

灵活使用 Rocky4ND 进行动态代码加密

1. 引言

Rockey4ND 是一款被广泛应用于软件保护的 32 位无驱加密狗，通过它，可以保护软件不被非法复制，非授权访问和使用。与其它一些同类产品不同，ROCKEY4ND 加密锁实际上是个小型计算机，它配有 CPU、存储器和特制的中间件，可以与应用程序交互运行。您可以通过在加密锁中编写复杂的算法，然后在程序中调用这些算法来实现加密。这种加密方法是我们所建议的，因为它具备了极高的安全性，很难破解。虽然 Rocky4ND 在硬件上已经足够强大，但是它如何应用于被保护的软件才是最重要的。大家想象一下，一个生活在与世隔绝的原始部落的人，即使拥有一部最先进的电脑，它也无法让它显示哪怕一秒钟的 Windows 画面。

相信大家对于如何利用 Rocky4ND 进行软件保护已经有了很深的理解，本文提出一个比较新颖的思路供大家借鉴，灵活使用 Rocky4ND 进行动态代码加密。

2. 动态代码加密

动态代码加密，简称 SMC (Self Modifying Code) 技术，就是一种将可执行文件中的代码或数据进行加密，防止别人使用逆向工程工具对程序进行静态分析的方法，只有程序运行时才对代码和数据进行解密，从而正常运行程序和访问数据。

下面是使用 SMC 技术的典型应用：

```
if (满足运行条件)
    // 解密某个函数或数据
    Decrypt (Adress_Of_Proc_Or_Data);
Else
    // 再对代码进行加密，防止程序被 Dump
    Encrypt (Adress_Of_Proc_Or_Data);
```

在自己的代码中使用 SMC (代码自修改) 技术可以大大提高软件的加密强度，保护私有数据和关键功能代码，对防止软件破解也可以起到很好的作用。下面我们就利用 Rocky4ND 来实现动态代码加密。

3. Rocky4ND 加密锁的硬件特性

ROCKEY4ND 加密锁硬件的核心是一个 CPU，加密锁的计算工作主要由它来完成。加密锁内部还有一片存储器芯片，存储的数据掉电后不会丢失。我们根据不同的功能把它划分成用户内存区、模块区和算法区以及用户 ID 区。开发者则可以将软件的一些重要信息 (如序列号等) 保存在加密锁中，需要注意的是，每个存储单元可写 10 万次，读的次数不受限制。10 万次是个很大的数，一般完全可以满足绝大多数开发者的要求，只要不当成内存单元来使用就好。专用芯片内置了随机数生成器、种子码生成器、用户自定义算法的解释器等。

4. 利用 Rocky4ND 加密锁进行动态代码加密

4.1 PE 文件格式格式简介

为了实现动态代码加密，首先我们得了解可执行文件的文件格式。Microsoft 为它的 32 位 Windows 系统设计了一种全新的可执行文件格式，被称为“Portable Executable”，也就是 PE 格式，PE 格式的可执行文件适用于包括 Windows 9X、Windows NT、Windows 2000、Windows XP 以及 Windows 2003 在内的所有 32 位操作系统。PE 文件格式将文件数据组织成一个线性的数据结构，图 1 展示了一个标准 PE 文件的映像结构：

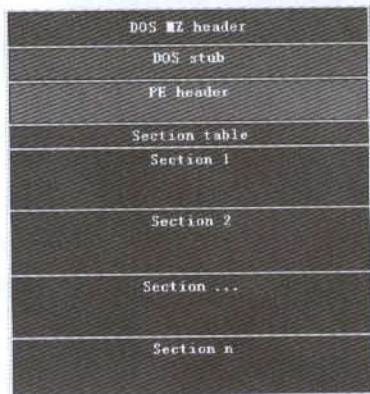


图 1 PE 文件映像结构

PE 格式文件的使用,使得 Windows 加载可执行文件不用再象以前一样将可执行文件拆开,在内存中东一块西一块地放置,取而代之的是一种简单的加载方式,就是按照顺序将 PE 文件读取到内存中,这也使得加载到内存中的 PE 文件和存放在磁盘上的 PE 文件具有相似的结构,只是各个段因为对齐方式的不同而导致偏移位置略有不同。本文只是简单的介绍了 PE 文件的格式及加载方式,如果要更深入的了解 PE 文件格式,请参阅专门的文档或书籍。

4.2 关键代码的定位

显然我们不能对全部代码段加密,因为如果整个程序代码都被加密,那么将没有机会再对其解密,从而造成程序加载运行失败。

首先我们要选择一个或多个函数,函数的选择应该具体针对性,应该选择那些关键的,而且会经常被调用的,这样对其加密才能起到效果,不然即使进行了加密,如果程序根本运行不到这个函数,也就失去加密的意义。选择好了进行加密的函数,接下来就需要定位进行动态代码加密的函数体的起始地址和长度。这里有个问题,我们是否要对整个函数体进行加密呢?在此我们的回答是不需要,因为每个函数体生成的二进制代码都有很多冗余,在函数体开始的地方有很多压栈操作,在函数体结束时有许多弹栈操作,这些都没有必要进行加密,即使加密了也没有太大的意义,我们只选择一些关键代码进行加密即可,这样可以提高加解密的效率,毕竟加密后的性能也是我们必须要考虑的因素,不能因为代码被加密就严重影响其性能。下面用一个简单的例子进行说明。

```
int Calculate(int x,int y)
{
    int z;
    z=x^y;
    return z;
}
```

