手把手教你栈溢出从入门到放弃

6 days ago

二进制安全 admin

本文来源：长亭技术专栏
作者：Jwizard

0x00 写在最前面

开场白：快报快报！今天是 2017 Pwn2Own 黑客大赛的第一天，长亭安全研究实验室在比赛中攻破 Linux 操作系统和 Safari 浏览器（突破沙箱且拿到系统最高权限），积分 14 分，在 11 支队伍中暂居 Master of Pwn 第一名。作为热爱技术乐于分享的技术团队，我们开办了这个专栏，传播普及计算机安全的“黑魔法”，也会不时披露长亭安全实验室的最新研究成果。

安全领域博大精深，很多童鞋都感兴趣却苦于难以入门，不要紧，我们会从最基础的内容开始，循序渐进地讲给大家。技术长路漫漫，我们携手一起出发吧。

0x10 本期简介

在计算机安全领域，缓冲区溢出是个古老而经典的话题。众所周知，计算机程序的运行依赖于函数调用栈。栈溢出是指在栈内写入超出长度限制的数据，从而破坏程序运行甚至获得系统控制权的攻击手段。本文将以 32 位 x86 架构下的程序为例讲解栈溢出的技术详情。

为了实现栈溢出，要满足两个条件。第一，程序要有向栈内写入数据的行为；第二，程序并不限制写入数据的长度。历史上第一例被广泛注意的“莫里斯蠕虫”病毒就是利用 C 语言标准库的 gets() 函数并未限制输入数据长度的漏洞，从而实现了栈溢出。

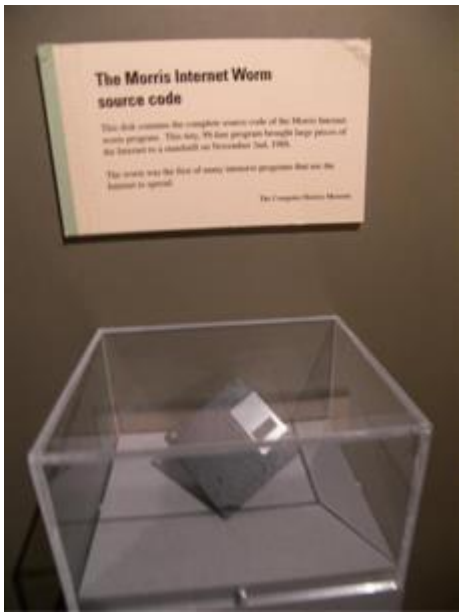

Fig 1. 波士顿科学博物馆保存的存有莫里斯蠕虫源代码的磁盘（source: Wikipedia）

如果想用栈溢出来执行攻击指令，就要在溢出数据内包含攻击指令的内容或地址，并且要将程序控制权交给该指令。攻击指令可以是自定义的指令片段，也可以利用系统内已有的函数及指令。

0x20 背景知识

在介绍如何实现溢出攻击之前，让我们先简单温习一下函数调用栈的相关知识。

函数调用栈是指程序运行时内存一段连续的区域，用来保存函数运行时的状态信息，包括函数参数与局部变量等。称之为“栈”是因为发生函数调用时，调用函数（caller）的状态被保存在栈内，被调用函数（callee）的状态被压入调用栈的栈顶；在函数调用结束时，栈顶的函数（callee）状态被弹出，栈顶恢复到调用函数（caller）的状态。函数调用栈在内存中从高地址向低地址生长，所以栈顶对应的内存地址在压栈时变小，退栈时变大。

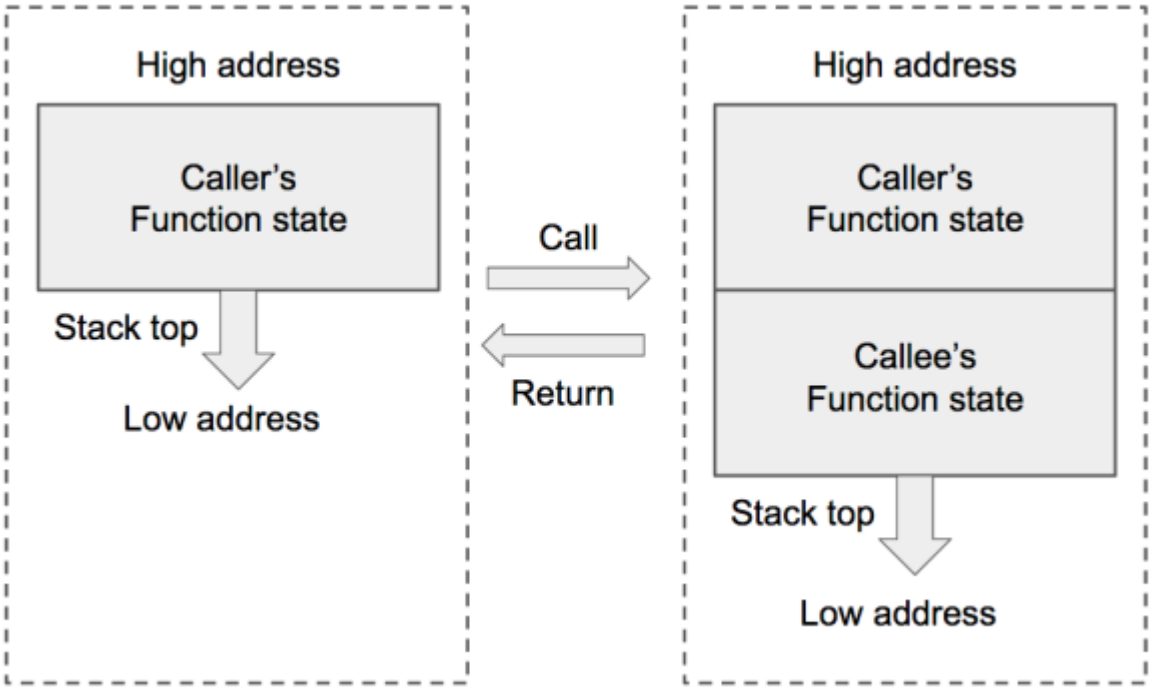

Fig 2. 函数调用发生和结束时调用栈的变化

函数状态主要涉及三个寄存器——esp，ebp，eip。esp 用来存储函数调用栈的栈顶地址，在压栈和退栈时发生变化。ebp 用来存储当前函数状态的基地址，在函数运行时不变，可以用来索引确定函数参数或局部变量的位置。eip 用来存储即将执行的程序指令的地址，cpu 依照 eip 的存储内容读取指令并执行，eip 随之指向相邻的下一条指令，如此反复，程序就得以连续执行指令。

下面让我们来看看发生函数调用时，栈顶函数状态以及上述寄存器的变化。变化的核心任务是将调用函数（caller）的状态保存起来，同时创建被调用函数（callee）的状态。

首先将被调用函数（callee）的参数按照逆序依次压入栈内。如果被调用函数（callee）不需要参数，则没有这一步骤。这些参数仍会保存在调用函数（caller）的函数状态内，之后压入栈内的数据都会作为被调用函数（callee）的函数状态来保存。

