-2781 (in Chinese)

(孙浩, 李会朋, 曾庆凯. 基于信息流的整数漏洞插装和验证[J]. 软件学报, 2013, 24(12): 2767-2781)

[56] Denning D E. A lattice model of secure information flow [J]. Communications of the ACM, 1976, 19(5): 236-243

[57] Wang T, Wei T, Lin Z, et al. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution [C] //Proc of NDSS'2009. Reston: Internet Society, 2009 [2014-12-14]. http://www.isoc.org/isoc/conferences/ndss/09/proceedings.shtml

[58] Liang Bin, Hou Kankan, Shi Wenchang, et al. A static vulnerabilities detection method based on security state tracing and checking [J]. Chinese Journal of Computers, 2009, 32(5): 899-909 (in Chinese)

(梁彬, 侯看看, 石文昌, 等. 一种基于安全状态跟踪检查的漏洞静态检测方法[J]. 计算机学报, 2009, 32(5): 899-909)

[59] Cowan C, Barringer M, Beattie S, et al. FormatGuard: Automatic protection from printf format string vulnerabilities [C] //Proc of the 10th USENIX Security Symp. Berkeley, CA: USENIX Association, 2001: No. 91

[60] Huang Qiang, Zeng Qingkai. Taint propagation analysis and dynamic verification with information flow policy [J]. Journal of Software, 2011, 22(9): 2036-2048 (in Chinese)

(黄强, 曾庆凯. 基于信息流策略的污点传播分析及动态验证[J]. 软件学报, 2011, 22(9): 2036-2048)

[61] Kemerlis V P, Portokalidis G, Jee K, et al. Libdft: Practical dynamic data flow tracking for commodity systems [J]. ACM SIGPLAN Notices, 2012, 47(7): 121-132

[62] Crane S, Liebchen C, Homescu A, et al. Readactor: Practical code randomization resilient to memory disclosure [C] //Proc of SP'2015. Piscataway, NJ: IEEE, 2015: 15

[63] Cowan C, Beattie S, Johansen J, et al. Pointguard TM: Protecting pointers from buffer overflow vulnerabilities [C] //Proc of the 12th USENIX Security Symp. Berkeley, CA: USENIX Association, 2003: 91-104

[64] Cadar C, Akritidis P, Costa M, et al. Data randomization, TR-2008-120 [R]. Redmond: Microsoft Research, 2008

[65] Backes M, Holz T, Kollenda B, et al. You can run but you can't read: Preventing disclosure exploits in executable code [C] //Proc of the 21st ACM Conf on Computer and Communications Security. New York: ACM, 2014: 1342-1353

[66] Forrest S, Somayaji A, Ackley D H. Building diverse computer systems [C] //Proc of the 6th Workshop on Hot Topics in Operating Systems. Piscataway, NJ: IEEE, 1997: 67-72

[67] Shacham H, Page M, Pfaff B, et al. On the effectiveness of address-space randomization [C] //Proc of the 11th ACM Conf on Computer and Communications Security. New York: ACM, 2004: 298-307

[68] Whitehouse O. An analysis of address space layout randomization on Windows Vista [J/OL]. Symantec Advanced Threat Research, 2007 [2014-03-12]. http://www.blackhat.com/ presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf

[69] Snow K Z, Monrose F, Davi L, et al. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization [C] //Proc of SP'2013. Piscataway, NJ: IEEE, 2013: 574-588

[70] Backes M, Nürnberger S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing [C] //Proc of the 23rd USENIX Security Symp. Berkeley, CA: USENIX Association, 2014: 433-447

[71] Chen P, Xiao H, Shen X, et al. DROP: Detecting return-oriented programming malicious code [G] //Information Systems Security. Berlin: Springer, 2009: 163-177

[72] Davi L, Sadeghi A R, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks [C] //Proc of the 6th ACM Symp on Information, Computer and Communications Security. New York: ACM, 2011: 40-51

[73] Onarlioglu K, Bilge L, Lanzi A, et al. G-Free: Defeating return-oriented programming through gadget-less binaries [C] //Proc of the 26th Annual Computer Security Applications Conf. New York: ACM, 2010: 49-58

[74] Cheng Y, Zhou Z, Miao Y, et al. ROPecker: A generic and practical approach for defending against ROP attacks [C] //Proc of NDSS'2014. Reston: Internet Society, 2014 [2015-10-12]. http://www.internetsociety.org/doc/ropecker-generic-and-practical-approach-defending-against-rop-attacks

[75] Heintze N, Tardieu O. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second [J]. ACM SIGPLAN Notices, 2001, 36(5): 254-263

Fu Jianming, born in 1969. Received his BSc and MSc degrees from Huazhong University of Science and Technology in 1991 and 1994 respectively, and his PhD from Wuhan University in 2000, Hubei, China. Now he is professor in Wuhan University. Senior member of China Computer Federation. His main research interests include software security and network security.

Liu Xiuwen, born in 1991. PhD candidate. Received his BSc degree from Yunnan University. Member of China Computer Federation. Her research interests include system security and software security.

Tang Yi, born in 1984. Master candidate. Received his BSc degree from the National University of Defense Technology. His research interests include system security and software security.

Li Pengwei, born in 1987. PhD candidate. Received his BSc degree from Sourthwest Jiaotong University. Member of China Computer Federation. His research interests include mobile cloud computing security and Android security.