//

How to get a SHA256 or SHA256D for Blockchains

____

maXbox Starter 54 - MS Cryptographic Service Provider

As you may know a SHA (Secure Hash Algorithm) is one of a number of cryptographic hash functions. A cryptographic hash is like a signature for a text or a binary data file. SHA2 algorithm generates an almost-unique, fixed size 256-bit (32-byte) hash result.

Hash is a one way function – it cannot be decrypted or reversed back. This makes it suitable for password validation, challenge hash authentication, securisation, anti-tamper, digital signatures and of course blockchains.

In the following I want to show 2 solutions, one with the advapi32.dll and a second one with a library from PascalCoin (www.pascalcoin.org) precompiled in maXbox! Small functions to build a micro-service.

Such a hash you can find for example in anti-virus services to recognize a file already uploaded:

https://www.virustotal.com/en/file/3a58a62b4a4959d1bc75c7ad698f3cb47ee85c52c4c3799d78b9bc862defda5a/analy

VirusTotal stores all the analyses it performs, this allows users to search for reports given an MD5, SHA1, SHA256 or URL. Search responses return the latest scan performed on the resource of interest. You see in the url above the SHA256 of the exe:

3a58a62b4a4959d1bc75c7ad698f3cb47ee85c52c4c3799d78b9bc862defda5a

Already scanned files can be identified by their known (e.g. by default) SHA256 hash without uploading complete files. O.K. lets build the script to get those hashes.

The script can be found at:
http://www.softwareschule.ch/examples/sha256.txt
pic: http://www.softwareschule.ch/images/sierpinski4realhash.png

The DLL solution is not the easiest one but it shows explicitly steps behind. Also you do have the flexibility to use larger values like SHA512. Our goal is to calculate SHA256 of maXbox4.exe. First we need some types and structures:

```
 type
    TCryptProv = THandle;
    TAlgID = integer;
    TCryptKey = PChar;
    TCryptHash = THandle; //or PChar;
    TCryptData = PChar;
    TSHA_RES3 = Array[1..32] of Byte;

 var hprov: TCryptProv;
     hhash: TCryptHash;
     hkey: TCryptKey;
     cbHashDataLen: dword; //byte;
     shares3: TSHA_RES3;
     shaStr: string;
```

The type TSHA_RES3 is kind of buffer for the 32-byte result in shaStr. I must admit that I managed to avoid pointers to pass so all the types are referenced and well managed. This seems very redundant but there is a very good default setting for all these parameter which makes sense for expressiveness, clarity and testing. On the other side PascalScript or Python cant handle pointers with one exception: PChar. And that was my helper. Next we define the const block:

```
 Const
    PROV_RSA_FULL = 1;
    PROV_RSA_AES = 24;
    CRYPT_VERIFYCONTEXT = $F0000000;
    CRYPT_NEWKEYSET = $00000008;
     // use with PROV_RSA_AES To get SHA-2 values.
     //http://www.tek-tips.com/faqs.cfm?fid=7423
    CALG_SHA256 = $0000800C;
    CALG_SHA384 = $0000800D;
    CALG_SHA512 = $0000800E;
    HP_HASHVAL = $0002;
    CRYPT32 = 'crypt32.dll';
    MS_ENHANCED_PROV = 'Microsoft Enhanced Cryptographic Provider v1.0';
    HASH256TEST= 'The quick brown fox jumps over the lazy dog';
```

By the way with OpenSSL and the well known libeay32.dll a further solution exists (just a type extract below) but in this article we focus on Win DLL.

```
type
   SHA_CTX2 = Record
     Unknown: Array[0.. 5] of LongWord;
     State: Array[0.. 4] of LongWord;
     Count: UInt64;
     Buffer: Array[0..63] of Byte;
   End;

function SHA256_CTX(nameform: DWord; namebuffer: array of char;
                      var nsize: DWord): boolean;
     external 'SHA256@libeay32.dll stdcall';

function libeay32version: pchar;
     external 'SSLeay_version@libeay32.dll stdcall';

procedure SHA256Init(var Context: SHA_CTX2);
     external 'SHA256_Init@libeay32.dll stdcall';
```

