# 背景介绍

软件漏洞是攻击计算机系统最常见的媒介。尽管人们在设计、构建和部署没有缺陷的软件方面付出了很大的努力，但即使是相对简单的软件系统中也仍然有漏洞存在。更加严重的是，即使在可信计算基中也可能存在足以令攻击者绕开系统中的安全机制的严重漏洞。

在所有的软件漏洞中，内存破坏型的漏洞是最常出现，也是最严重的漏洞之一。这有以下几个原因：首先，大量的程序是由C或C++这种类型不安全的语言编写的，这使得内存破坏型漏洞极为常见；同时，内存破坏漏洞的攻击能力非常强，很多攻击形式都利用内存破坏漏洞来完成攻击，可以导致的结果下至加密密钥口令等关键数据的泄露，上至系统最高权限被攻击者夺取。由于这些原因，内存破坏型的漏洞今天仍然是最常被利用的漏洞之一。根据微软公司2015年的报告[1]，基于内存破坏的攻击方式占据了远程代码执行类CVE（Common Vulnerabilities and Exposures）的主导地位。

鉴于内存破坏攻击的严重性与广泛性，研究者在如何防范这类攻击上进行了大量的工作。一般来说，现有的防御措施可以分为两类：内存错误的检测和特定攻击形式的缓解对策。

第一类防御措施致力于检测内存破坏攻击成立的源头。由于内存破坏攻击需要依靠内存错误来触发，因此如果能够阻止内存错误的发生，那么就可以从根源上阻止内存破坏公积。但是为了根除内存错误，需要在软硬件上付出比较很大的额外开销，根据Szekeres等人的总结[2]，这类措施带来的额外开销可以达到50%甚至超过100%。而当设计者需要在性能与安全之间进行取舍时，往往是安全特性要为性能让位，因此大多数的内存错误检测措施只用在离线的漏洞检测或者崩溃分析中。

第二类防御措施则为特定形式的攻击提供针对性的防御措施，致力于阻止攻击者实行有害系统安全的行为。内存破坏攻击的形式可以分为四类：代码破坏、控制流劫持、数据导向攻击和信息泄露攻击。对于每一种攻击形式，研究者们开发了有针对性的防御手段。例如，数据执行保护（Data Execution Prevention，DEP）机制要求一块内存区域不得同时具有写权限和执行权限，从而阻止控制流劫持中的代码注入攻击。针对性的防护措施具有额外开销小的优点，通常这类措施只会带来大约10%或更低的额外开销。但是针对性措施也有其缺点，那就是只能防止一种类型的攻击，这就导致这类防御措施可以被绕开。

总的来说，内存破坏型攻击的防御措施有这样的特点：能够提供强有力安全保障的解决方案会拖慢系统速度，而效率更高的解决方案则只能提供有限的防护。

# 攻击类型

为了应对内存破坏攻击，我们首先需要了解内存破坏攻击是如何引发并利用内存错误的。下图是内存破坏攻击的一般攻击模型，其中每个矩形框中表示的是攻击中的一步，最下面一行的椭圆框中表示的是具体的攻击种类，菱形框则表示在具体攻击方式中的选择。



图 1 内存破坏攻击的攻击模型

## 制造内存错误

内存破坏攻击首先需要制造内存错误，即图1中的最初两步。第一步中攻击者将制造一个无效的指针，第二步中攻击者将使用这个指针进行读写，即“解引用”（dereference），从而触发一个内存错误。制造无效指针的方法有两大类，一类是从空间上令指针无效，也就是制造一个越界的指针；另一类是让指针在时间上无效，例如解引用一个指向被删除对象的指针，这样的指针称为野指针。解引用越界指针所引发的内存错误称为“空间错误”，而由野指针引发的内存错误则称为“时间错误”。

空间错误方面，攻击者可以利用多种程序漏洞来制造越界指针。例如，攻击者可以触发一次指针的分配失败，如果这个失败事件因为程序漏洞而没有受到检查，那么攻击者就得到了一个空指针，并可以进一步利用这个空指针在内核空间中展开攻击{Kemerlis, 2012 #158}。另一种常见的制造越界指针的方法是在循环中不断地增加或减小数组指针的值，如果程序中没有进行边界检查，或者边界检查不完整，那么攻击者就得到了一个超出数组边界的指针，这种方法称为缓冲区上溢出和缓冲区下溢出。攻击者还可以通过整数的溢出、舍入错误、符号错误或者错误的类型转换来制造非法的数组下标，从而引发一个指针越界访问。最后，攻击者还可以利用已经触发的内存错误来制造更多的越界指针，从而扩大攻击的规模，如图1中向后的循环所示。

