漫谈兼容内核之七:
Wine 的二进制映像装入和启动

毛德操

上一篇漫谈中介绍了几种二进制可执行映像的识别方法，而识别的目的当然是为了要装入并启动这些映像的执行。映像的装入和启动一般总是和创建进程相连系，所以本来就是个相当复杂的过程。而对于 Wine，则在进程创建方面又增添了一些额外的复杂性。

为什么呢？我们这样来考虑：在 Windows 或 ReactOS 中，创建进程是由 CreateProcessW()完成的，系统中的“始祖”进程就是个 Windows 进程，代代相传下来，总是由 Windows 进程创建 Windows 进程，所以映像的装入和启动只发生在 CreateProcessW()中，所装入的也总是 Windows 或 DOS 上的二进制映像。而在 Linux 中，所谓“创建进程”实际上是将一个线程转化成进程，这是由 execve()一类的系统调用完成的，那也只是 Linux 进程的代代相传，所装入的也总是 Linux 上的二进制映像，包括 a.out 和 ELF 映像。

可是 Wine 就不同了。Wine 是在 Linux 内核上运行，系统里的“始祖”进程是 Linux 进程，但是却需要由作为其后代的 Linux 进程创建出 Windows 进程来。另一方面，创建出来的 Windows 进程则有可能通过 CreateProcessW()再创建新一代的 Windows 进程(实际上 Wine 还允许 Windows 进程创建 Linux 进程，这我们就不说了)。

兼容内核则与 Wine 相似，也有“从 Linux 到 Windows”和“从 Windows 到 Windows”两种不同的进程创建。那么，在这两种条件下的映像装入是否也有明显的区别呢？就 Wine 而言，这两种进程创建(从而映像装入)都是在 Linux 环境下在内核外面实现，区别应该是不大的。至于真正的从 Windows 到 Windows 的进程创建和映像装入，则应该再到 ReactOS 中去寻找借鉴。所以我们最好要分别考察 Wine 和 ReactOS 两个系统的进程创建和映像装入。在这篇漫谈中我们先考察 Wine 系统中的映像装入，而 Wine 系统中的映像装入又分两种，一种是在 Linux 环境下通过键盘命令启动一个 Windows 应用程序，另一种是由一个 Windows 应用程序通过 CreateProcessW()再创建一个 Windows 进程。

应该说，Wine 的映像装入和启动的过程是相当复杂的。要了解这个过程，我们不妨从一些装入工具着手，所以我们通过目录 wine/loader 下面的几个源码文件来说明问题。在这几个文件中，有两个.c 文件是有 main()函数的。一个是 main.c，另一个是 glibc.c。编译/连接以后，glibc.c 中的 main()进入工具 wine，而 main.c 中的 main()则分别进入 wine-kthread 和 wine-pthread 两个工具。在功能上 wine-kthread 和 wine-pthread 二者的作用是一样的，只是后者依靠程序库 libpthread.a 实现和管理线程，而前者则直接依靠内核实现和管理线程。另外还有一个文件 preloader.c，虽然并不带有函数 main()，但是编译/连接以后也成为一个可以独立运行的工具 wine-preloader。

我们先看由 Linux 的 shell 启动执行一个 Windows 应用软件的过程。

实际上 Windows 应用并不是由 Linux 的 Shell 直接启动的，而是通过一条类似于“wine notpad.exe”的命令由 Shell 启动 Wine 的装入/启动工具 wine，再由 wine 间接地装入/启动具体的应用程序。整个过程要依次经由 wine、wine-preloader、以及 wine-kthread 或 wine-pthread 共三个工具软件的接力才能完成。我们就顺着这个轨迹从 wine 开始。

可执行程序 wine 的入口 main()在 loader/glibc.c 中(不是在 main.c 中!)。

```
int main( int argc, char *argv[] )
{
    const char *loader = getenv( "WINELOADER" );
    const char *threads = get_threading();

    if (loader)
    {
        const char *path;
        char *new_name, *new_loader;

        if ((path = strrchr( loader, '/' ))) path++;
        else path = loader;

        new_name = xmalloc( (path - loader) + strlen(threads) + 1 );
        memcpy( new_name, loader, path - loader );
        strcpy( new_name + (path - loader), threads );

        /* update WINELOADER with the new name */
        new_loader = xmalloc( sizeof("WINELOADER=") + strlen(new_name) );
        strcpy( new_loader, "WINELOADER=" );
        strcat( new_loader, new_name );
        putenv( new_loader );
        wine_exec_wine_binary( new_name, argv, NULL, TRUE );
    }
    else
    {
        wine_init_argv0_path( argv[0] );
        wine_exec_wine_binary( threads, argv, NULL, TRUE );
    }
    fprintf( stderr, "wine: could not exec %s\n", argv[0] );
    exit(1);
}
```

先由 getenv()检查是否已经设置了环境变量 WINELOADER，如已设置则 getenv()返回其定义，否则返回 0。下面就可看到，检查的目的其实只是为了将其设置正确。接着的 get_threading()则是为了确定是否在使用 libpthread，从而确定下一步应该使用 wine-kthread 还是 wine-pthread。这二者之间的选择与线程的实现模式有关，pthread 中的’p’表示 Posix，而 kthread 中的’k’表示 Kernel。不过这是个专门的话题，在这里就不深入下去了，只是说明我们一般都使用 kthread，因此这个函数一般返回字符串“wine-kthread”，于是字符串 threads 的值就是“wine-kthread”。

如果环境变量 WINELOADER 有定义，即 loader 非 0，就要把该变量的值设置成 threads 的值。但是 WINELOADER 的值很可能包含着整个路径，所以需要先进行一些字符串的处理，把 threads 的值与原来的目录路径拼接在一起。然后就是关键所在、即对于函数 wine_exec_wine_binary()的调用了。

如果环境变量 WINELOADER 没有定义，那就不需要去设置它了。但是这里通过 wine_init_argv0_path()设置了 argv0_path 和 argv0_name 两个静态变量，然后也是对函数 wine_exec_wine_binary()的调用。

总之，wine_exec_wine_binary()是这里的关键。读者下面就会看到，对这个函数的调用要是成功就不会返回。此外，注意在这里调用这个函数时的最后一个参数都是 TRUE。

