Table of Contents

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 will be entirely transparent to VISTA users. And we will make no changes to any Kernel routines other than to `XUSHSH` itself.[1]

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. WorldVistA's version uses a *reversible* ASCII encoding. Replacements supporting hashes such as SHA1 and SHA512 have been developed. 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.

This package includes
- i.) A Python component (`xushsh.py`) which will reside on the host operating system
- ii.) This script is called from our new `^XUSHSH` using a MUMPS pipe command.
- iii.) The specifics of the call are configured in code stored at `^VA(200,"VWHSH")`.

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 `xushsh.py` to produce the required hash.

Consider the three distinct phases of this installation. All are aimed toward replacing weak or null hashes currently available with cryptographically strong options, with a strong preference for PBKDF2. The process described changes the hashing without changing any passwords.

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 version 2 VirtualBox VirtualMachine running on the same hardware.

1. `xushsh.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). A table of default values within `xushsh.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 `VWHSH00` to walk through New Person to convert to the new hash. (There is a backup global node created that **should be deleted** once you are satisfied with the conversion.)

Stage One: setup host's Python tools

On either platform we begin by setting up the host to support our Python script, `xushsh.py`.[2] An advantage of this is that the xushsh.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 style pipe to call it.

i. Python2.x required

Expect it to be included and accessible on any Linux host.

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

2 Recommended options:
- VPE is very helpful for configuring global nodes (especially `^VA(200,"VWHSH",*)`), for checking for proper results in New Person file, etc.
- git-core.
- I also installed GT.M's Posix plugin. This allows high-resolution timing to help choose the iteration parameter in the hash algorithm.

For example, testing on the dEWDrop virtual machine:

```
vista@dEWDrop:~$ pythonPython 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
[GCC 4.4.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

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.
>>>
```

ii. PBKDF2 hash implemented in Python

available as a zip or via git at: https://github.com/mitsuhiko/python-pbkdf2/[3]

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

This provides `pbkdf2_hex` which is imported by `xushsh.py`, our final host component...

iii. The Python script xushsh.py

This component provides a flexible interface that will be used by `XUSHSH.` access to a number of other useful hashes, and a set of defaults that can be easily changed by directly editing the script.

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

On dEWDrop I have copied `pbkdf2.py` and `xushsh.py` to `~/bin/` and this simple test confirms proper function:

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

This unembellished call to `xushsh.py` shows how it uses its set of hard-coded 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 so far.

And in Windows the simplest configuration puts both `xushsh.py` and `pbkdf2.py` in `C:\Python27\`, along with `python.exe` itself. And the call looks like this:

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

3 Copyright (c) 2011 by Armin Ronacher. license: BSD.

Now we can move directly to **Stage Two**... or play around with xushsh.py for a bit longer.

Here are a couple more calls to `xushsh.py`:

sha1 hash of "test":

```
c:\>c:\Python27\python c:\Python27\xushsh.py -h sha1 -p "test"
b56498f027060a5f4707bad6621d354d01672b86
```

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

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

pbkdf2 hash of "test" with 30,000 iterations, keylength 20:
(If any parameter is entered more than once the last one is used.)

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

`xushsh.py` will accept its six parameters in any order, using the defaults for any not entered. Both a short and long form may be used for each (See text box in Appendix A).

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, `VWHSH00` 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 `VWHSH00`.

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

We will use each `VWHSH0` tag in its turn:

i. TEST

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

A result other than NONE or LEGACY ends the installation. Only these two options are convertible to a cryptographically robust hash, which by definition is irreversible.

ii. SAVEOLD

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

```
MU-beta>do SAVEOLD^VWHSH0

Routine: XUSHSH       Loaded, cp: cannot stat `/home/vista/p/XUSHSH.m':
No such file or directory
Saved as VWHSHLEG
```

This error arose on dEWDrop because `XUSHSH.m` resides not in directory `/p/` but in `~/r/XUSHSH.m`. This required manually copying `XUSHSH.m` to `VWXUSHSH.m`.

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

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
```

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 `xushsh.py --hash=null -password=X`, a null operation but confirmation that the Python call is being found by our replacement `XUSHSH`.

iiii. MOVEIN

We can now 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")`. As we have just seen this is a rather convoluted way of doing very little. :-) 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 support the new version.

And now we are ready for the crucial data conversion stage.

Stage Three: REHASH Access / Verify Codes

We use routine `^VWHSH02 for this stage`. 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^VWHSH02(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^VWHSH02

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^VWHSH02 permits reversing the install if that is desired.