时间错误方面，常见的制造野指针的方法是利用异常处理程序或消息处理程序。例如在2011年发现的一个CVE中[CVE, 2011 #164]，攻击者可以改写“226 Transfer Complete”消息，从而在服务器的消息处理程序中触发一个漏洞。这个漏洞会导致指向应答消息的指针多于预期，在应答消息占用的空间被释放后，将会存在未被重新初始化的指针，利用这一指针攻击者可以在服务器上远程执行代码。在这个例子里，攻击者在对象已经被删除后再次“使用”了对象。由于这个特点，时间错误也被称为“use-after-free”漏洞，因为攻击者在野指针指向的内存区域已经被释放后对指针进行了解引用。

制造出无效的指针后，攻击者需要对该指针进行读或写操作来触发内存错误，这是图1中的第二步。无论是越界指针还是野指针，他们能够制造出的内存错误是相似的，能够进行的攻击也是相似的。只有一点限制，那就是对于野指针来说，已经被释放的内存对象所占用的空间需要被新的内存对象重新使用。

**（1）读无效指针**

当程序使用一个攻击者所制造的无效指针进行读操作时，所读取到的数据可能是由攻击者所控制的数据，也可能是系统中的敏感数据，这取决于攻击者将要实行什么样的攻击。例如，考虑下面一段代码，攻击者可以在user\_input中制造一个越界指针，这个指针指向了位于jump\_table之外的代码片段，通过这种方式，攻击者改变了程序正常的控制流，使程序执行自己所希望的代码片段。

|  |
| --- |
| func\_ptr jump\_table[3] = {fn\_0, fn\_1, fn\_2};  jump\_table[user\_input](); |

攻击者也可以通过无效指针来直接读取敏感数据，从而直接泄露系统信息。一个典型的例子是利用printf函数的格式化字符串所进行的格式化字符串攻击。假如printf函数的格式字符串被攻击者所控制，那么攻击者可以构造一个特殊的格式化字符串，使得printf函数在根据格式化字符串构造参数表的过程中制造出无效的指针。举一个简单的例子，在下面的printf用法中，如果用户输入的是”%x”，那么printf就会将栈中的内容打印出来。这样攻击者便利用越界指针直接获取了程序栈中的内容。另外，在这种攻击方式中，攻击者也可以利用格式化字符串中的%n参数来对程序中的变量内容直接进行修改。

|  |
| --- |
| printf(user\_input); |

另一种通过读取无效指针来进行攻击的方式是制造一个野指针，这个指针曾经指向的对象类型是A，在原来的内存对象被释放，内存空间重新分配后，这个指针指向一个由攻击者所控制的B类型对象。当程序按照原本的A类型对象调用虚函数时，查找的虚函数表内容其实是攻击者控制的新对象内容，这样一来，攻击者就可以通过操纵新对象内容来破坏虚函数的调用过程。另一种情况是，在重新分配的内存空间中包含了敏感数据，这时就可以利用A类型的指针直接对敏感数据区域进行读取，从而导致信息泄露。

**（2）写无效指针**

如果攻击者利用无效指针进行写操作的话，那么内存中的任何数据都有可能被攻击者修改，这包括了程序中的其他指针甚至程序代码本身。

这类内存错误的常见应用是利用缓冲区溢出攻击，或者数组下标类的漏洞来改写程序中的敏感数据，例如函数返回地址或者虚函数表。攻击者可以首先制造一次缓冲区溢出，越过数据的边界来破坏临近的一个虚函数表，那么接下来当程序使用这个虚函数表时，它将会使用一个被攻击者伪造的虚函数指针。由此可以看出，当攻击者制造了一次内存错误之后，他可以利用这次内存错误制造更多的的内存错误，从而更进一步地扩大攻击的范围。

另一类常见的攻击方法是“Double-free”，这种攻击中攻击者对已经被释放的野指针再次调用free()函数。在Double-free攻击中，攻击者会申请两块连续的内存空间，再分别释放掉以制造出一个野指针，接下来利用缓冲区溢出或其他方法向已经被释放的内存区域写入伪造的链表信息，使操作系统的内存管理程序认为该处的内存区域正在使用。这样一来，在对野指针再次调用free()函数时，攻击者保存在伪造链表信息中的代码指针就会覆盖掉程序中原有的代码指针，从而使得攻击者的代码可以得到执行。例如，可以用指向shellcode的代码指针覆盖程序退出时调用的exit()函数指针，这样在程序退出时就会执行攻击者插入的shellcode。