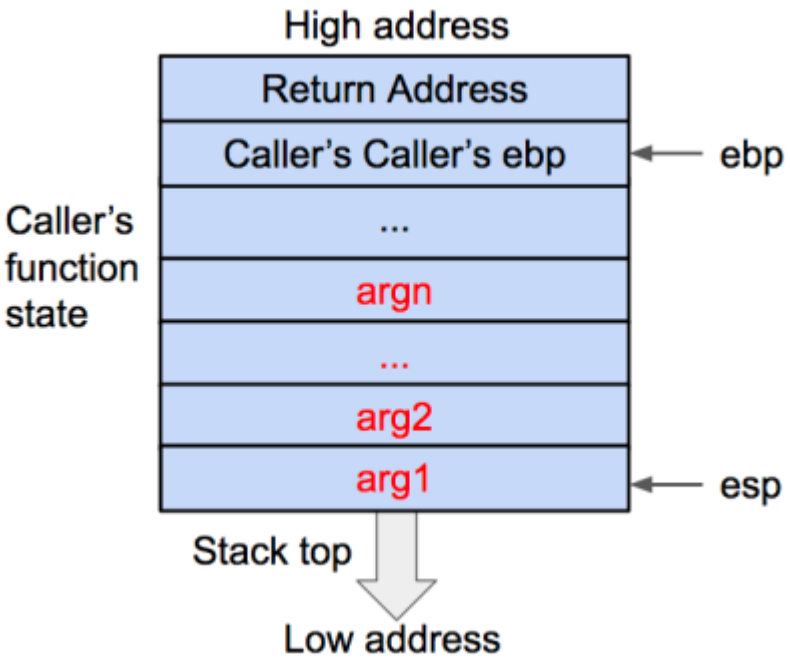

Fig 3. 将被调用函数的参数压入栈内

然后将调用函数（caller）进行调用之后的下一条指令地址作为返回地址压入栈内。这样调用函数（caller）的 eip（指令）信息得以保存。

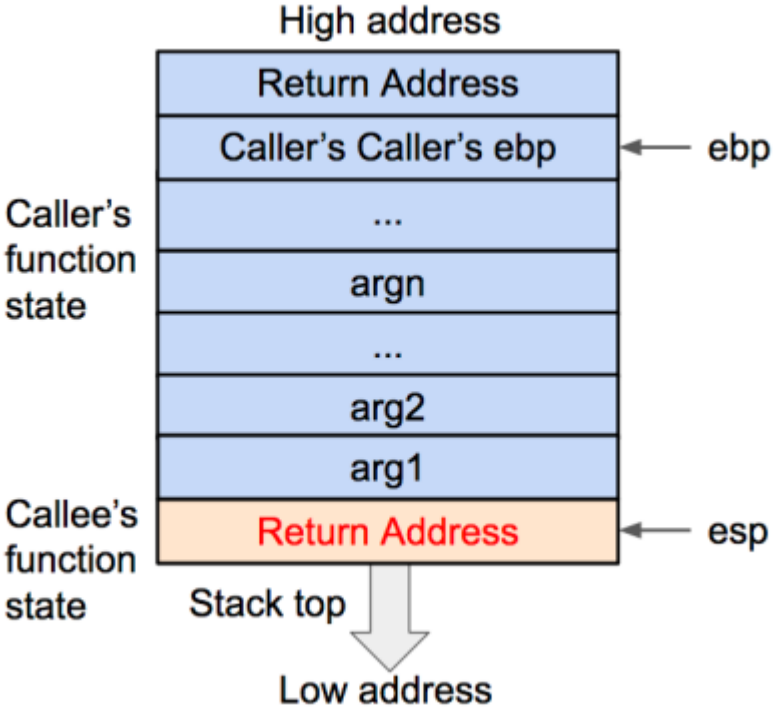

Fig 4. 将被调用函数的返回地址压入栈内

再将当前的 ebp 寄存器的值（也就是调用函数的基地址）压入栈内，并将 ebp 寄存器的值更新为当前栈顶的地址。这样调用函数（caller）的 ebp（基地址）信息得以保存。同时，ebp 被更新为被调用函数（callee）的基地址。

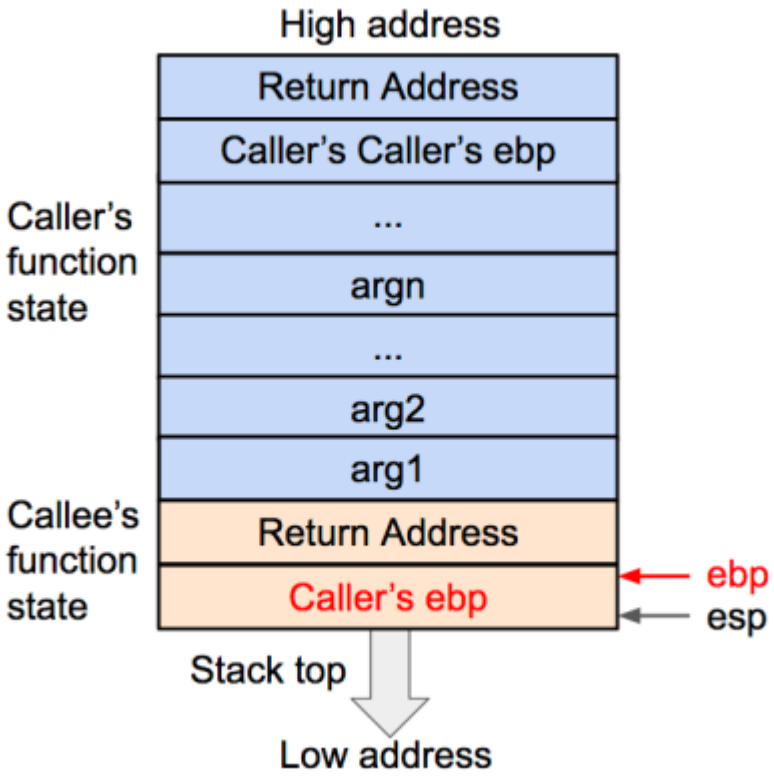

Fig 5. 将调用函数的基地址（ebp）压入栈内，并将当前栈顶地址传到 ebp 寄存器内

再之后是将被调用函数（callee）的局部变量等数据压入栈内。

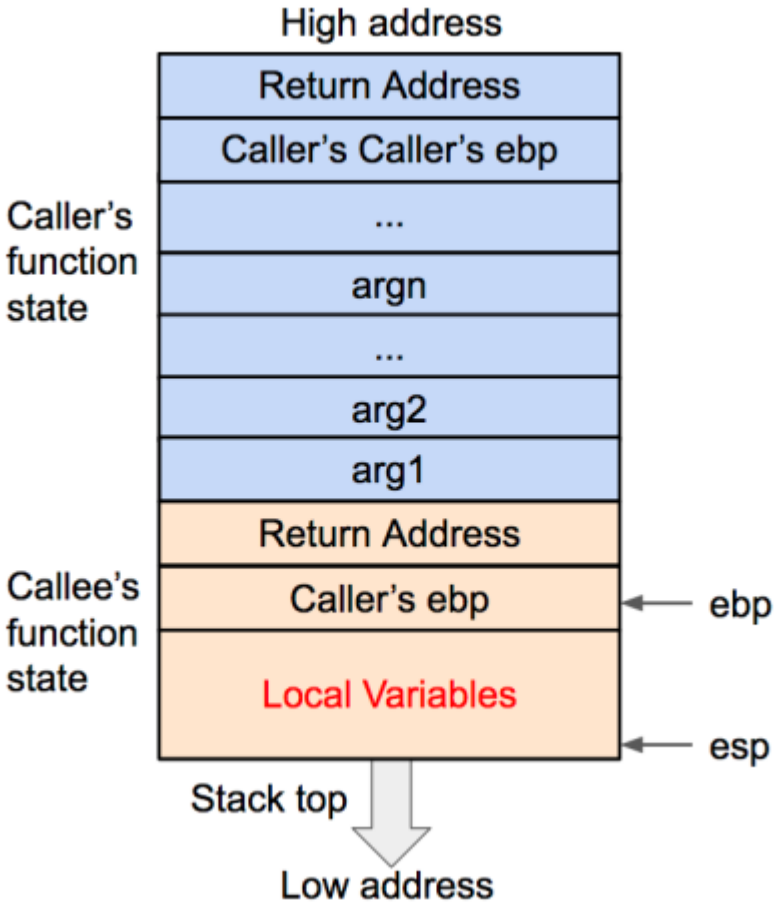

Fig 6. 将被调用函数的局部变量压入栈内

在压栈的过程中，esp 寄存器的值不断减小（对应于栈从内存高地址向低地址生长）。压入栈内的数据包括调用参数、返回地址、调用函数的基地址，以及局部变量，其中调用参数以外的数据共同构成了被调用函数（callee）的状态。在发生调用时，程序还会将被调用函数（callee）的指令地址存到 eip 寄存器内，这样程序就可以依次执行被调用函数的指令了。

看过了函数调用发生时的情况，就不难理解函数调用结束时的变化。变化的核心任务是丢弃被调用函数（callee）的状态，并将栈顶恢复为调用函数（caller）的状态。

首先被调用函数的局部变量会从栈内直接弹出，栈顶会指向被调用函数（callee）的基地址。

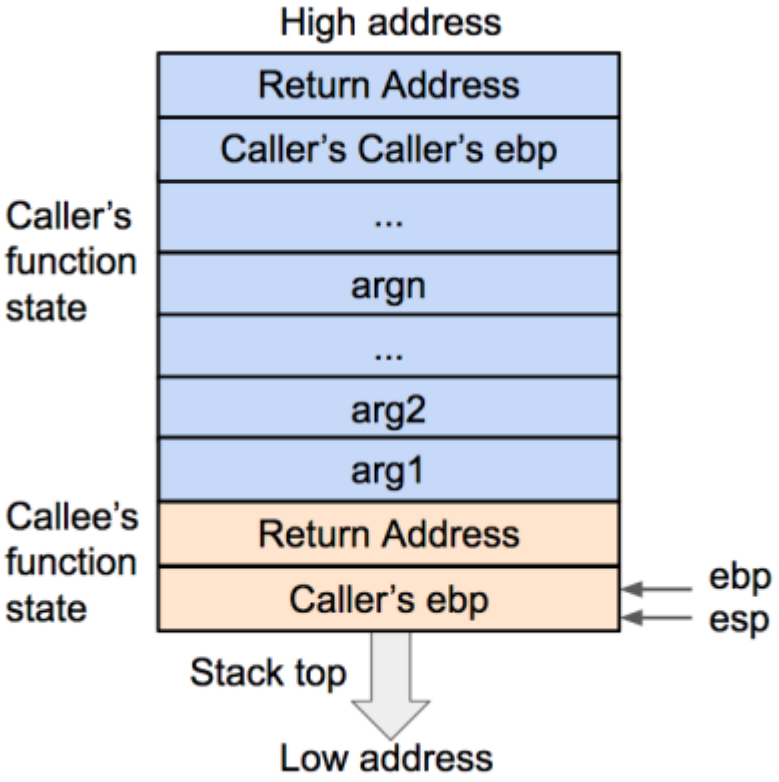

Fig 7. 将被调用函数的局部变量弹出栈外

然后将基地址内存储的调用函数（caller）的基地址从栈内弹出，并存到 ebp 寄存器内。这样调用函数（caller）的 ebp（基地址）信息得以恢复。此时栈顶会指向返回地址。