Probably the best way to get started with this sort of thing is to create a small test DLL, create a few functions with known parameters and call it. In our case we need 6 functions to declare:

```
 function CryptAcquireContext(out phProv: TCryptProv; szContainer:
 PChar; szProvider: PChar; dwProvType: DWord;
 dwFlags: DWord): boolean; //stdcall;
     External 'CryptAcquireContextA@advapi32.dll stdcall';

 function CryptCreateHash(phProv: TCryptProv; Algid: TAlgID; hKey:
 TCryptKey; dwFlags: DWord; out phHash: TCrypthash): boolean;
     External 'CryptCreateHash@advapi32.dll stdcall';

 function CryptHashData(phHash: TCryptHash; aRes: PChar; dwDataLen:
 DWord; dwFlags: DWord): boolean; //stdcall;
     External 'CryptHashData@advapi32.dll stdcall';

 function CryptGetHashParam(phHash: TCryptHash; dwParam: Dword;
 out pbdata: TSHA_RES3;
 var dwDataLen: DWord; dwFlags: DWord): Boolean; //stdcall;
     External 'CryptGetHashParam@advapi32.dll stdcall';
```

```
function CryptDestroyHash(phHash: TCryptHash): Boolean; //stdcall;
    External 'CryptDestroyHash@advapi32.dll stdcall';

function CryptReleaseContext(phProv: TCryptProv; dwFlags:DWord): boolean;
    External 'CryptReleaseContext@advapi32.dll stdcall';
```

The quality of a DLL function is the parameter documentation. So much the better you find a well based documentation concerning view the parameter and return types of a function!

https://technet.microsoft.com/en-us/library/cc962093.aspx

The Win module file format only provides a single text string to identify each function. There is no structured way to list the number of parameters, the parameter types, or the return type. However, some languages do something called function "decoration" or "mangling", which is the process of encoding information into the text string.

Our first and important call is CryptAcquireContext():

The CryptAcquireContext function is used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). A CSP is an independent module that performs all cryptographic operations.

At least one CSP is required with each application that uses cryptographic functions. A single application can occasionally use more than one CSP. This returned handle is used in calls to CryptoAPI functions that use the selected CSP, so the first 2 calls are:

```
writeln('context: '+botostr(CryptAcquireContext(hProv, '', '',
                  PROV_RSA_AES, CRYPT_VERIFYCONTEXT)));
```

The following code assumes that the handle of a cryptographic context has been acquired and that a hash object has been created and its handle (hHash) is available. So we dont need any pointers and I can script it in maXbox, Python or Powershell with call by references and a strict PChar with the ByteArray

```
  TSHA_RES3 = Array[1..32 ] of Byte;

 writeln('create: '+
     botostr(CryptCreateHash(hProv,CALG_SHA256,hkey,0,hHash)));
```

The CryptCreateHash() function initiates the hashing of a stream of data.

This handle is used in subsequent calls to CryptHashData and CryptHashSessionKey to hash session keys and other streams of data that we get we a filetoString():

```
  sr:= filetoString(exepath+'maXbox4.exe');
  writeln('cryptdata: '+botostr(CryptHashData(hhash,sr,length(sr),0)));
```

And the last step is to get the hash with CryptGetHashParam:

```
 cbHashDataLen:= 32;
 if (CryptGetHashParam(hHash, HP_HASHVAL, shares3,cbHashDataLen, 0))
 then begin
   for it:= 1 to cbHashDataLen do
     shastr:= shastr +UpperCase(IntToHex((shares3[it]),2));
   writeln('SHA256: '+shastr)
 end;
```

I do always evaluate on each function the boolean return value to make sure. When was the last time you saw the return value for a function checked?

