CS 8803: Introduction to Information Security
Final Project: A Security Analysis, Proof of Concept, and Suggested Improvements to KeePassX, an Open Source Password Manager

David Tomaschik
david@systemoverlord.com

I. Introduction

KeePassX is a portable version of KeePass, a popular password management tool. According to the KeePassX homepage, "KeePassX is an application for people with extremly high demands on secure personal data management."[1] The most recent version, 0.4.3, was released on March 7, 2010.[2] This indicates that KeePassX is under reasonably active development.

In June of 2009, a discussion on the cryptography@metzdowd.com mailing list revealed that KeePassX fails to "lock" (prevent from being swapped to disk) certain data into memory.[3] A cursory examination of the source of the most recently released version (0.4.3) reveals that no improvements to locking have been implemented. Further examination of the source also indicated that sensitive data is left in memory when the application exits. This lead to a concern that data from an application claiming to be for "people with extremely high demands on secur[ity]" might leak some information after being closed.

II. Background

Modern operating systems use areas of the disk known as "swap space" to extend the systems "virtual memory," allowing applications that need more memory to have it allocated by writing parts of memory back to disk in what is called a "swap out". When that data is again needed, it can be "swapped in" by reading the data back into the system memory. While this has performance benefits, it has the disadvantage of leaving a copy of memory contents on the hard disk of the computer, often without the knowledge of the user. In the case of programs that store sensitive data (such as a password

1 KeePassX Homepage. http://www.keepassx.org/
2 KeePassX 0.4.3 Released. http://www.keepassx.org/news/2010/03/213
3 P. Metzger, Re: password safes for mac. Cryptography@metzdowd.com. http://www.mail-archive.com/cryptography@metzdowd.com/msg10580.html

manager), this can result in passwords being written to disk in an unencrypted state.

Because of these implications, the POSIX 2001 specification includes a set of functions to prevent regions of memory from being swapped to disk. These functions include `mlock` and `mlockall`. According to the `mlock(2)` manual page, "`mlock()` and `mlockall()` respectively lock part or all of the calling process's virtual address space into RAM, preventing that memory from being paged to the swap area."[4] Programs storing sensitive data should take steps to prevent that data from being swapped to disk, including ensuring any memory pages with sensitive data are locked into RAM. Memory must be locked before any sensitive data is written to it to ensure that it is not possible to swap out that memory in the interim period.

Sensitive data should be overwritten before being returned to the operating system via `delete` or `free`. Data that is not overwritten is may be vulnerable to data remnance attacks and cold-boot attacks.[5] Operating systems generally do not overwrite data before allocating it to a program, meaning that data left behind by an application might be read by another, potentially malicious, application.

III. KeePassX's Design

KeePassX is written in C++ using the Qt Framework from Nokia[6] for user-interface components and many data structures. This choice seems to stem from a desire to easily port the application to a variety of platforms, and has allowed KeePassX to be built for Windows, Mac, and Linux.[7] KeePassX encrypts its database files using a user's choice of either AES (internally referred to by its AES-competition name, Rijndael) or another AES competitor, Twofish. Users may supply a password, key file, or a combination of both to provide key material for encrypting the user's password database.[8]

One notable strength of KeePassX is that the password provided by the user is significantly

---

4 mlock(2) manual page, as included with Ubuntu Linux 10.10.

5 J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM* 52, 5 (May 2009), 91-98. DOI=10.1145/1506409.1506429 http://doi.acm.org/10.1145/1506409.1506429

6 Qt – A Cross-Platform Application and UI Framework. http://qt.nokia.com/

7 KeePassX – Downloads. http://www.keepassx.org/downloads/

8 KeePassX – Features. http://www.keepassx.org/features/

strengthened by performing an SHA-256 hash of the provided password, then encrypting that hash with Rijndael/AES or Twofish through a user-configurable number of rounds, and finally taking an SHA-256 hash of this resultant value. This key strengthening process has the advantage of significantly slowing an attack against the user's password and inhibiting the ability of an attacker to predict any part of the key actually used to encrypt the database file.