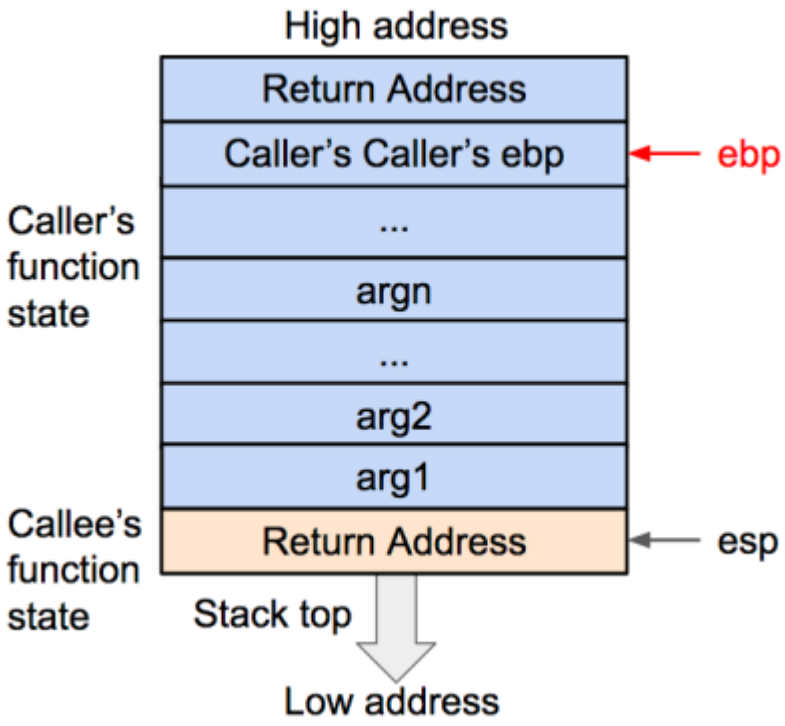

Fig 8. 将调用函数（caller）的基地址（ebp）弹出栈外，并存到 ebp 寄存器内

再将返回地址从栈内弹出，并存到 eip 寄存器内。这样调用函数（caller）的 eip（指令）信息得以恢复。

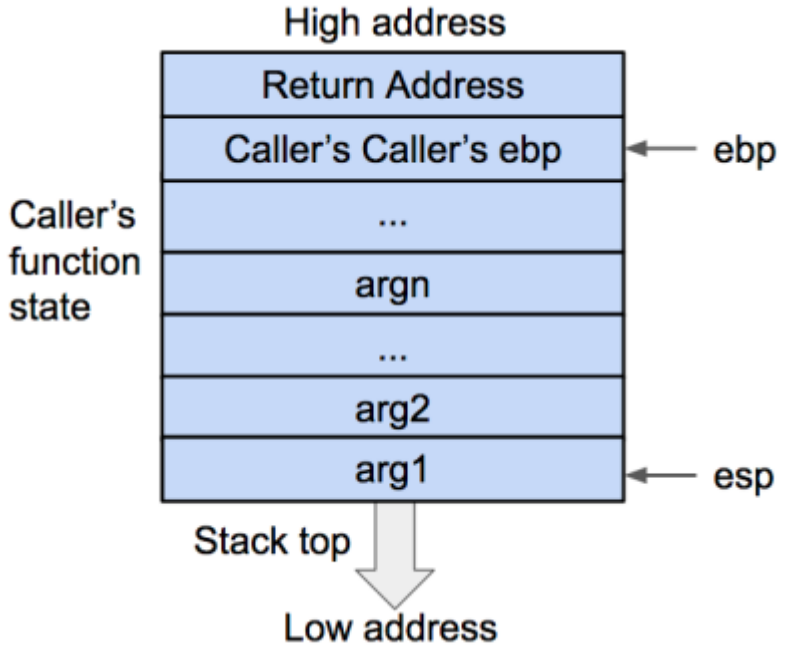

Fig 9. 将被调用函数的返回地址弹出栈外，并存到 eip 寄存器内

至此调用函数（caller）的函数状态就全部恢复了，之后就是继续执行调用函数的指令了。

0x30 技术清单

介绍完背景知识，就可以继续回归栈溢出攻击的主题了。当函数正在执行内部指令的过程中我们无法拿到程序的控制权，只有在发生函数调用或者结束函数调用时，程序的控制权会在函数状态之间发生跳转，这时才可以通过修改函数状态来实现攻击。而控制程序执行指令最关键的寄存器就是 eip（还记得 eip 的用途吗？），所以我们的目标就是让 eip 载入攻击指令的地址。

先来看看函数调用结束时，如果要让 eip 指向攻击指令，需要哪些准备？首先，在退栈过程中，返回地址会被传给 eip，所以我们只需要让溢出数据用攻击指令的地址来覆盖返回地址就可以了。其次，我们可以在溢出数据内包含一段攻击指令，也可以在内存其他位置寻找可用的攻击指令。

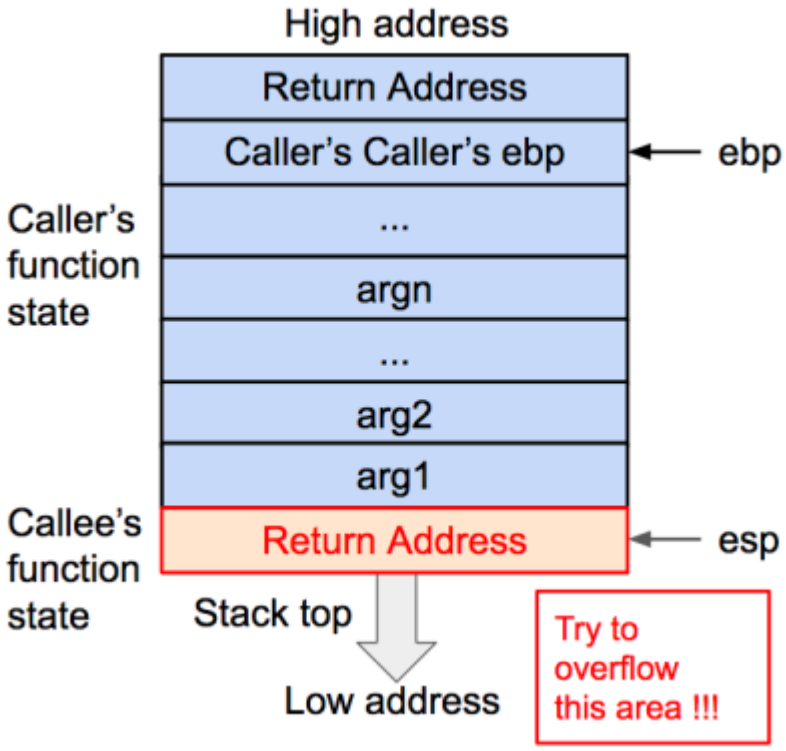

Fig 10. 核心目的是用攻击指令的地址来覆盖返回地址

再来看看函数调用发生时，如果要让 eip 指向攻击指令，需要哪些准备？这时，eip 会指向原程序中某个指定的函数，我们没法通过改写返回地址来控制了，不过我们可以“偷梁换柱”——将原本指定的函数在调用时替换为其他函数。

所以这篇文章会覆盖到的技术大概可以总结为（括号内英文是所用技术的简称）：

- 修改返回地址，让其指向溢出数据中的一段指令（shellcode）
- 修改返回地址，让其指向内存中已有的某个函数（return2libc）
- 修改返回地址，让其指向内存中已有的一段指令（ROP）
- 修改某个被调用函数的地址，让其指向另一个函数（hijack GOT）

本篇文章会覆盖前两项技术，后两项会在下篇继续介绍。（所以请点击“关注专栏”持续关注我们吧 ^_^ ）

0x40 Shellcode

——修改返回地址，让其指向溢出数据中的一段指令

根据上面副标题的说明，要完成的任务包括：在溢出数据内包含一段攻击指令，用攻击指令的起始地址覆盖掉返回地址。攻击指令一般都是用来打开shell，从而可以获得当前进程的控制权，所以这类指令片段也被成为“shellcode”。shellcode 可以用汇编语言来写再转成对应的机器码，也可以上网搜索直接复制粘贴，这里就不再赘述。下面我们先写出溢出数据的组成，再确定对应的各部分填充进去。

