Table of Contents robust ^XUSHSH

This document's.pdf format was generated and is editable by LibreOffice... improvements welcome

Overview:

The purpose of this distribution is to enhance VISTA's security by providing cryptographically robust hashing for Access and Verify Codes. The end result should be entirely transparent to VISTA users. The New Person file, including Code expiration dates and AOLD, VOLD cross-references is unchanged except for re-hashing of ALL Access/Verify Code information. And we will make no changes to any Kernel routines other than to XUSHSH itself.

The call $$EN^XUSHSH(X) sits at the core of the sign-on process of any VISTA implementation. Whether approaching through CPRS, a web application, or on a character-based terminal, a user is prompted for both Access and Verify Codes. And then these are passed in turn through XUSHSH and the result used (Access Code) to look-up the user in New Person cross reference "A" and then (Verify Code) to check the against the content of the Verify Code field, #11. See detail at XUS, under the hood.

Current sign-on security:

distribution	$$EN^XUSHSH("test")	hash
FOIA	test	"NONE"
OpenVista	test	"NONE"
WorldVistA	115116101116	"LEGACY"
vxVistA	098F6BCD4621D373CADE4E832627B4F6	md5

Weak implementations of this critical function cannot be considered adequate for sensitive healthcare information. Of those listed here, only md5 can be properly called a hash and it is cryptographically weak and deprecated for password hashing. WorldVistA's version uses reversible ASCII encoding not a true hash. Replacements for XUSHSH have been developed that support hashes from SHA1 up to SHA512 This implementation builds on that experience and hopes to provide a state of the art option with installation tools to support an orderly transition to a more secure VISTA.[1]

This package includes

i.) A Python component (xus.py) which resides on the host operating system.

ii.) This script is called from our new ^XUSHSH using a MUMPS pipe command.
(Separate versions for GT.M and for Caché accommodate their different versions of the pipe command. This insures identical, distribution independent hashing outcome.)

iii.) The specifics of the call are configured and modifiable in code stored at ^VA(200,"VWHSH"). This permits very flexible configuration of the system's behavior. If that is desired for some reason. The default is very simple to install. And, if desired, the configuration can be hard-coded into ^XUSHSH itself.

1 This package was built and tested on Caché for Windows (x86-64) 2012.1.2 running on Windows 7 Home Premium, [AMD athlon II X2 235e Processor 2.70 GHz 4 GB RAM] and, for GT.M 5.4-002B, on a dEWDrop versions 2&3 VirtualBox VirtualMachine running on the same hardware.

A call to `XUSHSH(X)` Xecutes the code stored at `^VA(200,"VWHSH")` to assemble the desired pipe call into a command stored in variable, `VWCALL`, which is passed through the pipe for processing by `xus.py` to produce the required hash.

1. `xus.py` (see below and the source code in Appendix A to see its structure.) This Python program provides hooks allowing a MUMPS pipe to smoothly call a library of hashes (including md5, sha1, sha256, sha512, and PBKDF2)[2]. A table of default values within `xus.py` simplifies the interface and can be easily edited if desired.

2. MUMPS routine, `VWHSH0`, supports the installation of the new `XUSHSH`, `VWHSH3` for Caché or `VWHSH8` for a GT.M installation. Then, the exact behavior of the newly installed version of `$$EN^XUSHSH(X)` is configured with a global array at `^VA(200,"VWHSH")`. Initially it can be configured to just call the old version, which we save at `VWHSHLEG` until we are ready to convert to the new hash.

3. Our final step which uses routine other tags of `VWHSH0` to walk through New Person to convert to the new hash. (There is a backup global node created in `^XTMP` that **should be deleted** once a successful conversion has been confirmed.)

4. A final option would be to move the Python call out of `^VA(200,"VWHSH")` and to hard-code it into the `^XUSHSH` routine itself.

Stage One: setup host's Python tools

On either platform we begin by setting up the host to support our Python script, `xus.py`.[3]
An advantage of this is that the xus.py script performs identically on either a Linux host or on Windows. Caché on Linux will use the same setup as GT.M at this stage and the Caché pipe to call it.

i. Python2.x required

Expect it to be included and accessible on any Linux host. We will test for proper function at the command line. For example on the dEWDrop virtual machine:

```
vista@dEWDrop:~$ python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
```

All examples: Prompt: `Blue`
You input: **`green`** {You can usually execute code by careful copy / paste of the green stuff.}
System returns: `brown`

2 Jan 17 2014 version adds **scrypt** and **SHA3** to the list.

3 Recommended options:
- VPE is very helpful for configuring global nodes (especially `^VA(200,"VWHSH",*)`), for checking results, etc.
- I also installed GT.M's Posix plugin. On dEWDrop this allows high-resolution timing to help choose the iteration parameter in PBKDF2; see here.

```
...
```

Windows hosts will most likely require an install. It's available at http://www.python.org/download/. (Python3.2.3 will no work.) The full path to python.exe is needed unless it is added to PATH:

```
c:\>C:\Python27\python
Python 2.7.3 (default, Apr 10 2012, 23:24:47) [MSC v.1500 64 bit (AMD64)] on
win32
Type "help", "copyright", "credits" or "license" for more information.
>>>exit()
```

ii. The Python script xus.py

This provides a flexible interface that will be used by `XUSHSH`. It's primary function is to calculate the pbkdf2 hash[4]. It provides access to a number of other useful hashes, and a set of defaults that can be altered by directly editing the script.

(See Appendix A, `xushsh.py, defaults`.)

On dEWDrop I have placed `xus.py` in `~/bin/` and this simple test confirms proper function:

```
vista@dEWDrop:~$ python /home/vista/bin/xus.py
f4ca507c07d0bd31bc779a08756826a6fd9dd97d43ac25e4
```

This unembellished call to `xus.py` shows how we can use the defaults for all supported parameters, including a dummy input, "password". So we are seeing the pbkdf2 hash of the string, "password" confirming that everything works thus far.

And in Windows the simplest configuration puts both `xus.py` in `C:\Python27\`, along with `python.exe` itself. And the call looks like this:

```
C:\>C:\Python27\python c:\Python27\xus.py
f4ca507c07d0bd31bc779a08756826a6fd9dd97d43ac25e4
```

Now we can move directly to **Stage Two**... or play around with xus.py for a bit longer.
For example, here is the sha1 hash of the string "test":

```
c:\>c:\Python27\python c:\Python27\xus.py -h sha1 -d "test"
b56498f027060a5f4707bad6621d354d01672b86
```

The "null" hash of default input, which is "password":

```
vista@dEWDrop:~$ python /home/vista/bin/xus.py --hash=null
password
```

---

4 PBKDF2 hash implemented in Python, from: https://github.com/mitsuhiko/python-pbkdf2/

```
$ git clone git://github.com/mitsuhiko/python-pbkdf2.git
```

To simplify installation, `pbkdf2_hex` from this package has been incorporated into `xus.py`.