[main() > wine_exec_wine_binary()]

```
/* exec a wine internal binary (either the wine loader or the wine server) */
void wine_exec_wine_binary( const char *name, char **argv, char **envp, int use_preloader )
{
    const char *path, *pos, *ptr;

    if (name && strchr( name, '/' ))
    {
        argv[0] = (char *)name;
        preloader_exec( argv, envp, use_preloader );
        return;
    }
    else if (!name) name = argv0_name;

    /* first, try bin directory */
    argv[0] = xmalloc( sizeof(BINDIR "/") + strlen(name) );
    strcpy( argv[0], BINDIR "/" );
    strcat( argv[0], name );
    preloader_exec( argv, envp, use_preloader );
    free( argv[0] );

    /* now try the path of argv0 of the current binary */
    . . . . . .

    /* now search in the Unix path */
    . . . . . .
}
```

在我们这个情景中，这里的参数 name 是“wine-kthread”，参数 use_preloader 是 TRUE，而 argv[]数组中各项则依次是“wine”和“notepad.exe”，相当于命令行“wine notepad.exe”。这里实质性的操作显然是 preloader_exec()。但是在调用 preloader_exec()之前把 argv[0]换成了“wine-kthread”。这样，对于传给 preloader_exec()的 argv[]，与其相当的命令行就变成了“wine-kthread　notepad.exe”。当然，这是在为执行另一个工具 wine-kthread 作准备。

如果原来的 argv[0]是个路径，那就把程序名 wine 替换掉以后加以调用就是。而如果只是个程序名而并不包括目录路径的话，那就要反复尝试，首先是目录/usr/local/bin，然后是当前目录，在往下就是逐一尝试环境变量 PATH 中定义的各个路径。在正常的情况下，preloader_exec()是不返回的(所以 wine_exec_wine_binary()不返回)，如果返回就说明在给定

的目录中找不到 wine-kthread。所以，要是逐一尝试全都失败的话，wine_exec_wine_binary()就会返回而在 main()中显示出错信息。

接着往下看 preloader_exec()的代码。

[main() > wine_exec_wine_binary() > preloader_exec()]

```
/* exec a binary using the preloader if requested; helper for wine_exec_wine_binary */
static void preloader_exec( char **argv, char **envp, int use_preloader )
{
#ifdef linux
    if (use_preloader)
    {
        static const char preloader[] = "wine-preloader";
        char *p, *full_name;
        char **last_arg = argv, **new_argv;

        if (!(p = strrchr( argv[0], '/' ))) p = argv[0];
        else p++;

        full_name = xmalloc( p - argv[0] + sizeof(preloader) );
        memcpy( full_name, argv[0], p - argv[0] );
        memcpy( full_name + (p - argv[0]), preloader, sizeof(preloader) );

        /* make a copy of argv */
        while (*last_arg) last_arg++;
        new_argv = xmalloc( (last_arg - argv + 2) * sizeof(*argv) );
        memcpy( new_argv + 1, argv, (last_arg - argv + 1) * sizeof(*argv) );
        new_argv[0] = full_name;
        if (envp) execve( full_name, new_argv, envp );
        else execv( full_name, new_argv );
        free( new_argv );
        free( full_name );
        return;
    }
#endif
    if (envp) execve( argv[0], argv, envp );
    else execv( argv[0], argv );
}
```

当然，条件编译控制量 linux 是有定义的，而参数 use_preloader 则为 TRUE。这里实质性的操作是系统调用 execve()或 execv()，视调用参数 envp 是否为非 0、即是否需要传递环境变量而定。但是这是一段有点“奥妙”的代码。奥妙之处是把原来的 argv[]扩大成了 new_argv[]，使原来的 argv[0]变成了 new_argv[1]，而 new_argv[0]则固定设置为“wine-preloader”，并保持原有的目录路径不变。由于传给 execve()或 execv()的是 new_argv[]，

这就相当于在命令行“wine-kthread　wine notepad.exe”前面添上了一项，变成了这样(如果忽略目录路径)：

“wine-preloader　wine-kthread　wine notepad.exe”

这就是说，本来是要启动装入工具 wine-kthread，现在却变成了 wine-preloader，而原来的命令行则变成了传给 wine-preloader 的命令行参数。

限于篇幅，这里就不介绍 Linux 内核如何装入 ELF 格式可执行映像的过程了。只是指出：wine-preloader 是个 ELF 格式的可执行映像，但这是个特殊的 ELF 可执行映像。特殊在哪里呢？可以看一下 Makefile 中对于如何编译/连接这个程序的说明：

```
wine-preloader: preloader.o Makefile.in
    $(CC) -o $@ -static -nostartfiles -nodefaultlibs -Wl,-Ttext=0x78000000 preloader.o \
                                                    $(LIBPORT) $(LDFLAGS)
```

注意这里的连接可选项-nostartfiles 和-nodefaultlibs，这说明在连接时不使用通常都要使用的 C 程序库。那么，C 程序库中都有些什么函数呢？有些是大家都很熟悉的。例如 printf()，malloc()等等，再如 C 库对系统调用的包装 open()、read()、 mmap()等等，就是大家所熟知的。但是还有一些则知道的人不多，现在就涉及到这些函数了。

大家知道 C 程序的总入口是 main()，可是这只是就程序设计而言。其实操作系统内核在装入可执行映像以后最初跳转进入的是一个名为_start()的函数。是这个函数为 main()和其余目标程序的运行做好各种准备、即系统层次上的初始化，然后再调用 main()的。不管用户程序是什么样，干什么事，这一部分的操作都是一样的，所不同的只是一些参数(例如程序段和数据段的大小和位置等等)，而这些参数都存放在具体可执行映像的头部。这样，与此有关的代码就不需要由用户的程序员来反复地编写，而是统一放在 C 库中供大家连接使用。正因为如此，_start()对于一般的程序员是不可见的，因而也就不为人所知了。而 main()之所以是“总入口”，只是因为标准 C 库中的有关程序总是在完成初始化以后调用 main()。对于 wine-preloader，上述可选项的作用就是告诉连接程序 ld，让它别用 C 库，而由 preloader.c 自己来提供_start()。既然 preloader.c 中的_start()是自己写的，当然就有了自由度，而不必使用 main()这个函数名了，这就是 preloader.c 中没有 main()的原因。