```
MU-beta>do REVERT^VWHSH02
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^VWHSH02

```
MU-beta>Do KILL^VWHSH02
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.[4]

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 NONE in which `XUSHSH(X)` simply returns `X` unaltered. WorldVistA's version uses a simple and easily reversible ASCII

4 http://en.wikipedia.org/wiki/Cryptographic_hash_function

encoding algorithm which I will term LEGACY. Of course, neither NONE nor LEGACY is a cryptographic hash.[5] And neither meets standards for a system entrusted with patient care data.[6]

> This package *will* convert LEGACY to NONE 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 NONE to a cryptographically robust hash, which in my opinion should be PBKDF2. The xushsh.py script can support sha1, sha2, etc. for other uses. For password protection, again IMHO, these weaker alternatives are becoming obsolete.

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[7] 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 has was showing its age with a feasible attack identified. NTSI[8] 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.[9] Eventually both openssl and GnuPG were supported with consistent results for md5 thru sha512 on both GT.M and Caché. However,

---

5 http://en.wikipedia.org/wiki/Kerckhoffs%27_principle#Security_through_obscurity
http://slashdot.org/features/980720/0819202.shtml

6 https://www.cchit.org/documents/18/158304/CCHIT+Certified+2011+Security+Test+Script+ALL+DOMAINS.pdf

7 sed magic: `s SED="sed -e 's/$/\r/'|"` made my Linux hash look like the Windows one.

8 National Institute of Standards and Technology: http://csrc.nist.gov/groups/ST/hash/statement.html April 25, 2006

9 I long ago used pgp to encrypt MailMan messages written to MSDOS and read back into MSM.

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.[10] Turns out there are very interesting disadvantages to using efficient hashing algorithms. So new, intentionally slower algorithms have emerged.[11] Currently available are bcrypt, pbkdf2 and scrypt. Why PBKDF2?

The purpose of `xushsh.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`[12] to support PBKDF2.

The first security improvement compared to storing passwords in clear text is to only store a hashed or encrypted form of the password. Hashing is preferable over encryption in this context, because with encryption it is possible to go back to the unencrypted form (if you know the key) whereas with a hash aims to make it computationally infeasible to do so and forces an attacker into what amount to fancy guessing games.

Next 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[13] 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.

Iteration means to compute the hash function not once, but hundreds or even 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.)

Why PBKDF2?

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.

(There are versions of both bcrypt and scrypt written in Python but available implementations are painfully slow and therefore not practical alternatives for this approach.)

10 http://mytechencounters.wordpress.com/2011/04/03/gpu-password-cracking-crack-a-windows-password-using-a-graphic-card/
11 http://blog.josefsson.org/2011/01/07/on-password-hashing-and-rfc-6070/#more-228
12 The work of Armin Ronacher https://github.com/mitsuhiko/python-pbkdf2/
13 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[14]

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`

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

So, 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 `xushsh.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 `xushsh.py` →
                Pipe to `Python xushsh.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.[15] Scrypt is arguably the strongest.[16]

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.

Details:

a.)
Under Linux, where best to locate `xushsh.py`,and `pbkdf2.py`? I have them 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.

c.)

Salt.

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

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.[17] 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.

---

17 (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

```
#!/usr/bin/python

import getopt
import sys
import hashlib
import binascii
from pbkdf2 import pbkdf2_hex

"""defaults:
"""
hsh="pbkdf2u"
input="password"
salt=""
cycles=10000
keylen=24
func="sha512"

u="$"
flag=0
```

Input parameters:

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

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

md5, **sha1**, **sha256**, **sha512, null**

--password=<string> or **-p <string>**

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

These are used only if **hash=pbkdf2(u).**

--cycles=<integer> or **-c <integer>**

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

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

```
options, remainder = getopt.gnu_getopt(sys.argv[1:], 'h:p:s:i:k:f:c:',
['hash=','password=','input=','salt=','iterations=','cycles','keylen=','function=',] )

for opt, arg in options:
    if opt in ('-h', '--hash'):
        hsh = str(arg)
    if opt in ('-i', '-p', '--password', '--input'):
        input = str(arg)
    if opt in ('-s', '--salt'):
        salt = str(arg)
    if opt in ('-c', '--cycles', '--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" hash without detail:
"""
if (hsh=="pbkdf2u"):
    print pbkdf2_hex(input, salt, cycles, keylen, hashfunc)
```