And pbkdf2 hash of "test" with 30,000 iterations, keylength 20:

```
vista@dEWDrop:~$ python /home/vista/bin/xus.py -p "test" -c 20000 -k 20
--iterations=30000 -h pbkdf2
pbkdf2$$30000$20$sha512$$08b61e482481cb425dfe62c9bddce2a4a6581f0d
```

A call to produce a random number, useful for generating a salt:

```
vista@dEWDrop:~$ python /home/vista/bin/xus.py -h random
541e44897d28a94b96a4a2469be1da306653a9009929fc81
```

When called `xus.py` will accept its six parameters in any order, using the defaults for any not entered and ignoring irrelevant input (e.g. iterations for sha1). Both a short and long form may be used for each. See text box in Appendix A. (If any parameter is entered more than once the last one is used. This is useful in building a command string in M to send to the Python script.)

Stage Two: prepare MUMPS

We are now ready to hook these tools into the MUMPS code of the Kernel. Our distribution provides a configuration utility, `VWHSH0` will be used to prepare the ground and then to install the proper version tailored to either GT.M or Caché. A configuration array stored at `^VA(200,"VWHSH")` is built, ready to redirect control to pbkdf2. Final conversion of the NEW PERSON file is done, in Stage 3 by calls into `VWHSH0`.

routine	*entry points*
VWHSH0	TEST, SAVEOLD, BUILD(), MOVEIN
	TO(FROMHASH,TOHASH), REVERT, KILL
VWHSH3	Caché version XUSHSH
VWHSH8	GT.M version XUSHSH

i. TEST

```
MU-beta>do TEST^VWHSH0
OK, current HASH is identified as LEGACY.
```

A result other than NONE or LEGACY would require manual rebuilding of all passwords.

ii. SAVEOLD

simply copies current, legacy, routine `XUSHSH` to `VWHSHLEG`.

```
EHR>do SAVEOLD^VWHSH0
```

Note: This entry point ***does not work on dEWDrop*** because the call uses `ZTMGRSET` which is not able to accommodate dEWDrop's GT.M routine directory structure. Therefore this step requires manually copying r/`XUSHSH.m` to `VWXUSLEG`.

iii. BUILD

```
MU-beta>do BUILD^VWHSH0()

This constructs the following array:

MU-beta>zwr ^VA(200,"VWHSH",:)
^VA(200,"VWHSH")="SET VWCALL=""python /home/vista/bin/xus.py --input=""""""_X
          _""""""""" XECUTE ^VA(200,""VWHSH"",""LEGACY"")"
^VA(200,"VWHSH",0)="Set X=$$EN^VWHSHLEG(X) Kill VWCALL"
^VA(200,"VWHSH","LEGACY")="SET VWCALL=VWCALL_"" --input=""""""_$$EN^VWHSHLEG(X)_
          """""" -h null"",vwcall=VWCALL"
^VA(200,"VWHSH","NONE")="SET VWCALL=VWCALL_"" --hash=null"""
^VA(200,"VWHSH","PBKDF2")="Set VWCALL=VWCALL_"" --hash=pbkdf2"" XECUTE ^VA(200,"
          "VWHSH"",""SALT"")"
^VA(200,"VWHSH","SALT")="SET VWCALL=VWCALL_"" --salt=""""5ce3e82f22c2c78a3e6694d
          c87ce78f091b66440883c6daf""""""""  1) ^VA(200,"VWHSH") = SET
  6) ^VA(200,"VWHSH","SALT") = SET VWCALL=VWCALL_"
            --salt=""5ce3e82f22c2c78a3e6694dc87ce78f091b66440883c6daf"""
```

If **`"python /home/vista/bin/xus.py..."`** does not match the location established in Section One, the node must be edited to reflect the correct position. Use VPE for that.

Now we can test for proper function at this stage by calling either $$EN^VWHSH3, or $$EN^VWHSH8, depending on our M version. So for GT.M:

```
MU-beta>w $$EN^VWHSH8("test")
115116101116
MU-beta>w $$EN^XUSHSH("test")
115116101116
```

And in Caché:

```
EHR>w $$EN^VWHSH3("test")
115116101116
```

Controlled by our newly built ^VA(200,"VWHSH"), the planned replacement for XUSHSH returns the same result as the old version still in place. VWHSH8 calls the old version, the newly copied to VWHSHLEG and then passes the result through xus.py --**hash=null** -password=X, a null operation but confirmation that the Python call is being found by our replacement XUSHSH. (Note that, if one wishes to leave hash at the LEGACY setting, the shortcut of setting X and then killing VWCALL eliminates the call to Python.)

iii. MOVEIN

We can now safely overwrite the old XUSHSH with the new.

```
MU-beta>Do MOVEIN^VWHSH0
```

At this point our new XUSHSH is in place and under control of ^VA(200,"VWHSH"). This is a rather convoluted way marching in place. :-) But it allows the new XUSHSH to mimic the version it is to replace. And it requires only a tweak of ^VA(200,"VWHSH") to finish the change.

(**My recommendation** would be to distribute VISTA with password set to NONE and with this new XUSHSH in place and ready for a single-step conversion to PBKDF2, scrypt or SHA3 as desired.)

Stage Three: REHASH Access / Verify Codes

We continue to call ^VWHSH0. It converts the two fields ACCESS CODE and VERIFY CODE. And it must kill and then rebuild, hashed, the cross references: "A", "AOLD", and "VOLD".

i. TO^VWHSH0(TOHASH)

At this entry point FROMHASH can here be only "NONE" or "LEGACY" while TOHASH should be "NONE" or "PBKDF2". "LEGACY" is run through $$UN(X) to unwind it's ASCII wrapping before the new hash is applied.

ii. REVERT^VWHSH0

The installation is, of course, intended to be *irreversible* since that is the whole point of a strong hash. During the process, however, we can hedge a bit since the conversion begins by backing up the entire ^VA(200) global array to ^XTMP("VWH VA(200) <datetime>,200). This allows the cross references, "A", "AOLD", and "VOLD" to be killed and then rebuilt from ^XTMP. With that task completed the ^XTMP node ought to be kept only long enough to confirm a successful install. The entry tag REVERT^VWHSH0 permits reversing the install if that is necessary:

```
MU-beta>do REVERT^vwhsh0
VWH VA(200) 3120911.151441
VWH VA(200) 3120911.153447
Returning to Latest entry:
^XTMP(VWH VA(200) 3120911.1534470) =
          3120913^3120911.153447^LEGACY

Do you wish to proceed now? //Yes/No Yes
```

iii. KILL^VWHSH0

The final step is to not leave free-text codes sitting in the ^XTMP backup global;

```
MU-beta>Do KILL^VWHSH0
VWH VA(200) 3120911.151441

3120913^3120911.151441^LEGACY
Are we certain we want to kill this backup? //Yes/No
3120911.153447

3120913^3120911.153447^LEGACY
Are we certain we want to kill this backup? //Yes/No Yes
```

Zero-Stage:

WHY? background, culture, et cetera

The VISTA kernel routine XUSHSH should support the kernel's the log-on processes while maintaining the user's credentials safe from unauthorized use/abuse. When a user seeks access to VISTA, he/she enters Access and Verify Codes; and then each is passed through $$EN^XUSHSH(X) and compared in turn to values stored in the New Person file.