总的来说，攻击者能够利用内存错误对程序的内存状态进行非法的读取或修改。由之前几种攻击方式的例子可以看出，无论怎样组合无效指针类型和无效指针的使用方式，攻击者都可以获取程序内部的敏感数据、对程序状态进行修改或者利用已经触发的内存错误制造更多的无效指针。前文提到的制造与利用无效指针的方法其根源都是利用了程序在指针管理上的漏洞，例如没有进行边界检查，野指针未清空等等。由于C和C++程序中，这类工作全部由程序员来负责，所以在这些程序中此类漏洞非常常见。

本节所描述的内存错误都违反了内存安全的性质，这一性质要求程序运行过程中不出现任何的空间错误（越界指针）和时间错误（野指针）。C和C++程序本质上是内存不安全的。根据C和C++的标准，越过数组边界的写操作、解引用空指针，或者读取未初始化的变量会导致未定义的行为，因此，C和C++是无法在标准上避免出现内存错误的。与之类似，需要程序员直接管理内存的汇编语言存在内存错误的隐患。汇编语言、C或者C++只是类型不安全语言中的代表，但由这些语言编写的代码数量已经非常庞大，从中找出所有的程序漏洞是不现实的，更不用说修复所有的漏洞。因此，我们需要额外的机制来确保现有的程序能够遵守内存安全的性质，或者能够在图1中后面所展示的攻击过程中阻止攻击者得逞。本文接下来将更具体地讨论各种攻击的形式。

## 代码破坏攻击

攻击者如果想要改变程序的执行行为，那么最直接的方法就是用内存错误来改写内存中的程序代码。代码一致性要求程序代码不能被修改，如果程序能够保证代码一致性的性质，那么就可以避免直接的代码破坏攻击。

为了满足代码一致性的性质，可以将所有保存有代码的内存区域访问权限限制为不可写。现代处理器所使用的页式内存管理方式允许将所有包含代码的页设置为只读，从而满足代码一致性的性质。但是页式内存管理的权限设置单位是内存页，而程序代码并不能准确地填满内存页，对内存页中剩余空间的利用便成了问题。

除此之外，代码一致性并不能适用于自修改类的代码，以及即时编译（Just-In-Time Compilation）类型的程序。现在几乎所有的网络浏览器中都会配备一个JIT编译器用来支持JavaScript和Flash程序。为了支持这些程序，JIT编译器通常会提供一个指令缓存，在这部分内存区域中生成将要执行的代码，这就使得这部分内存区域有一段时间需要既可写又可执行，因此为攻击者提供了攻击的时间窗口。由于JIT程序的广泛存在，程序员无法完全确保代码一致性的性质。

## 控制流劫持攻击

相比直接修改代码的代码破坏攻击，攻击者在更多情况下是利用内存破坏漏洞来影响程序的控制流，这就是控制流劫持攻击。假如系统已经保证了代码完整性的性质，攻击者无法直接修改程序代码，但是程序进行间接转移时使用的代码指针不属于程序代码的一部分，因此这一影响程序控制流方向的关键数据没有受到代码一致性的保护。这就使得攻击者可以通过破坏代码指针的方式来操纵程序的行为，正因如此我们需要的不仅仅是代码一致性。

如图1中所示，控制流劫持攻击可以分为三步：修改代码指针、加载代码指针来劫持程序控制流以及最后的攻击负载的执行。其中修改代码指针这一步可以进一步细分为两个部分，其一是找到将要破坏的指针位置，其二是找到攻击负载代码所处的位置并正确地覆盖指针原内容。之所以这样分开是因为现代的操作系统中大多采用了地址空间随机化（Address Space Randomization），这使得攻击者无法简单地找到攻击负载代码所处的位置，而如果这个位置无法确定，那么即使代码指针被破坏，攻击者也很难发起有意义的攻击。

假如攻击者成功地将代码指针覆盖成了负载代码入口位置，那么下一步就是让处理器加载并执行这一代码指针。只有通过间接转移指令，被破坏的代码指针才能影响到处理器的PC寄存器，因此一种防范控制流劫持攻击的方法是针对所有的间接转移指令进行检查来发现是否有控制流劫持攻击的发生。