payload : padding1 + address of shellcode + padding2 + shellcode

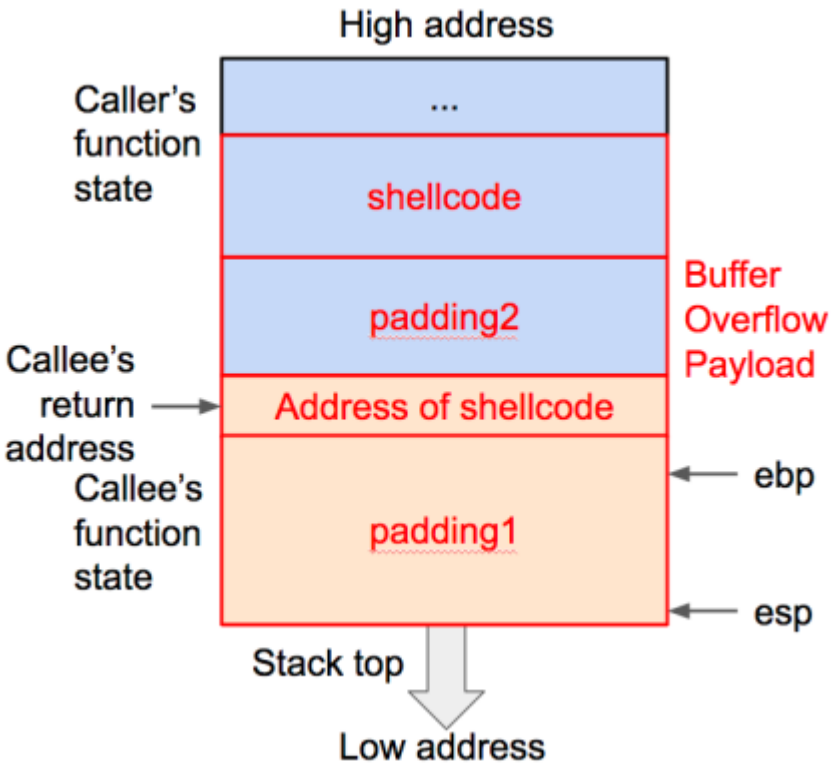

Fig 11. shellcode 所用溢出数据的构造

padding1 处的数据可以随意填充（注意如果利用字符串程序输入溢出数据不要包含 “\x00” ，否则向程序传入溢出数据时会造成截断），长度应该刚好覆盖函数的基地址。address of shellcode 是后面 shellcode 起始处的地址，用来覆盖返回地址。padding2 处的数据也可以随意填充，长度可以任意。shellcode 应该为十六进制的机器码格式。

根据上面的构造，我们要解决两个问题。

1. 返回地址之前的填充数据（padding1）应该多长？

我们可以用调试工具（例如 gdb）查看汇编代码来确定这个距离，也可以在运行程序时用不断增加输入长度的方法来试探（如果返回地址被无效地址例如“AAAA”覆盖，程序会终止并报错）。

2. shellcode 起始地址应该是多少？

我们可以在调试工具里查看返回地址的位置（可以查看 ebp 的内容然后再加 4（32 位机），参见前面关于函数状态的解释），可是在调试工具里的这个地址和正常运行时并不一致，这是运行时环境变量等因素有所不同造成的。所以这种情况下我们只能得到大致但不确切的 shellcode 起始地址，解决办法是在 padding2 里填充若干长度的 “\x90”。这个机器码对应的指令是 NOP (No

Operation)，也就是告诉 CPU 什么也不做，然后跳到下一条指令。有了这一段 NOP 的填充，只要返回地址能够命中这一段中的任意位置，都可以无副作用地跳转到 shellcode 的起始处，所以这种方法被称为 NOP Sled（中文含义是“滑雪橇”）。这样我们就可以通过增加 NOP 填充来配合试验 shellcode 起始地址。

操作系统可以将函数调用栈的起始地址设为随机化（这种技术被称为内存布局随机化，即 Address Space Layout Randomization (ASLR)），这样程序每次运行时函数返回地址会随机变化。反之如果操作系统关闭了上述的随机化（这是技术可以生效的前提），那么程序每次运行时函数返回地址会是相同的，这样我们可以通过输入无效的溢出数据来生成 core 文件，再通过调试工具在 core 文件中找到返回地址的位置，从而确定 shellcode 的起始地址。

解决完上述问题，我们就可以拼接出最终的溢出数据，输入至程序来执行 shellcode 了。

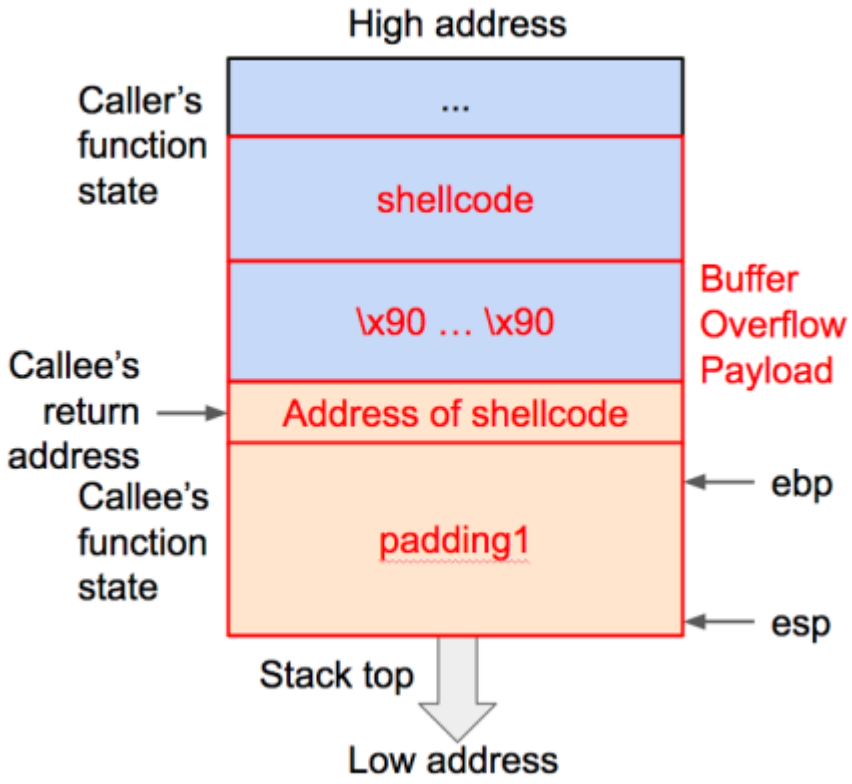

Fig 12. shellcode 所用溢出数据的最终构造

看起来并不复杂对吧？但这种方法生效的一个前提是在函数调用栈上的数据（shellcode）要有可执行的权限（另一个前提是上面提到的关闭内存布局随机化）。很多时候操作系统会关闭函数调用栈的可执行权限，这样 shellcode 的方法就失效了，不过我们还可以尝试使用内存里已有的指令或函数，毕竟这些

部分本来就是可执行的，所以不会受上述执行权限的限制。这就包括 return2libc 和 ROP 两种方法。

0x50 Return2libc

——修改返回地址，让其指向内存中已有的某个函数

根据上面副标题的说明，要完成的任务包括：在内存中确定某个函数的地址，并用其覆盖掉返回地址。由于 libc 动态链接库中的函数被广泛使用，所以有很大概率可以在内存中找到该动态库。同时由于该库包含了一些系统级的函数（例如 system() 等），所以通常使用这些系统级函数来获得当前进程的控制权。鉴于要执行的函数可能需要参数，比如调用 system() 函数打开 shell 的完整形式为 system(“/bin/sh”)，所以溢出数据也要包括必要的参数。下面就以执行 system(“/bin/sh”) 为例，先写出溢出数据的组成，再确定对应的各部分填充进去。

payload: padding1 + address of system() + padding2 + address of “/bin/sh”

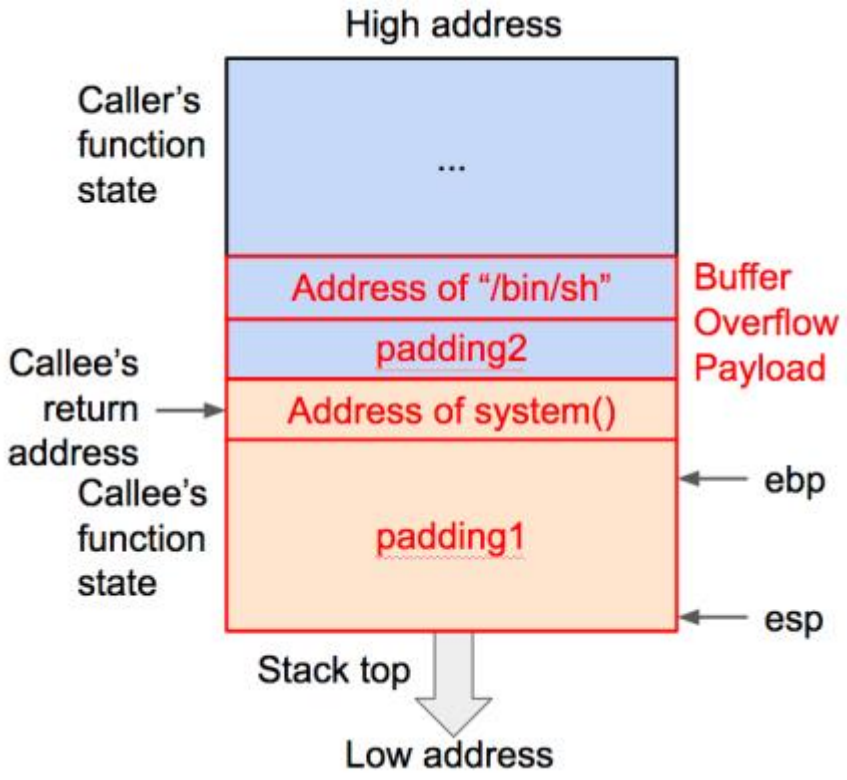

Fig 13. return2libc 所用溢出数据的构造

padding1 处的数据可以随意填充（注意不要包含 “\x00” ，否则向程序传入溢出数据时会造成截断），长度应该刚好覆盖函数的基地址。address of

system() 是 system() 在内存中的地址，用来覆盖返回地址。padding2 处的数据长度为 4（32 位机），对应调用 system() 时的返回地址。因为我们在这里只需要打开 shell 就可以，并不关心从 shell 退出之后的行为，所以 padding2 的内容可以随意填充。address of “/bin/sh” 是字符串 “/bin/sh” 在内存中的地址，作为传给 system() 的参数。