Why improve XUSHSH? Because our open source versions of XUSHSH are just shells and provide no real protection of user's credentials. They cannot properly protect a health-care database.

Let's start with a definition of hash:

> A **cryptographic hash** function... is an algorithm that takes an arbitrary block of data and returns a fixed-size bit string, the hash value...
>
> The ideal cryptographic hash function has four main or significant properties:
> - it is easy to compute the hash value for any given message
> - it is infeasible to generate a message that has a given hash
> - it is infeasible to modify a message without changing the hash
> - it is infeasible to find two different messages with the same hash
>
> ….
>
> **Password verification**
> … Passwords are usually not stored in clear-text, but instead in digest form, to improve security. To authenticate a user, the password presented by the user is hashed and compared with the stored hash.[5]

The system does not need to *store* the password in order to *authenticate* it.

Note the difference between cryptographically hashed input and *encrypted* input. An encrypted password could be protected with an algorithm that locks the password up with a *key*, which could be used to unlock and return the free-text password. The logic behind using a hash is that the hash is not reversible. There is no key to be maintained, compromised or lost. And the password itself is never stored in the system (and, ideally, not written to disc).

The Problem:

The VA does not include its internal version of `XUSHSH` in FOIA releases and protects itself partly by obscurity. In the open source arena Medsphere's OpenVista uses what I will term hash NULL in which `XUSHSH(X)` simply returns `X` unaltered. WorldVistA's version uses a simple and easily reversible ASCII encoding algorithm which I will term LEGACY. Of course, neither NULL nor LEGACY is a cryptographic hash.[6] And neither meets standards for a system entrusted with patient care data.[7]

> This package *will* convert LEGACY to NULL if that is desired for development or educational implementations. In my opinion, it is preferable to have passwords stored in the open than with the reversible pseudo-hash that LEGACY provides.

But the goal of this project is to move existing VISTA implementations from either LEGACY or NULL to a cryptographically robust hash, which in my opinion should be PBKDF2 or scrypt. xus.py can support sha1, sha2, etc. for other uses. For password protection, again IMHO, these weaker alternatives are now obsolete.

---

5 http://en.wikipedia.org/wiki/Cryptographic_hash_function
6 http://en.wikipedia.org/wiki/Kerckhoffs%27_principle#Security_through_obscurity
http://slashdot.org/features/980720/0819202.shtml
7 https://www.cchit.org/documents/18/158304/CCHIT+Certified+2011+Security+Test+Script+ALL+DOMAINS.pdf

Members of WorldVistA have been working for the past several years to implement a standards-based XUSHSH routine. Early attempts to give VISTA cryptographically robust hashing worked by writing a file to the host, then calling openssl or sha1sum to perform the hash followed by reading the result back into MUMPS. cf. Appendix B

```
X ▯  write X to hostfile1
call host-routine(hostfile1) ▯  hostfile2
read hostfile2 ▯  X
remove hostfile1,hostfile2
```

Both GT.M and Caché provide Pipe functions that have removed the need to manage writing, reading, and *deleting* host system files, see Appendix A XUSHSH.

One persistent difficulty: Calls from GT.M and Caché to host system libraries were not producing the same hash for a given input. The problem turned out to be due to end of line differences in the host systems, not a GT.M vs Caché difference. A bit of sed magic[8] gave us identical output under Windows and Linux. (I define magic as good stuff that I can use without necessarily bothering to understand... usually something supplied by a generous colleague.)

In June 2009 Chris Uyehara programmed the algorithm for sha1 in pure MUMPS and achieved correct results in both GT.M and Caché. I include all 232 lines in Appendix B out of respect for the effort involved. Meanwhile however, the standards have been shifting. Our chosen sha1 was already showing its age with a feasible attack identified. NTSI[9] had already recommended "rapid transition" to the stronger sha2 family of hash functions *for digital signature applications*.

I initially approached the problem using GnuPG in preference to openssl.[10] Eventually both openssl and GnuPG were supported with consistent results for md5 thru sha512 on both GT.M and Caché. However, new standards have been evolving because of emerging threats to even sha512. For that the reader should Google "GPU hash cracking" for lots of interesting reading.[11] Turns out there are very interesting disadvantages to using efficient hashing algorithms. So new, intentionally slower algorithms have emerged.[12] Currently available are bcrypt, pbkdf2 and scrypt. The purpose of `xus.py` is to provide a readily accessible interface between MUMPS and the host system's cryptographic resources. It imports `hashlib` which supports md5, sha1, sha2, etc. and imports `pbkdf2_hex` from `pbkdf2.py`[13] to support PBKDF2. With this latest release scrypt has been imported from py-scrypt.[14] And sha3 has likewise just been added.[15]

---

8 sed magic: `s SED="sed -e 's/$/\r/'|"` made my Linux hash look like the Windows one.
9 National Institute of Standards and Technology: http://csrc.nist.gov/groups/ST/hash/statement.html April 25, 2006
10 I long ago used pgp to encrypt MailMan messages written to MSDOS and read back into MSM.
11 http://mytechencounters.wordpress.com/2011/04/03/gpu-password-cracking-crack-a-windows-password-using-a-graphic-card/
12 http://blog.josefsson.org/2011/01/07/on-password-hashing-and-rfc-6070/#more-228
13 The work of Armin Ronacher https://github.com/mitsuhiko/python-pbkdf2/
14 https://pypi.python.org/packages/source/s/scrypt/scrypt-0.6.1.tar.gz
15 https://pypi.python.org/pypi/pysha3/

Consider salting and iteration:

Salting means to include something other than the password in the input to the hash function, typically some random string. Without salting, a pre-computed rainbow table[16] can support a dictionary attack against a hashed password database (one rainbow table per hashing algorithm). By using a salt, a rainbow table for one system is not applicable to another system. Since generating a rainbow table is time consuming, this adds another barrier to an attack.

In this context “iteration” or “cycling” refers to calculating the hash function not once, but thousands of times. The extra time is hopefully not noticeable for the user, but for an attacker who is testing an entire dictionary of common passwords against your hash, making the cost thousand times higher quickly makes the attack astronomically more costly. Adjustable iteration allows tuning the application to maximize security while maintaining user acceptability... (A pause that would be entirely unacceptable at a menu prompt is barely noticeable during sign on... and can even be seen as a sign that strong security is on the job.)

PBKDF2 vs scrypt?

PBKDF2 combines **hashing**, **iteration**, **salt** and **function** into one construct. The function parameter uses hashlib.sha1 by default, adjustable to the other hashes available from hashlib.

PBKDF2 is theoretically not as strong a solution as the newer scrypt algorithm which adds an adjustable memory cost to iterations. However, PBKDF2 has a longer history and is thus theoretically less vulnerable to some yet undiscovered attack than the new kid on the block.

