

MS Word 加密算法弱点利用

MS Word 是微软公司出品的一款常用的文字编辑软件, 是该公司 OFFICE 系列软件的一种。该软件提供加密被编辑文档内容的功能。当前, Word97 (包括 Word97) 以后的各个版本的 Word 为了保持向下兼容, 默认情况下均使用 97/2000 兼容的加密算法。但是该加密算法在实现时存在一些弱点, 加密强度在最好的情况下也仅相当于 40 位密钥的加密强度, 存在安全隐患。

本文通过阅读开源软件 wvWare 的源代码, 分析了 Word97/2000 加密算法的细节, 最后利用 wvWare 软件中提供的一些公用库, 编写了一个对使用 Word97/2000 加密算法进行攻击并理论上可以保证 100% 破解率的程序。

试验环境:

本文所涉及的所有运行环境是: 一台安装 Redhat 9.0 的 VMware 虚拟机。当前 wvWare 的最高版本为 1.2.1。但因为在 Redhat 9.0 中安装该版本的 wvWare 时要升级 gObject 在内的多个组件, 所以试验中我取了较低版本的 wvWare。具体是 wvWare 0.7.2 (源码包可以从网上自由下载, 或从本文附带的光盘中获得)

wvWare 简介及安装过程:

wvWare 是一款可以转换 MS Word 中内容其他格式的软件, 该软件的主页地址为: <http://wvware.sourceforge.net/index.html>

从 sourceforge.net 网站的 CVS 中下载 wvWare 0.7.2。执行 `tar zfxv wv-0.7.2.tar.tar` 解包, 之后 `cd wv-0.7.2` 进入 wvWare 的目录, 依次执行 `./configure`、`make`、`make install` 三条命令安装 wvWare。安装完毕后就可以使用 wvWare 了。

wvWare 可以通过用户输入的口令来读取并转换加密的 Word 的内容。口令的参数是: `--password=xxxx` (其中 xxxx 代表真正的口令)。如我们可以用光盘中的 123.doc 来做测试。(光盘中的 123.doc 文件的加密口令是 123, 文档的内容是 1234, 是使用 Word 2003 创建的) 键入命令 `wvWare -password=123 123.doc` 我们得到 123.doc 转换成 html 格式后的结果。

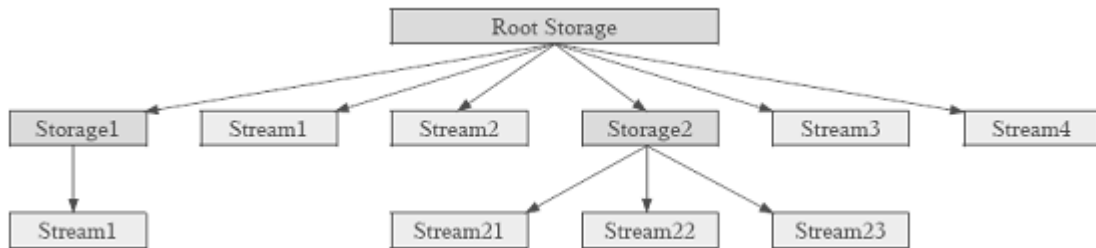
有关 wvWare 对 Word 文档进行解密的代码大部分在源码包中的 `decrypt97.c` 文件中, 另外 `wvWare.c` 中有少部分预处理代码。所以我主要对 `decrypt97.c` 进行了注释。详见附件光盘。

Word 文件二进制结构

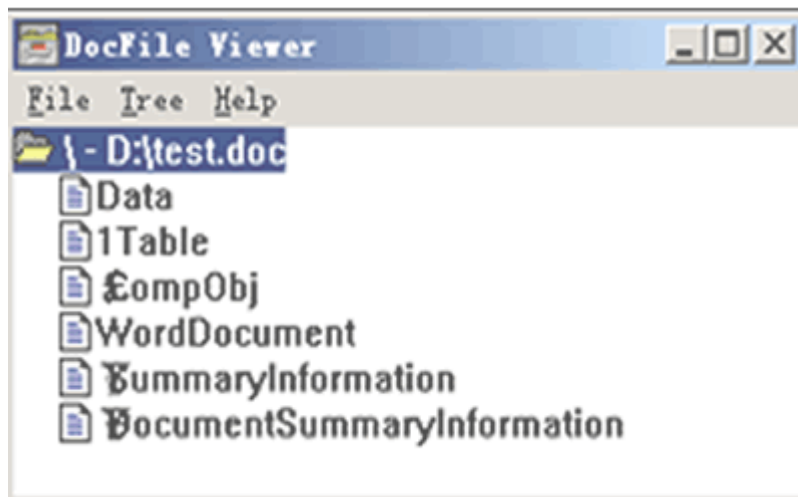
office 系列软件所产生的文档是 Microsoft Compound Document (微软复合文档) 的一种, 其文档格式与微软复合文档的文档格式一致。(注: 微软复合文档在 WINDOWS 操作系统中有着广泛的应用, 比如 WIN-XP 操作系统中的 Thumbs.db 文件也是一种微软复合文档)。