根据上面的构造，我们要解决个问题。

1. 返回地址之前的填充数据（padding1）应该多长？

解决方法和 shellcode 中提到的答案一样。

2. system() 函数地址应该是多少？

要回答这个问题，就要看看程序是如何调用动态链接库中的函数的。当函数被动态链接至程序中，程序在运行时首先确定动态链接库在内存的起始地址，再加上函数在动态库中的相对偏移量，最终得到函数在内存的绝对地址。说到确定动态库的内存地址，就要回顾一下 shellcode 中提到的内存布局随机化（ASLR），这项技术也会将动态库加载的起始地址做随机化处理。所以，如果操作系统打开了 ASLR，程序每次运行时动态库的起始地址都会变化，也就无从确定库内函数的绝对地址。在 ASLR 被关闭的前提下，我们可以通过调试工具在运行程序过程中直接查看 system() 的地址，也可以查看动态库在内存的起始地址，再在动态库内查看函数的相对偏移位置，通过计算得到函数的绝对地址。

最后，“/bin/sh” 的地址在哪里？

可以在动态库里搜索这个字符串，如果存在，就可以按照动态库起始地址＋相对偏移来确定其绝对地址。如果在动态库里找不到，可以将这个字符串加到环境变量里，再通过 getenv() 等函数来确定地址。

解决完上述问题，我们就可以拼接出溢出数据，输入至程序来通过 system() 打开 shell 了。

0x60 半途小结

小结一下，本篇文章介绍了栈溢出的原理和两种执行方法，两种方法都是通过覆盖返回地址来执行输入的指令片段（shellcode）或者动态库中的函数（return2libc）。需要指出的是，这两种方法都需要操作系统关闭内存布局随机化（ASLR），而且 shellcode 还需要程序调用栈有可执行权限。下篇会继续介绍另外两种执行方法，其中有可以绕过内存布局随机化（ASLR）的方法，敬请关注。

0x00 写在前面

首先还是广播一下 2017 Pwn2Own 大赛的最终赛果，本次比赛共发现 51 个漏洞，长亭安全实验室贡献 11 个，以积 26 分的总成绩，在 11 支参赛团队中名列第三！同时，也祝贺国内的安全团队包揽本次大赛的前三名！

0x10 上期回顾

上篇文章介绍了栈溢出的原理和两种执行方法，两种方法都是通过覆盖返回地址来执行输入的指令片段（shellcode）或者动态库中的函数（return2libc）。本篇会继续介绍另外两种实现方法。一种是覆盖返回地址来执行内存内已有的代码片段（ROP），另一种是将某个函数的地址替换成另一个函数的地址（hijack GOT）。

0x20 相关知识

0x21 寄存器

在上篇的背景知识中，我们提到了函数状态相关的三个寄存器——esp，ebp，eip。下面的内容会涉及更多的寄存器，所以我们大致介绍下寄存器在执行程序指令中的不同用途。

32 位 x86 架构下的寄存器可以被简单分为**通用寄存器**和**特殊寄存器**两类，通用寄存器在大部分汇编指令下是可以任意使用的（虽然有些指令规定了某些寄存器的特定用途），而特殊寄存器只能被特定的汇编指令使用，不能用来任意存储数据。

32 位 x86 架构下的通用寄存器包括一般寄存器（eax、ebx、ecx、edx），索引寄存器（esi、edi），以及堆栈指针寄存器（esp、ebp）。

一般寄存器用来存储运行时数据，是指令最常用到的寄存器，除了存放一般性的数据，每个一般寄存器都有自己较为固定的独特用途。eax 被称为累加寄存器（**A**ccumulator），用以进行算数运算和返回函数结果等。ebx 被称为基址寄存器（**B**ase），在内存寻址时（比如数组运算）用以存放基地址。ecx 被称为记数寄存器（**C**ounter），用以在循环过程中记数。edx 被称为数据寄存器（**D**ata），常配合 eax 一起存放运算结果等数据。

索引寄存器通常用于字符串操作中，esi 指向要处理的数据地址（**S**ource **I**ndex），edi 指向存放处理结果的数据地址（**D**estination **I**ndex）。

堆栈指针寄存器（esp、ebp）用于保存函数在调用栈中的状态，上篇已有详细的介绍。

32 位 x86 架构下的特殊寄存器包括段地址寄存器（ss、cs、ds、es、fs、gs），标志位寄存器（EFLAGS），以及指令指针寄存器（eip）。

现代操作系统内存通常是以分段的形式存放不同类型的信息的。我们在上篇谈及的函数调用栈就是分段的一个部分（Stack Segment）。内存分段还包括堆（Heap Segment）、数据段（Data Segment），BSS 段，以及代码段（Code Segment）。代码段存储可执行代码和只读常量（如常量字符串），属性可读可执行，但通常不可写。数据段存储已经初始化且初值不为 0 的全局变量和静态局部变量，BSS 段存储未初始化或初值为 0 的全局变量和静态局部变量，这两段数据都有可写的属性。堆用于存放程序运行中动态分配的内存，例如 C 语言中的 malloc() 和 free() 函数就是在堆上分配和释放内存。各段在内存的排列如下图所示。

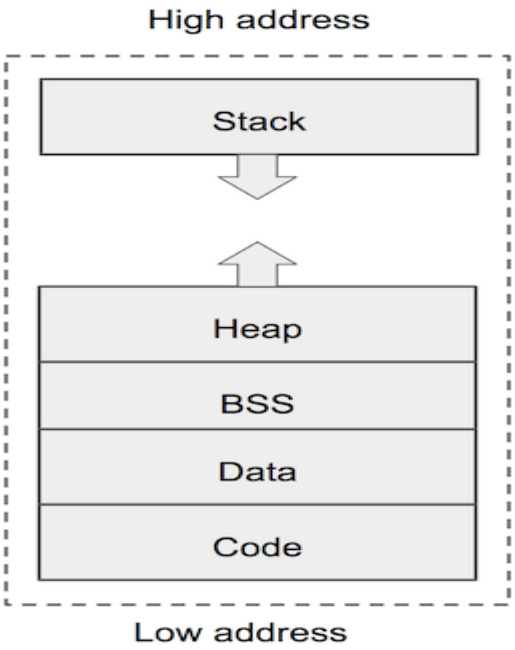

Fig1. 内存分段的典型布局

段地址寄存器就是用来存储内存分段地址的，其中寄存器 ss 存储函数调用栈（**S**tack **S**egment）的地址，寄存器 cs 存储代码段（**C**ode **S**egment）的地址，寄存器 ds 存储数据段（**D**ata **S**egment）的地址，es、fs、gs 是附加的存储数据段地址的寄存器。

标志位寄存器（EFLAGS）32 位中的大部分被用于标志数据或程序的状态，例如 OF（**O**verflow **F**lag）对应数值溢出、IF（**I**nterrupt **F**lag）对应中断、ZF（**Z**ero **F**lag）对应运算结果为 0、CF（**C**arry **F**lag）对应运算产生进位等等。

指令指针寄存器（eip）存储下一条运行指令的地址，上篇已有详细的介绍。

0x22 汇编指令

为了更好地理解后面的内容，一点点汇编语言的知识也是必要的，本节会介绍一些基础指令的含义。32 位 x86 架构下的汇编语言有 Intel 和 AT&T 两种格式，本文所用汇编指令都是 Intel 格式。两者最主要的差别如下。

Intel 格式，寄存器名称和数值前无符号：

“指令名称 目标操作数 DST，源操作数 SRC”

AT&T 格式，寄存器名称前加“%”，数值前加“$”：

“指令名称 源操作数 SRC，目标操作数 DST”

一些最常用的汇编指令如下：

- MOV：数据传输指令，将 SRC 传至 DST，格式为

```
MOV DST, SRC;
```

- PUSH：压入堆栈指令，将 SRC 压入栈内，格式为

```
PUSH SRC;
```

- POP：弹出堆栈指令，将栈顶的数据弹出并存至 DST，格式为

```
POP DST;
```

- LEA：取地址指令，将 MEM 的地址存至 REG ，格式为

```
LEA REG, MEM;
```

- ADD / SUB：加 / 减法指令，将运算结果存至 DST，格式为

```
ADD/SUB DST, SRC;
```

- AND / OR / XOR：按位与 / 或 / 异或，将运算结果存至 DST ，格式为

```
AND/OR/XOR DST, SRC;
```

- CALL：调用指令，将当前的 eip 压入栈顶，并将 PTR 存入 eip，格式为

```
CALL PTR;
```

- RET：返回指令，操作为将栈顶数据弹出至 eip，格式为

```
RET;
```

介绍完以上背景知识，就可以继续栈溢出技术之路了。

0x30 ROP (Return Oriented Programming)