16 http://en.wikipedia.org/wiki/Rainbow_table

Demonstration of effect of iterations on time it takes to compute log-on hash (running on dEWDrop virtual machine using `$$ZHOROLOG^%POSIX`):

iterations	seconds	hash of "test"
00001	0.037	579bb89e389d45611e735d85c926b20ea0276d2fee5c5b95
00010	0.027	ffb03d6ff7cfadbb99ee72885522e66209956bf32e37c458
00100	0.048	d7a0e3b279505ebf37fb3783060d8120cc86616755be7b06
01000	0.087	ce979774611b26b29d2760eb522a47254a51d392729ad657
10000	0.55	9e27047241e3f9bc8d03bc1f590ff7792d5b32713faeca9d
20000	1.08	a8661f93452892f8e3d6249173fb59911e45b8b7e2adcc37
30000	1.62	8e8086478433afe834b6a9b3b723e00f77fe0b3f7d32b3af
40000	2.18	a8616f276e2f7064c2d39a39372710a727cef4007e671384
50000	2.79	48b104d53e33fea3e21be6b216bc44613cad0f1af481c97c
60000	3.29	2984ee4b89341d34d7fde9dd17f9f5a7daef67dcb87859b3
70000	3.96	cffccdd28e7e6a146b7d9c3d09360ddb45754eaf62873a32
80000	4.55	5e6c28eb7ff0b6f57f6dfa94af67e0079a95bbc8f47a7b69
90000	5.12	fb423815326261679a6f92a2ee49bbc16746eed3aa22db98

100000 3.95 ▯ `pbkdf2.py` uses recursion and segfaults after about 93,540 iterations[17]

XUS interface, under the hood

Routines in the `XUS*` name space accept users' `ACCESS` and `VERIFY` codes, and test them against two relevant fields in the NEW PERSON file. As far as I can tell, ALL user log-on, terminal, web-based, and CPRS, flows through `^XUS` (~line 115 in VOE).

```
S X=$P(X1,";") Q:X="^" -1 S:XUF %1="Access: "_X
Q:X'?1.20ANP 0
S X=$$EN^XUSHSH(X) I '$D(^VA(200,"A",X)) D LBAV Q 0
S %1="",IEN=$O(^VA(200,"A",X,0)),XUF(.3)=IEN D USER(IEN)
S X=$P(X1,";",2) S:XUF %1="Verify: "_X S X=$$EN^XUSHSH(X)
I $P(XUSER(1),"^",2)'=X D LBAV Q 0
```

> Or, in other words, the sign-on process passes in a variable, `X1`, containing something like, **`"WORLDVISTA6;$#HAPPY7"`**. The string **`"WORLDVISTA6"`** is passed through `XUSHSH` and the result used to look for an entry in the `"A"` index of the New Person file. If found, that returns the user's `IEN` which is used to look up several nodes in the file. In turn **`"$#HAPPY7"`** is run through `$$EN^XUSHSH` and the result is compared to the Verify Code field stored in field #11 and, for the moment, in `XUSER(1)`.

FINALLY see Appendix C for the source of our new `XUSHSH`

17 **mitsuhiko** commented: "That's because it's recursively chaining iterators and cpython has a crappy detection for when it runs out of stack space..."

To describe the configuration and function of this routine:

XUSHSH finds the Python script by fetching a new global node, ^VA(200,"VWHSH"), which holds the host system's call to xus.py. See examples in the XUSHSH code below. This mechanism removes all the details from XUSHSH but makes the options available by editing ^VA(200,"VWHSH").

So we have a layer cake:

Access Code/Verify Code entered into XUS →

^XUS calls $$EN^XUSHSH(X) →
Xecute ^VA(200,"VWHSH") →
configures XUSHSH →
build call to xus.py →
Pipe to Python xus.py →
returns hash →

^XUS compares to hash stored in New Person file

Choice of hash:

See discussion of details of pbkdf2 vs bcrypt and scrypt.[18] Scrypt is arguably the strongest.

One weakness of PBKDF2 is that while its *c* parameter can be adjusted to make it take an arbitrarily large amount of computing time, it can be implemented with a small circuit and very little RAM, which makes brute-force attacks using ASICs or GPUs relatively cheap[15]. The bcrypt key derivation function requires a larger (but still fixed) amount of RAM and is slightly stronger against such attacks, while the more modern scrypt[15] key derivation function can use arbitrarily large amounts of memory and is much stronger. [19]

But scrypt is also the least vetted of these newer options. So PBKDF2 remains a viable choice. SHA3 has just been added, but it is not designed to be costly of either CPU or memory.

Details:

a.) Under Linux, where best to locate xus.py? I have it in ~/bin/ for now. Perhaps they belong somewhere in /opt/ ? Need guidance and a rationale for general GT.M installations as well as for dEWDrop specifically.

Likewise is there a more standard location for these programs on Windows ?

b.) Where best to put "VWHSH"? Now at ^VA(200,"VWHSH"). Would ^%ZOSF(<somewhere>) be better. See hard-coding option Appendix A which may be useful step to take after final configuration is set.

18 http://security.stackexchange.com/questions/4781/do-any-security-experts-recommend-bcrypt-for-password-storage
19 http://www.tarsnap.com/scrypt/scrypt.pdf

c.) Salt. Need direction on what to use here. I have left the default blank for this field.

ACCESS CODE is used for user look-up during sign-on [code: **I '$D(^VA(200,"A",X))**] That precludes using a user-specific salt there. So a system wide (or at least ACCESS CODE-wide) salt is the only option.[20] I am not sure what would work best here.

VERIFY CODE could use a unique salt for each New Person with improved resistance to rainbow table type attack. In fact the internal, "pre-hashed" contents of the ACCESS CODE field which is held at $Piece(^VA(200,IEN,0),"^",3) appears to be a ready-made candidate.

But to use it ^XUSHSH itself must detect that it has been called to hash a VERIFY CODE. Serendipity provides an opportunity here. The following works well during the sign-on process. But I have put it aside because ***setting*** a VERIFY CODE occurs in several places and is not so easily identifiable.

Consider ^XUS, again around line 115:

```
  S X=$$EN^XUSHSH(X) I '$D(^VA(200,"A",X)) D LBAV Q 0
  S %1="",IEN=$O(^VA(200,"A",X,0)),XUF(.3)=IEN D USER(IEN)
  S X=$P(X1,";",2) S:XUF %1="Verify: "_X S X=$$EN^XUSHSH(X)
 ....
USER(IX) ;Build XUSER
  S XUSER(0)=$G(^VA(200,+IX,0)),XUSER(1)=$G(^(.1)),XUSER(1.1)=$G(^(1.1))
  Q
```

Since the call to USER(IEN) builds the XUSER array between the two calls to XUSHSH, we could use the presence of the variable XUSER(0) to trigger our user-specific salt. And the hashed ACCESS CODE is already in hand! So I would like to substitute the ACCESS CODE hash for the default SALT:

```
Set:$Data(XUSER(0)) SALT=$Piece(XUSER(0),U,3)
```

Or we could *append* to the default SALT, known as "**salt and pepper**":

```
Set:$Data(XUSER(0)) SALT=SALT_$Piece(XUSER(0),U,3)
```

Since to accomplish user sign-on, all roads pass through ^XUS, this little hack works just fine with CPRS. But the problem comes with any other code that stores a new or changed VERIFY CODE.
So I have shelved this little enhancement for the time being. It can be turned on at any time by changing "PBKDF2" to "PBKPLUS" in ^VA(200,"VWHSH"). It functions transparently during user sign-on, obtaining the hashing parameters from the New Person Verify Code field, #11.

