近年随着计算机技术飞速发展，由于漏洞利用或恶意软件传播导致的直接与间接经济损失也在逐年增加，软件与系统安全逐渐引起各界的高度重视。据 Trustwave 发布的《2017 全球安全报告》显示，99.7%的 web 应用程序至少存在一个漏洞；另据中国信息安全测评中心发布的《2016 年国内外信息安全漏洞态势报告》显示，新增漏洞数量呈在逐年上升趋势，自 2012 年起年均增长 7000 个以上，其中开源软件漏洞频发，全球网站和信息系统正遭遇严重安全威胁。

然而漏洞只是攻击的靶向，恶意软件也只是传播载体，真正在注入攻击中起作用的是用于软件漏洞利用的载荷，即 shellcode。早期 shellcode 主要被发送到特定服务器并建立一个高权限的 shell。随着技术的不断发展，shellcode 不再局限于单一功能，已经可以起到删改系统重要文件、窃取数据、上传病毒甚至格式化硬盘的作用。shellcode 注入已经严重威胁着信息系统安全以致国家与人民群众的财产安全。因此，对 shellcode 在多平台实现原理的研究就变得十分有必要。

每个进程的信息都存放在进程环境块（PEB）中，进程的每一个线程的信息存放在线程环境块（TEB）中，因此可以借助 PEB 找到当前 kernel32.dll 在内存中的位置。但是自 Windows xp 起引入了 PEB 和 TEB 随机化技术，其基址不再固定。但是开发者可借助 FS 寄存器，该寄存器中存储的是 TEB 在全局描述符表（GDT）中的序号，通过 GDT 即可获得 TEB 基址，而 PEB 结构体相对于 TEB 的偏移量固定，因此可定位 PEB。而后通过 PEB 定位 kernal32.dll 的基址。具体实现方法如下：

1.定位 PEB

PEB 相对于 TEB 的偏移量为 0x30，TEB 基址通过 fs 寄存器获得，因此在汇编代码中 PEB 基址为 fs:[30h]

2.定位 LDR

Ldr 是 PEB 结构中的一项，存放一些指向动态链接库信息的链表地址，能得到进程加载的所有模块，它的类型为结构体指针，在 PEB 中的偏移量为 0Ch。

3.定位 LDR_DATA_TABLE_ENTRY 及定位 dll 基址

Ldr 结构体中的一项为 InMemoryOrderModuleList，其为模块加载队列，而在进程中每装入一个模块时，要为其分配一个 LDR_DATA_TABLE_ENTRY 数据结构，将其挂载到模块加载队列中，因此可通过计算 kernal32.dll 与第一个被加载模块的偏移量来从模块加载队列中获得 kernal32.dll 的 LDR_DATA_TABLE_ENTRY 结构，其中包含了 kernal32.dll 的基址（通常情况下 dll 基址在 LDR_DATA_TABLE_ENTRY 结构中的偏移量为 0X18）。但是计算偏移量时需要注意的是，InMemoryOrderModulwList 字段是一个指针，指向 LDR_DATA_TABLE_ENTRY 结构

体上的 LIST_ENTRY 字段。但是它不是指向 LDR_DATA_TABLE_ENTRY 起始 位置的指针，而是指向这个结构的 InMemoryOrderLinks 字段。在 windows xp 和 windows 7 以上的系统中，kernal32.dll 在模块加载队列中的位置不同，故编写 shellcode 时要根据具体平台计算 dll 相对于 PEB 基址的偏移量。

至此，我们获得了已加载进程中的 kernal32.dll 的基址。

函数原型：

```
FARPROC GetProcAddress(
    HMODULE hModule, // DLL 模块句柄
    LPCSTR lpProcName //  函数名
);
```

如果函数调用成功，返回值是 DLL 中的输出函数地址。
如果函数调用失败，返回值是 NULL。得到进一步的错误信息，调用函数 GetLastError。

dll 文件也属于 PE 文件的一种，因此我们可借助 PE 文件的导出表来进行函数定位，通过遍历导出表匹配对应函数名获取函数地址。DLL 基址处存储的是 PE 文件的 DOS 头，在 DOS 头偏移 0x03c 处存储 NT 头的偏移，在 NT 头偏移 0x18 处存储的是 NT 可选头的地址，NT 可选头的偏移 0x60（0x70）处存储的是 DataDirectory 数组，该数组的第一项是导出表信息结构体，包含导出表地址与大小。获取导出表的地址后，在偏移 0x20 处获取函数名列表的 RVA，遍历函数名列表，与目标函数名进行比较，确定其在函数名列表中的索引，然后在函数索引列表中该索引处获取目标函数在函数地址列表中的索引，将在函数地址列表的索引处获取到的相对虚拟地址（RVA）与 DLL 基址相加即获得该函数的基址。
由此我们可以得到 kernel32.dll 中 LoadLibrary 函数的地址。

在 kernal32.dll 定位方法方面：
1.由 FS 寄存器中存储 TEB 的 GDT 序号改为在 GS 寄存器中存储。In Win32, the PEB lives at [fs:30h] whereas in Win64 the PEB is at [gs:60h].
2.各数据结构的偏移量有所不同,例如在 WIN32 平台下，Ldr 在 PEB 结构中的偏移

