Flare-On 3: Challenge 3 Solution - unknown

Challenge Author: Dominik Weber

The third challenge is written in C and compiled into a Windows X86 executable that takes the password to complete this challenge as its command-line argument. We will brute-force this password by dumping a set of 26 32-bit hashes; one for every character of the password.

The trick

The download is named *unknown*; this is one hint that we have to find the proper executable name – the extension would be exe. A hint is the existence of a RSDS section with a PDB path. This should trigger the IDA Pro confirmation prompt from Figure 1, provided the message has not been disabled. In the message text we see that the PDB path is *C:\extraspecial.pdb*. This path hints at the executable name: *extraspecial.exe*

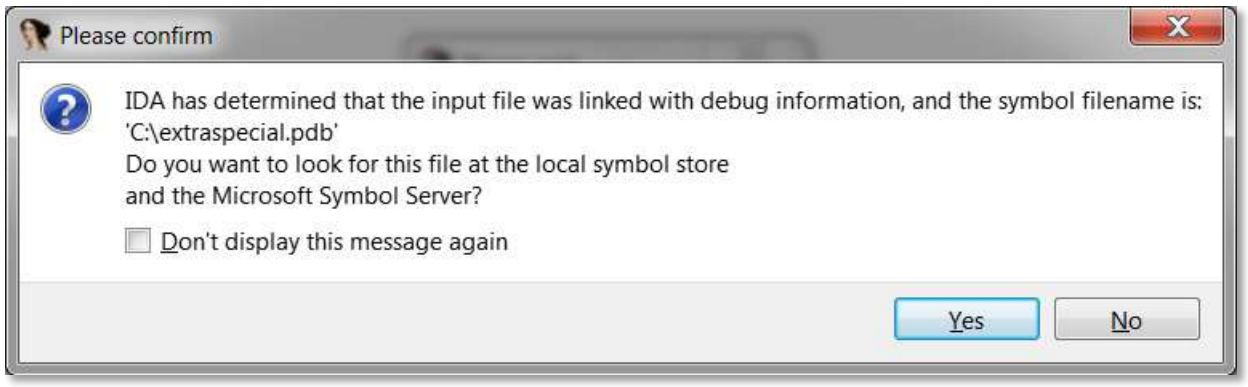

Figure 1: IDA Pro confirmation to look for the debug information for our sample

At the beginning of the main function, the executable path (from *argv[0]*) is searched with *wcsrchr* for the last occurrence of a lowercase 'r' and the following characters (aspecial.exe) are hashe d with a 32-bit hash yielding a four-byte *exePathHash*. This hash function is from the Windows XP registry name hashing and is shown in Figure 2.

```
uint32 GetXpHash(ctstring s) {
  uint sum = 0;
  while (*s) {
    sum = sum * 0x24 + sum + *s++;
  }
  return sum;
}
```

Figure 2: 32-bit string hash

The hash of the exe along with the RSDS GUID and the password length are used in deriving a 16-byte RC4 decryption key for the initially decrypted set of password hashes. This process is illustrated in Figure 3.