---

20 (As already described, we have extended the $$EN^XUSHSH(X) call to include optional variables, including optional salt, $$EN^XUSHSH(X,salt,cycles,hashlength). However, to avoid venturing into multiple coding changes in the ^XUS* name-space we must use only (X) during user sign-on.)

Credits / Thanks

Mahalo to Chris Uyehara for MUMPS SHA1, and original UPDATE subroutine. (More careful reading of the early work would have saved me some thrashing about.)

Thanks to KS Bhaskar for GT.M pipe & for pointing out Caché's pipe as well. And for his deep well of common sense about security... and exotic food.

Lars Nooden, for timely advice on sed.

Jim Self, thru Kevin Toppenberg, for unhash(LEGACY).

Nancy Anthracite, for the many things she has done including long hours hacking away at XUSHSH with me.

Appendix A VWHSH code

These listings are included to help understanding of the software and are not necessarily complete. See the actual routines for complete details, including the licensing, and probably a few late tweeks.

xushsh.py

Obsolete, replaced by xus.py

see below:

Input parameters:

--hash=<option> or **-h <option>**

options: **pbkdf2**, **pbkdf2u, md5,**

sha1, **sha256**, **sha512, null**

--data=<string> or **-d <string>**

--salt=<string> or **-s <string>**

--iterations=<integer> or **-i <integer>**

--keylen=<integer> or **-k <integer>**

--function=<sha1,...sha512> or **-f <sha1,...>**

```
#!/usr/bin/python

import hmac
import hashlib
from struct import Struct
from operator import xor
from itertools import izip, starmap

import getopt
import sys
import binascii
import os

_pack_int = Struct('>I').pack

"""
    pbkdf2
    Module from https://github.com/mitsuhiko/python-pbkdf2
                implements pbkdf2 for Python.
    :copyright: (c) Copyright 2011 by Armin Ronacher.
    :license: BSD
     Cut and pasted here from his pbkdf2.py to simplify distribution.
"""

def pbkdf2_hex(data, salt, iterations=1000, keylen=24, hashfunc=None):
    return pbkdf2_bin(data, salt, iterations, keylen, hashfunc).encode('hex')

def pbkdf2_bin(data, salt, iterations=1000, keylen=24, hashfunc=None):
    """Returns a binary digest for the PBKDF2 hash algorithm of `data`
    with the given `salt`.  It iterates `iterations` time and produces a
    key of `keylen` bytes.  By default SHA-1 is used as hash function,
    a different hashlib `hashfunc` can be provided.
    """
    hashfunc = hashfunc or hashlib.sha1
    mac = hmac.new(data, None, hashfunc)
    def _pseudorandom(x, mac=mac):
        h = mac.copy()
        h.update(x)
        return map(ord, h.digest())
    buf = []
    for block in xrange(1, -(-keylen // mac.digest_size) + 1):
```

```
        rv = u = _pseudorandom(salt + _pack_int(block))
        for i in xrange(iterations - 1):
            u = _pseudorandom(''.join(map(chr, u)))
            rv = starmap(xor, izip(rv, u))
        buf.extend(rv)
    return ''.join(map(chr, buf))[:keylen]

    raise SystemExit(bool(failed))
    """GpaZ: knows not what this line does.
    """

"""defaults:
"""

hsh="pbkdf2"
input="password"
salt=""
#for example: salt="bfe102f3b4877733e8dfe2877a860606f69f900d865b3df3"
cycles=10000
keylen=24
func="sha512"

u="$"
flag=0

options, remainder = getopt.gnu_getopt(sys.argv[1:], 'h:d:s:i:k:f:',
['hash=','data=','salt=','iterations=','keylen=','function=',] )

for opt, arg in options:
    if opt in ('-h', '--hash'):
        hsh = str(arg)
    if opt in ('-d', '--data'):
        input = str(arg)
    if opt in ('-s', '--salt'):
        salt = str(arg)
    if opt in ('-i', '--iterations'):
        cycles = int(arg)
    if opt in ('-k', '--keylen'):
        keylen = int(arg)
    if opt in ('-f', '--function'):
        func = str(arg)

# options for internal hash used by pbkdf2:

if (func=="sha1"):
    hashfunc=hashlib.sha1
if (func=="sha256"):
    hashfunc=hashlib.sha256
if (func=="sha512"):
    hashfunc=hashlib.sha512

# "naked" pbkdf2 hash without detail:

if (hsh=="pbkdf2"):
    print pbkdf2_hex(input, salt, cycles, keylen, hashfunc)
```

```
    exit()

# output returned in "$" delimited string:
#    pbkdf2$<salt>$<cycles>$<keylength>$<function>$$<hash>

if (hsh=="pbkdf2u"):
    print "pbkdf2"+u+salt+u+str(cycles)+u+str(keylen)+u+func+u+u+pbkdf2_hex(input,
salt, cycles, keylen, hashfunc)
    exit()

""" older hashes: md5, sha1, sha256, sha512, null
    From here down may all be redundant crap and could be done away with.
"""

if (hsh=="md5"):
     hash = hashlib.md5
     flag=1
if (hsh=="sha1"):
     hash = hashlib.sha1
     flag=1
if (hsh=="sha256"):
     hash = hashlib.sha256
     flag=1
if (hsh=="sha512"):
     hash = hashlib.sha512
     flag=1

# $<hsh>$<salt>$<result>

if (flag==1):
     print hash(input + salt).hexdigest()
     exit()

if (hsh=="crc32"):
     print binascii.crc32(input) & 0xffffffff
     exit()

if (hsh=="random"):
    hash = os.urandom(keylen).encode('hex')
    print hash
    exit()

if (hsh=="null"):
     print input
     exit()

print "error: << " + hsh + " >> is not supported."
exit()

"""
    :copyright: (c) Copyright 2012 by JohnLeo Zimmer.
    :license: BSD, see LICENSE for more details.

"""
```

xus.py

Input parameters:

--hash=<option> or **-h <option>**

options: **pbkdf2**, **pbkdf2u**, **scrypt**, **sha3**, **md5**, **sha1**, **sha256**, **sha512**, **null**