而微软复合文档则是一种结构化的储存文件，这种文件由微软的 OLE Structured Storage API 来创建和管理。

下面这张图就是复合文件的一个例子：每个文件都有且只有一个根目录，根目录下可以有若干个 stream 和 storage，stream 中存放数据，而 storage 中存放其他 stream 和 storage。



下面这张图是一个典型的带图片的 word 文档的结构，用的是链表式的数据结构。（使用 VC 自带的 DocFile Viewer）



data stream 中（在一个 Word 文档中只有带了图片才会有该 stream）是图片数据，WordDocument stream 中是文本数据，SummaryInformation 和 DocumentSummaryInformation stream 中是摘要信息，等等。详见《OpenOffice.org's Documentation of the Microsoft Compound Document File Format》（下载地址：<http://sc.openoffice.org/compdocfileformat.pdf>）

wvWare 中通过 wvInit () 和 wvInitParser () 两个函数来初始化一个 word 文档并提取其中有关 stream。其中 1Table stream0 的首地址被放在 ps->tablefd 中；（ps 是一个 wvParseStruct 的指针）WordDocument stream 的首地址被放在 ps->mainfd 中。


Word 加密机制

当一个 word 文档被加密后，并不是文档中所有的数据都会被加密，只有 1Table、WordDocument 等带有文本、图片等数据的 stream 才会被加密。（仔细分析附带光盘中带有注释的 decrypt97.c 的 172~244 行代码，你会发现 Word 加密各个 stream 时使用的是同一个密钥，这一做法也是违反密码学原则的。详见《The Misuse of RC4 in Microsoft Word and Excel》（下载地址：

http://eprint.iacr.org/2005/007.pdf)) 请注意 1Table stream, 它是我分析的重点: 如果文档被加密了测试有关口令正确与否的信息就存放在该 stream 中。如图 1 所示:

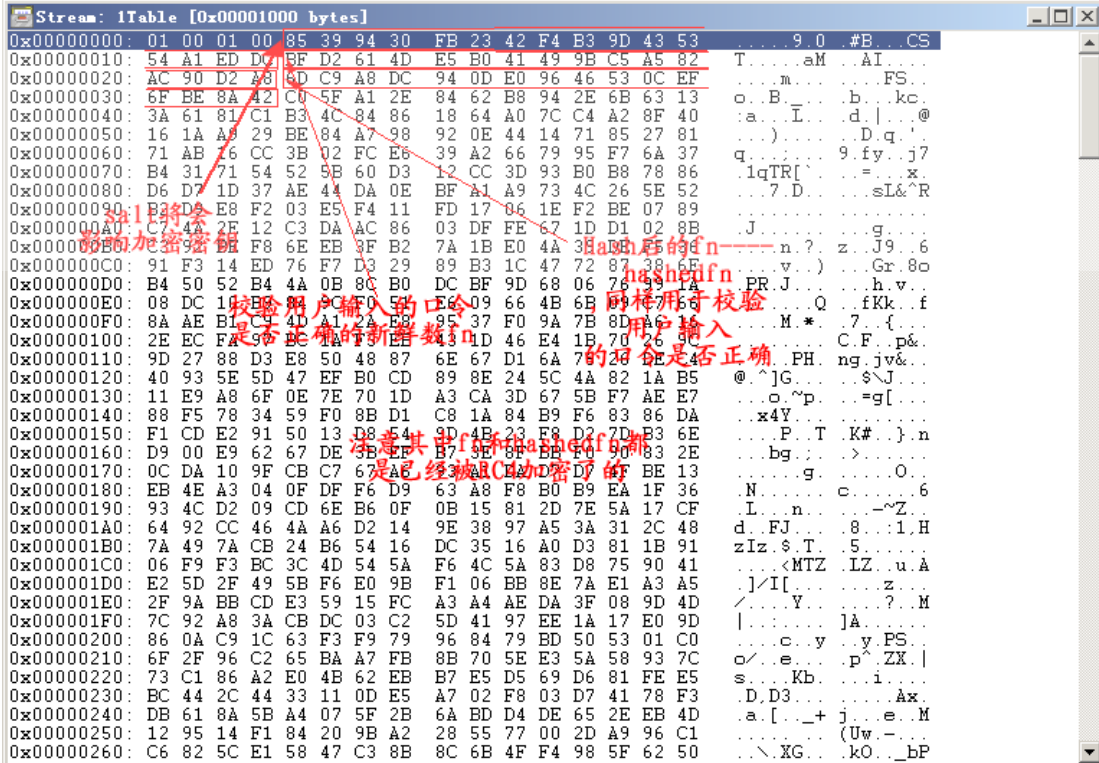
1Table 中一共有 3 个 16 字节的域, 其中第一个域中存放的是 salt, 用来与用户输入的口令一起生成加/解密的密钥(计算过程如图 2 所示)。第二个域中存放的是系统随机产生的一个 128 位的新鲜数被 RC4 加密后的结果。第二个域中的新鲜数被加密前先将其扩展为一个 64 字节的字节串, 然后对所得的 64 位字节串计算 MD5 hash 的结果, 这个 hash 值再被 RC4 加密后被存放在第三个域中。如图 3 所示。

