# 二进制炸弹与缓冲区总结

## 考察的知识点：

**内存布局:**

**32位虚拟内存布局**

在32位模式下虚拟地址空间总是一个4GB的内存地址块。这些虚拟地址通过页表（page table）映射到物理内存，页表由操作系统维护并被处理器引用。每一个进程拥有一套属于它自己的页表，但是还有一个隐情。只要虚拟地址被使用，那么它就会作用于这台机器上运行的所有软件，包括内核本身。因此一部分虚拟地址必须保留给内核使用：

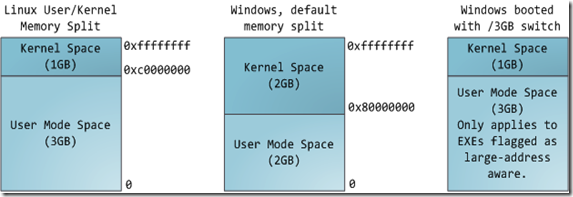


图 1

这并不意味着内核使用了那么多的物理内存，仅表示它可支配这么大的地址空间，可根据内核需要，将其映射到物理内存。内核空间在页表中拥有较高的特权级（ring 2或以下），因此只要用户态的程序试图访问这些页，就会导致一个页错误（page fault），用户程序不可访问内核页。在Linux中，内核空间是持续存在的，并且在所有进程中都映射到同样的物理内存。内核代码和数据总是可寻址的，随时准备处理中断和系统调用。与此相反，用户模式地址空间的映射随进程切换的发生而不断变化：

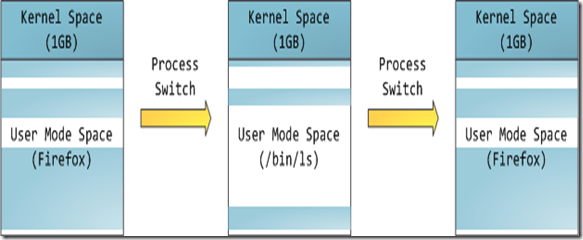


图 2

图2中，蓝色区域表示映射到物理内存的虚拟地址，而白色区域表示未映射的部分。在上面的例子中，Firefox使用了相当多的虚拟地址空间，因为它是传说中的吃内存大户。地址空间中的各个条带对应于不同的内存段（memory segment），如：堆、栈之类的。记住，这些段只是简单的内存地址范围，与Intel处理器的段没有关系。

**32位经典内存布局**

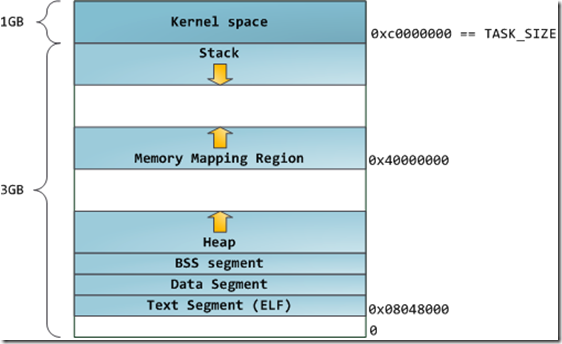


图 3

32位经典内存布局，程序起始1GB地址为内核空间，接下来是向下增长的栈空间和由0×40000000向上增长的mmap地址。而堆地址是从底部开始，去除ELF、代码段、数据段、常量段之后的地址并向上增长。但是这种布局有几个问题，首先是容易遭受溢出攻击；其次是，堆地址空间只有不到1G有木有？如果mmap内存比较少地址很浪费有木有？所以后来就有了另一种内存布局

**32位默认内存布局**

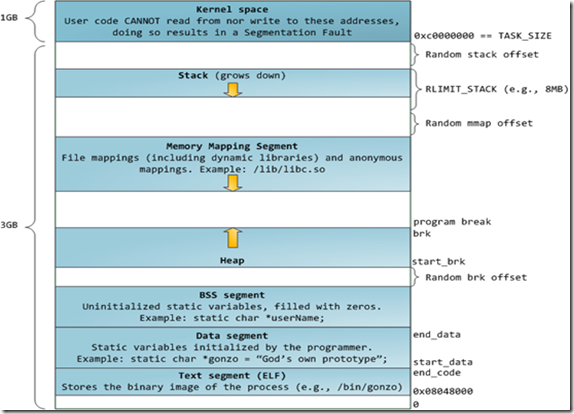


图 4

当计算机开心、安全、可爱、正常的运转时，几乎每一个进程的各个段的起始虚拟地址都与图4完全一致，这也给远程发掘程序安全漏洞打开了方便之门。一个发掘过程往往需要引用绝对内存地址：栈地址，库函数地址等。远程攻击者必须依赖地址空间布局的一致性，摸索着选择这些地址。如果让他们猜个正着，有人就会被整了。因此，地址空间的随机排布方式逐渐流行起来。Linux通过对栈、内存映射段、堆的起始地址加上随机的偏移量来打乱布局。不幸的是，32位地址空间相当紧凑，给随机化所留下的空当不大，削弱了这种技巧的效果。