--data=<string> or **-d <string>**

--salt=<string> or **-s <string>**

--iterations=<integer> or **-i <integer>**

--keylen=<integer> or **-k <integer>**

--function=<sha1,...sha512> or **-f <sha1,...>**

```
#!/usr/bin/python

import os
import getopt
import sys
import binascii
import imp
import string
from ctypes import (cdll, POINTER,
      pointer, c_char_p, c_size_t,
      c_double, c_int, c_uint64,
      c_uint32,
create_string_buffer)
import hmac
import hashlib
from operator import xor
from itertools import izip, starmap
import getopt
import subprocess

from struct import Struct
_pack_int = Struct('>I').pack

def pbkdf2_hex(data, salt, iterations=1000, keylen=24, hashfunc=None):
    return pbkdf2_bin(data, salt, iterations, keylen, hashfunc).encode('hex')

def pbkdf2_bin(data, salt, iterations=1000, keylen=24, hashfunc=None):
    """Returns a binary digest for the PBKDF2 hash algorithm of `data`
    with the given `salt`.  It iterates `iterations` time and produces a
    key of `keylen` bytes.  By default SHA-1 is used as hash function,
    a different hashlib `hashfunc` can be provided.
    """
    hashfunc = hashfunc or hashlib.sha1
    mac = hmac.new(data, None, hashfunc)
    def _pseudorandom(x, mac=mac):
        h = mac.copy()
        h.update(x)
        return map(ord, h.digest())
    buf = []
    for block in xrange(1, -(-keylen // mac.digest_size) + 1):
        rv = u = _pseudorandom(salt + _pack_int(block))
        for i in xrange(iterations - 1):
            u = _pseudorandom(''.join(map(chr, u)))
            rv = starmap(xor, izip(rv, u))
        buf.extend(rv)
    return ''.join(map(chr, buf))[:keylen]
    raise SystemExit(bool(failed))

_scrypt = cdll.LoadLibrary(imp.find_module('_scrypt')[1])
_crypto_scrypt = _scrypt.exp_crypto_scrypt
```

```
_crypto_scrypt.argtypes = [c_char_p, # const uint8_t *passwd
                           c_size_t, # size_t passwdlen
                           c_char_p, # const uint8_t *salt
                           c_size_t, # size_t saltlen
                           c_uint64, # uint64_t N
                           c_uint32, # uint32_t r
                           c_uint32, # uint32_t p
                           c_char_p, # uint8_t *buf
                           c_size_t, # size_t buflen
                           ]
_crypto_scrypt.restype = c_int

def scrypt(data, salt, N=1 << 14, r=8, p=1, buflen=64):
    return _crypto_scrypt(data, len(data), salt, len(salt), N, r, p, outbuf, buflen)

""" defaults:
"""

hsh="sha256"
input="password"
salt="Salty"  #for example: salt="bfe102f3b4877733e8dfe2877a860606f69f900d865b3df3"
salt = "1340f327dc01a6cb7bc8d5446149e614ba62c79d20a6b8cd"
func="sha1"
N=1
r=1
p=1
keylen=24
u="$"
flag=0

options, remainder = getopt.gnu_getopt(sys.argv[1:], 'h:d:s:i:n:k:f:m:z:',
['hash=','data=','salt=','iterations=','keylen=','function=','mods=','Z='] )

for opt, arg in options:
    if opt in ('-h', '--hash'):
        hsh = str(arg)
    if opt in ('-d', '--data'):
        input = str(arg)
    if opt in ('-s', '--salt'):
        salt = str(arg)
    if opt in ('-i', '-n', '--iterations'):
        N = int(arg)
    if opt in ('-s', '--salt'):
        salt = str(arg)
    if opt in ('-k', '--keylen'):
        keylen = int(arg)
    if opt in ('-f', '--function'):
        func = str(arg)
    if opt in ('-m', '--mods'):
        modifiers = str(arg)
    if opt in ('-z', '--Z'):
        Z=str(arg).split('$')
        hsh=str(Z[0])
        mod=str(Z[1]).split()
        if (hsh=='pbkdf2'):
```

```
            N=int(mod[0])
            func=str(mod[1])
            keylen=int(mod[2])
            salt=str(Z[2])
        if (hsh=='scrypt'):
            N=int(mod[0])
            p=int(mod[1])
            r=int(mod[2])
            keylen=int(mod[3])
            salt=str(Z[2])

""" options for internal hash used by pbkdf2:
"""

if (func=="sha1"):
    hashfunc=hashlib.sha1
if (func=="sha256"):
    hashfunc=hashlib.sha256
if (func=="sha512"):
    hashfunc=hashlib.sha512

""" output returned in "$" delimited string:
    pbkdf2$<salt>$<N>$<keylength>$<function>$$<hash>
"""

if (hsh=="pbkdf2u"):
    if (salt=="random"):
        salt=binascii.hexlify(os.urandom(keylen))
    print "pbkdf2"+u+str(N)+" "+str(func)+" "+str(keylen)+u+salt+u+pbkdf2_hex(input,
salt, N, keylen, hashfunc)
    exit()

if (hsh=="pbkdf2"):
    if (salt=="random"):
        salt=binascii.hexlify(os.urandom(keylen))
    print "pbkdf2"+u+str(N)+u+salt+u+pbkdf2_hex(input, salt, N, keylen, hashfunc)
    exit()

if (hsh=="scrypt"):
    outbuf = create_string_buffer(keylen)
    if (salt=="random"):
        salt=binascii.hexlify(os.urandom(keylen))

    result = scrypt(input, salt, N, r, p, keylen)
    if result:
        print "error in scrypt"
        print result
        exit()
    if (N==1):
        print "scrypt"+u+str(N)+u+binascii.hexlify(outbuf.raw)
        exit()
    print "scrypt"+u+str(N)+" "+str(r)+" "+str(p)+" "+str(keylen)
+u+salt+u+binascii.hexlify(outbuf.raw)
    exit()
```

```
""" older hashes: md5, sha1, sha256, sha512, null
"""

if (hsh=="md5"):
     hash=hashlib.md5
     flag=1
if (hsh=="sha1"):
     hash=hashlib.sha1
     flag=1
if (hsh=="sha256"):
     hash=hashlib.sha256
     flag=1
if (hsh=="sha512"):
     hash=hashlib.sha512
     flag=1

if (flag==1):
     print hsh+"$"+hash(input + salt).hexdigest()
     exit()
if (hsh=="crc32"):
     print binascii.crc32(input) & 0xffffffff
     exit()
if (hsh=="random"):
    hash = os.urandom(keylen).encode('hex')
    print hash
    exit()
if (hsh=="null"):
     print input
     exit()

print "error: << " + hsh + " >> is not supported."
exit()

"""
    :license: BSD, see LICENSE for more details. copyright: (c)
    :Copyright 2012 by John Leo Zimmer.
                      johnleozim@gmail.com
 pbkdf2
 Module from https://github.com/mitsuhiko/python-pbkdf2 implements PBKDF2 hash.
 :copyright: (c) Copyright 2011 by Armin Ronacher. license: BSD
 Cut and pasted here from his pbkdf2.py to simplify distribution.

  Scrypt was created by Colin Percival and is licensed as 2-clause BSDscrypt.py
  Author: Magnus Hallin
  Home Page: http://bitbucket.org/mhallin/py-scrypt
  License: 2-clause BSD
  wget https://pypi.python.org/packages/source/s/scrypt/scrypt-0.6.1.tar.gz
  tar zxvf scrypt-0.6.1.tar.gz

      $ cd py-scrypt
      $ python setup.py build

      Become superuser (or use virtualenv):
      $ sudo python setup.py install
```

```
      Run tests after install:
      $ python setup.py test

"""
```