```
    exit()

""" output returned in "$" delimited string:
    pbkdf2$sha1$<salt>$<cycles>$<hash>
"""
if (hsh=="pbkdf2"):
    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
"""
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=="null"):
     print input
     exit()

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

"""
    :copyright: (c) Copyright 2012 by John Leo Zimmer.
                     johnleozim@gmail.com
    :license: GNU Affero General Public License version 3,
        details in file LICENSES or at <http://fsf.org/>

"""
```

XUSHSH: (VWHSH8, VWHSH3)

```
XUSHSH   ;IA/GpZ-ROBUST(PBKDF2)HASHING UTILITY v1.0; 10/1/12 6:26pm
V        ;;8.0;KERNEL;;Jul 10, 1995
        ;
        ;--------------------------------------------------------------------
        ; Copyright (c) 2012 John Leo Zimmer Email: johnleozim@gmail.com      ;
        ; All rights reserved. Glenwood, Iowa                                 ;
        ;                                                                     ;
        ; This program is free software: You can redistribute it and/or modify ;
        ; it under the terms of the GNU Affero General Public License as      ;
        ; published by the Free Software Foundation, either version 3 of the  ;
        ; License, or (at your option) any later version.                     ;
        ;                                                                     ;
        ; See COPY in distribution package or <http://www.gnu.org/licenses/>. ;
        ;--------------------------------------------------------------------
         ;
A        SET X=$$EN(X) QUIT
         ;
EN(X)    ;
         NEW VWCALL
         XECUTE ^VA(200,"VWHSH")
         QUIT:$L($G(VWCALL))=0 X  ;Short circuit to support LEGACY or NULL hash.
         QUIT $$HOSTPIPE(VWCALL)
         ;
         ; -----------GT.M-PIPE-Magic----------------
HOSTPIPE(CALL) ;
         New X
         Open "PIPE":(command=CALL:READONLY)::"PIPE"
         Use "PIPE" READ X
         Close "PIPE"
         Use 0
         QUIT X
```

ONLY DIFFERENCE between GT.M version (^VWHSH8) and Caché version (^VWHSH3) is in the HOSTPIPE(CALL)

```
         ; -----------Cache-PIPE-WaveWand------------
HOSTPIPE(CALL) ;
         New ZUT,X
         Set ZUT=$ZUTIL(68,40,1)
         Open CALL:"Q"  Use CALL Read X
         Close CALL
         Set ZUT=$ZUTIL(68,40,ZUT)
         Use 0
         Quit X
```

Appendix B

These listings are included as history, out of respect for the work involved in their creation, and simply for my own satisfaction. Enjoy:

CKUSHA1 Chris Uyehara

available at: https://www.box.com/shared/xq7j2yhukq

```
CKUSH1  ; Authored by Chris Uyehara, chris.uyehara@gmail.com 6/10/
        ; SHA1, a cryptographic hash function designed by the National Security Agency (NSA) and
        ; published by the NIST as a U.S. Federal Information Processing Standard. SHA-1 is the
        ; best established of the existing SHA hash functions, and is employed in several widely
        ; used security applications and protocols. In 2005, security flaws were identified in SHA-1,
        ; namely that a possible mathematical weakness might exist, indicating that a stronger hash
        ; function would be desirable.
        ; -------------------------------------------------------------------------------------------
        ; Version 0.1b - Initial build; runs on GT.M and Cache.
        ; -------------------------------------------------------------------------------------------

        ; Copyright (C) 2009 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 2
        ; 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, write to the Free Software
        ; Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.

        Q ; Direct calling prohibited.

DIGEST(MESSAGE) ; Entry Point. Extrinsic Function. Pass the String to be hashed as the
        ; MESSAGE parameter and expect the SHA1 hash to be returned.
        N DIGESTMESSAGE,H,M
        D PREINIT