**栈**

进程地址空间中最顶部的段是栈，大多数编程语言将之用于存储局部变量和函数参数。调用一个方法或函数会将一个新的栈桢（stack frame）压入栈中。栈桢在函数返回时被清理。也许是因为数据严格的遵从LIFO的顺序，这个简单的设计意味着不必使用复杂的数据结构来追踪栈的内容，只需要一个简单的指针指向栈的顶端即可。因此压栈（pushing）和退栈（popping）过程非常迅速、准确。另外，持续的重用栈空间有助于使活跃的栈内存保持在CPU缓存中，从而加速访问。进程中的每一个线程都有属于自己的栈。

通过不断向栈中压入的数据，超出其容量就有会耗尽栈所对应的内存区域。这将触发一个页故障（page fault），并被Linux的expand\_stack()处理，它会调用acct\_stack\_growth()来检查是否还有合适的地方用于栈的增长。如果栈的大小低于RLIMIT\_STACK（通常是8MB），那么一般情况下栈会被加长，程序继续愉快的运行，感觉不到发生了什么事情。这是一种将栈扩展至所需大小的常规机制。然而，如果达到了最大的栈空间大小，就会栈溢出（stack overflow），程序收到一个段错误（Segmentation Fault）。当映射了的栈区域扩展到所需的大小后，它就不会再收缩回去，即使栈不那么满了。这就好比联邦预算，它总是在增长的。

动态栈增长是唯一一种访问未映射内存区域（图中白色区域）而被允许的情形。其它任何对未映射内存区域的访问都会触发页故障，从而导致段错误。一些被映射的区域是只读的，因此企图写这些区域也会导致段错误。

**内存映射段**

在栈的下方，是我们的内存映射段。此处，内核将文件的内容直接映射到内存。任何应用程序都可以通过Linux的mmap()系统调用（实现）或Windows的CreateFileMapping() / MapViewOfFile()请求这种映射。内存映射是一种方便高效的文件I/O方式，所以它被用于加载动态库。创建一个不对应于任何文件的匿名内存映射也是可能的，此方法用于存放程序的数据。在Linux中，如果你通过malloc()请求一大块内存，C运行库将会创建这样一个匿名映射而不是使用堆内存。‘大块’意味着比MMAP\_THRESHOLD还大，缺省是128KB，可以通过mallopt()调整。

**堆**

说到堆，它是接下来的一块地址空间。与栈一样，堆用于运行时内存分配；但不同点是，堆用于存储那些生存期与函数调用无关的数据。大部分语言都提供了堆管理功能。因此，满足内存请求就成了语言运行时库及内核共同的任务。在C语言中，堆分配的接口是malloc()系列函数，而在具有垃圾收集功能的语言（如C#）中，此接口是new关键字。

如果堆中有足够的空间来满足内存请求，它就可以被语言运行时库处理而不需要内核参与。否则，堆会被扩大，通过brk()系统调用（实现）来分配请求所需的内存块。堆管理是很复杂的，需要精细的算法，应付我们程序中杂乱的分配模式，优化速度和内存使用效率。处理一个堆请求所需的时间会大幅度的变动。实时系统通过特殊目的分配器来解决这个问题。堆也可能会变得零零碎碎，如下图所示：



图 5

**BSS 数据段 代码段**

最后，我们来看看最底部的内存段：BSS，数据段，代码段。在C语言中，BSS和数据段保存的都是静态（全局）变量的内容。区别在于BSS保存的是未被初始化的静态变量内容，它们的值不是直接在程序的源代码中设定的。BSS内存区域是匿名的：它不映射到任何文件。如果你写static int cntActiveUsers，则cntActiveUsers的内容就会保存在BSS中。