注意, 第二个域和第三个域中存放的数据是用来校验用户输入的口令是否正确的, 具体过程如下: 当一个被加密的 word 文档被打开时, Word 程序结合用户输入的口令和第一个域中的 salt 计算出 40 位的决定 RC4 初始化向量的数, 再用它产生 RC4 初始化向量, 进而使用 RC4 解出第二个域和第三个域中存放的新鲜数及其 hash 的明文, Word 将再次计算新鲜数的 MD5 hash, 并将其结果与第三个域中存放的结果相比对, 如果二者相同, 则说明用户输入了正确的口令; 反之, 则说明用户输入了错误的口令。如图 4 所示。

在上述各个图中, 为了便于描述和理解, 我做了一些简化, 即 decrypt97.c 中整个 makekey() 函数我直接用一个  符号加以代替了, 实际上该函数的处理过程如图 5 所示。

需要说明的是: 在 decrypt97.c 源码中 docid 起的是 salt 的作用, 而 salt 和 hashedsalt 实际上是新鲜数及其 hash。命名上有点不规范。但是因为攻击软件编写过程中援引了 decrypt97.c 中一些代码, 所以就将错就错对新鲜数及其 hash 仍然使用了 salt 和 hashedsalt 的名字。

更多内容详见 decrypt97.c 中的注释。



```
Stream: 1Table [0x00001000 bytes]
0x00000000: 01 00 01 00 85 39 94 30 FB 23 42 F4 B3 9D 43 53 .....9.0.#B..CS
0x00000010: 54 A1 ED DC BF D2 61 4D E5 B0 41 49 9B C5 A5 82 T.....aM..AI...
0x00000020: AC 90 D2 A8 BD C9 A8 DC 94 0D E0 96 46 53 0C EF .....m.....FS..
0x00000030: 6F BE 8A 42 C0 5F A1 2E 84 62 B8 94 2E 6B 63 13 o..B.....b...kc.
0x00000040: 3A 61 81 C1 B3 4C 84 86 18 64 A0 7C C4 A2 8F 40 :a..I.....d.|...@
0x00000050: 16 1A A8 29 BE 84 A7 98 92 0E 44 14 71 85 27 81 :.).....D.q...
0x00000060: 71 AB 16 CC 3B 02 FC E6 39 A2 66 79 95 F7 6A 37 q.....9.fy..j7
0x00000070: B4 31 71 54 52 9B 60 D3 12 CC 3D 93 B0 B8 78 86 ..lqTR[.....=...x
0x00000080: D6 D7 1D 37 AE 44 DA 0E BF A1 A9 73 4C 26 5E 52 ...7.D.....sL&^R
0x00000090: B3 D8 E8 F2 03 E5 F4 11 FD 17 06 1E F2 BE 07 89 .....J.....g....
0x000000A0: C7 4A 2E 12 C3 DA AC 86 03 DF FE 67 1D D1 02 8B .....J.....g....
0x000000B0: F8 6E EB 3F B2 7A 1B E0 4A 3B 0E 4E 43 0E 44 .....n?..z..J9..6
0x000000C0: 91 F3 14 ED 76 F7 D3 29 89 B3 1C 47 72 87 38 6E .....n?..z..J9..6
0x000000D0: B4 50 52 B4 4A 0B 8C B0 DC BF 9D 68 06 76 99 7A .....PR.J.....h.v..
0x000000E0: 08 DC 19 B8 9C F0 5B E6 09 66 4B 6B 07 69 7A .....Q.....fKk..f
0x000000F0: 8A AE B1 94 4D A1 2A EB 95 37 F0 9A 7B 8D A6 16 .....M.*..7.{...
0x00000100: 2E EC FA 99 0E 2A 1F EB 43 1D 46 E4 1B 70 26 9C .....C.F..p&..
0x00000110: 9D 27 88 D3 E8 50 48 87 6E 67 D1 6A 7B 18 E4 4 .....PH..ng.jv&..
0x00000120: 40 93 5E 5D 47 EF B0 CD 89 8E 24 5C 4A 82 1A B5 @.^jG.....$J...
0x00000130: 11 E9 A8 6F 0E 7E 70 1D A3 CA 3D 67 5B F7 AE E7 .....o.^p.....=g[...
0x00000140: 88 F5 78 34 59 F0 8B D1 C8 1A 84 B9 F6 83 86 DA .....x4Y.....
0x00000150: F1 CD E2 91 50 13 D8 54 9D 4B 23 F8 D2 7D B3 6E .....P..T..K#..}..n
0x00000160: D9 0A E9 62 67 DE 3B EF B7 0E BF BB 51 90 83 2E .....bg.....>...
0x00000170: 0C DA 10 9F CB C7 67 AE 12 0A 07 07 07 07 07 07 .....g.....O...
0x00000180: E8 4E A3 04 0F DF F6 D9 63 A8 F8 B0 B9 EA 1F 36 .....N.....c.....6
0x00000190: 93 4C D2 09 CD 6E B6 0F 0B 15 81 2D 7E 5A 17 CF .....L.....n.....~Z..
0x000001A0: 64 92 CC 46 4A A6 D2 14 9E 38 97 A5 3A 31 2C 48 d..FJ.....8...1.H
0x000001B0: 7A 49 7A CB 24 B6 54 16 DC 35 16 A0 D3 81 1B 91 zIz.$..T...5...
0x000001C0: 06 F9 F3 BC 3C 4D 54 5A F6 4C 5A 83 D8 75 90 41 .....<MTZ..LZ..u.A
0x000001D0: E2 5D 2F 49 5B F6 E0 9B F1 06 BB 8E 7A E1 A3 A5 ..]I[.....z...M
0x000001E0: 2F 9A BB CD E3 59 15 FC A3 A4 AE DA 3F 08 9D 4D /...Y.....?..M
0x000001F0: 7C 92 A8 3A CB DC 03 C2 5D 41 97 EE 1A 17 E0 9D |.....]A.....
0x00000200: 86 0A C9 1C 63 F3 F9 79 96 84 79 BD 50 53 01 C0 .....c..y...y.PS..
0x00000210: 6F 2F 96 C2 65 BA A7 FB 8B 70 5E E3 5A 58 93 7C o/.e.....p..ZX..|
0x00000220: 73 C1 86 A2 E0 4B 62 EB B7 E5 D5 69 D6 81 FE E5 s...Kb.....i...
0x00000230: BC 44 2C 44 33 11 0D E5 A7 02 F8 03 D7 41 78 F3 .D.D3.....Ax..
0x00000240: DB 61 8A 5B A4 07 5F 2B 6A BD D4 DE 65 2E EB 4D .a.[...+j...e..M
0x00000250: 12 95 14 F1 84 20 9B A2 28 55 77 00 2D A9 96 C1 .....(Uw...-
0x00000260: C6 82 5C E1 58 47 C3 8B 8C 6B 4F F4 98 5F 62 50 ...\.XG...kO...bP
```