另一方面，对于映像装入内存后的地址也作了明文规定，就是从 0x78000000 开始。

不光是 wine-preloader 特殊，wine-kthread 和 wine-pthread 的编译/连接也有点特殊，它们的装入地址是可以浮动的。虽然它们各自都有个 main()，实际上却接近于共享库、即.so 模块。

对于 wine-preloader，我们从_start()开始看它的代码。这是一段汇编代码。

```
__ASM_GLOBAL_FUNC(_start,
                  "\tmovl %esp,%eax\n"
                  "\tleal -128(%esp),%esp\n"    /* allocate some space for extra aux values */
                  "\tpushl %eax\n"              /* orig stack pointer */
                  "\tpushl %esp\n"              /* ptr to orig stack pointer */
                  "\tcall wld_start\n"
                  "\tpopl %ecx\n"               /* remove ptr to stack pointer */
                  "\tpopl %esp\n"               /* new stack pointer */
                  "\tpush %eax\n"               /* ELF interpreter entry point */
                  "\txor %eax,%eax\n"
```

```
        "\txor %ecx,%ecx\n"
        "\txor %edx,%edx\n"
        "\tret\n")
```

注意这里的‘\t’就是作为分隔符的 TAB，所以例如“\tpushl %eax\n”就是“pushl %eax”。初始堆栈的位置和内容都是内核为其准备好的，堆栈的内容就是 argc，argv[]、envp 等等。至于 argv[]的内容，则相当于命令行“wine-preloader wine-kthread notepad.exe”。

我们暂且不去深究这些汇编代码，但是显然这里主要的操作是对 wld_start()的调用，所以我们直接看 wld_start()。不过有兴趣的读者不妨自己抠一下这里对于堆栈的处理。

```
/*
 *    wld_start
 *
 *    Repeat the actions the kernel would do when loading a dynamically linked .so
 *    Load the binary and then its ELF interpreter.
 *    Note, we assume that the binary is a dynamically linked ELF shared object.
 */
void* wld_start( void **stack )
{
    int i, *pargc;
    char **argv, **p;
    char *interp, *reserve = NULL;
    ElfW(auxv_t) new_av[12], delete_av[3], *av;
    struct wld_link_map main_binary_map, ld_so_map;
    struct wine_preload_info **wine_main_preload_info;

    pargc = *stack;
    argv = (char **)pargc + 1;
    if (*pargc < 2) fatal_error( "Usage: %s wine_binary [args]\n", argv[0] );

    /* skip over the parameters */
    p = argv + *pargc + 1;

    /* skip over the environment */
    while (*p)
    {
        static const char res[] = "WINEPRELOADRESERVE=";
        if (!wld_strncmp( *p, res, sizeof(res)-1 )) reserve = *p + sizeof(res) - 1;
        p++;
    }

    av = (ElfW(auxv_t)*) (p+1);
    page_size = get_auxiliary( av, AT_PAGESZ, 4096 );
    page_mask = page_size - 1;
```

```
    preloader_start = (char *)_start - ((unsigned int)_start & page_mask);
    preloader_end = (char *)((unsigned int)(_end + page_mask) & ~page_mask);

#ifdef DUMP_AUX_INFO
    wld_printf( "stack = %x\n", *stack );
    for( i = 0; i < *pargc; i++ ) wld_printf("argv[%x] = %s\n", i, argv[i]);
    dump_auxiliary( av );
#endif

    /* reserve memory that Wine needs */
    if (reserve) preload_reserve( reserve );
    for (i = 0; preload_info[i].size; i++)
        wld_mmap( preload_info[i].addr, preload_info[i].size,
          PROT_NONE, MAP_FIXED | MAP_PRIVATE | MAP_ANON | MAP_NORESERVE,
            -1, 0 );

    /* load the main binary */
    map_so_lib( argv[1], &main_binary_map);

    /* load the ELF interpreter */
    interp = (char *)main_binary_map.l_addr + main_binary_map.l_interp;
    map_so_lib( interp, &ld_so_map);

    /* store pointer to the preload info into the appropriate main binary variable */
    wine_main_preload_info = find_symbol( main_binary_map.l_phdr,
                                          main_binary_map.l_phnum,
                                          "wine_main_preload_info" );
    if (wine_main_preload_info) *wine_main_preload_info = preload_info;
    else wld_printf( "wine_main_preload_info not found\n" );

#define SET_NEW_AV(n,type,val) new_av[n].a_type = (type); new_av[n].a_un.a_val = (val);
    SET_NEW_AV( 0, AT_PHDR, (unsigned long)main_binary_map.l_phdr );
    SET_NEW_AV( 1, AT_PHENT, sizeof(ElfW(Phdr)) );
    SET_NEW_AV( 2, AT_PHNUM, main_binary_map.l_phnum );
    SET_NEW_AV( 3, AT_PAGESZ, page_size );
    SET_NEW_AV( 4, AT_BASE, ld_so_map.l_addr );
    SET_NEW_AV( 5, AT_FLAGS, get_auxiliary( av, AT_FLAGS, 0 ) );
    SET_NEW_AV( 6, AT_ENTRY, main_binary_map.l_entry );
    SET_NEW_AV( 7, AT_UID, get_auxiliary( av, AT_UID, wld_getuid() ) );
    SET_NEW_AV( 8, AT_EUID, get_auxiliary( av, AT_EUID, wld_geteuid() ) );
    SET_NEW_AV( 9, AT_GID, get_auxiliary( av, AT_GID, wld_getgid() ) );
    SET_NEW_AV(10, AT_EGID, get_auxiliary( av, AT_EGID, wld_getegid() ) );
    SET_NEW_AV(11, AT_NULL, 0 );
```

```
#undef SET_NEW_AV

    i = 0;
    /* delete sysinfo values if addresses conflict */
    if (is_in_preload_range( av, AT_SYSINFO )) delete_av[i++].a_type = AT_SYSINFO;
    if (is_in_preload_range( av, AT_SYSINFO_EHDR ))
        delete_av[i++].a_type = AT_SYSINFO_EHDR;
    delete_av[i].a_type = AT_NULL;

    /* get rid of first argument */
    pargc[1] = pargc[0] - 1;
    *stack = pargc + 1;

    set_auxiliary_values( av, new_av, delete_av, stack );

#ifdef DUMP_AUX_INFO
    wld_printf("new stack = %x\n", *stack);
    wld_printf("jumping to %x\n", ld_so_map.l_entry);
#endif

    return (void *)ld_so_map.l_entry;
}
```