另一方面，数据段保存在源代码中已经初始化了的静态变量内容。这个内存区域不是匿名的。它映射了一部分的程序二进制镜像，也就是源代码中指定了初始值的静态变量。所以，如果你写static int cntWorkerBees = 10，则cntWorkerBees的内容就保存在数据段中了，而且初始值为10。尽管数据段映射了一个文件，但它是一个私有内存映射，这意味着更改此处的内存不会影响到被映射的文件。也必须如此，否则给全局变量赋值将会改动你硬盘上的二进制镜像，这是不可想象的。

下图中数据段的例子更加复杂，因为它用了一个指针。在此情况下，指针gonzo（4字节内存地址）本身的值保存在数据段中。而它所指向的实际字符串则不在这里。这个字符串保存在代码段中，代码段是只读的，保存了你全部的代码外加零零碎碎的东西，比如字符串字面值。代码段将你的二进制文件也映射到了内存中，但对此区域的写操作都会使你的程序收到段错误。这有助于防范指针错误，虽然不像在C语言编程时就注意防范来得那么有效。下图展示了这些段以及我们例子中的变量：

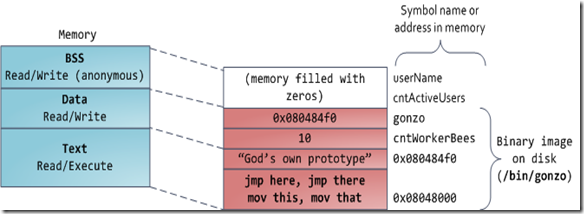


图 6

你可以通过阅读文件/proc/pid\_of\_process/maps来检验一个Linux进程中的内存区域。记住一个段可能包含许多区域。比如，每个内存映射文件在mmap段中都有属于自己的区域，动态库拥有类似BSS和数据段的额外区域。下一篇文章讲说明这些“区域”（area）的真正含义。有时人们提到“数据段”，指的就是全部的数据段+ BSS + 堆。

**汇编指令**

IA32机器码以及汇编代码都与原始的C代码有很大不同，因为一些状态对于C程序员来说是隐藏的。例如包含下一条要执行代码的内存位置的程序指针(program counter or PC)以及8个寄存器。还要注意的一点是：汇编代码的ATT格式和Intel格式。ATT格式是GCC和objdump等工具的默认格式，在CSAPP中一律使用这种格式。而Intel格式则通常会在Intel的IA32架构文档以及微软的Windows技术文档中碰到。两者的主要区别有：

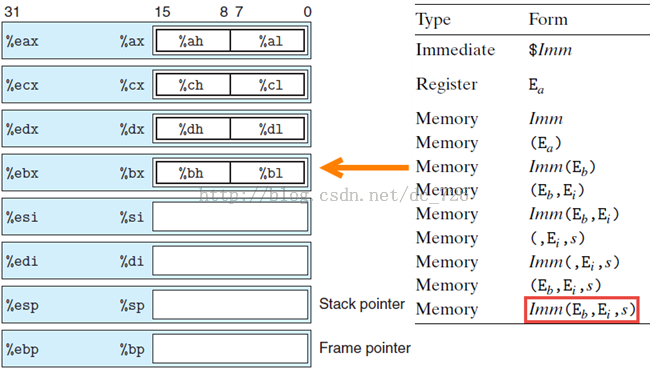
* Intel格式忽略指令中暗示操作数长度的后缀，例如mov而不是ATT格式的movl。
* Intel格式忽略寄存器名称前的%，例如esp而不是ATT格式的%esp。
* Intel格式用不同的方式描述内存位置，例如DWORD PTR [ebp+8]而不是ATT格式的8(%ebp)。
* Intel格式指令的操作数顺序与ATT格式的完全相反，ATT格式总是最后一个操作数是目标，例如movl %eax, (%edx)。

此外，作为16位处理器架构的遗留产物，如今的指令依旧用word指2个字节16位，而用double word指4个字节。所以指令中通常使用B、W、L表示操作数是1、2、4个字节的指令，例如数据移动指令的三个版本movb、movw、movl。

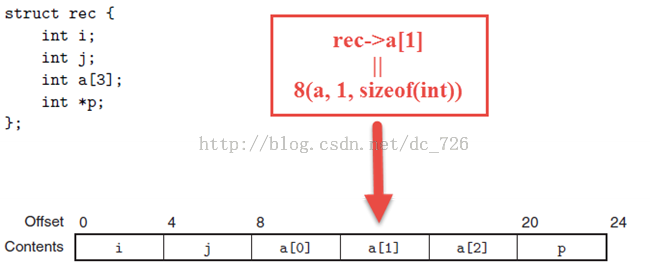
为什么逆向工程很难？因为源代码与编译后的代码往往不是一一对应的。编译器会引入源代码中不存在的新变量，同时为了节约寄存器的使用，编译器也经常将多个值映射到一个寄存器。对于循环来说，通过观察寄存器是如何在循环前初始化，在循环内的更新和条件检测以及循环后的使用，能够得到一些线索。