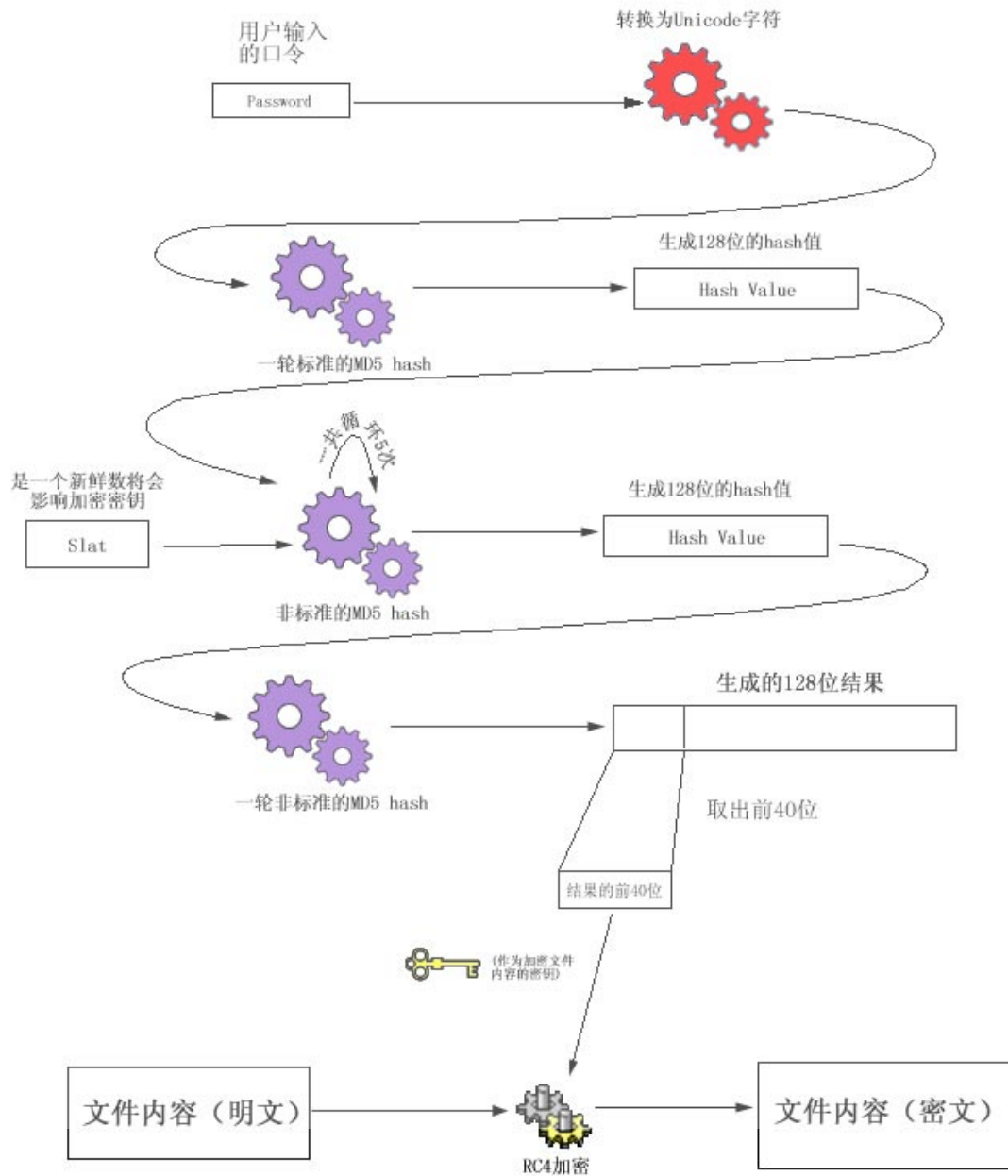


图 2 Word 文档加密算法

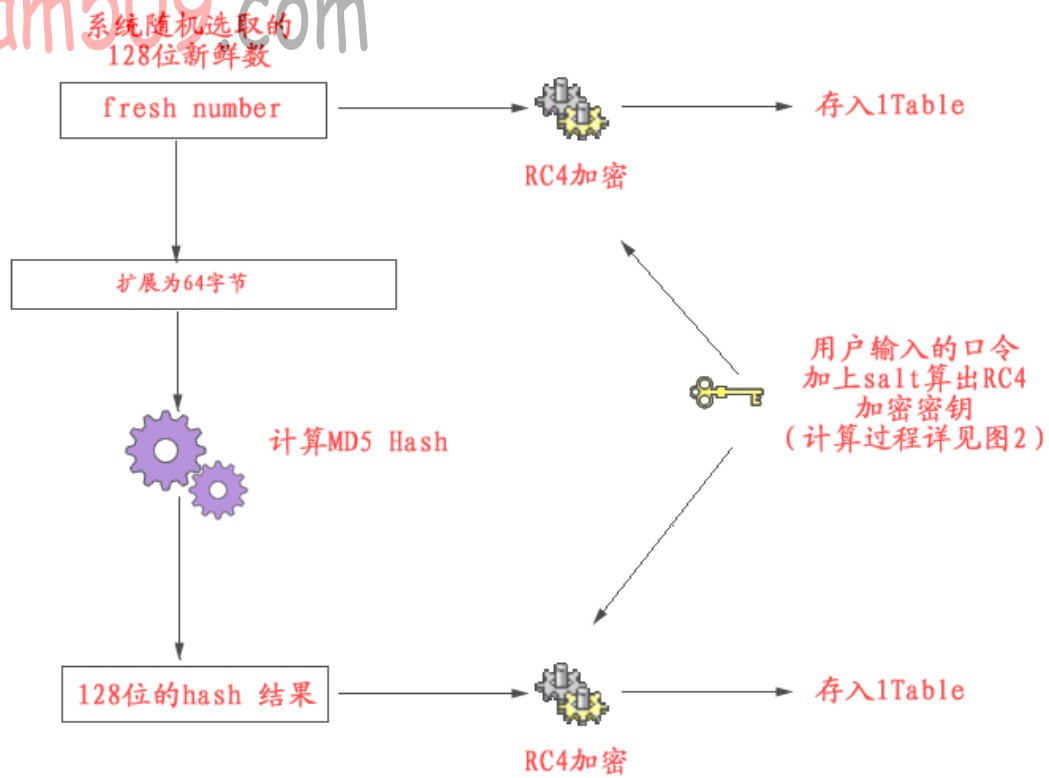


图 3

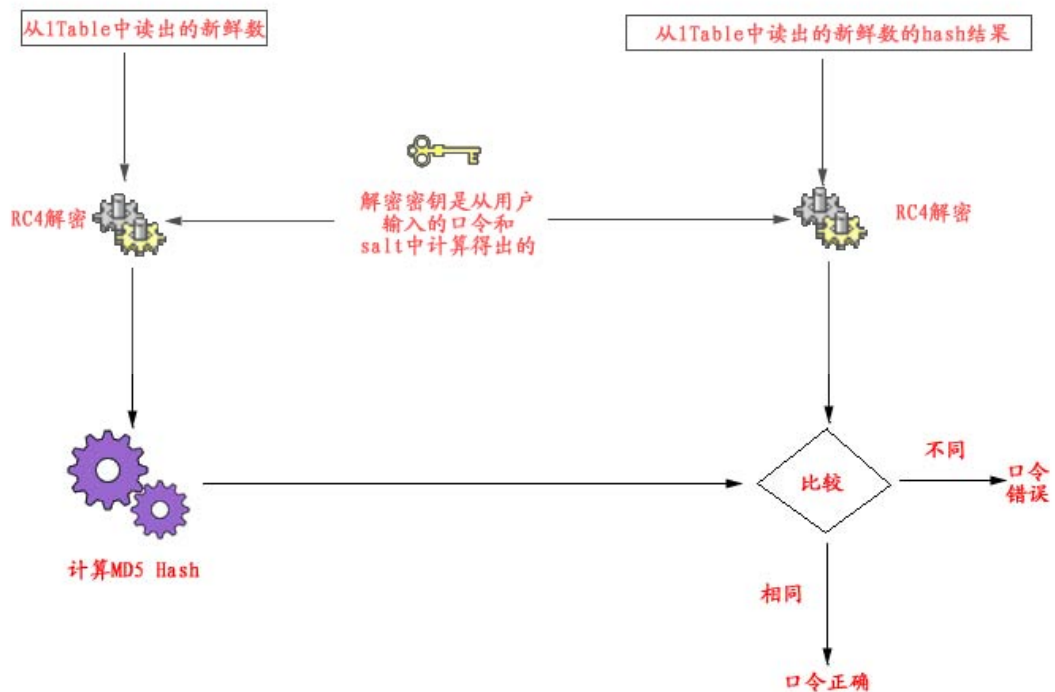


图 4

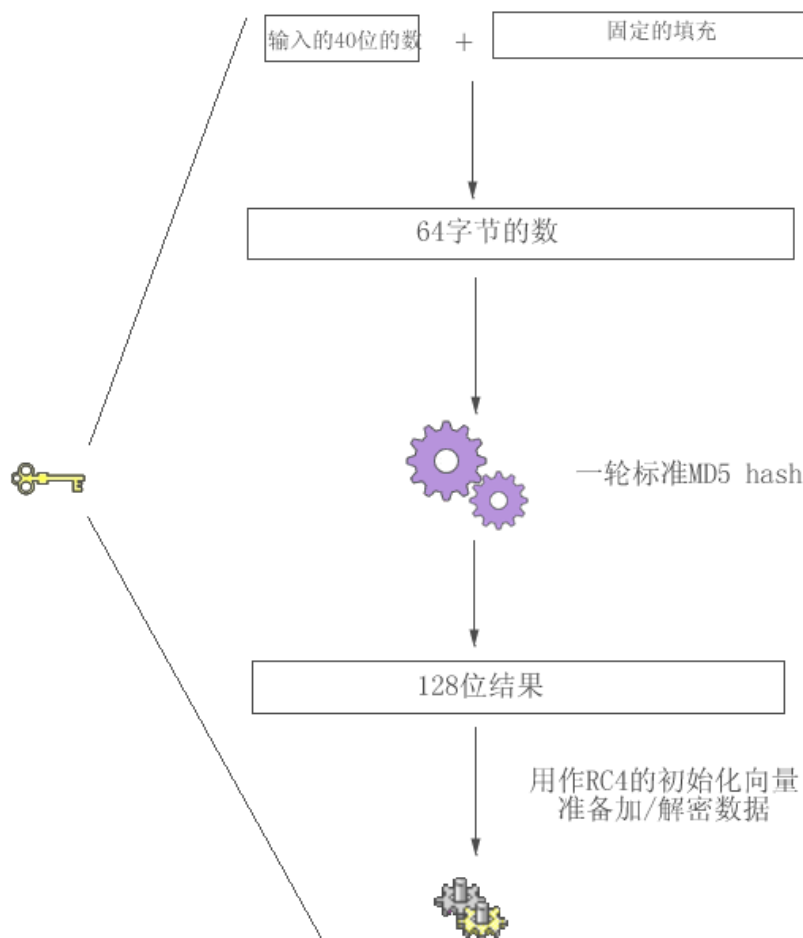



图 5

了解了 Word 的有关加密算法后，就可以着手进行攻击了。

两种攻击方案的比较：

第一种攻击方案：猜测用户的口令字，每猜一个口令就加上 1Table 中存储的 salt 计算出决定 RC4 初始化向量的那个 40 位的数（按图 2 所示的流程）。再按图 4 所示的流程校对这个 40 位数是否正确。进而推出所猜的口令字是否正确。如果正确则停止攻击返回正确的口令字，如果不正确则继续猜测下一个口令。

