《电子证据分析》课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 实验名称 |  | 姓名 |  |
| 班级 |  | 学号 |  |
| 实验日期 |  | 指导老师 |  |
| 实验环境  (实验软硬件要求) | Windows10、windowsXP、DAMN Hash Calculator、MS word、记事本 | | |
| 实验目的 | （给出本次实验所涉及并要求掌握的知识点）  了解字符编码的差异  了解对字符串和文件进行hash的操作 | | |

|  |  |
| --- | --- |
| 实验内容 | 1. 在win10中对字符串的哈希值进行记录，并且比较当文件内容与字符串哈希值一致时，两者哈希值是否一致，观察现象并且简述这样的原因。（哈希选择sha256）   在win10中，首先对数字进行哈希值的比较。  使用工具DAMN Hash Calculator。  在Text一栏中输入自己的手机号：    可以得到，我的手机号的哈希值为： 12EF0600FCC423BC6E06C2598ED65EDF4610231BD5C8A9A0C0780F248E16D4CB  随后，创建txt文件，内容为手机号：    计算hash得到：    发现两者的hash值是一致的，数字的hash在win10上默认是一样的。  随后尝试改变一下编码，看看文件的哈希会不会变化。将其进行另存为，将编码选择为Unicode编码，名称设置为2.txt    对其进行计算hash值，得到如下：    因为哈希计算的是文件在数据区的二进制流，所以可以肯定这个哈希的变化绝对不是因为自己命名不同导致，就是二进制流不同，打开winhex进行比较可以看到两者的16进制具体情况：      可以看到因为编码的变化，两者的十六进制也是截然不同的，而且有一个很重要的原因，txt文档使用unicode编码时会在开头默认带上0xFFFE，而在正常的情况下，win10直接打开txt文本的时候编码时utf-8，这种开头是没有0xFFFE的。除去头的区别，还能看到很重要的一点，虽然数字表示上两种编码都是一致的，但是使用Unicode编码时是宽字节的，一个字会占用两个字节也就是16bit（可能这就是Unicode也是UTF-16 LE的表现之一吧）而UTF-8一个字只会占用一个字节也就是8bit，显然这也导致了两者的流是不一样的，进而哈希也会不同。那么win10默认使用的编码是什么呢？这里将继续进行研究。（虽然字符串和初始UTF-8编码的txt文档哈希值一致，也许可能就是使用的UTF-8，但是这只是数字的哈希一样，接下来会尝试中文）  接下来使用中文进行同样的操作。  输入字符串“圆球”，然后可以得到哈希值如下：    可以知道hash值为：9175CF9887BF4A9589A85C30FD4DE90DFCA32285AE237432B80A2BD21B31A5C6  然后一样的操作，新建文档内容为“圆球”，随后计算hash：    果然，之前认为win10的默认编码就是UTF-8是不对的，此处哈希值不同，因此可以认为两者的编码也是不同的。  继续对这个文档进行另存为，修改编码格式。  此时选择为Unicode编码进行操作。    随后计算其哈希值：    值为DE92AAD1A72760FBD4E4039CD13C87C7AC9874CE47665B3356A53E2105553E5B  很显然不一致，那么就能说明不是了吗？    不一定，在使用winhex将txt文件开头的0xFFFE删除之后，再一次进行计算哈希值。得到如下：    这下可以确定的确不是Unicode编码了。  接下来使用ASNI试试看。  创建txt并且使用ASNI。    直接计算其hash值    确实是一样的。ASNI是这个ASCII编码的升级版，打开winhex尝试查看它和Unicode编码的区别。发现两个完全不一样……没得比较，如下：      通过上课时老师的讲解，得到了结果。在中国大陆，应该就是GB2312  这里使用VScode新建一个文档，然后将编码格式设置为GB2312，随后输入内容“圆球”，如下：    然后保存后，计算其hash值。    哈希值为：9175CF9887BF4A9589A85C30FD4DE90DFCA32285AE237432B80A2BD21B31A5C6  可以看到，它的哈希值是和之前的字符串以及ANSI的哈希值一致的。  查看GBK编码版本的二进制流：    就是与之前的ASNI一致的。  所以说，在win10（中国版）默认的编码就是GB2312。  另外，关于ASNI编码，在不同的地区是不一样的。以下是百度的解释：     1. 在XP中对字符串的哈希值进行记录，并且比较当文件内容与字符串哈希值一致时，两者哈希值是否一致，观察现象并且简述这样的原因。（哈希选择sha256）   首先输入字符串“圆球” ，如下：    得到的哈希值为：9175CF9887BF4A9589A85C30FD4DE90DFCA32285AE237432B80A2BD21B31A5C6  发现竟然还是默认使用GBK编码。  然后新建一个txt文档，输入内容“圆球”    保存后计算其哈希值。发现也是一样的。    果然是一致的，查看，因为XP默认直接使用的是ASNI……  然后将其另存为“新建 文本文档1.txt”，编码格式修改为Unicode：    计算哈希，果然也是不一样的。毕竟编码是不同的。    最后，可以得到这么一个结论。不同放入系统同一个程序得出的结果也会是不同的。像是在win10里面，它默认记事本编码是UTF-8，所以在计算哈希的时候自然是不一样的，但是在XP环境的时候，因为记事本使用的就是ASNI，因此就是会得到一样的哈希值。   1. 编写一个能够将三个文件进行异或的小程序，并且将异或之后得到的结果存放在另外一个文件中。   