控制流劫持攻击的最后一部是执行攻击者所期望的恶意代码，也就是攻击负载。典型的控制流劫持攻击中，攻击者会首先向内存中注入一段恶意代码，例如shellcode。Shellcode是一段在执行后会打开一个shell终端的代码，通过将程序控制流导向shellcode，攻击者可以打开一个终端，从而几乎完全控制受攻击的主机。这种向内存中注入代码的攻击被称为代码注入，这一类攻击可以通过保证“数据不可执行”的性质来阻止。如果我们将数据不可执行性质与代码完整性性质相结合，我们就可以得到性质[van de Ven, 2004 #165]。要求一个内存页面要么是可写的，要么是可执行的，但两者不能同时成立，即可写与可执行是一个异或的关系。现代的内存管理系统只需要对内存页的权限标记进行少量的检查就可以很方便地确保这一性质，因此是一种非常实用的内存破坏攻击防范手段。但是与代码完整性类似，在包含自修改代码的程序和需要JIT编译的程序中无法得到保证，考虑到网页浏览器中JIT程序的广泛性，即使内存管理系统提供了性质的支持，整个系统中仍然给攻击者留下了可以攻击的空间。另一种避免攻击者进行代码注入攻击的方式是对指令本身进行加密，这就是指令系统随机化（Instruction Set Randomization）。ISR将程序中的指令进行加密，如果攻击者不知道正确的密钥的话，那么他注入的指令就无法被处理器正确地执行。但是相比于方式，ISR将会在指令执行时引入额外的解密开销，这对ISR的应用是一个巨大的阻碍。

除了JIT程序这种性质的盲点，攻击者还可以将内存中已经存在的代码。攻击者可以利用已有的完整函数，或者是将内存中的代码片段连接在一起形成一段有意义的攻击代码。最常见的连接方法是利用返回指令将不同位置的片段连接在一起，正因如此，这种攻击方法被称为返回导向编程（Return Oriented Programming）。由于返回指令是间接转移中的一种，ROP也可以更一般化，不只使用返回指令，也利用其他的间接转移指令来将各处的代码片段连接在一起，这种更一般化的方式被称为转移导向编程（Jump Oriented Programming）。JOP和ROP所能够利用的代码片段不仅仅包括程序中原有的一段指令序列，在x86等变长指令系统中，攻击者还可以将攻击负载的入口设定在原有指令的中间部分，这种情况下处理器将把原有的二进制序列译码为另外的指令，攻击者可以利用这种方式来构造出不存在于原始程序中的指令。

JOP和ROP所利用的都是内存中原本存在的合法代码片段，因此如果攻击者已经成功地将PC寄存器替换为攻击代码的入口的话，那么后续的攻击将很难阻止。可以阻止代码重用攻击的位置有两处，一处是对间接转移指令进行检测，防止攻击者劫持控制流；另一处则是尝试消除掉攻击者可以利用的代码片段。G-Free[Onarlioglu, 2010 #166]是一个在编译阶段消除可利用代码片段的方案，G-Free向在编译时向代码中插入空指令，使得无论从二进制代码的什么位置开始译码，在若干条指令内一定会重新按源代码的指令执行，这样就使攻击者很难利用x86指令系统的变长特性来创造自己需要的指令。除了在编译时对程序作出修改外，还可以利用二进制代码重写的方式来消除可利用的代码片段，例如[Keromytis, 2012 #167]可以离线地对应用所需的库文件进行分析，产生一个随机化之后的库文件池，这样经过相应修改的系统服务或者加载器就可以在应用每一次启动时加载一个不同的库文件，从而减少攻击者成功攻击的可能性。不过需要注意的是，这些试图消除可利用代码片段的方法并不能消除所有的可利用代码片段，只能让JOP和ROP攻击更难完成。另一方面，有研究证明即使不使用代码片段，利用完整库函数的return-to-libc攻击也能够做到图灵完备的计算[Tran, 2011 #168]，这使得消除可利用代码片段的方式看起来不那么吸引人。

## 数据导向攻击

## 信息泄露攻击

# 基于随机化的防御策略

<https://ieeexplore.ieee.org/document/6234437/> 随机化指令位置

# 内存安全策略

# 控制流完整性策略

# 参考文献

[1] RAINS T, MILLER M, WESTON D. Exploitation trends: From potential risk to actual risk; proceedings of the RSA Conference, F, 2015 [C].

[2] SZEKERES L, PAYER M, WEI T, et al. SoK: Eternal War in Memory; proceedings of the 2013 IEEE Symposium on Security and Privacy, F 19-22 May 2013, 2013 [C].