简单地说一下这段程序的作用，就是：

- 保留本进程用户空间(虚拟地址)的某些区间，将来用于 PE 映像的装入。
- 然后通过 map_so_lib()装入 wine-kthread 的映像，但是要避开保留的那些区间。
- 再通过 map_so_lib()装入启动 ELF 映像所需的辅助工具 ld-linux.so.2。
- 设置一些辅助变量(程序中的 new_av[ ])，主要是把有关 wine-kthread 映像的信息(如程序头数组的位置、数组的大小等等)传递给解释器。
- 返回 ld-linux.so.2 的程序入口地址。

这里，眼下要装入的是 wine-kthread 的映像，但是最终要装入的却是 Windows 应用软件的 PE 格式映像。PE 格式可执行程序映像的装入地址是固定的，所以不能让 wine-kthread 映像占了它的位置。后者本身的装入位置是固定的，并且不与前者冲突，但是其启动和实际运行却需要动态分配空间，这就可能发生冲突。因此，需要先把 PE 映像需要用到的地方先加以保留，把它占住，然后才能装入 wine-kthread 的映像。完成了装入以后，最终就转入到被装入映像 wine-kthread 中的 main()。那么需要为 PE 映像保留那些区间呢？数组 preload_info[]对此作出了规定：

```
static struct wine_preload_info preload_info[] =
{
    { (void *)0x00000000, 0x00110000 },    /* DOS area */
    { (void *)0x80000000, 0x01000000 },    /* shared heap */
    { (void *)0x00110000, 0x1fef0000 },    /* PE exe range (may be set with
```

```
                                        WINEPRELOADRESERVE),
                                        defaults to 512mb */
    { 0, 0 }                             /* end of list */
};
```

首先，从虚拟地址 0 开始的 1MB 加 64KB 的区间是为 DOS 及其软件保留的。虽然 Windows 软件已经不是 DOS 软件，但这是从 DOS 发展过来的，仍有可能要用到这块空间，所以仍需保留。再往上，从虚拟地址 0x00110000 开始，大小为 0x1fef0000 的区间是为 PE 格式映像本身保留的。实际上这两块空间是连续的，合在一起占了从 0 到 0x20000000、即 512MB 的空间。除此之外，从地址 0x80000000、即 2GB 边界开始，大小为 16MB 的空间是为 PE 映像保留的 heap 空间、即动态分配的虚拟地址空间(传统上 Windows 对于 32 位地址空间的划分是 2G+2G，即用户空间和系统空间各 2GB，现在也可以像 Linux 那样划分成 3G+1G)。这些都是 Windows 软件、即 PE 格式映像的约定，既要装入执行这样的映像，就必须遵守。而保留这些区间的手段，则就是通过上面代码中调用的 wld_mmap()，实际上就是系统调用 mmap()。注意这样保留的只是当前进程的虚拟地址空间资源，而并不立即就消耗物理存储空间。

另一方面，wine-preloader 本身所占用的虚拟地址空间，则是从 0x78000000 开始的一块不大的空间，反正不会超过 0x02000000、即 32MB。这样，从 0x20000000 到 0x78000000，大约 1.4GB 的地址空间仍是空闲的，足够容纳 wine-kthread 的映像及有关的.so 映像。

为最终要装入的 PE 映像保留好地址空间以后，就通过 map_so_lib()装入 wine-kthread 的映像。如前所述，此时的 argv[1]就是“wine-kthread”，所以装入的就是它的映像。函数 map_so_lib()的代码这里就不看了，从函数名看，这个函数是用来装入.so 模块、即共享库(动态连接库)的映像的，但实际上也可以用来装入别的 ELF 映像，所以这里 wine-kthread 的映像也由 map_so_lib()装入。ELF 格式的目标映像在装入时需要受到一些协助，以完成与动态连接库、即.so 模块的连接。这些协助要由一个工具软件来提供，称为“解释器(interpreter)”。不过这样的解释器并不是作为独立的进程运行，而是本身就以(无需动态连接的)共享库的形式装入目标映像的进程空间中运行；只是这里的动态连接很简单，只要由解释器软件提供一个总的入口即可。解释器模块与创建目标 ELF 映像时所使用的编译/连接工具是配套的，模块的文件名就写在目标 ELF 映像的头部数据结构中。所以，一旦装入了目标映像，就可以知道应该配套使用什么解释器，例如“/lib/ld-linux.so.2” (读者不妨这样试一下：“strings wine-kthread | grep ld-linux”，就可以看到 wine-kthread 的配套解释器就是/lib/ld-linux.so.2)。这里的 main_binary_map.l_interp 就是该文件名字符串在映像中的位移。所以，上面第二次调用 map_so_lib()的目的就是装入 ELF 映像的解释器。

此外，这里还通过 find_symbol()在已装入的 wine-kthread 映像中寻找一个名为 wine_main_preload_info 的变量，找到后就把结构数组 preload_info[]的起始地址填写到这个变量中，这是因为 wine-kthread 也需要知道为 PE 映像和 DLL 所保留的空间。

最后，wld_start()返回 ld_so_map.l_entry，这就是 ELF 映像解释器的入口地址。我们知道，函数的返回值是通过寄存器%eax 传递的。回到前面_start()的汇编代码，可以看到它把%eax 的内容压入堆栈，然后执行了一条 ret 指令，这就跳转到了解释器的入口地址。解释器的细节已经不在本文的范围之内，概而言之就是它会装入目标映像运行所需的共享库(例如 linc.so)，并完成目标映像与共享库的动态连接，最后跳转到目标映像的入口。然后，目标映像在完成了自身的初始化以后，就会调用目其 main()。这当然就是 wine-kthread 的 main()，也就是 loader/main.c 中的 main()了。此外，wld_start()在返回之前对堆栈进行了调整，其效果是在原来的 argv[]中跳过了 argv[0]，使原来的 argv[1]变成了进入 main()时的 argv[0]。于

是，对于 main()来说，这就相当于命令行“wine-kthread notepad.exe”。

```
int main( int argc, char *argv[] )
{
    char error[1024];
    int i;

    if (wine_main_preload_info)
    {
        for (i = 0; wine_main_preload_info[i].size; i++)
            wine_mmap_add_reserved_area( wine_main_preload_info[i].addr,
                                         wine_main_preload_info[i].size );
    }

    wine_init( argc, argv, error, sizeof(error) );
    fprintf( stderr, "wine: failed to initialize: %s\n", error );
    exit(1);
}
```