对于这个简单的函数,它的关键代码就是 $z=x^y$;只要这条语句对应的二进制代码被加密了,我们的目的也就达到了。现在的问题就是我们如何定位到这条语句对应的二进制代码。对于这个问题,可能有很多不同的解决办法,利用特征码就是一种常见

的简便的方法。使用嵌入式汇编是最简单的嵌入特征码的方法,如下所示:

```
int Calculate(int x,int y)
{
    int z;
    goto Label;
    __asm{
        _emit 'S';
        _emit 't';
        _emit 'a';
        _emit 'r';
        _emit 't';
        _emit 0x00;
    }
Label:
z=x^y;
return z;
__asm{
    _emit 'E';
    _emit 'n';
    _emit 'd';
    _emit 0x00;
}
}
```

利用嵌入的特征码很容易就定位到“ $z=x^y$ ”对应的二进制代码,并获取它的长度。但这种方法有一定局限性,一来熟悉汇编语言的人不多,二来在程序中会产生很多跳转语句。本文使用一种使用 C/C++ 的语句构造特征码的方法。在 C/C++ 语言中,将常数赋值给某个变量的简单赋值语句,通常可以被翻译成一条简单的汇编代码,以下面的 C/C++ 代码为例:

```
DWORD dwTagCode = 0; // 定义一个全局变量
dwTagCode = 0xAFAFAF;
这条赋值语句汇编成机器代码后就是:
mov DWORD PTR [AAAAAAAA], AFAFAFH
```

这里 AAAAAAAH 表示的是 dwTagCode 的地址，最终生成的二进制代码就是：

```
C7 05 AA AA AA AF AF AF AF
```

我们以这 10 个字节为标记，在想要被加密的代码之前之后加入这样的赋值语句，通过查找标记就可以很方便的获取要加密代码的起始地址和长度。

加入标记的函数如下所示：

```
DWORD dwTagBegin=0;// 全局变量
DWORD dwTagEnd=0;// 全局变量
int Calculate(int x,int y)
{
    int z;
    dwTagBegin =0xAFAFAFAF;// 开始标记
    z=x^y;
    dwTagEnd =0xAEAEAEAE;// 结束标记
    return z;
}
```

查找标记的函数如下所示：

```
BYTE* FindTagCode(void *pStartAddr, unsigned
long *pTagLoc, unsigned long lTagValue, int
nSearchLength)
```

```
{
    int nPos = -1;
    int i = 0;
    unsigned char *pAddr = (unsigned char *)
pStartAddr;
```

```
// 如果是 Debug 版本，则需要跳过调试信息
```

```
// 判断是否是跳转指令
```

```
if(*((unsigned char*)pAddr)==0xe9)
```

```
{
    pAddr++;// 跳过 0xe9 指令
    //多加4个字节是因为操作数本身是4个字节
    pAddr+=((* (int*)pAddr)+4);
}
```

```
while(i < nSearchLength)
```

```
{
    // 查找 mov 指令
    if((*pAddr == 0xC7) && (*(pAddr + 1)
== 0x05))
    {
        unsigned long *Loc = (unsigned
long *)((unsigned char*)pAddr + 2);
        // 此处的数据 *Loc 就是全局静态变
//量的地址
        if(*Loc == (unsigned long)pTagLoc)
        {
            unsigned long *Val = (unsigned
long *)((unsigned char*)pAddr + 6);
            // 此处的数据 *Val 就是常数
//lTagValue 值
            if(*Val == lTagValue)
            {
                nPos = i;
                return (unsigned char*)
pAddr+10;
            }
        }
        pAddr++;
        i++;
    }
    return NULL;
}
```

其中需要指出的是，函数中有一段跳过调试信息的代码。Debug 版本的程序与 Release 版本的程序的函数地址是不一样的，Debug 版本通常将一些调试信息放在函数代码开始之前，所以函数开始位置被转向到了一条跳转指令 jmp (0xE9)，这样 VC 的调试器就可以根据函数名定位到函数的调试信息，而这条跳转指令又能保证函数体代码被正确地执行，所以我们必须跳过这段调

试信息，才能得到函数代码的真正开始位置。

通过调用

```
PBYTE pBeginPos=FindTagCode((void*)
Calculate, &dwTagBegin, 0xAF0AF0AF, 0x200);
```

就可以获得关键代码的在内存里的起始地址。

```
PBYTE pEndPos=FindTagCode(pBeginPos,
&dwTagEnd, 0xAE0AE0AE, 0x200);
```

```
DWORD length=pEndPos-pBeginPos-10;
```