——修改返回地址，让其指向内存中已有的一段指令

根据上面副标题的说明，要完成的任务包括：在内存中确定某段指令的地址，并用其覆盖返回地址。可是既然可以覆盖返回地址并定位到内存地址，为什么不直接用上篇提到的 return2libc 呢？因为有时目标函数在内存内无法找到，有时目标操作并没有特定的函数可以完美适配。这时就需要在内存中寻找多个指令片段，拼凑出一系列操作来达成目的。假如要执行某段指令（我们将其称为“gadget”，意为小工具），溢出数据应该以下面的方式构造（padding 长度和内容的确定方式参见上篇）：

payload : *padding + address of gadget*

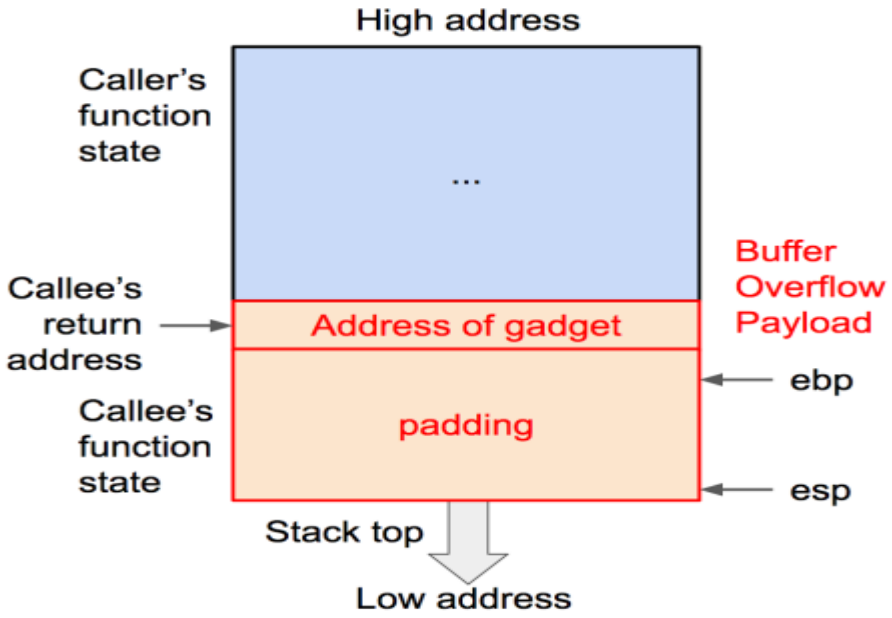

Fig 2. 包含单个 gadget 的溢出数据

如果想连续执行若干段指令，就需要每个 gadget 执行完毕可以将控制权交给下一个 gadget。所以 gadget 的最后一步应该是 RET 指令，这样程序的控制权（eip）才能得到切换，所以这种技术被称为返回导向编程 (**R**eturn **O**riented **P**rogramming)。要执行多个 gadget，溢出数据应该以下面的方式构造：

payload : *padding + address of gadget 1 + address of gadget 2 + ... + address of gadget n*

在这样的构造下，被调用函数返回时会跳转执行 gadget 1，执行完毕时 gadget 1 的 RET 指令会将此时的栈顶数据（也就是 gadget 2 的地址）弹出至 eip，程序继续跳转执行 gadget 2，以此类推。

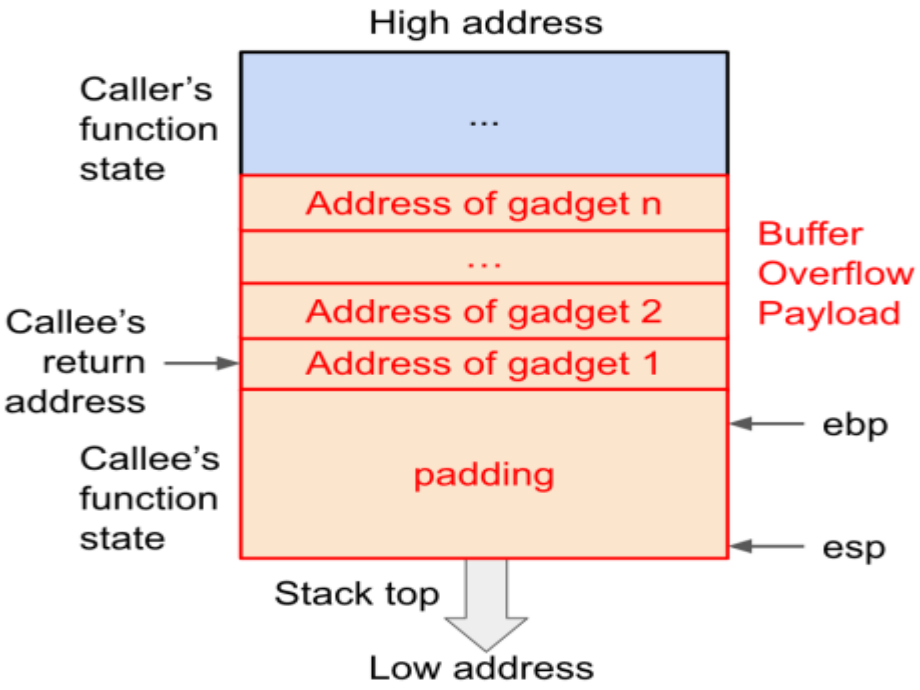

Fig 3. 包含多个 gadget 的溢出数据

现在任务可以分解为：针对程序栈溢出所要实现的效果，找到若干段以 ret 作为结束的指令片段，按照上述的构造将它们的地址填充到溢出数据中。所以我们要解决以下几个问题。

首先，栈溢出之后要实现什么效果？

ROP 常见的拼凑效果是实现一次系统调用，Linux 系统下对应的汇编指令是 int 0x80。执行这条指令时，被调用函数的编号应存入 eax，调用参数应按顺序存入 ebx，ecx，edx，esi，edi 中。例如，编号 125 对应函数

```
mprotect (void *addr, size_t len, int prot)
```

，可用该函数将栈的属性改为可执行，这样就可以使用 shellcode 了。假如我们想利用系统调用执行这个函数，eax、ebx、ecx、edx 应该分别为“125”、内存栈的分段地址（可以通过调试工具确定）、“0x10000”（需要修改的空间长度，也许需要更长）、“7”（RWX 权限）。

其次，如何寻找对应的指令片段？

有若干开源工具可以实现搜索以 ret 结尾的指令片段，著名的包括 **ROPgadget**、**rp++**、**ropeme** 等，甚至也可以用 grep 等文本匹配工具在汇编指令中搜索 ret 再进一步筛选。搜索的详细过程在这里就不再赘述，有兴趣的同学可以参考上述工具的说明文档。

最后，如何传入系统调用的参数？

对于上面提到的 mprotect 函数，我们需要将参数传输至寄存器，所以可以用 pop 指令将栈顶数据弹入寄存器。如果在内存中能找到直接可用的数据，也可以用 mov 指令来进行传输，不过写入数据再 pop 要比先搜索再 mov 来的简单，对吧？如果要用 pop 指令来传输调用参数，就需要在溢出数据内包含这些参数，所以上面的溢出数据格式需要一点修改。对于单个 gadget，pop 所传输的数据应该在 gadget 地址之后，如下图所示。

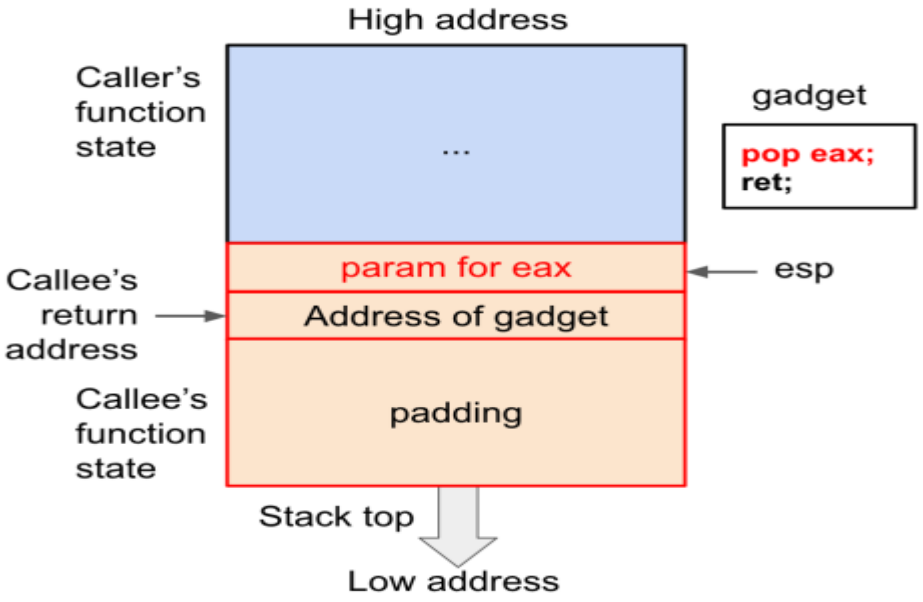

Fig 4. gadget “pop eax; ret;”

