Tales from the Crypt: Safer Anonymization with SHA

Andy Smith, Amadeus Software Ltd
So, you want to share detailed datasets (microdata) with 3rd parties, e.g. Researchers, Statisticians...
AND you want to let your audience join tables based on Key Fields...
AND those Key Fields could be a unique identifier to an Individual...
AND you need to preserve Anonymity, so as not to disclose sensitive information...

Then you could...
IF you have lots of time...

AND you are a Wizard at Data Manipulation...

AND you are prepared for a lot of ongoing maintenance...

THEN Create, Maintain and Synchronize anonymous Surrogate Keys across all of your tables...

Don’t want to do that..? I really don’t blame you...
How about Encrypting those Key Fields? Great idea!

IF you are Alan Turing, (yes, really him) why not create your own Cipher?
IF you are not Alan Turing (hint: you are not)

AND you want to something Quick, Easy and Proven

THEN why not use... SHA (Secure Hash Algorithm)?

The SHA256 function converts a string, based on the SHA256 algorithm, to a 256-bit hash value
This resultant HASH Value is unique, and secure
SHA256 is commonly used in password storage

It’s a One Way function, so results cannot be functionally “UnHashed” or “UnEncrypted”...
data _null_;  
  y=sha256('abc');  
  z=sha256('access method');  
  put y=$hex64.;  
  put z=$hex64.;  
run;

The following output is displayed for ASCII systems:

y=BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD
z=F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915

But Wait! There’s a problem...
A would-be Attacker can exploit output from SHA256!

By pre-hashing valid Key Field Values, and storing them in so-called Rainbow Tables
And then using these Rainbow Tables to reverse lookup the Hash Value to the Key Field Value
AND it’s not difficult to do... (you don’t need to be Alan Turing)

<table>
<thead>
<tr>
<th>Hashed_Value</th>
<th>Unhashed_Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BA7816BF8F01CFEA414140DE5DAE2223B00361A396177A9CB410FF61F20015AD</td>
<td>abc</td>
</tr>
<tr>
<td>F2758E91725621F59F2F80D15DE8824560EDC471EBE40A83BA6D1259B1605915</td>
<td>access method</td>
</tr>
</tbody>
</table>
What’s the answer? Add a little Salt...

A Salt is a Secret text value added to the value that’s being Hashed, e.g. 16 Bytes

AND renders a Rainbow Table powerless

AND it’s so simple to implement, you really don’t need to be Alan Turing
data _null_;   
if _n_ = 1 then set secure.salt(keep=salt);  
/* It’s a good idea to secure your salt table in a Metadata Bound Library */  
y=sha256(catt(salt,'abc')); /* concatenate the salt and value to be hashed */  
z=sha256(catt(salt,'access method')); /* other concatenation functions are available */  
put y=$hex64.;  
put z=$hex64.;  
run;

The following output is displayed for ASCII systems:

y=SOMEOTHERHASHEDVALUETHATYOUCANONLYREVERSELOOKUPIFYOUNOWTHESALT*
z=ANDEVENIFYOUKNWTHESALTYOUWOULDHAVETOBUILDARAINBOWTABLEFROMSCRATCH*

* Not the actual hashed values, but you get the idea…
The Storage Requirement and Raw Computational Power to create Rainbow Tables for all valid potential Salt values is STAGGERINGLY MASSIVE, so your data is Very Safe

Still feeling vulnerable? You could always use Salt AND Pepper

```plaintext
hash = sha256(catt(salt, value, pepper));
```

And push it real good...

(I do apologize...)

Safer Anonymization with Secure Hash Algorithm
Andy Smith
Amadeus Software Ltd
So... what have we learned?

SHA256 is a FANTASTIC way to Encrypt your Key Fields
Although Out of the Box it is vulnerable to Attack – EEK!
And with minimal effort, you can protect yourself against Attacks
All you need is a little bit of NaCl... and maybe some Pepper to taste...
SAS® GLOBAL FORUM 2016
IMAGINE. CREATE. INNOVATE.

LAS VEGAS | APRIL 18-21
#SASGF