XUSHSH Chris Uyehara

```
XUSHSH ;CKU/HAWAII - PASSWORD ENCRYPTION ;11/01/07 15:09 ;
 ;;8.0;KERNEL;;Jul 10, 1995
 ;; This is the public domain version of the VA Kernel.
 ;; Use this routine for your own encryption algorithm
 ;; Input in X
 ;; Output in X
 ;; 11/01/07 - Routine currently uses SHA-512 HASH.
 ;; SHA-512 Description: http://en.wikipedia.org/wiki/SHA_hash_functions
 ;;
 ;; Copyright 2007 Chris Uyehara.
 ;;
 ;; This program is free software: you can redistribute it and/or modify
 ;; it under the terms of the GNU General Public License as published by
 ;; the Free Software Foundation, either version 3 of the License, or
 ;; (at your option) any later version.
 ;;
 ;; This program is distributed in the hope that it will be useful,
 ;; but WITHOUT ANY WARRANTY; without even the implied warranty of
 ;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 ;; GNU General Public License for more details.
 ;;
 ;; You should have received a copy of the GNU General Public License
 ;; along with this program. If not, see <http://www.gnu.org/licenses/>.
 ;;
A
      N EXT S EXT=0 G NONEXT
EN(X)
NONEXT
      N Y,Z S Y=""
        ZSYSTEM "umask 177 ; echo -n """_X_""" | openssl dgst -sha512 &>
/tmp/login_"_$J_".hash"

        N LOGINTMP S LOGINTMP="/tmp/login_"_$J_".hash"
        O LOGINTMP:(REWIND:EXCEPTION="Q")
        U LOGINTMP:(EXCEPTION="G EOF")
        F  U LOGINTMP R Z S Y=Y_Z
        Q

EOF
        I '$ZEOF ZM +$ZS
        C LOGINTMP:(DELETE)
      I $D(EXT)&(EXT=0) S X=Y Q
        Q Y
```

```
UPDATE
      N I,Q S I=0
      I $D(^VA(200,"XUSHSH")) W !," * * YOU HAVE ALREADY INSTALLED THIS ENHANCEMENT!!
* * " Q

      W !," * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *"
      W !," WARNING: ONCE THE UPDATE HAS COMPLETED YOU **CANNOT** "
      W !," UNINSTALL or ROLLBACK UNLESS YOU HAVE BACKED UP YOUR DATA "
      W !," * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *"
      W !!," DO YOU WANT TO CONTINUE?? YES or NO?? NO// " R Q
      I Q'="YES" Q

      F  S I=$ORDER(^VA(200,I)) QUIT:+I=0 D
      . N X I $G(^VA(200,I,0))'="" S X("A")=$P(^VA(200,I,0),"^",3)
      . I $G(^VA(200,I,.1))'="" S X("V")=$P(^VA(200,I,.1),"^",2)
      . I X("A")=""!(X("A")=";") Q
      . N Y S Y("A")=$$EN(X("A"))
      . S Y("V")=$$EN(X("V"))
      . S $P(^VA(200,I,0),"^",3)=Y("A"),$P(^VA(200,I,.1),"^",2)=Y("V")
      . S X("AXREF")=$QS($Q(^VA(200,"A",X("A"))),4)
      . K ^VA(200,"A",X("A")) S ^VA(200,"A",Y("A"),X("AXREF"))=+$H

      W !!," * * INSTALL COMPLETED SUCCESSFULLY!! * * ",!!
      S ^VA(200,"XUSHSH")="BETA"
      Q
```

XUSHSH FOIA

```
XUSHSH ;SF-ISC/STAFF - PASSWORD ENCRYPTION ;3/23/89 15:09 ;
 ;;8.0;KERNEL;;Jul 10, 1995
 ;; This is the public domain version of the VA Kernel.
 ;; Use this routine for your own encryption algorithm
 ;; Input in X
 ;; Output in X
A Q
EN(X) Q X
```

XUSHSH vxVistA

Note: I have just barely cracked open this interesting distribution.

Most striking is its use of the DSS namespaced file

`^VFD(21614.1,0) = VFD SUPPORTED API^21614.1^14^14`

XUSHSH, listed below consists of a look-up and Xecute of a field in "ONE-WAY HASH" an entry in that file, details below. The result is an md5 hash of X.

After copying in and compliling ^VWHSH3 (Cache version of XUSHSH) and running BUILD^VWHSH0 to create ^VA(200,"VWHSH") We were able to simply change to **"PBKDF2 HASH"** See below for the simple entry in file VFD SUPPORTED API

Although md5 is a very old hash it is a true hash unlike WorldVista, LEGACY and therefore the only practical way of switching from "ONE-WAY HASH" aka md5 to PBKDF2, or any of the others offered by this package is to enter new AC/VC. This works just fine... and is another reason to distribute test/educational/set-up versions of a distribution with passwords set to NONE.

```
XUSHSH ;SF-ISC/STAFF - PASSWORD ENCRYPTION ;3/23/89 15:09 ; 2/16/07
9:41am
;; 8.0;KERNEL;;Jul 10, 1995
;; DSS Version: 1.0
;
;; DSS/LM  entire routine modified to support MD5 encryption
;
A ;;
S X=$$EN(X)
Q
;
EN(X) ;;
D X^VFDXTX("ONE-WAY HASH") Q X
;;; Change to  D X^VFDXTX("PBKDF2 HASH") Q X
;
UC(X) ;;
Q $$UP^XLFSTR(X)
;
```

```
^VFDXTX
...
X(VFDAPI) ; Xecute an implementation-specific or supported API
; VFDAPI - req - Supported API exact name
;
Q:'$L($G(VFDAPI)) NVFDSIEN,VFDX
S VFDSIEN=$O(^VFD(21614.1,"B",VFDAPI,0)) Q:'VFDSIEN
; check for implementation specific xecute first, then check for
; supported API default xecute
S VFDX=$G(^VFD(21614,VFDSIEN,1))
I VFDX="" S VFDX=$G(^VFD(21614.1,VFDSIEN,2))
I VFDX'="" X VFDX
```

```
Q
;
OUTPUT FROM WHAT FILE: VFD SUPPORTED API//

NAME: ONE-WAY HASH                      APPLICATION GROUP: KERNEL SECURITY
  DATE CREATED: JAN 12, 2009            RESPONSIBLE PERSON: LLOYD
SUPPORTED VARIABLE: X                   ALWAYS DEFINED: YES
  BRIEF DESCRIPTION: Unencrypted string
  DEFAULT XECUTE:
 S X=$$UP^XLFSTR($$MAIN^XUMF5BYT($$HEX^XUMF5AU($$MD5R^XUMF5AU(X))))
 REMARKS: Set variable X equal to a hash of the unencrypted argument X.

 NAME: PBKDF2 HASH                       APPLICATION GROUP: KERNEL SECURITY
  DATE CREATED: SEP 15, 2012            RESPONSIBLE PERSON: JLZ
SUPPORTED VARIABLE: X                   ALWAYS DEFINED: YES
  BRIEF DESCRIPTION: Unencrypted string
  DEFAULT XECUTE: S X=$$EN^VWHSH3(X)
 REMARKS: CALLS ^VWHSH3, Cache version of XUSHSH. Uses ^VA(200,"VWHSH") to set
    PYTHON and PARAMS variables prior to call to OS via PIPE.
    Returns PBKDF2 hash of X.

 ^VFD(21614.1,1,0)="ONE-WAY HASH^1^3090112^LLOYD"
^VFD(21614.1,1,1,0)="^21614.11^1^1"
^VFD(21614.1,1,1,1,0)="X^1^Unencrypted string"
^VFD(21614.1,1,1,"B","X",1)=""
^VFD(21614.1,1,2)=
"S X=$$UP^XLFSTR($$MAIN^XUMF5BYT($$HEX^XUMF5AU($$MD5R^XUMF5AU(X))))"
^VFD(21614.1,1,3,0)="^21614.13^1^1^3090820^^^"
^VFD(21614.1,1,3,1,0)="Set variable X equal to a hash of the unencrypted argument X."
```

Appendix C README

The green text is designed to work with copy and paste into a terminal/console.