如前所述，由于 wld_start()的配合，这里的指针 wine_main_preload_info 已经指向由 wine-preloader 提供的保留地址区间表。实际上每个 Linux 进程都有自身的保留区间队列，因为例如从地址 0xc0000000 往下的区间就是要保留用于堆栈的。所以这里要把它们合并到自身的保留区间队列中。

而 main()的主体，则毫无疑问是对 wine_init()的调用。

[main() > wine_init()]

```
void wine_init( int argc, char *argv[], char *error, int error_size )
{
    char *wine_debug;
    int file_exists;
    void *ntdll;
    void (*init_func)(void);

    build_dll_path();
    wine_init_argv0_path( argv[0] );
    __wine_main_argc = argc;
    __wine_main_argv = argv;
    __wine_main_environ = environ;
    mmap_init();

    if ((wine_debug = getenv("WINEDEBUG")))
    {
        if (!strcmp( wine_debug, "help" )) debug_usage();
```

```
        wine_dbg_parse_options( wine_debug );
    }

    if (!(ntdll = dlopen_dll( "ntdll.dll", error, error_size, 0, &file_exists ))) return;
    if (!(init_func = wine_dlsym( ntdll, "__wine_process_init", error, error_size ))) return;
    init_func();
}
```

由于篇幅所限，我们只好长话短说了。这里的 build_dll_path()根据环境变量 WINEDLLPATH 设置好装入各种 DLL 的目录路径。这是为装入 DLL 做好准备。注意这里把调用参数 argc、argv、environ 转移到了__wine_main_argc 等全局量中。

下面还有两件事是实质性的：

1. 通过 dlopen_dll()装入由 Wine 提供的动态连接库 ntdll.dll。由于 Wine 特殊的系统结构，它不能使用微软的 ntdll.dll，而只能使用 Wine 自己的 DLL，这样的 DLL 称为“内置(built-in)”DLL。
2. 执行内置 ntdll.dll 中的函数__wine_process_init()。先通过 wine_dlsym()取得这个函数的入口地址，然后通过指针 init_func 进行调用。之所以要以这样的方式调用，是因为这个函数在 ntdll.dll 中，而 dlopen_dll()只是装入了这个 DLL、却并未完成与此模块的动态链接。

Windows 的 DLL 是 PE 格式的，而在 Linux 环境下由 gcc 生成的.so 文件为 COFF 或 ELF 格式(但是 PE 格式的主体部分就是 COFF，二者基本上只差一个头)。一般而言，Wine 既可以安装使用本身的“内置”动态库，也可以安装使用 Windows 上相应的“原装(native)”动态库。但是，其中有几个动态库是特殊的，因而只能使用 Wine 自己的版本，ntdll 就是其一。实际上 Windows 上的 ntdll.dll 中根本不会有__wine_process_init()这么个函数。Wine 自己的“内置”DLL 实际上就是 Linux 的.so 模块，是在 Linux 环境下由 gcc 产生的。GNU 的 C 库 glibc 中提供了一组用来处理.so 模块的函数，上面的 dlopen_dll()和 wine_dlsym()最终都是调用这些库函数(例如 dlopen()和 dlsym())来完成操作。

下面就是执行由 ntdll.dll 提供的__wine_process_init()了。

[main() > wine_init() > __wine_process_init()]

```
void __wine_process_init( int argc, char *argv[] )
{
    static const WCHAR kernel32W[] = {'k','e','r','n','e','l','3','2','.','d','l','l',0};

    WINE_MODREF *wm;
    NTSTATUS status;
    ANSI_STRING func_name;
    void (* DECLSPEC_NORETURN init_func)();
    extern mode_t FILE_umask;

    thread_init();

    /* retrieve current umask */
```

```
    FILE_umask = umask(0777);
    umask( FILE_umask );

    /* setup the load callback and create ntdll modref */
    wine_dll_set_callback( load_builtin_callback );

    if ((status = load_builtin_dll( NULL, kernel32W, 0, &wm )) != STATUS_SUCCESS)
    {
        MESSAGE( "wine: could not load kernel32.dll, status %lx\n", status );
        exit(1);
    }
    RtlInitAnsiString( &func_name, "__wine_kernel_init" );
    if ((status = LdrGetProcedureAddress( wm->ldr.BaseAddress, &func_name,
                                          0, (void **)&init_func )) != STATUS_SUCCESS)
    {
        MESSAGE(
          "wine: could not find __wine_kernel_init in kernel32.dll, status %lx\n", status );
        exit(1);
    }
    init_func();
}
```

也许需要加深一下读者的印像，现在是在 wine-kthread 中运行，目的是要装入目标 PE 格式映像、对于我们这儿是 notepad.exe 的映像。

简而言之，函数__wine_process_init()主要完成以下操作：

一、 thread_init()

a. 分配并初始化 TEB, info 等数据结构。

b. 通过 server_init_process()与服务进程建立 socket 连接。在此过程中，如果连接失败就说明服务进程尚不存在，此时要通过 start_server()先 fork()一个子进程，让其执行 wine_exec_wine_binary()，以装入并运行 wineserver，再试图与其建立连接。

c. 请求服务进程执行 init_thread()。

二、由 load_builtin_dll("kernel32.dll")装入 Wine 的另一个“内置”动态连接库 kernel32.dll。可见 kernel32.dll 也必须是个“built-in(内置)”DLL。

三、由 LdrGetProcedureAddress() 从装入的 kernel32.dll 映像中取得函数__wine_kernel_init()的入口。

四、执行 kernel32.dll 中的函数__wine_kernel_init()。

读者也许注意到这里装入 kernel32.dll 和从中获取函数入口时所调用的函数与前面装入 ntdll.dll 时所用的不同，但是这只是一些细节上的不同，在这里可以忽略。