        ; Setup each message (M) with appropriate variables.
        N I,EOM S EOM("POS")=0,I=1 F  D  Q:EOM("POS")>$LENGTH(MESSAGE)
        . S M(I)=$ZBITSTR(512,0)
        . N J,TMPHEXMSG S TMPHEXMSG="",EOM("POS")=EOM("POS")+1,EOM("CNT")=0
        . F EOM("POS")=EOM("POS"):1:EOM("POS")+63 D  Q:EOM("POS")>$LENGTH(MESSAGE)
        . . Q:EOM("POS")>$LENGTH(MESSAGE)  S EOM("CNT")=EOM("CNT")+1
        . . S TMPHEXMSG=TMPHEXMSG_$$DEC2HEX($A($E(MESSAGE,EOM("POS"))))
        . S EOM("MSG")=I I '$LENGTH(TMPHEXMSG) S I=I+1 Q
        . N TMPBINMSG S TMPBINMSG=$$HEX2BIN(TMPHEXMSG)
        . N K F K=1:1:$ZBITLEN(TMPBINMSG) D
        . . S M(I)=$ZBITSET(M(I),K,$ZBITGET(TMPBINMSG,K))
        . S I=I+1
        I EOM("CNT")>55 S M(I)=$ZBITSTR(512,0)

        ; Padding the end of the message with a one.
```

```
        I EOM("POS")<64 S EOM("MSG")=1
        I EOM("POS")>64 F  D  Q:EOM("POS")<64
        . S EOM("POS")=EOM("POS")-64
        S EOM=(EOM("POS")-1)*8
        S M(EOM("MSG"))=$ZBITSET(M(EOM("MSG")),EOM+1,1)

        ; Append the length of the original message to the end of the new message.
        N MINDEX D
        . N I S I="M" F  S I=$QUERY(@I) Q:I=""  S MINDEX=I
        N MESSAGELEN S MESSAGELEN=$LENGTH(MESSAGE)*8
        S MESSAGELEN=$$DEC2BIN(MESSAGELEN)
        N I,J S J=512-$ZBITLEN(MESSAGELEN)+1 F I=1:1:$ZBITLEN(MESSAGELEN) D
        . S @MINDEX=$ZBITSET(@MINDEX,J,$ZBITGET(MESSAGELEN,I)),J=J+1

        ; Build the hash :]
        N II F II=1:1:$QS(MINDEX,1) D
        . N A,B,C,D,E,W
        . D POSTINIT(H(0),H(1),H(2),H(3),H(4),M(II)) D
        . N I F I=1:1:79 D
        . . N TEMP S TEMP=$$COMPUTET(I,A(I-1),B(I-1),C(I-1),D(I-1),E(I-1),$
$COMPUTEW(I))
        . . S E(I)=D(I-1),D(I)=C(I-1),C(I)=$$ROTL(B(I-1),30),B(I)=A(I-1),A(I)=TEMP
        . S H(0)=$$BINPLUSBIN(H(0),A(79)),H(1)=$$BINPLUSBIN(H(1),B(79))
        . S H(2)=$$BINPLUSBIN(H(2),C(79)),H(3)=$$BINPLUSBIN(H(3),D(79))
        . S H(4)=$$BINPLUSBIN(H(4),E(79))
        Q $$BIN2HEX(H(0))_$$BIN2HEX(H(1))_$$BIN2HEX(H(2))_$$BIN2HEX(H(3))_$
$BIN2HEX(H(4))

PREINIT ; Pre initialization of H0 ... H4
        S H(0)=$$HEX2BIN("67452301")
        S H(1)=$$HEX2BIN("efcdab89")
        S H(2)=$$HEX2BIN("98badcfe")
        S H(3)=$$HEX2BIN("10325476")
        S H(4)=$$HEX2BIN("c3d2e1f0")
        Q

POSTINIT(AA,BB,CC,DD,EE,WW) ; Pre initialization of A0 ... E0 and W0
        N I,BITMARK S BITMARK=1 F I=0:1:15 D
        . N BINTMP,J S BINTMP=$ZBITSTR(32,0) F J=1:1:32 D
        . . S BINTMP=$ZBITSET(BINTMP,J,$ZBITGET(WW,BITMARK))
        . . S BITMARK=BITMARK+1
        . S W(I)=BINTMP

        N T S T=$$COMPUTET(0,AA,BB,CC,DD,EE,W(0))
        S E(0)=DD
        S D(0)=CC
        S C(0)=$$ROTL(BB,30)
        S B(0)=AA
        S A(0)=T
        Q

HEX2BIN(HEXVALUE) ; Convert a hexadecimal formatted string to a binary formatted bit
string.
        N HEXLEN,TMPLEN,BINRESULT,BINVALUES,AVALUE,TMPVALUE,TMPRESULT,I,J
        S TMPLEN=$LENGTH(HEXVALUE),BINVALUES=8421,TMPRESULT=""
```