在调用 mprotect() 为栈开启可执行权限之后，我们希望执行一段 shellcode，所以要将 shellcode 也加入溢出数据，并将 shellcode 的开始地址加到 int 0x80 的 gadget 之后。但确定 shellcode 在内存的确切地址是很困难的事（想起上篇里面艰难试探的过程了吗？），我们可以使用 push esp 这个 gadget（加入可以找到的话）。

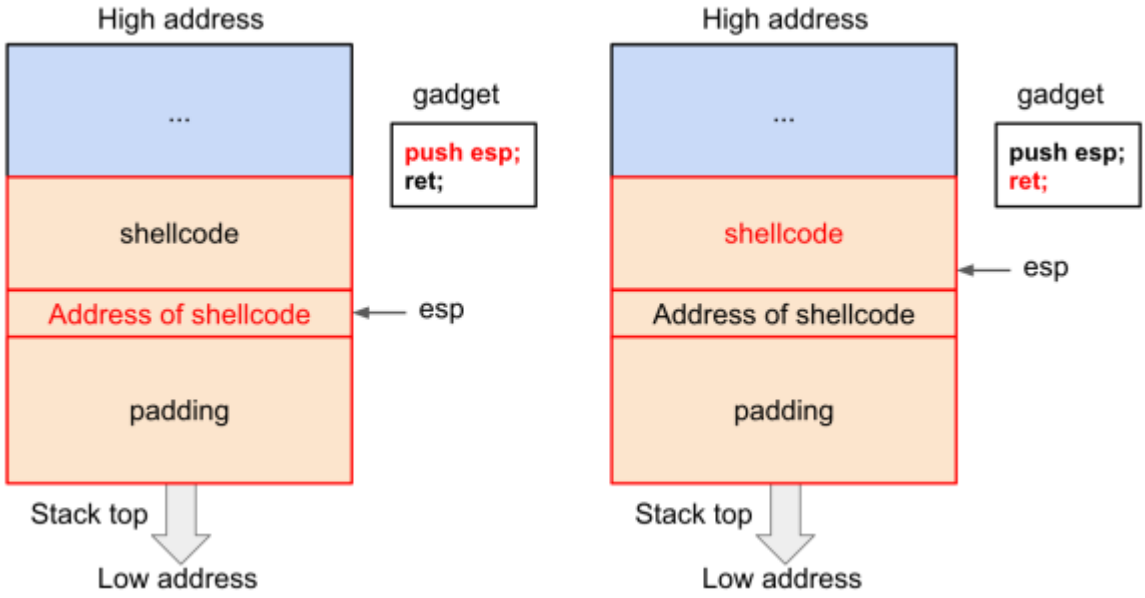

Fig 5. gadget “push esp; ret;”

我们假设现在内存中可以找到如下几条指令：

```
pop eax; ret;    # pop stack top into eax

pop ebx; ret;    # pop stack top into ebx

pop ecx; ret;    # pop stack top into ecx

pop edx; ret;    # pop stack top into edx

int 0x80; ret;   # system call

push esp; ret;   # push address of shellcode
```

对于所有包含 pop 指令的 gadget，在其地址之后都要添加 pop 的传输数据，同时在所有 gadget 最后包含一段 shellcode，最终溢出数据结构应该变为如下格式。

payload : *padding + address of gadget 1 + param for gadget 1 + address of gadget 2 + param for gadget 2 + ... + address of gadget n + shellcode*

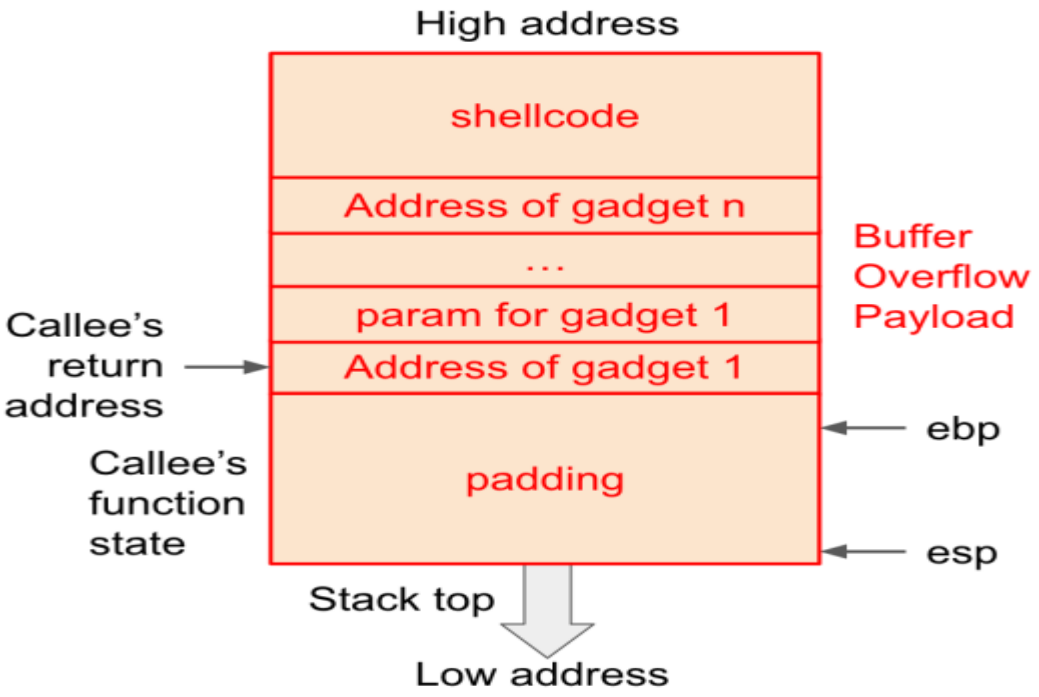

Fig 6. 包含多个 gadget 的溢出数据（修改后）

此处为了简单，先假定输入溢出数据不受“\x00"字符的影响，所以 payload 可以直接包含 “\x7d\x00\x00\x00”（传给 eax 的参数 125）。如果希望实现更为真实的操作，可以用多个 gadget 通过运算得到上述参数。比如可以通过下面三条 gadget 来给 eax 传递参数。

```
pop eax; ret;          # pop stack top 0x1111118e into eax

pop ebx; ret;          # pop stack top 0x11111111 into ebx

sub eax, ebx; ret;     # eax -= ebx
```

解决完上述问题，我们就可以拼接出溢出数据，输入至程序来为程序调用栈开启可执行权限并执行 shellcode。同时，由于 ROP 方法带来的灵活性，现在不再需要痛苦地试探 shellcode 起始地址了。回顾整个输入数据，只有栈的分段地址需要获取确定地址。如果利用 gadget 读取 ebp 的值再加上某个合适

的数值，就可以保证溢出数据都具有可执行权限，这样就不再需要获取确切地址，也就具有了绕过内存随机化的可能。

出于演示的目的，我们假设（简直是*欲点*）了所有需要的 gadget 的存在。在实际搜索及拼接 gadget 时，并不会像上面一样顺利，有两个方面需要注意。

第一，很多时候并不能一次凑齐全部的理想指令片段，这时就要通过数据地址的偏移、寄存器之间的数据传输等方法来“曲线救国”。举个例子，假设找不到下面这条 gadget

```
pop ebx; ret;
```

但假如可以找到下面的 gadget

```
mov ebx, eax; ret;
```

我们就可以将它和

```
pop eax; ret;
```

组合起来实现将数据传输给 ebx 的功能。上面提到的用多个 gadget 避免输入“\x00”也是一个实例应用。

第二，要小心 gadget 是否会破坏前面各个 gadget 已经实现的部分，比如可能修改某个已经写入数值的寄存器。另外，要特别小心 gadget 对 ebp 和 esp 的操作，因为它们的变化会改变返回地址的位置，进而使后续的 gadget 无法执行。

0x40 Hijack GOT

——修改某个被调用函数的地址，让其指向另一个函数

根据上面副标题的说明，要完成的任务包括：在内存中修改某个函数的地址，使其指向另一个函数。为了便于理解，不妨假设修改 printf() 函数的地址使其指向 system()，这样修改之后程序内对 printf() 的调用就执行 system() 函数。要实现这个过程，我们就要弄清楚发生函数调用时程序是如何“找到”被调用函数的。

程序对外部函数的调用需要在生成可执行文件时将外部函数链接到程序中，链接的方式分为静态链接和动态链接。静态链接得到的可执行文件包含外部函数的全部代码，动态链接得到的可执行文件并不包含外部函数的代码，而是在运

行时将动态链接库（若干外部函数的集合）加载到内存的某个位置，再在发生调用时去链接库定位所需的函数。

可程序是如何在链接库内定位到所需的函数呢？这个过程用到了两张表——GOT 和 PLT。GOT 全称是全局偏移量表（**G**lobal **O**ffset **T**able），用来存储外部函数在内存的确切地址。GOT 存储在数据段（Data Segment）内，可以在程序运行中被修改。PLT 全称是程序链接表（**P**rocedure **L**inkage **T**able），用来存储外部函数的入口点（entry），换言之程序总会到 PLT 这里寻找外部函数的地址。PLT 存储在代码段（Code Segment）内，在运行之前就已经确定并且不会被修改，所以 PLT 并不会知道程序运行时动态链接库被加载的确切位置。那么 PLT 表内存储的入口点是什么呢？就是 GOT 表中对应条目的地址。