The CryptGetHashParam function retrieves data that governs the operations of a hash object. The actual hash value can be retrieved by using this function. Dont forget to free handles and structure:

```
 println('Destroy hash-hnd: '+botostr(CryptDestroyHash(hhash)));
 println('Crypt_ReleaseContext: '+botostr(CryptReleaseContext(hProv, 0)));
```

A second way to test the resulting hash is

```
 writeln('SHA256: '+(binToHEX_Str(shares3)))
```

I did also test this on a Ubuntu 16 Mate with Wine and IT works too!
pic: 675_virtualbox_ubuntu_sha256_advapi32dll.png
http://www.softwareschule.ch/images/virtualbox_ubuntu_advapi32dll.png

maXbox Output:

```
context: TRUE
create: TRUE
cryptdata: TRUE
SHA256: 3A58A62B4A4959D1BC75C7AD698F3CB47EE85C52C4C3799D78B9BC862DEFDA5A
test length: 32
SHA256: 3A58A62B4A4959D1BC75C7AD698F3CB47EE85C52C4C3799D78B9BC862DEFDA5A
destroy hash-hnd: TRUE
Crypt_ReleaseContext: TRUE
```

The binToHEX_Str function is an effective way to get a HEX result test:

```
Const HexSymbols = '0123456789ABCDEF';

function binToHEX_Str(const bin: array of byte): string;
var i: integer;
begin
  SetLength(Result, 2*Length(bin));
  writeln('test length: '+itoa(length(bin)))
  for i:= 0 to Length(bin)-1 do begin
    Result[1 + 2*i + 0]:= HexSymbols[1+bin[i] shr 4];
    Result[1 + 2*i + 1]:= HexSymbols[1+bin[i] and $0F];
  end;
end;
```

Lets make an overview of the 6 functions used:

```
1. CryptAcquireContext  Get handle to current key container of particular CSP.
2. CryptCreateHash      Creates an empty hash object.
3. CryptHashData        Hashes a block of data, adding it to spec. hash object.
4. CryptGetHashParam    Retrieves a hash object parameter.
5. CryptDestroyHash     Destroys a defined hash object.
6. CryptReleaseContext  Releases handle acquired by the CryptAcquireContext().
```

By the way Indy retrieves SHA1 and with Indy 10:

```
 function SHA1ADirect3(const fileName: string): string;
 var fs: TFileStream;
 begin
   with TIdHashSHA1.Create do begin
     fs:= TFileStream.Create(fileName,fmOpenRead);
     try
       result:= AsHex(HashValue(fs));
     finally
       fs.Free;
       Free
     end;
   end;
 end;
```

Next we step to the double SHA256 called SHA256D and block generation. Its important to realize that block generation is not a long, set problem (like doing a million hashes), but more like a lottery. Each hash basically gives you a random number between 0 and the maximum value of a 256-bit number (which is huge). If your hash is below the target, then you win. If not, you increment the nonce (completely changing hash) and try again to mine. With the SHA256 lib of PascalCoin the function is simpler to use in comparison to the DLL:

Example:

```
   sr:= filetoString(Exepath+'maXbox4.exe')
   writeln(SHA256ToStr(CalcSHA256(sr)))
```

or more simpler with an alias in maXbox:

```
   writeln(GetSHA256(sr))
```

```
function GetSHA256(Msg: AnsiString): string; //overload;
var Stream: TMemoryStream;
begin
  Stream:= TMemoryStream.Create;
  try
    Stream.WriteBuffer(PAnsiChar(Msg)^,Length(Msg));
    Stream.Position:= 0;
    Result:= SHA256ToStr(CalcSHA256(Stream));
  finally
    Stream.Free;
  end;
end;
```

Imagine now the double hash. It is also a crypto hash function, mainly used to ensure integrity of the encrypted message of the block, i.e. if you manipulate the message it will be visible, because the hash will also change. It also guarantees the uniqueness of a message or block of data.