This is the README from Github prettied up to assist easy installation.

```
Rapid recipe for the already-believer.
For long-winded (and perhaps soon completed) cookbook see the rest of xushsh_pbkdf2.pdf (this very document).
(Purists can read the README file for this same info in plain black and white.)

We will replace XUSHSH and use a global node,^VA(200,"VWHSH"), to configure & control the new XUSHSH routine.
After a demonstration of pbkdf2, we will set it to return the old hash unchanged until we are ready to convert
the NEW  PERSON file's ACCESS/VERIFY CODE fields and their "A","AOLD", and "VOLD" cross-references.
The final result is a conversion to pbkdf2 that is entirely transparent to the ordinary VISTA user.

Note: We can also convert WorldVistA's legacy hash to no-hash, if that is desired for
development/teaching/installation-tweeking/etc., and then convert that to pbkdf2 at any time.  Of course, the
reversal of pbkdf2 is NOT feasible.  There is a backup created as a precaution during installation.  But that
should be deleted as the final step of this installation once a successful conversion is confirmed.

GT.M host, this example was run on a dEWDrop virtual machine:

Python will be already installed and in the PATH.

vista@dEWDrop:~$ mkdir git   <<<--- or somewhere of <your> choice.
            :~$ cd git
         :~/git$ git clone git://github.com/grapaZ/xushsh.git   <<<---- or download a zip from
                                                                        https://github.com/grapaZ/xushsh
         :~/git$ cd xushsh
  :~/git/xushsh$ cp xus.py ~/bin     <<<--- or some other location; but be prepared to alter
                                                                        ^VA(200,"VWHSH") in a later step.
  :~/git/xushsh$ python ~/bin/xus.py  <<<-- testing w defaults in xus.py
f4ca507c07d0bd31bc779a08756826a6fd9dd97d43ac25e4   <<<---- this is the pbkdf2 hash of "password" with no salt,
                                                           10000iterations, internal function sha512,
                                                           keylength 24(hex digits).
This is a good time to add a salt which can be generated like so:
            :~$ python ~/bin/xus.py -h random
3731f02531b2157558140f0c222ac4aedfa9486bd8889aca    <<<--- for example

            :~$ python ~/bin/xus.py --salt="3731f02531b2157558140f0c222ac4aedfa9486bd8889aca"
f6534f8db4d7f8f3808d0735da1f38c487f539d801ce7dfd
```

Further, ~/bin/xus.py can (and should) be edited now to lock-in your salt as a permanent default (not subject to casual change).

_____manual labor option _____

```
            :~$ cp ~/git/xushsh/*.m ~/p/
```

Now we can go into MUMPS:

```
            :~$ mumps -dir

        MU-beta> zlink "VWHSH8.m","VWHSH0.m","VWHSHLEG.m"
```

_____or we can do: _____

```
        MU-beta> DO ^%RI

                 Formfeed delimited <No>?
                 Input device: <terminal>: /home/vista/git/xushsh/vwhshGTM.ro
                 VWHSH* for GT.M                          NOTE--->^<-- for Cache' use vwhshCACHE.ro
                 GT.M 19-OCT-2012 12:32:40


                 Output directory : /home/vista/p/

                 VWHSH0    VWHSH8    VWHSHLEG


                 Restored 226 lines in 3 routines.
```

_____continue _____

```
        MU-beta> W $$EN^XUSHSH("test")   <<<------------LEGACY hash
115116101116
        MU-beta> W $$EN^VWHSHLEG("test")
115116101116
```

^VWHSH8 will be able to replace ^XUSHSH once global node ^VA(200,"VWHSH") is properly configured by the following call to ^VWHSH0:

```
       MU-beta> DO BUILD^VWHSH0()    <<<--------configuration will default to "LEGACY"
       MU-beta> w $$EN^VWHSH8("test")
115116101116                        <<<----------and NOW that is what ^VWHSH8 will return.
```

```
       MU-beta> DO SET^VWHSH0("PBKDF2") <<<-----LEGACY, NONE, and PBKDF2 are supplied by BUILD()
       MU-beta> W $$EN^VWHSH8("test")
f054d357dfc8464f110cd32b36423acead8e1bcbf1bd8197  <<<--- If you have not set a default salt.

Lots of other stuff could be done with ^VA(200,"VWHSH"), but KeepItSimple for now.
I recommend looking at ^("VWHSH") with VPE.
Some VISTA configurations may require editing the location of xus.py

       MU-beta> DO SET^VWHSH0("LEGACY")   <<<---- We are now ready to replace the old XUSHSH.
       MU-beta> zsy "cp ~/p/VWHSH8.m ~/p/XUSHSH.m"  <<<----overwrite ^XUSHSH with ^VWHSH8
       MU-beta> zlink "XUSHSH.m"

We now have the new XUSHSH replacing the legacy version.
Nothing else has changed.

   <<<NOW>>> we are ready to switch to PBKDF2...


       MU-beta> DO CONVERT^VWHSH0("PBKDF2")    <<<---------more or less irreversible step
( or   MU-beta> DO CONVERT^VWHSH0("NONE")     <<<---- less irreversible option :-)


CONVERT^VWHSH0()  is going to change fields #2 and #11

^VA(200,D0,0)=  ^ ^ (#2) ACCESS CODE [3F] ^
^VA(200,D0,.1)= ^ (#11) VERIFY CODE [2F] ^

And Cross-References:

     "A"    MUMPS
            Field:  ACCESS CODE  (200,2)
                    1)= S ^VA(200,"A",X,DA)=+$H
                    2)= K ^VA(200,"A",X,DA)
                    3)= ACCESS CODE lookup

  "AOLD"    MUMPS
            Field:  ACCESS CODE  (200,2)
       Description:  This is a list of used ACCESS CODES that may not be used
                     again untill the OLD ACCESS CODE PURGE option is run.
                     1)= Q
                     2)= S ^VA(200,"AOLD",X,DA)=+$H
                     3)= OLD ACCESS CODES
```

```
"VOLD"    MUMPS
          Field:  VERIFY CODE  (200,11)
    Description:  This builds a list of old VERIFY CODEs that this user has
                  had in the past.  It is cleaned  out with the same option
                  the purges the old access code X-ref.
                  1)= Q
                  2)= S ^VA(200,DA,"VOLD",X)=+$H
```

BUT the conversion is NOT going to reset any cross reference $H, etc.
The intent is to simply rehash ALL of these global nodes without changing anything else.
Everything will look the same to users, including their passwords, expiration dates, etc.

© Licenses.txt