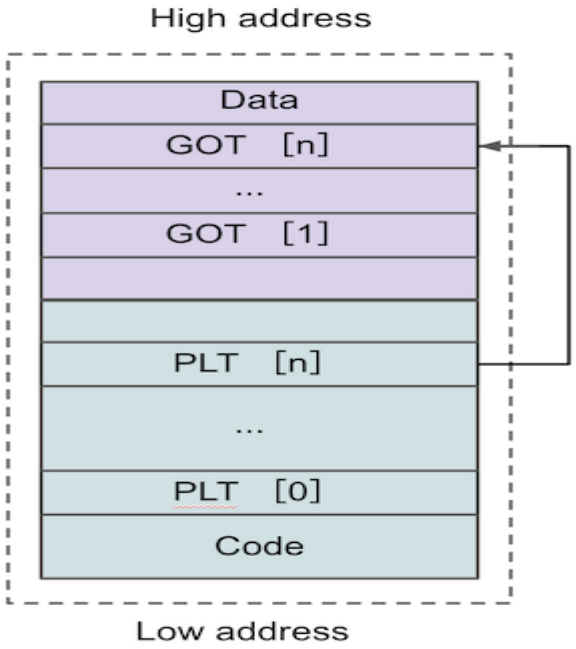

Fig 7. PLT 和 GOT 表

等等，我们好像发现了一个不合理的地方，外部函数的内存地址存储在 GOT 而非 PLT 表内，PLT 存储的入口点又指向 GOT 的对应条目，那么程序为什么选择 PLT 而非 GOT 作为调用的入口点呢？在程序启动时确定所有外部函数的内存地址并写入 GOT 表，之后只使用 GOT 表不是更方便吗？这样的设计是为了程序的运行效率。GOT 表的初始值都指向 PLT 表对应条目中的某个片段，这个片段的作用是调用一个函数地址解析函数。当程序需要调用某个外部函数时，首先到 PLT 表内寻找对应的入口点，跳转到 GOT 表中。如果这是第一次调用这个函数，程序会通过 GOT 表再次跳转回 PLT 表，运行地址解析程序来确定函数的确切地址，并用其覆盖掉 GOT 表的初始值，之后再执行函数调用。当再次调用这个函数时，程序仍然首先通过 PLT 表跳转到 GOT

表，此时 GOT 表已经存有获取函数的内存地址，所以会直接跳转到函数所在地址执行函数。整个过程如下面两张图所示。

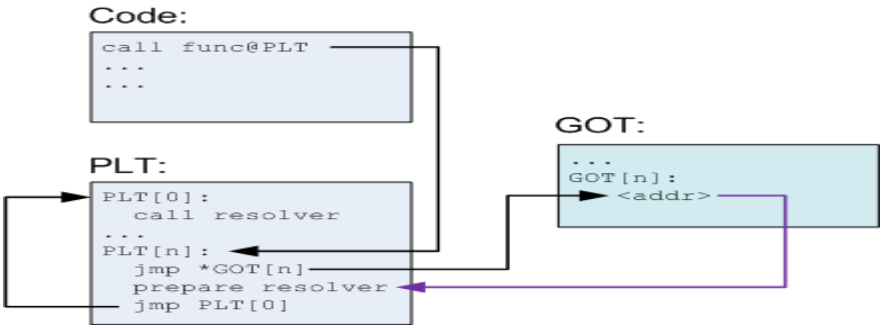

Fig 8. 第一次调用函数时解析函数地址并存入 GOT 表

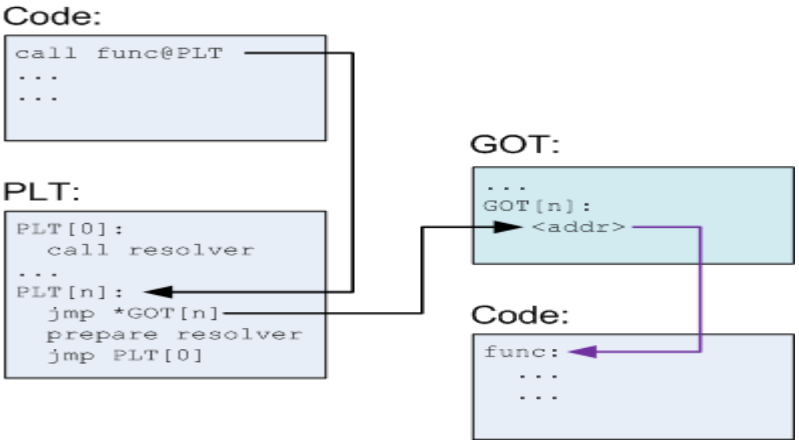

Fig 9. 再次调用函数时直接读取 GOT 内的地址

上述实现遵循的是一种被称为 ***LAZY*** 的设计思想，它将需要完成的操作（解析外部函数的内存地址）留到调用实际发生时才进行，而非在程序一开始运行时就解析出全部函数地址。这个过程也启示了我们如何实现函数的伪装，那就是到 GOT 表中将函数 A 的地址修改为函数 B 的地址。这样在后面所有对函数 A 的调用都会执行函数 B。

那么我们的目标可以分解为如下几部分：确定函数 A 在 GOT 表中的条目位置，确定函数 B 在内存中的地址，将函数 B 的地址写入函数 A 在 GOT 表中的条目。

首先，如何确定函数 A 在 GOT 表中的条目位置？

程序调用函数时是通过 PLT 表跳转到 GOT 表的对应条目，所以可以在函数调用的汇编指令中找到 PLT 表中该函数的入口点位置，从而定位到该函数在 GOT 中的条目。

例如

```
call 0x08048430 <printf@plt>
```

就说明 printf 在 PLT 表中的入口点是在 *0x08048430*，所以 *0x08048430* 处存储的就是 GOT 表中 printf 的条目地址。

其次，如何确定函数 B 在内存中的地址？

如果系统开启了内存布局随机化，程序每次运行动态链接库的加载位置都是随机的，就很难通过调试工具直接确定函数的地址。假如函数 B 在栈溢出之前已经被调用过，我们当然可以通过前一个问题的答案来获得地址。但我们心仪的攻击函数往往并不满足被调用过的要求，也就是 GOT 表中并没有其真实的内存地址。幸运的是，函数在动态链接库内的相对位置是固定的，在动态库打包生成时就已经确定。所以假如我们知道了函数 A 的运行时地址（读取 GOT 表内容），也知道函数 A 和函数 B 在动态链接库内的相对位置，就可以推算出函数 B 的运行时地址。

最后，如何实现 GOT 表中数据的修改？

很难找到合适的函数来完成这一任务，不过我们还有强大的 ROP（DIY 大法好）。假设我们可以找到以下若干条 gadget（继续*欸点*），就不难改写 GOT 表中数据，从而实现函数的伪装。ROP 的具体实现请回看上一章，这里就不再赘述了。

```
pop eax; ret;       # printf@plt -> eax

mov ebx [eax]; ret; # printf@got -> ebx

pop ecx; ret;       # addr_diff = system - printf -> ecx

add [ebx] ecx; ret;     # printf@got += addr_diff
```

从修改 GOT 表的过程可以看出，这种方法也可以在一定程度上绕过内存随机化。

0x50 防御措施

介绍过几种栈溢出的基础方法，我们再来补充一下操作系统内有哪些常见的措施可以进行防御。首先，通常情况下程序在默认编译设置下都会取消栈上数据的可执行权限，这样简单的 shellcode 溢出攻击就无法实现了。其次，可以在操作系统内开启内存布局随机化（ASLR），这样可以增大确定堆栈内数据和动态库内函数的内存地址的难度。编译程序时还可以设置某些编译选项，使程序在运行时会在函数栈上的 ebp 地址和返回地址之间生成一个特殊的值，这个值被称为“金丝雀”（关于这个典故，请大家自行谷歌）。这样一旦发生了栈溢出并覆盖了返回地址，这个值就会被改写，从而实现函数栈的越界检查。最后值得强调的是，尽可能写出安全可靠的代码，不给栈溢出提供写入越界的可能。

0x60 全篇小结

本文简要介绍了栈溢出这种古老而经典的技术领域，并概括描述了四种入门级的实现方法。至此我们专栏的第一讲就全部结束啦，接下来专栏会陆续放出计算机安全相关的更多文章，内容也会涵盖网络安全、Web 渗透、密码学、操作系统，还会有 ctf 比赛题解等等.....

0x70 号外

给大家推荐几个可以练习安全技术的网站：

Pwnhub（ pwnhub | Beta ）：长亭出品，题目丰富，积分排名机制，还可以兑换奖品，快来一起玩耍吧！

Pwnable.kr（ http://pwnable.kr ）：有不同难度的题目，内容涵盖多个领域，界面很可爱

Pwnable.tw（ Pwnable.tw ）：由台湾 CTF 爱好者组织的练习平台，质量较高

Exploit Exercises（ https://exploit-exercises.com ）：有比较完善的题目难度分级，还有虚拟机镜像供下载

最后，放出一张长亭战队在 PWN2OWN 的比赛精彩瞬间，No Pwn No Fun ！ 也祝长亭战队再创佳绩！

References：

- 《Hacking: Art of Exploitation》
- https://sploitfun.wordpress.com/2015/