In terms of Bitcoin or PascalCoin, it guarantees the uniqueness of each coin. So you cannot just copy the same set of data over and over again. The function is

```
 Function CalcDoubleSHA256(Msg : AnsiString) : TSHA256HASH;
 Function SHA256ToStr( Hash : TSHA256HASH) : String;

   sr:= filetoString(Exepath+'maXbox4.exe')
   writeln(SHA256ToStr(CalcDoubleSHA256(sr)))

 >>> 7DECBAE2 2C539395 8C3707E9 080281CE 06F45779 BFBBB81F 9954E031 982A505E
```

It appears to be double SHA256. In other words:

```
   SHA256D(x) =SHA256(SHA256(x)).
```

SHA256 (and thus SHA256D) is a cryptographic hash function (it performs a 1-way transformation on an input value) that forms the proof-of-work algorithm used when adding blocks to the blockchain in bitcoin. You are hashing the hexadecimal representation of the first hash. You need to hash the actual hash, the binary data that the hex represents.

Just semantics, but to avoid a common misunderstanding: SHA256 and others does hashing, not encoding. Encoding is something entirely different. For one it implies it can be decoded, whereas hashing is strictly a one-way (and destructive) operation!

There’s no guarantee that every single value in a hash function is reachable, depending on the hash algorithm. For some cryptographic algorithms, it is likely that less than half of the output keyspace is reachable for any given input. However, this may not hold true for every single cryptographic hash algorithm, and it is computationally unfeasible to verify.

There is also no proof that every output of common hash functions is reachable for some input, but it is expected to be true. No method better than brute force is known to check this, and brute force is entirely impractical.

Ref:

http://www.pascalcoin.org/
https://en.bitcoin.it/wiki/Target
https://bitcoinwisdom.com/
https://maxbox4.wordpress.com
http://www.xorbin.com/tools/sha256-hash-calculator
http://www.softwareschule.ch/examples/sha256.txt

https://sourceforge.net/projects/maxbox/files/Examples/13_General/778_advapi32_dll_SHA256.txt/download
https://sourceforge.net/projects/maxbox/files/Examples/13_General/675_bitcoin_doublehash2.txt/download

https://maxbox4.wordpress.com/2017/08/23/five-steps-to-get-sha256-or-other-ciphers/

Doc: SHA256 Lib **Interface**:

```
procedure SIRegister_USha256(CL: TPSPascalCompiler);
begin
 type TSHA256HASH', 'array[0..7] of Cardinal');
 type TSHAChunk', 'array[0..7] of Cardinal');
 //TSHA256HASH = array[0..7] of Cardinal;
 Function CalcDoubleSHA256( Msg : AnsiString) : TSHA256HASH;
 Function CalcSHA256( Msg : AnsiString) : TSHA256HASH;
 Function CalcSHA2561( Stream : TStream) : TSHA256HASH;
 Function SHA256ToStr( Hash : TSHA256HASH) : String;
 Function CanBeModifiedOnLastChunk( MessageTotalLength: Int64; var startBytePos : integer) : Boolean');
 Procedure PascalCoinPrepareLastChunk( const messageToHash : AnsiString; var stateForLastChunk : TSHA256HASH; var bufferForLastChunk : TSHAChunk);
 Function ExecuteLastChunk(const stateForLastChunk: TSHA256HASH; const bufferForLastChunk: TSHAChunk; nPos : Integer; nOnce, Timestamp : Cardinal) : TSHA256HASH;
 Function ExecuteLastChunkAndDoSha256(const stateForLastChunk: TSHA256HASH; const bufferForLastChunk: TSHAChunk; nPos : Integer; nOnce, Timestamp : Cardinal) : TSHA256HASH;
 Procedure PascalCoinExecuteLastChunkAndDoSha256( const stateForLastChunk : TSHA256HASH; const bufferForLastChunk: TSHAChunk; nPos : Integer; nOnce, Timestamp : Cardinal; var ResultSha256 : AnsiString);
 Function Sha256HashToRaw( const hash : TSHA256HASH) : AnsiString;
 Function GetSHA256( Msg : AnsiString) : string;;
 function GetDriveNumber(const Drive: string): Integer;
 function HardDiskSerial(const Drive: string): DWORD;
 function IsDriveReady2(const Drive: string): Boolean;
 function Touchfile(const FileName: string): Boolean;
 function URLFromShortcut(const Shortcut: string): string;
```