这样，就可以获得关键代码的长度。减10是因为要去除10个字节的标记。

4.3 利用Rockey4ND进行动态代码生成

通过上一节，我们已经可以定位到加密的函数体，下面我们利用Rockey4ND的内部存储区域，来实现代码的动态生成。

首先，在被加密的程序中选择想要动态代码生成的函数，从函数体中选择一段关键代码，并按4.2节所述在关键代码的前后设置好标记。

接下来，就是动态代码生成过程了。由于我们计划利用Rockey4ND来存储被选择的关键代码，所以动态代码生成的过程就是从Rockey4ND中指定的内存区域中读取数据并覆盖到内存中标记所指定的位置。通常代码段所在的内存是不允许写入的，我们需要改写其属性，这里有两种方法，用户可以选择使用。一种是利用windows API：

```
VirtualProtect(LPVOID lpAddress, DWORD
dwSize, DWORD flNewProtect, PDWORD
lpflOldProtect);
```

另一种是利用编译选项：

```
#pragma comment(linker, "/section:.text,
ERW")
```

然后从锁中的指定位置读出数据并覆盖到标记指定的内存区域，即可完成动态代码生成。如下所示：

```
BOOL Decrypt(BYTE *pPos, DWORD length)
{
    // 从锁中读取指定的代码，
    // 并恢复到相应的位置
```

```
WORD handle, p1, p2, p3, p4, retcode;
DWORD lp1, lp2;
BYTE buffer[1024];
p1 = 0xc44c;
p2 = 0xc8f8;
retcode = Rockey(RY_FIND, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, buffer);
if(retcode)return FALSE;
retcode = Rockey(RY_OPEN, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, buffer);
if(retcode)return FALSE;

p1=0;
p2=length;
retcode=Rockey(RY_READ, &handle, &lp1, &lp2,
&p1, &p2, &p3, &p4, pPos);
if(retcode)return FALSE;

retcode=Rockey(RY_CLOSE, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, buffer);

return TRUE;
}
```

值得注意的是，关键代码的长度和锁中保存的代码长度必须相同，否则运行时会发生不可预见的错误。另外为防止运行时的异常错误，在运行包含动态生成函数代码的地方，最好加入异常处理机制，捕获异常，避免程序崩溃。

接下来，我们利用一个加密程序对PE文件进行加密处理，将关键代码从PE文件中去除，并复制到锁中，在运行时再自动生成。这样，如果没有正确的锁，就无法恢复关键代码，从而使得程序的功能得到限制或者根本无法使用。

如何定位PE文件中关键代码的位置？我们知道，windows加载PE文件只是将PE文件简单的映像到内存中，加载到内存中的PE文件和存放在磁盘上的PE文件具有相似的结构。因为我们已经在代码中做了标

记，所以我们只需要得到代码段的起始地址，然后开始在代码段中查找指定的标记即可定位关键代码，然后将关键代码写入到锁中，再将原关键代码用随机数填充，或者更改为其它代码，只要其它人无法从更改后的代码恢复到原有代码即可。最好的办法是更改一些运算再写回去，这们程序可以运行，但运行结果面目全非。比如将 $z=x^y$ ；改为 $z=x&y$ ；这样虽然程序运行不会出错，但函数的功能完全改变了。另外需要注意：更改后的代码与原代码的长度一定要保持一致。

加密函数如下所示：

```

BOOL Encrypt( BYTE *pPos, DWORD length)
[
    // 加密代码段
    // 将代码段写入锁内，然后再将原代码清除或
    // 写入错误的代码来混淆视线
    WORD handle, p1, p2, p3, p4, retcode;
    DWORD lp1, lp2;
    BYTE buffer[1024];
    p1 = 0xc44c;
    p2 = 0xc8f8;
    retcode = Rockey(RY_FIND, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, buffer);
    if(retcode)return FALSE;
    retcode = Rockey(RY_OPEN, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, buffer);
    if(retcode)return FALSE;

    p1=0;
    p2=(WORD)length;

```

```

    retcode=Rockey(RY_WRITE, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, (unsigned char*)pPos);
    if(retcode)return FALSE;

    retcode=Rockey(RY_CLOSE, &handle, &lp1,
&lp2, &p1, &p2, &p3, &p4, (unsigned char*)pPos);

    // 清除源代码
    while(length-->0)
    {
        // 也可以为随机数，或专门设计的代码
        *pPos=0x00;
        pPos++;
    }
    return TRUE;
}

```

5. 小结

本文提出一种利用 Rockey4ND 进行动态代码生成的方案，给用户使用 Rockey4ND 提供了一种新的思路，由于 Rockey4ND 的限制，保存在 Rockey4ND 中的代码无法在锁中直接运行，这多少有点缺憾。但幸运的是，飞天诚信科技有限公司的高端产品 Rockey6Smart，具备在锁中直接运行代码的能力，如果采用这款产品，就可以得到更加完善的加密方案。总之，加密锁的本身是固定的，灵活运用才是关键，相信在用户的不断探索和发掘下，会有更多更新的方法出现，使我们的软件保护方案更加完美。