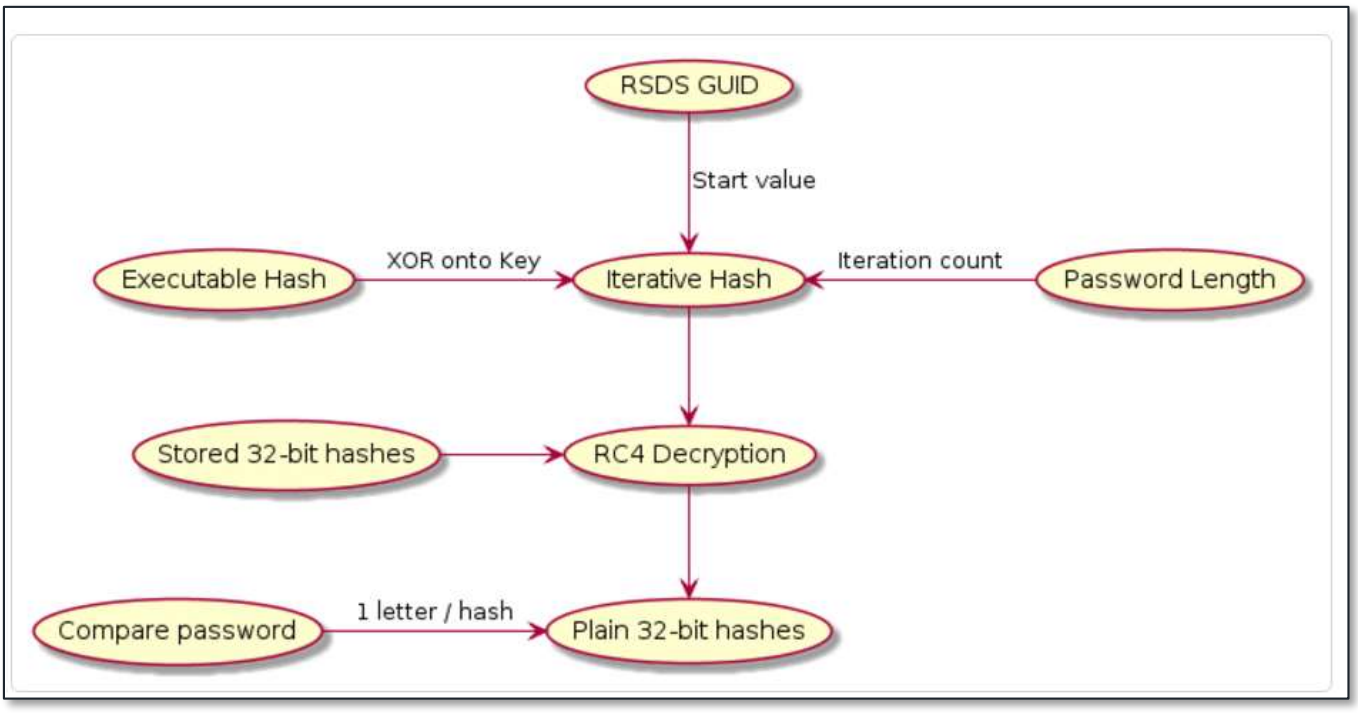

Figure 3: Inputs to the hash decryption.

The details of the RC4key generation are below, but all that is needed is to dump the decrypted array of 32-bit hashes from Figure 5 on page 5.

Things to note are:

1. Iterative hashing is used to hinder brute-forcing using the unmodified executable.

2. The hash function used in the iterative hashing is not MD5, but a modified variant of it. This is to prevent the use of existing hash libraries.
3. Most of the code has been written to use the same number of cycles for a given password length (there is a check in the final loop that I missed), to hinder timing analysis.
4. The use of the password length in the iteration count for the iterative hashing is another complication (see US Patent 8,238,552), since the stored 32-bit hashes will get decrypted incorrectly. Thus, the length of the password must be derived from the array of hashes.
5. The use of the GUID from the RSDS blob from Figure 4 to derive the hash decryption key is done as a hint to draw attention to the PDB path.

```
000142A0: 52 53 44 53 D0 2D 1E B6 23 1B 0E 46 AD 92 A0 98  RSDS.-..#..F....
000142B0: E9 B7 87 79 01 00 00 00 43 3A 5C 65 78 74 72 61  ...y....C:\extra
000142C0: 73 70 65 63 69 61 6C 2E 70 64 62 00 00 00 00 00  special.pdb.....
```

Figure 4: RSDS blob at file offset 0x142A0

Breaking the hashes

For every character of the password, a suffix of `<0x60 + index>FLARE On!` Is being added. The resulting string is then hashed with the algorithm from Figure 2, e.g. yielding 0xEE613E2F for "O`FLARE On!" Thus every hash can be brute-forced in less than 127 tries, yielding a password of `Ohs0pec1alpwd@flare-on.com` for this stage.

Appendix A: Iterative hashing

The MD5 version has different constants in the transformation phase. While this normally unadvisable, it is only used as a one-way hash function in key derivation scheme. The state of the iterative hash is 16 bytes, with the first four being XORed with the *exePathHash*. These are then hashed with the modified MD5 and the process repeats for a total of 0x10000 * passwordLength times. The use of the password length as the iteration count is a subtle but important complication of this challenge.

Appendix B: 32-bit Hashes

The list of hashes for comparison can be found in Figure 5. For every line, the first hash is the encrypted data in the sample and the second one is the properly decrypted one. The list ends with a NULL hash as a marker.

```
uint32 Hashes[0x1B] = {
  0x2227F967,//  0xEE613E2F,// O`FLARE On!
  0xAE88DDA3,//  0xDE79EB45,// haFLARE On!
  0x19C84D84,//  0xAF1B2F3D,// sbFLARE On!
  0xC4D6886B,//  0x8747BBD7,// 0cFLARE On!
  0x709494CD,//  0x739AC49C,// pdFLARE On!
  0x85EBB2F9,//  0xC9A4F5AE,// eeFLARE On!
  0x6F460D9D,//  0x4632C5C1,// cfFLARE On!
  0xB9CF2FF5,//  0xA0029B24,// 1gFLARE On!
  0x7A057016,//  0xD6165059,// ahFLARE On!
  0x1D84EA61,//  0xA6B79451,// liFLARE On!
  0x2D57C396,//  0xE79D23BA,// pjFLARE On!
  0x74CABD17,//  0x8AAE92CE,// wkFLARE On!
  0x93EEC970,//  0x85991A18,// dlFLARE On!
  0xE3AA9FA9,//  0xFEE05899,// @mFLARE On!
  0xC883A61A,//  0x430C7994,// fnFLARE On!
  0x2963E4A8,//  0x1AB9F36F,// loFLARE On!
  0xC62D8758,//  0x70C42481,// apFLARE On!
  0xAC8E881F,//  0x05BD27CF,// rqFLARE On!
  0xF8B26100,//  0xC4FF6E6F,// erFLARE On!
  0x72FC130D,//  0x5A77847C,// -sFLARE On!
  0x0C6E19DD,//  0xDD9277B3,// otFLARE On!
  0x9EC02B6B,//  0x25843CFF,// nuFLARE On!
  0xE1B5B4BB,//  0x5FDCA944,// .vFLARE On!
  0x5E453F35,//  0x8EE42896,// cwFLARE On!
  0x0C46282B,//  0x2AE961C7,// oxFLARE On!
  0x9B64EDBB,//  0xA77731DA,// myFLARE On!
  0x00000000//  0x00000000,//
};
```

Figure 5: List of 32-bit hashes for the password