有了 ntdll 和 kernel32 两个动态连接库，就可以通过__wine_kernel_init()装入目标程序的映像并加以执行了。这个函数的伪代码如下：

[main() > wine_init() > __wine_process_init() > __wine_kernel_init()]

```
void __wine_kernel_init(void)
{
        process_init();             //源码中说：Initialize everything
        跳过__wine_main_argv[ 0]，即“wine-kthread”，使目标程序名变成新的 argv[0]。
        将目标程序名转换成 unicode。
        通过 find_exe_file()找到并打开目标映像文件，例如“notepad.exe”。
        MODULE_GetBinaryType();          //取得目标文件的类型。
        Switch(目标文件类型)
        {
        case BINARY_PE_EXE:
            if ((peb->ImageBaseAddress = load_pe_exe( main_exe_name, main_exe_file )))
              goto found;                  //装入映像成功。
             . . . . . .
        . . . . . .
        }
found:
    /* build command line */
    set_library_wargv( __wine_main_argv );
    if (!build_command_line( __wine_main_wargv )) goto error;

    stack_size =
        RtlImageNtHeader(peb->ImageBaseAddress)->OptionalHeader.SizeOfStackReserve;

    /* allocate main thread stack */
    if (!THREAD_InitStack( NtCurrentTeb(), stack_size )) goto error;

    /* switch to the new stack */
    wine_switch_to_stack( start_process, NULL, NtCurrentTeb()->Tib.StackBase );

 error:
    ExitProcess( GetLastError() );
}
```

这里的第一个函数 process_init()是个很大的过程，包括向 wineserver 登记、对包括 PEB 在内的各种数据结构的初始化、还有对注册本的查询、对当前目录和环境的设置等等，所以源码中说是“initialize everything”。其实这是很好理解的：因为目标映像、这里是 notepad.exe、可不会向 wineserver 登记，在 Windows 平台上根本就没有 wineserver。另一方面，在目标进程开始运行之前，还得为其做好许多准备，例如开始运行时的工作目录在那里，人机界面上应使用哪一种文字，等等。注意这里所说的进程既是指内核意义上的进程，更是指 Wine 层面上的进程。从内核的意义上说，眼下正在运行的程序就属于当前进程；而从 Wine 的意义上说，则目标进程尚未开始运行，还正在为此进行准备。

到 process_init()执行完毕的时候，为 Wine 进程进行的准备就基本就绪了。可是，舞台搭好了，演员却还没有到，目标映像本身尚未装入。于是接着就通过 find_exe_file()找到并

打开目标映像，准备装入。之所以先要找到，而不是直接打开，是因为可能需要按若干不同的路径寻找目标映像文件。

然后，打开目标映像文件以后，就通过 MODULE_GetBinaryType()获取文件的类型，读者已经在上一篇漫谈中看到这个函数是怎样实现此项功能的了。

下面的操作就要视目标映像的类型而定了。在这里我们只关心类型为 BINARY_PE_EXE 的 32 位.exe 映像。当然，那是 PE 格式的。从代码中看到，BINARY_PE_EXE 映像是由 load_pe_exe()装入的。注意这只是装入目标映像，而并不包括 DLL 以及目标映像与所需 DLL 的动态连接，那还在后面。

装入了目标 EXE 映像以后，回到__wine_kernel_init()的代码中，下面为目标映像的执行准备好 argc,、argv[]、以及命令行等参数。就我们这个情景而言，已经只剩下“notepad.exe”一项了。然后是通过 THREAD_InitStack()为目标进程的主线程分配堆栈，映像头部的 OptionalHeader 中提供了所建议的堆栈大小。

最后的 wine_switch_to_stack(start_process, NULL, NtCurrentTeb()->Tib.StackBase)是最为关键的。这是一小段汇编代码，实现的是对函数 start_process()的调用，而对 start_process()的调用在正常情况下是不返回的。

[main() > wine_init() > __wine_process_init() > __wine_kernel_init() > wine_switch_to_stack()]

```
#if defined(__i386__) && defined(__GNUC__)
__ASM_GLOBAL_FUNC( wine_switch_to_stack,
                   "movl 4(%esp),%ecx\n\t"    /* func */
                   "movl 8(%esp),%edx\n\t"    /* arg */
                   "movl 12(%esp),%esp\n\t"    /* stack */
                   "pushl %edx\n\t"
                   "xorl %ebp,%ebp\n\t"
                   "call *%ecx\n\t"
                   "int $3" /* we never return here */ );
```

我把这几行汇编代码留给读者，只是说明：这里的 call 指令所调用的就是 start_process()，对这个函数的调用不应返回，如果返回就执行指令“int 3”，那是用于 Debug 的自陷指令。

下面接着看 start_process()：

[main() > wine_init() > __wine_process_init() > __wine_kernel_init() > wine_switch_to_stack()
 > start_process()]

```
static void start_process (void *arg )
{
    __TRY
    {
        PEB *peb = NtCurrentTeb()->Peb;
        IMAGE_NT_HEADERS *nt;
        LPTHREAD_START_ROUTINE entry;
```

```
        LdrInitializeThunk( main_exe_file, 0, 0, 0 );     //装入所需的 DLL 并完成动态连接。
        nt = RtlImageNtHeader( peb->ImageBaseAddress );
        entry = (LPTHREAD_START_ROUTINE)((char *)peb->ImageBaseAddress +
                                                nt->OptionalHeader.AddressOfEntryPoint);
        . . . . . .
        ExitProcess( entry( peb ) );
    }
    __EXCEPT(UnhandledExceptionFilter)
    {
        TerminateThread( GetCurrentThread(), GetExceptionCode() );
    }
    __ENDTRY
}
```

这是一段带出错处理的代码，意思是这样：在执行__TRY{}里面的代码之前，先把陷阱(Trap)响应/处理的向量指向这里的 UnhandledExceptionFilter()。这样，如果在执行的过程中落下了陷阱，例如访问了某个未经映射、或者受到保护的地址，就会转到UnhandledExceptionFilter()来执行。

显然，在目标映像投入运行前，还得根据映像中提供的信息装入所有需要直接、间接用到的 DLL，并建立好与这些 DLL 的动态连接，这是由 LdrInitializeThunk()完成的。所以，LdrInitializeThunk()是个十分重要的过程，本文因篇幅所限就不深入下去了，以后再回到这个话题上来。