```
        N UPPER,LOWER S UPPER="ABCDEFGHIJKLMNOPQRSTUVWXYZ",LOWER="abcdef"
        S HEXVALUE=$TRANSLATE(HEXVALUE,UPPER,LOWER)
        S HEXLEN=$LENGTH(HEXVALUE)
        I HEXLEN'=TMPLEN Q -1
        S BINRESULT=$ZBITSTR(HEXLEN*4,0)
        N I F I=1:1:$LENGTH(HEXVALUE) D
        . S AVALUE=$E(HEXVALUE,I)
        . I AVALUE?1L S AVALUE=$A(AVALUE)-87
        . N J F J=1:1:4 D
        . . S TMPVALUE=$E(BINVALUES,J)
        . . I (AVALUE>TMPVALUE)!(AVALUE=TMPVALUE) S
AVALUE=AVALUE#TMPVALUE,TMPRESULT=TMPRESULT_"1"
        . . E  S TMPRESULT=TMPRESULT_"0"
        I $ZBITLEN(BINRESULT)'=$LENGTH(TMPRESULT) Q -1
        N I F I=1:1:$LENGTH(TMPRESULT) D
        . S BINRESULT=$ZBITSET(BINRESULT,I,$E(TMPRESULT,I))
        Q BINRESULT

BIN2HEX(BIN) ; Convert a binary formatted bit string to hexadecimal.
        Q $$DEC2HEX($$BIN2DEC(BIN))

DEC2HEX(DEC) ; Convert a single decimal value to hexadecimal.
        I DEC<16 Q $$SDEC2HEX(DEC)
        N HEXVALUE,TMPMOD,HEXRESULT S HEXRESULT=""
        F  D  Q:DEC<16
        . S TMPMOD=DEC#16,HEXRESULT=HEXRESULT_$$SDEC2HEX(TMPMOD),DEC=DEC\16
        . I DEC<16 S HEXRESULT=HEXRESULT_$$SDEC2HEX(DEC)
        I $LENGTH(HEXRESULT)#2 S HEXRESULT=HEXRESULT_"0"
        Q $REVERSE(HEXRESULT)

SDEC2HEX(DEC) ; Helper sub-routine for DEC2HEX.
        Q

$SELECT(DEC=10:"a",DEC=11:"b",DEC=12:"c",DEC=13:"d",DEC=14:"e",DEC=15:"f",DEC<16&DEC>
-1:DEC,1:-1)

DEC2BIN(DEC) ; Convert a single decimal value to binary.
        Q $$HEX2BIN($$DEC2HEX(DEC))

BIN2DEC(BIN) ; Convert a binary formatted bit string to a decimal value.
        N BITLEN,RESULT S BITLEN=$ZBITLEN(BIN)-1,RESULT=0
        N I F I=1:1:$ZBITLEN(BIN) D
        . I $ZBITGET(BIN,I) S RESULT=RESULT+(2**BITLEN)
        . S BITLEN=BITLEN-1
        Q RESULT

SHR(WORD,NUM) ; Shift right operation.
        N BINRESULT S BINRESULT=$ZBITSTR($ZBITLEN(WORD),0)
        N I F I=1:1:$ZBITLEN(WORD)-NUM D
        . S BINRESULT=$ZBITSET(BINRESULT,I+NUM,$ZBITGET(WORD,I))
        Q BINRESULT
ROTR(WORD,NUM) ; Circular bit shift right operation.
        N BINRESULT
        N I F I=1:1:NUM D
        . S BINRESULT=$$SHR(WORD,1)
```

```
        . S BINRESULT=$ZBITSET(BINRESULT,1,$ZBITGET(WORD,$ZBITLEN(WORD)))
        . S WORD=BINRESULT
        Q BINRESULT

SHL(WORD,NUM) ; Shift left operation.
        N BINRESULT S BINRESULT=$ZBITSTR($ZBITLEN(WORD),0)
        N I F I=1:1:$ZBITLEN(WORD)-NUM D
        . S BINRESULT=$ZBITSET(BINRESULT,I,$ZBITGET(WORD,I+NUM))
        Q BINRESULT

ROTL(WORD,NUM) ; Circular bit shift left operation.
        N BINRESULT
        N I F I=1:1:NUM D
        . S BINRESULT=$$SHL(WORD,1)
        . S BINRESULT=$ZBITSET(BINRESULT,$ZBITLEN(WORD),$ZBITGET(WORD,1))
        . S WORD=BINRESULT
        Q BINRESULT