第二种攻击方案：直接猜测决定 RC4 初始化向量的那个 40 位的数（就是从  的位置开始攻击），每猜一个就按图 4 所示流程校对它是否正确。如果正确就停止攻击，直接解出整个文档的明文，如果不正确，则继续猜测下一个 40 位数。

显然对方案一如果用户选择了一个弱口令，我们可以较快的猜出用户的口令，这是方案一好的一面。但是如果用户选择了一个强度较大的口令时，该种攻击很可能就无效了，这也就是为什么用“Accent Office Password Recovery”、

“Advanced Office Password Recovery”之类的软件对 office 文档进行破解时，但是一旦被加密的文件口令大于 7 到 8 个字符就基本上不能破解出加密的信息的原因。

我们现在考虑密钥/口令字空间问题：决定 RC4 初始化向量的那个 40 位的数的空间是 2^{40} （近似于 10^{12} ）。另外我们设用户的口令全由数字构成，那么假设我们猜测一个最大长度为 n 位的口令字，口令的密钥空间就等于 $10^n + 10^{n+1} + \dots + 10$ ，显然当 $n \geq 12$ 时，直接猜测 40 位的 RC4 加密密钥就比较合算。（这里我们忽略了将口令转换为 RC4 加密密钥的开销）。如果用户口令更复杂一些，比如包含字母，大小写以及特殊字符的情况， n 的值就会进一步减小。显然对于一些高强度的口令使用方案二更有效。

方案二的问题在于它不能返回用户的口令字！但是有些时候我们还是需要口令字的，所以作为方案二的一个改进，我们可以在获得决定 RC4 初始化向量的那个 40 位的数后，再行猜测口令字。当然由于已经有了决定 RC4 初始化向量的那个 40 位的数我们可以简化一下方案一，即每猜一个口令就加上 lTable 中存储的 salt 计算出这个 40 位的数后就可以直接判断该 40 位数是否正确了。（这一步改进在试验攻击程序中没有实现）

口令字的有关问题：

但是我们回过头来想一下获得正确的口令字可能吗？

显然对于给定的一个 salt（即对于一个特定的文档）可以有多个口令字（其中有一个是用户的原始口令）可以正确的解密整个 Word 文档！听上去有点不可思议对吗？但事实确实如此！