将一个 PE 映像装入内存以后，其“进程环境块”PEB 中的 ImageBaseAddress 指向这个映像的起点，映像的开头是一个 IMAGE_DOS_HEADER 数据结构，里面有个指针指向映像的主体部分，而主体部分的开头则是一个 IMAGE_NT_HEADERS 数据结构。RtlImageNtHeader()就根据这个关系找到 IMAGE_NT_HEADERS 数据结构(在内存中)的起点。IMAGE_NT_HEADERS 是个比较复杂的多层数据结构，里面的成分 OptionalHeader.AddressOfEntryPoint 就是可执行程序的入口地址(相对于映像起点的位移)。所以，entry 就是目标程序在内存中的入口地址。最后，目标程序、在这里是 notepad.exe、的执行是以 entry(peb)的形式实现的，即把入口地址当成一个函数的起点，而把已经设置好的 PEB 数据结构作为参数。这样，当 CPU 从 entry()返回时，下一步就是 ExitProcess()、即退出/终止 Windows 进程了。事实上 ExitProcess()里面的最后一个系统调用是 exit()，所以这既是作为 Windows 进程的终结，也是作为 Linux 进程的终结。

注意在此过程中 wine-preloader、wine-kthread、以及目标映像 notepad.exe 都是在同一个进程、即同一个地址空间中活动。先是 wine-preloader 转化成了 wine-kthread，然后 wine-kthread 又转化成 notepad.exe，就像对动态库的调用一样。这中间并没有创建/启动新的进程。而从 wine 到 wine-preloader 则略有不同，那是一个进程通过 execv()“从新做人”，变成了另一个进程。如果从“进程控制块”、即 task_struct 数据结构的角度看，则从 wine 开始一直到 notepad.exe 都是在同一个进程内变迁。但是，从效率的角度看，则先后装入了四个软件的映像(不包括“解释器”、例如/lib/ld-linux.so.2 的映像)。相比之下，Linux 内核在装入普通的 ELF 目标映像时只要装入一个映像就够了(也不包括“解释器”的映像)。

应该说，整个过程确实很复杂。为什么要这么复杂呢？这又要讲到 Wine 的宗旨，那就

是不触动内核，也即“内核差异核外补”。这里的内核差异在于 Linux 内核不能装入 PE 格式的映像，也不能实施 PE 映像与 DLL 之间的动态连接。于是，要实施 PE 映像与 DLL 之间的动态连接，就得在核外启动 wine-kthread。这倒没有什么，Linux 在装入 ELF 映像时也要启动 ld-linux.so。但是在装入 wine-kthread 时又得避开 PE 映像所需要的地方，所以就需要先启动 wine-preloader。另一方面，Linux 内核并不知道 wine-preloader，也不知道该用 wine-kthread 还是 wine-pthread，于是就又得通过另一个工具 wine 再过渡一下。复杂性就这么层层加码累积起来了。类似的过程，如果是由核内核外分工合作，就可以简化和提高效率。例如，内核读取目标映像的头部就可以知道这是个 PE 映像，从而需要动用 wine-kthread，并且在装入 wine-kthread 时需要避开应该为 PE 映像预留的空间，而这对于内核是不难办到的，这样 wine 和 wine-preloader 都可以不需要了。进一步，wine-kthread 所做的事情也有许多可以移到内核中去做，这样既可提高效率又可简化程序。

上面所说的是在 Linux 环境下通过键盘命令启动 Windows 软件的过程，也就是前面所说的“从 Linux 进程到 Windows 进程”的过程。下面再看一下由一个 Windows 进程通过 CreateProcessW()创建另一个 Windows 进程的过程，即“从 Windows 到 Windows”过程，这样才算完整。

```
BOOL WINAPI CreateProcessW( LPCWSTR app_name, ……)
{
    ……
    get_file_name( app_name, cmd_line, name, sizeof(name), &hFile );
    ……
    switch( MODULE_GetBinaryType( hFile, &res_start, &res_end ))
    {
    case BINARY_PE_EXE:
        TRACE( "starting %s as Win32 binary (%p-%p)\n",
               debugstr_w(name), res_start, res_end );
        retv = create_process( hFile, name, tidy_cmdline, envW, cur_dir,
                               process_attr, thread_attr, inherit, flags, startup_info,
                               info, unixdir, res_start, res_end );
        break;
    case BINARY_OS216:
    case BINARY_WIN16:
    case BINARY_DOS:
        TRACE( "starting %s as Win16/DOS binary\n", debugstr_w(name) );
        retv = create_vdm_process( name, tidy_cmdline, envW, cur_dir, process_attr,
                                   thread_attr, inherit, flags, startup_info, info, unixdir );
        break;
    case BINARY_PE_DLL:
        TRACE( "not starting %s since it is a dll\n", debugstr_w(name) );
        SetLastError( ERROR_BAD_EXE_FORMAT );
        break;
    case BINARY_UNIX_LIB:
        . . . . . .
```

```
            break;
        case BINARY_UNKNOWN:
            /* check for .com or .bat extension */
            if ((p = strrchrW( name, '.' )))
            {
                if (!strcmpiW( p, comW ) || !strcmpiW( p, pifW ))
                {
                    TRACE( "starting %s as DOS binary\n", debugstr_w(name) );
                    retv = create_vdm_process( name, tidy_cmdline, envW, cur_dir, process_attr,
                                               thread_attr, inherit, flags, startup_info, info, unixdir );
                    break;
                }
                if (!strcmpiW( p, batW ))
                {
                    TRACE( "starting %s as batch binary\n", debugstr_w(name) );
                    retv = create_cmd_process( name, tidy_cmdline, envW, cur_dir, process_attr,
                                               thread_attr, inherit, flags, startup_info, info );
                    break;
                }
            }
            /* fall through */
        case BINARY_UNIX_EXE:
            {
                ……
            }
            break;
        }
        CloseHandle( hFile );
        …….
        return retv;
}
```

看到这段代码，读者也许会想起前边的__wine_kernel_init()，那是在工具 wine-kthread 中执行的。但是这里的相似只是形式上的，而不是实质上的。

这里 MODULE_GetBinaryType()的作用是判断目标映像的类型，这大家都知道了。而我们主要关心的是 32 位 Windows 应用，即 PE 格式的 EXE 映像，所以只关心 create_process()。

[CreateProcessW() > create_process()]

```
static BOOL create_process (HANDLE hFile, LPCWSTR filename, LPWSTR cmd_line, . . .)
{
    设置好目标进程的环境和参数。
    创建两对 pipe，  startfd[]和 execfd[]。
    fork()
```

子进程：

read(startfd[0]； //企图从一个 pipe 中读，实际上是等待父进程发来开始运行的指令。

close(startfd[0])；//关闭该 pipe。

wine_exec_wine_binary(NULL, argv, NULL, TRUE);

//启动“wine-preloader wine-kthread …”。

write (execfd[1], &err, sizeof(err))；//通过另一个 pipe 向父进程发送运行后的出错代码。

_exit(1); //退出运行并终结。

父进程：

SERVER_START_REQ(new_process)

. . . . . .