**寄存器与寻址**

程序执行的很大一部分时间都是在将数据挪来挪去的。所以处理器支持只使用寄存器的1、2、4个字节，同时并且支持多种寻址方式。如下图右半边的表格中所示，这样我们就可以灵活地从内存中加载数据到寄存器，或者将寄存器中的值保存到内存。

****

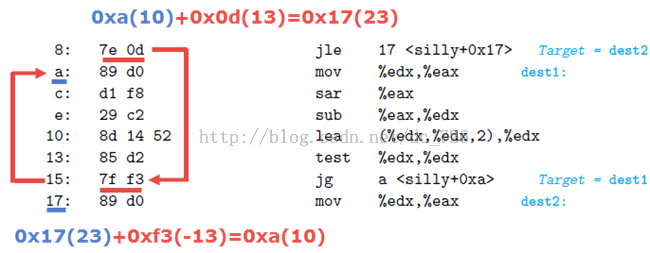
虽然看起来有些眼花缭乱，但实际上最基本的形式就是最后一种：Imm(Eb, Ei, s)=Imm+R[Eb]+R[Ei]\*s (R[X]指寄存器X的值)。一共四个参数控制寻址，看起来有些过于灵活，那就让我们想象一下它的应用场景。先不考虑Imm，那么最典型的应用就是访问数组中某个数据项。假如数组为int x[4]，则此时Eb就是数组的首地址，相当于x，而Ei就是要访问数据项的下标，而s就是数组中数据类型的长度。例如我们要访问x[3]，那么就相当于(x, 3, sizeof(int))=x+3\*4。用C语言来写就是\*(x+3)，因为C语言自动按照指针的类型长度进行移动(编译器自动生成正确的代码)，所以我们并不用自己计算偏移量乘以sizeof(int)，但这都是后话了。那再加上Imm又能有何种应用场景，其实很简单，就是访问struct中的数组中的某一项。如下图所示，直接一条指令就能访问到结构中的数组中的某一项。

****

**常用指令**

下面是一些最常见的汇编指令及其含义：

* mov：数据移动。IA32强加了一条限制：一条移动指令的两个操作数不能都是内存地址。所以从一个内存位置拷贝数据到另一个内存位置是需要两条指令的。
* leal：加载地址。效果就是mov Imm(%a, %b, s), %x会将%x赋值为Imm+%a+s\*%b，而不是M[Imm+%a+s\*%b]，所以有两个很有用的场景：1）拷贝地址。例如int \*x=a汇编为mov (%eax), %edx，那么int x=&a汇编为leal (%eax), %edx。所以leal不会真的将a的值(即(%eax))保存到x(即%edx)，而只是将a的地址(其实就是%eax)保存到x。2）简单算术运算。第二个很自然会想到的应用就是使用leal一条指令压缩简单的算术运算，例如leal 7(%edx, %edx, 4)=5x+7。
* jmp：直接跳转到标签，或间接跳转到寄存器中指定的地址。对于直接跳转，在汇编语言中通常就是符号化的标签表示。但之后汇编器或链接器要对其进行编码，最常见的编码方式就是PC相对地址。即用1、2、4字节的偏移量表示跳转目标地址与jmp指令紧接着的下一条指令的地址，如下图所示。但为什么是紧接着jmp指令的下一条指令的地址而不是jmp这一条的？其实也是有历史原因的，因为早期的处理器实现是先更新PC计数器作为第一步，然后再执行当前指令的。所以指令在执行的时候，其实PC已经指向下一条指令了，因此跳转的偏移量也就要相对下一条指令来说了。

****

**局部变量其实就在寄存器里**

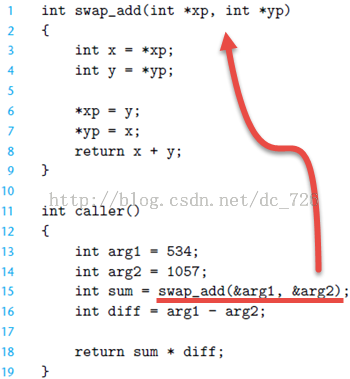
其实局部变量是直接存储在寄存器的，大部分情况下都会一直在寄存器中，而不会落地到内存。例如函数swap\_add()，函数运行时栈帧(内存)实际上没有保存任何局部变量。整个函数的局部变量和逻辑都在寄存器和ALU中执行完成。