COMPUTET(T,A,B,C,D,E,W) ; Compute T. Used for t0, t1 ... t79
        N ROTL,F,RESULT
        S ROTL=$$ROTL(A,5),F=$$COMPUTEF(T,B,C,D)
        N ROTLDEC,FDEC,EDEC,KDEC,WDEC,DECRESULT
        S ROTLDEC=$$BIN2DEC(ROTL),FDEC=$$BIN2DEC(F),EDEC=$$BIN2DEC(E)
        S KDEC=$$BIN2DEC($$COMPUTEK(T)),WDEC=$$BIN2DEC(W)
        S DECRESULT=ROTLDEC+FDEC+EDEC+KDEC+WDEC
        F  D  Q:DECRESULT<(2**32)
        . I DECRESULT>(2**32) S DECRESULT=DECRESULT-((2**32))
        S DECRESULT=$$DEC2BIN(DECRESULT)
        I $ZBITLEN(DECRESULT)<32 D
        . N TMPBIN S TMPBIN=$ZBITSTR(32,0)
        . N I,J S J=$ZBITLEN(DECRESULT) F I=32:-1:32-J+1 D
        . . S TMPBIN=$ZBITSET(TMPBIN,I,$ZBITGET(DECRESULT,J)),J=J-1
        . S DECRESULT=TMPBIN
        Q DECRESULT

COMPUTEF(T,X,Y,Z) ; Compute F. Used for f0, f1 ... f79
        I (T>-1)&(T<20) Q $ZBITXOR($ZBITAND(X,Y),$ZBITAND($ZBITNOT(X),Z))
        I (T>19)&(T<40) Q $ZBITXOR(X,$ZBITXOR(Y,Z))
        I (T>39)&(T<60) Q $ZBITXOR($ZBITAND(X,Y),$ZBITXOR($ZBITAND(X,Z),
$ZBITAND(Y,Z)))
        I (T>59)&(T<80) Q $ZBITXOR(X,$ZBITXOR(Y,Z))
        Q -1

COMPUTEK(T) ; Compute K. Constant value to be used for iteration t of the hash
computation.
        I (T>-1)&(T<20) Q $$HEX2BIN("5a827999")
        I (T>19)&(T<40) Q $$HEX2BIN("6ed9eba1")
        I (T>39)&(T<60) Q $$HEX2BIN("8f1bbcdc")
        I (T>59)&(T<80) Q $$HEX2BIN("ca62c1d6")
        Q -1

COMPUTEW(T) ; The T-th W-bit word of the message schedule.
        I (T>-1)&(T<16) Q W(T)
        I (T>15)&(T<80) S W(T)=$$ROTL($ZBITXOR(W(T-3),$ZBITXOR(W(T-8),$ZBITXOR(W(T-
14),W(T-16)))),1)
```

```
        Q W(T)

BINPLUSBIN(B1,B2) ; Binary addition.
        N D1,D2,DSUM S D1=$$BIN2DEC(B1),D2=$$BIN2DEC(B2),DSUM=D1+D2
        F  D  Q:DSUM<(2**32)
        . I DSUM>(2**32) S DSUM=DSUM-(2**32)
        Q $$DEC2BIN(DSUM)

MUNIT ; MUMPS Unit Test
        N CMD,RESP,EXRESP
        ; Test #1
        W !!,"Test #1"
        D UNITTEST("abc","a9993e364706816aba3e25717850c26c9cd0d89d")
        ; Test #2
        W !!,"Test #2"
        D
UNITTEST("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq","84983e441c3bd26e
baae4aa1f95129e5e54670f1")
        ; Test #3
        W !!,"Test #3"
        D
UNITTEST("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq12345678","9ef5c682
d93914e77a5d345abb957436445a6fb6")
        Q

UNITTEST(MESSAGE,EV) ; The message to test and the expected value.
        N AV S AV=$$DIGEST(MESSAGE)
        W !,"Testing SHA1 -> "_MESSAGE
        W !,"Actual Value -> "_AV
        W !,"Expected Value -> "_EV
        W !,"Test Result -> "_$SELECT(AV=EV:"PASS! ;]",AV'=EV:"FAIL! :[")
        Q
```

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."
```

© Licenses.txt