证明如下：

对于给定的 salt，对于每个口令会有一个且仅有一个决定 RC4 初始化向量的 40 位的数与之对应（这个 40 位数正确与否先不去管他），但是反过来，每个决定 RC4 初始化向量的 40 位的数只对应一个口令吗？显然不是。因为口令的空间是无限大的（设口令长度不限），但是 40 位数的空间却只有 2^{40} 个，根据鸽笼原理，每个决定 RC4 初始化向量的 40 位的数就会对应多个口令。而由于 MD5 hash 将每个口令字都均匀的映射到 40 位数的空间上，所以我们平均在任意的 2^{40} 个口令字中（原始口令是否在其中无关）就可以找到一个口令对应于正确的那个 40 位数！即，这个口令字可以正确的解密整个 Word 文档。

举个不恰当的例子，你用方案一攻击一个 Word 文档（口令最好不是纯数字的）只使用数字进行暴力攻击，当长度大于等于 12 时，你会得到一个可以打开文档的解！

也就是说：对于一个以强口令保护的 Word 文档是不能保证恢复原始口令的！你最多只能得到一个可能是部分 Hash 冲突的解！（之所以称之为**部分** hash 冲突的解是因为它只需要在 128 位的 MD5 hash 结果中前 40 位冲突就可以了）

嗯，如此说来方案二就没有存在必要了，因为用方案一也可以达到同样的效果！

但是方案二还是有一定的优势的，我们在前面的分析中强调：这里我们忽略了将口令转换为决定 RC4 初始化向量的那个 40 位的数时的开销！事实上这个开销还是比较大的我们不能予以忽略！所以在同样遍历 2^{40} 大小的空间时，方案二比方案一快许多倍！（粗略的计算一下：由于 MD5 是整个算法中出现次数最多，

且最慢的算法，所以我们忽略memcmp、RC4等算法的时间开销，只计算MD5的计算次数。方案一中每测试一个口令字需要计算9次MD5 hash，但是方案二中每测试一个40位数只需要计算2次MD5 hash，效率是方案一的4.5倍！)

攻击试验程序说明

基于上面的分析我们选择使用第二种攻击方案进行攻击，写出了附带光盘中crack.c文件。其中wvMD5StoreDigest()、makekey()是基本直接使用decrypt97.c中的函数的，hashed_pwd是我们增加的，表示决定RC4初始化向量的那40位的数，所以是5字节大小的。verify_pwd()是用来测试某个特定的40位数是否正确的，attack()函数是用来遍历40位的空间的（使用递归的方法），正常攻击时，attack()函数的第一个参数x应该是0，（这个参数表示从hashed_pwd数组的第几个成员开始遍历）但是由于完成一个正常的攻击要花费很长的时间，所以，在测试时我们选择这个x为2（即只遍历hashed_pwd[2]、hashed_pwd[3]、hashed_pwd[4]这24位数）。

在编译crack.c之前我们还要小小的作弊一下。

我们在decrypt.c的第101行（未注释的版本）插入如下代码，“请”wvWare先告诉我们hashed_pwd的值是多少：

```
int i;
for (i=0; i<5; i++) {
    printf("0x%x ", valContext.digest[i]);
}
```

Printf("\n");

保存后，重新make和make install。

再次执行wvWare -password=123 123.doc |more

我们发现在输出的第一行有“0x4b 0x74 0x41 0x61 0xc6”

所以我们在main函数中插入代码设置hashed_pwd的值。

```
hashed_pwd[0]=0x4b;
hashed_pwd[1]=0x74;
hashed_pwd[2]=0x00;
hashed_pwd[3]=0x00;
hashed_pwd[4]=0x00;
```

我只设了前两位，后3位没有设，目的就是测试这个crack.c是否能找到正确的40位数。

我们把crack.c和123.doc（因为在crack.c中我是把123.doc以硬编码的方式放入的）拷入wvWare所在的目录中，然后键入“gcc -I. -I./glib-wv crack.c -L. -lwv -lm -lglib”编译，编译时会有警告信息输出，但没关系，因为在makewvWare时同样有这样的警告信息。

之后我们键入“./a.out”运行攻击程序。

等了一段时间后（因试验机的硬件配置不同，等待的时间会不同），屏幕输出：

Correct!

0x4b 0x74 0x41 0x61 0xc6

攻击成功!

我的虚拟机中这一过程持续了约 1 分钟,也就是说遍历 3 字节的 1/4 我用了约 1 分钟,所以遍历整个 3 字节可能要花 4 分钟左右的时间。你可以把 crack.c 中 hashed_pwd 初始化代码改为下面的情况:

```
hashed_pwd[0]=0x4b;  
hashed_pwd[1]=0x73;  
hashed_pwd[2]=0x41;  
hashed_pwd[3]=0x61;  
hashed_pwd[4]=0xc6;
```

再把 `attack(2, hashed_pwd, salt, hashedsalt);` 一句改成 `attack(1, hashed_pwd, salt, hashedsalt);` 就可以比较精确的测出遍历 3 字节的数所需要的时间了。

由于每增加一位,攻击时间加倍,所以遍历完一个 word 的 40 位数我们需要约 80 天时间,但这是最差情况,平均来讲只要遍历 40 位空间中的一半就能找到正确的 40 位数,所以一次完整的攻击可能只需要 40 天时间。