在以下情况，局部变量会被保存在内存中(栈上)：

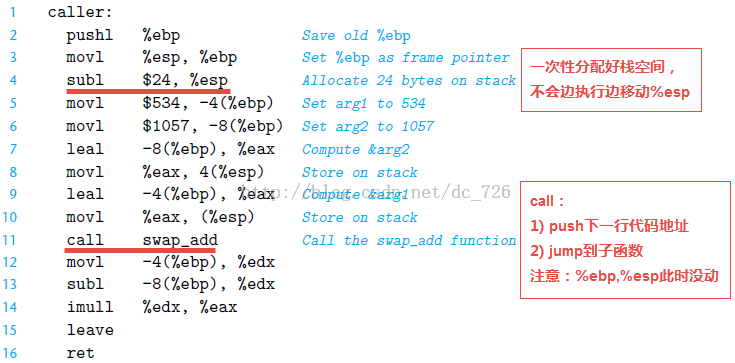
* 当没有足够的寄存器来保存所有局部变量时。毕竟寄存器只有八个。
* 一些局部变量是数组或struct，因此必须通过指针访问。
* 当对局部变量进行取地址&运算时，因此必须产生一个内存地址给它。

**运行时的代码与栈**

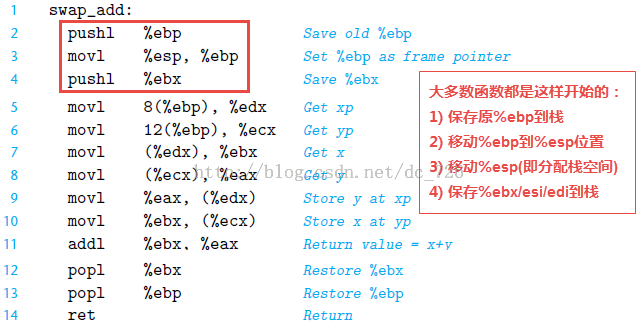
下面来看一个函数调用的例子，深入学习代码底层是如何运行的。



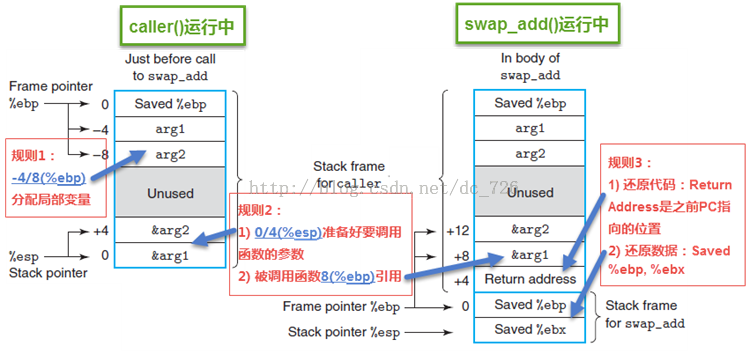
caller()代码如下：



swap\_add()代码如下：

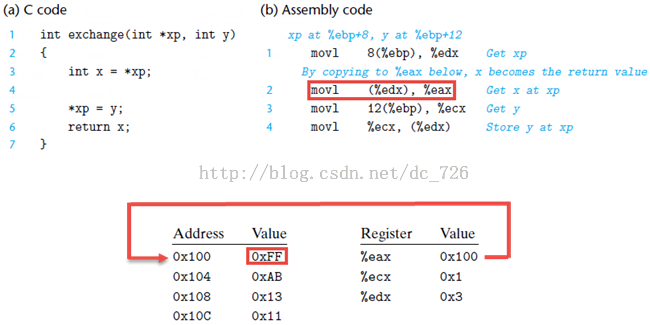


编译器生成的代码会遵守一定的规则，这样在执行各种跳转、函数调用时才不会发生数据覆盖等问题，从而使程序正确的运行。



**指针的本质**

也许之前也曾听过，指针本质上就是一个内存地址。但之前没有顿悟，现在通过研究底层知识来强化理解。从下图可以看出，指针取值实际上是一种很自然的操作，因为大多数时候我们没法在一个寄存器里放下一个变量表示的全部数据，例如数组或结构。如果寄存器能够放下整个数组和结构，那我们当然没必要用指针了。所以很自然地，我们就会先加载数据的首地址的内存地址(就是指针！)到寄存器，然后再去访问寄存器指向的内存位置。

****