由于太久没有写过C语言程序了……手都生了，指针什么的忘记了，决定使用python来写了……  下面是源码，都有注释……就不多说了   1. # -\*- coding: gbk -\*- 2. **import** binascii 3. **import** struct  6. **def** BytToStr(string): 7. """此函数的作用就是将读取到的16进制字符流转变成十六进制字符串""" 8. hexs = [] 9. **for** s **in** string: 10. temp = hex(s).replace('0x','') 11. **if** len(temp) != 2: 12. temp.zfill(2) 13. # print(temp) 14. hexs.append(temp) 15. **return** hexs 17. **if** \_\_name\_\_ == "\_\_main\_\_": 18. f1 = open(r'1.txt','rb') 19. f2 = open(r'2.txt','rb') 20. f3 = open(r'3.txt','rb') 21. # 打开文件 23. hex1 = [] 24. hex2 = [] 25. hex3 = [] 26. # 每一个文件的数据会转换为十六进制被分别存放进其中 28. **while** True: 29. temp1 = f1.read() 30. temp2 = f2.read() 31. temp3 = f3.read() 32. # 每次操作先分别读取三个文件512个字节 34. **if** ((temp1 == b'') **or** (temp2 == b'') **or** (temp3 == b'')): 35. **break** 36. # 判断当读取文件获得为空的时候就直接跳出读取操作 38. hex1.extend(BytToStr(temp1)) 39. hex2.extend(BytToStr(temp2)) 40. hex3.extend(BytToStr(temp3)) 41. # 将获取到的字符流转换成了十六进制(字符串形式，这个后续还会改成int型) 43. f1.close() 44. f2.close() 45. f3.close() 46. # 读取完成关闭文件 48. len1 = len(hex1) 49. len2 = len(hex2) 50. len3 = len(hex3) 51. minlen = min(len1,len2,len3) 52. # 计算三个文件数据可以无脑异或的部分，后面会解释为啥需要这么做 53. midlen = max(min(len1,len2),min(max(len1,len2),len3)) 54. # 计算中间部分，这个部分属实麻烦…… 55. maxlen = max(len1,len2,len3) 56. # 计算最简单的部分，这个部分都不需要进行异或了，和0异或就是自己本身~ 58. **if** (len1 > len2): 59. **if** (len1 > len3): 60. **if** (len2 > len3): 61. **pass** 62. # 最好的情况，原本文件长度为1>2>3 63. **elif** (len3 > len2): 64. hex2,hex3 = hex3,hex2 65. # 此时原本的文件长度为1>3>2 66. **elif** (len3 > len1): 67. hex1,hex2,hex3 = hex3,hex1,hex2 68. # 此时原本的文件长度为3>1>2 69. **elif** (len2 >len1): 70. **if** (len1 > len3): 71. hex1,hex2,hex3 = hex2,hex1,hex3 72. # 此时原本文件长度为2>1>3 73. **elif** (len3 > len2): 74. hex1,hex2,hex3 = hex3,hex2,hex1 75. # 此时原本文件长度为3>2>1 76. **elif** (len2 > len3): 77. hex1,hex2,hex3 = hex2,hex3,hex1 78. # 此时原本文件长度为2>3>1  81. fout = open('out.txt','wb+') 82. **for** i **in** range(minlen): 83. a = int(hex1[i][0],16)\*16+int(hex1[i][1],16) 84. b = int(hex2[i][0],16)\*16+int(hex2[i][1],16) 85. c = int(hex3[i][0],16)\*16+int(hex3[i][1],16) 86. d = struct.pack('B',a^b^c) 87. fout.write(d) 88. # 这是三个文件数据中可以无脑异或的部分，这时候三个文件的数据都是需要进行异或的 89. **for** i **in** range(minlen,midlen): 90. a = int(hex1[i][0],16)\*16+int(hex1[i][1],16) 91. b = int(hex2[i][0],16)\*16+int(hex2[i][1],16) 92. c = struct.pack('B',a^b) 93. fout.write(c) 94. # 这是有一个文件的长度已经结束，但是还有两个文件需要异或的部分 95. **for** i **in** range(midlen,maxlen): 96. a = struct.pack('B',int(hex1[i][0],16)\*16+int(hex1[i][1],16)) 97. fout.write(a) 98. # 这是最后只有一个文件还没有结束的时候。 100. fout.close()   以上就是源码了，简单地对这个程序验证一下吧。因为这个程序是使用同目录下的1.txt、2.txt、3.txt作为输入，out.txt作为输出的。所以，我这边直接用十六进制查看器打开对应的文件。  1.txt的文件内容：    十六进制为：    2.txt的文件内容：    十六进制为：    3.txt的文件内容：    十六进制为：    就以第一个半个字节的直接为例吧。  分别为：e、e、c  进行异或的结果为c    使用python进行验算（拿python的shell验算python写的程序，不愧是我）    out.txt的十六进制就是如此~  实验完成。 |
| 实验总结 | （对本次实验涉及到的知识点的重难点归纳、实验心得、思考与建议）  本次实验1和2没有太大的难度，实验3因为比较长时间没有进行过编程了，手比较生了，花了比较长的时间。  编码在对于文件的hash是比较重要的，同样的文件内容因为使用不一样的编码也会导致不一样的结果，这一点是一定需要注意的。 |
| 评分 |  |