SERVER_END_REQ; //向 wineserver 登记子进程。

write(startfd[1], &dummy, 1)； //向子进程发送 1，令其运行。

read(execfd[0]); //企图从另一个 pipe 中读，实际上是等待子进程的运行结果。

WaitForSingleObject(process_info, INFINITE); //等待目标进程运行。

SERVER_START_REQ(get_new_process_info)

. . . . . .

SERVER_END_REQ; //取得有关子进程运行的信息，这是要返回给调用者的。

}

这里的操作分两个方面。一方面是子进程的创建，父、子进程间的同步，与 wineserver 的交互等等。另一方面是创建子进程的具体方法和过程。关于后者，在这里就是对 wine_exec_wine_binary()的调用。读者在前边已经看到过对这个函数的调用，那是在工具 wine 中执行的，在 loader/glibc.c 的 main()中受到调用。

二者都调用 wine_exec_wine_binary()，而且在调用时的最后一个参数 use_preloader 也都是 TRUE，表示应该使用 wine-preloader。显然，从 wine_exec_wine_binary()开始，下面的装入/启动过程就都与前面所述完全相同了。

所以，二者在这方面的相似性才是实质性的。实际上，当前进程、即调用 CreateProcessW()的进程，起着与 wine 相同的作用，即启动“wine-preloader wine-kthread …”的运行。其实工具 wine 是可以省略的，例如我们既然知道应该用 wine-kthread，就可以这样来启动 notepad.exe：“wine-preloader wine-kthread notepad.exe”。但是，要求使用者每次都要这样来键入命令行似乎不现实。

我们不妨再深入一些，看看问题的焦点在那里。显然，现有的 Linux 内核(到 2.4.14 为止)并不具备直接装入 PE 映像的能力，所以需要在用户空间有个 wine-kthread 来完成目标 PE 映像和 DLL 的装入，并完成目标映像和 DLL 之间的动态连接，最后跳转到目标映像的入口。所以 wine-kthread 对于 PE 映像起着类似于 ELF“解释器”的作用，但是又远比后者复杂，因为它还负责 PE 映像的装入。既然这么复杂，就不能像 ELF“解释器”那样把它做成一个小小的共享库了。于是就来了空间冲突的问题。

映像 wine-preloader 和 wine-kthread 的装入地址都是固定的。前者的装入地址是 0x78000000，大小不超过 8KB；后者的装入地址则是 0x7bf00000，大小不超过 32KB。于是这里就有了两个问题：

1) 既然 wine-kthread 的装入地址是 0x7bf00000，与为 PE 格式映像所保留的空间并不冲突，那为什么还要用 wine-preloader 来保留这些空间呢？

2) 即使真有必要为 PE 格式映像保留空间，而 PE 映像又是由 wine-kthread 装入的，那

为什么不可以由 wine-kthread 自己来保留这些空间呢？比方说，在它的 main()中一开始就保留好这些空间，然后才进行其余的活动。

对这两个问题的回答是互相连系在一起的。原来，ELF 映像头部所提供的装入位置、大小等等只是静态的数据，并不表示目标映像在运行时就只占这么一点地方。特别重要的是，表面上 wine-preloader 只装入了 wine-kthread 和 ld-linux.so.2 这么两个 ELF 映像，但是实际上需要装入的还不止于此。这是因为 wine-kthread 的运行本身还需要得到某些共享库的支持。我们在 wine-kthread 的代码中看到了对 getenv()、strlen()、malloc()等等函数的调用，可是这些函数都在共享库 libc.so 中。事实上，wine-kthread 的运行需要得到 libc.so 和 libwine.so 两个共享库的支持。共享库的装入地址都是浮动的，其代码在编译时都要在命令行中加上-fPIC 选项。可是，虽然是浮动的，但是在通过 mmap()将其影像“装入”虚存空间时却非常可能(甚至肯定)会动用本应该为 PE 映像保留的区间中。另一方面，共享库在本进程空间的装入、以及目标映像(在这里是 wine-kthread)与共享库的动态连接是由“解释器”在启动目标映像之前完成的。当 CPU 进入目标映像的 main()时，共享库的装入和连接早就完成了，到这时候再来为 PE 映像保留空间早就为时已晚。正因为如此，又不想触及内核，那就只能由 wine-preloader 过渡一下了。然而，要保留本进程的一部分用户空间，如果是在内核中实现的话，那是轻而易举的事。而 Wine 之所以搞得这么繁琐，完全就是因为不想触及内核。

从以上的叙述和分析可以看出，Wine 的 PE 映像装入/启动过程存在着两方面的问题。首先是心理和使用习惯方面的，要用户在需要启动 notepad.exe 时必须键入“wine notepad.exe”会使用户感到别扭。另一方面则是效率方面的。

其实前者是比较容易解决的，最简单的办法是为每个目标程序准备一个脚本，例如为 notepad.exe 准备一个脚本 notepad，其内容可以是类似于这样：

```
#! /bin/sh
wine-preloader wine-kthread notepad.exe
```

当然，这样一来效率比使用工具 wine 更低了，但是这毕竟是在人机界面上。

更好的办法当然是从内核着手。比较简单的办法是让内核在检测到目标映像是 PE 格式时把命令行“notepad …”改成“wine-preloader wine-kthread notepad.exe …”，然后实际装入并启动 wine-preloader 的映像(这是 ELF 格式的映像)。这样效率比前者高一些，但还是多转了一道弯，所以还可以改进。再进一步，可以在内核中先为 PE 映像保留好空间，而跳过 wine-preloader。

如果说通过键盘命令启动 Windows 软件毕竟是人机交互，效率低一些也关系不大；那么通过 CreateProcessW()创建新进程的过程就比较敏感了。对于经常要创建新进程的应用，这种效率上的降低可能是很容易被觉察的。