我再进一步:光拿到 40 位数的攻击还不是很直观,我们干脆把文档的内容解密出来,这一步骤实际上很简单:我直接抄 decrypt97.c 中 `wvDecrypt97()` 中第 170 行一下的代码,当然略做了调整,然后把 crack.c 命名为 decrypt97.c 把原来的 decrypt97.c 覆盖掉,然后重新 `make`、`make install`

然后键入 “`wvWare -password=xxx 123.doc`” (这时不用输入正确的口令,因为它不会被真正传递进来运算) 经过一些时候的等待后, `wvWare` 就把正确的文档内容转换成 HTML 格式输出了 😊

进一步的讨论

实际上这个 crack.c 的攻击还是有提速可能的。如我们可以使用多台计算机进行网络分布式破解,另外我们还可以用硬件来实现攻击。这样就能把攻击时间缩短到一个可以接受的时间了。

附：文中涉及的密码学知识：

1、hash 函数

密码学中所指的 hash 函数又称单向不可逆函数，它的定义较数据结构中使用的 hash 表的定义更加严格。本文中的 hash 函数是指这样一种计算过程：把任意长的输入消息串变化成固定长的输出串的一种函数。这个输出串称为该消息的 hash 值。一个安全的 hash 函数应该至少满足以下几个条件：

①输入长度是任意的。

②输出长度是固定的，根据目前的计算技术应至少取 128bits 长，以便抵抗生日攻击。

③对每一个给定的输入，计算输出即 hash 值是很容易的。

④给定杂凑函数的描述，找到两个不同的输入消息杂凑到同一个值是计算上不可行的，或给定杂凑函数的描述和一个随机选择的消息，找到另一个与该消息不同的消息使得它们杂凑到同一个值是计算上不可行的。

目前 hash 函数已有很多方案。这些算法都是伪随机函数，对于任意一个输入而言任何 hash 值都是等可能的。输出并不以可辨别的方式依赖于输入；在任何输入串中单个比特的变化，将会导致输出比特串中大约一半的比特发生变化。

Hash 函数主要用于完整性校验和提高数字签名的有效性，我们在下文可以看到：office 文件正是使用了 hash 函数对用户输入的口令字进行校验的

2、RC4 流加密算法

RC4 是一种流式加密算法，同时也是一种对称体制的加密算法，它由 RSA 安全公司的设计，流式加密将固定长度的密钥（IV 初始化向量）展开成为长度无限的伪随机数值串。这个伪随机数值串就是密钥流，将明文和密钥流进行异或就可以产生密文，解密时用户也必须拥有密钥，通过将密钥展开为密钥流（这个密钥流与加密时产生的密钥流是完全一样的），然后将这个密钥流与密文进行异或操作就可以还原明文了。在 Office 中对文件内容的加密是使用 RC4 算法的。

3、salt

一般 Hash 函数加密用户口令的过程是这样的：首先由于计算机在打开文件之前必须要识别用户输入的口令是否正确，所以必须把口令存放在文件的某个位置中，但是我们又不能把口令明文直接存放在文件中——因为这样攻击者就可以轻易的获得口令，获得加密后的信息——所以我们采用了一个变通的办法：我们对用户输入的口令计算 hash，把所得的结果存入文件。由于知道一个 hash 值是不能推出口令明文的（见 hash 的性质 4），所以这样可以安全的存放口令。当用户试图打开文件时，计算机把用户输入的口令再进行一次 hash 后将结果与存放在文件内的 hash 值作比较，如果相同则说明口令正确，不通则说明口令错误。

但是这种做法还是又一个问题：不能抵抗“预计算”攻击。我们假设口令只有两种可能或是 1，或是 0。使用上面所述的方法把口令计算 hash 后的结果存放在文件中，但是攻击者可以事先计算出 1 和 0 的 hash 值分别是多少，然后再与文件中存放的 hash 值做比较，如果文件中存放的 hash 值与 1 的 hash 值相同，则说明 1 是口令，如果文件中存放的 hash 值与 0 的 hash 值相同，则说明 0 是口令。该中攻击方式是相当有效的，最著名的软件是 rainbowcrack，它可以在 20 分钟通过比较预计算的结果内破解 15 位一下的 Windows 开机口令。

为了预防预计算攻击，我们使用 salt 方法。即在口令的 hash 计算过程中加入一个随机数从而使预计算的难度以几何级数的速度递增，最终使预计算的方法在计算量上不可行。我们的做法如下：还是设口令只有两种可能或是 1，或是 0，

但是我们在计算口令的 hash 时随机选择一个随机数 (salt) 比如 “689041”，然后把口令加上 salt，再计算其 hash——即计算 1689041 的 hash。另外这个 salt 也是以明文形式存放在文件中，这样当计算机验证用户输入的口令时就将用户输入的口令加上 salt 后与文件中的 hash 进行比较，而对于攻击者而言由于 salt 的存在它就无法只是简单的计算 0 和 1 的 hash，而是要不得不计算 0~199999 之间所有的数的 hash (我们设 salt 是 5 位的)，攻击难度就大大增加了。另外在 Word 和 Excel 中由于使用了 salt，即使是使用相同的口令对相同的两个文件进行加密，结果也是不同的。