量为 0xC，而在 WIN64 平台下为 0x18.其余结构的相对基址也后延
在函数定位方法方面：导入表相对 NT 可选头的偏移量不同
在函数调用方面：
1.新增 8 个通用寄存器，r8，r9，r10，r11，r12，r13，r14 和 r15。所有寄存器扩充为 64 位，寄存器的名称发生变化，采用 R 字母开头。这些寄存器也可以分解为 32,16 和 8 位版本
2.程序在每一次函数调用起始处分配其所需的全部堆栈空间，函数结束时释放，调用过程中不会再开辟新的堆栈空间。
3.入栈操作改为 mov 指令，rsp 指针完成所有栈操作。

LINUX：
大多数操作系统都向应用程序提供了一些核心功能，通过使用系统提供的核心功能，应用程序可以很轻松的访问文件，检测用户和组的权限，访问网络资源，以及接收与显示数据等，这些核心功能被称作系统调用。在 linux 平台下，shellcode 的实现是通过启用系统调用来实现预期功能，具体流程包括：

首先将系统调用编号载入寄存器 EAX 中，在 linux 系统中，EAX 寄存器专门用来存储系统调用的编号；
随后，把系统调用需要使用的参数压入其他寄存器，分别保存到 EBX、ECX、ESI、EDI 和 EBP 里；
执行 int 0x80 指令，系统调用软件中断发生，根据 eax 中所定的系统调用去调用系统调用所对应的的内核函数，然后根据相关寄存器内所保存的数据实现一次系统调用。

在 shellcode 的编写中，主要使用 execve 系统调用来执行其他程序，通过将对应程序所在路径作为参数传递个 execve 系统调用，触发软中断即可执行对应程序。

1.将寄存器 eax 清零；
xor eax,eax
2.将 eax 入栈；
push eax; 通过 push eax 来保证字符串后面的数所据是 0，也即字符串结束符
3.将//sh 入栈；
push 0x68732f2f

4.将/bin 入栈；

push 0x6e69622f

5.设置 execve 系统调用，x86 系统下其系统调用号为 11，转换为 16 进制即为 0xb；

mov ebx,esp；将字符串地址赋给 ebx(系统调用的第一个参数）

push edx ; 将 argv[1](值为 NULL) 放到栈上

push ebx ; 将 argv[0]("/bin//sh")放到栈上

mov ecx,esp; argv 是系统调用第二参数，需要将它赋给 ecx

xor eax,eax

mov al,0xb

int 0x80; 使用 int 0x80 进行系统调用

在 linux 平台下 shellcode 的开发中还存在一大难题。当进程调用 execve 系统调用时，如果调用成功则新加载的程序从启动代码开始执行，当前进程的用户空间代码和数据完全被新程序所替换。因而调用成功后，程序不再返回，没有返回值即不能判断何时 execve 调用的程序运行结束，何时 shellcode 应进行下一步操作。通常解决方案为将 linux 系统调用中的 fork 和 wait 方法与 execve 方法配合使用。fork 函数用于创建子进程，在子进程中，fork 函数返回 0，在父进程中，fork 返回新创建子进程的进程 ID，我们可以通过 fork 返回的值来判断当前进程是子进程还是父进程。wait 函数的主要功能是如果父进程的所有子进程都还在运行，调用 wait 将使父进程阻塞，子进程结束后获取子进程状态改变信息，wait 函数能获知内核中子进程的退出信息，并清空该信息所占用的内存空间。因此，linux 平台下 shellcode 开发的基本思想为 shellcode 执行后先使用 fork 通过系统调用创建一个与原来进程几乎完全相同的进程，接下来在子进程中使用 execve 来启动其他能完成 shellcode 功能的程序，即产生一个新的任务，子进程运行过程中，由 wait 系统调用获知当前状态，一旦子进程运行结束，wait 不再阻塞当前父进程的执行，程序继续顺序去执行剩余代码。fork、wait、execve 配合进行 shellcode 开发的思路如下：

1.在寄存器的使用方面：x64 平台下使用 64 位寄存器，寄存器个数由 8 个扩展为 16 个，前八个寄存器与 x86 平台相比首字母由 e 改为 r，后八个寄存器名为 r8 至 r15。

2.在系统调用号方面：x64 和 x86 平台的系统调用号有很大不同，例如 x86 平台下，fork 的系统调用号为 2，wait 为 7，execve 为 11；而在 x64 平台下，对应的

系统调用号为 57,61 和 59。

3.在运行系统调用方面：x86 平台下采用 int 0x80 来启动系统调用，因为 linux 系统调用位于中断 0x80，执行 INT 指令时，所有操作转移到内核中的系统调用处理程序，完成后执行转移到 INT 指令之后的下一条指令。而在 x64 系统中，多数使用 syscall 的方法启动系统调用，它本质上调用 C 函数，从用户态转入内核态。

4、在函数传参方面，x86 系统中参数都保留在栈上，但在 x64 系统中，前六个参数只需要直接保存在 rdi, rsi, rdx, rcx, r8 和 r9 寄存器中，如果还有更多的参数的话才会保存在栈上。

5、内存地址大小不同，在 x64 系统中，虽然内存地址是 64 位长，但用户空间只使用前 47 位，因此地址不能大于 0x00007fffffffffff，否则会抛出异常。