One "feature" of KeePassX that disappoints me is the password strength meter provided by the user interface. It shows an improvement of 8 bits per character of the password, but this is clearly overstating the strength of the password. Even a random ASCII password has a maximum entropy of 6.6 bits/character, assuming the use of the full 94 printable ASCII characters.[9] NIST has published various formulas for estimating the entropy of a password[10] based on work reaching as far back as Shannon's work in 1948-1951.[11]

IV. Source Analysis & Vulnerabilities

I began by reviewing the source code to the version of KeePassX that is available as a part of Ubuntu Linux 10.10/Maverick Meerkat – version 0.4.3-1. I retrieved the source by executing "apt-get source keepassx", which extracted the source into a working directory.

Because KeePassX is built for Windows, Apple's Mac OS X and a variety of Linux platforms, there are some sections of code which use `#define`s for portability reasons. Because of my familiarity with the platform, I focused my analysis and improvements on the Linux build.

One platform-related difference is in memory locking – on Windows, `VirtualLock` is used, and on POSIX-compliant systems, `mlock` is used. Due to this, the original authors wrote a function called lockPage as a wrapper to either locking function. Searching through the source code reveals that only a variable labeled as "`sessionkey`" is locked into RAM. This `sessionkey` is a randomly-generated value used as a key to encrypt (with an RC4 stream cipher) the database master key and

9 NIST publication 800-63, Appendix A, p 63.
10 *Ibid.*
11 C. E. Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal, v. 27, pp. 379-423, 623-656, July, October 1948, http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html

individual passwords while the program is running. This encryption is handled by a `SecString` class that provides lock and unlock operations to transform between RC4 encrypted and plaintext. This encrypted data is not locked into RAM, however the plaintext is overwritten when it is locked back. However, if swapping were to occur while the `SecString` was "unlocked", plaintext passwords could be written to disk.

In several places, various copies are made of the database master key, including into plain `QByteArray`s which offer no protection against swapping to disk (neither locking nor encryption) and are not overwritten before being discarded. This is particularly obvious within `src/Kdb3Database.cpp` where the functions for setting up the final key do not overwrite their data when copying among different character sets (used for legacy database versions). These functions include `Kdb3Database::setPasswordKey`, `Kdb3Database::setFileKey`, and `Kdb3Database::setCompositeKey`.

Just as bad is `Kdb3Database::loadReal`, where the strengthened version of the master key (called FinalKey) is stored in a plain 32-byte `quint8` array. This is not locked against swapping, and is not overwritten before the function exits (the array exists on the stack). This leads to the key that is actually being used to encrypt/decrypt the database being stored in unprotected memory and potentially being left behind after program execution.

A major shortcoming in the security of KeePassX is in fact within the Qt libraries used by the application. Passwords are edited and displayed with a control called "`QLineEdit`". Because passwords are passed to the control, it is hard to discern whether or not the password will be overwritten in memory at the end of use. The copy of the password inside KeePassX is overwritten, and if only a pointer is used by `QLineEdit`, it is likely comparatively safe, however, without an in-depth analysis of the Qt libraries, it's impossible to ascertain if the value is copied at any point.

Overall, the authors of KeePassX have done a fairly good job of writing a well-planned password storage application. I found no indication of any backdoors or flaws in the file handling that would lead to KeePass databases being compromised. While the risk of having sensitive data swapped or left behind is small, research such as the Cold Boot Attacks by Halderman, et al., shows that this is a new frontier for attacking cryptographic products. While KeePassX appears strong, it can be improved to resist the next generation of attacks.

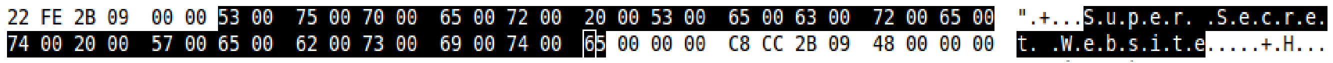
V. Proof of Concept

Because of the difficulty in attacking a physical system for a cold-boot attack, I utilized a virtual environment for my proof of concept. Using QEMU[12] (an Open Source Virtualization tool), I created a fresh Ubuntu 10.10 install with 512MB of RAM, installed the distributed version of KeePassX. I then built a small test KeePass Database with 2 accounts, including one labeled "Secret Site" with username "david" and password "secretpassword." QEMU offers a command pmemsave that allows you to dump memory to a file on disk.[13] I used "`pmemsave 0 0x20000000 FILE.img`" to dump memory images.

At first, I attempted to use grep to find relevant portions of the memory region, but was unsuccessful. I realized that the data handled by the application was stored as `QString`s, which are UTF-16 encoded strings, similar to Java. I wrote a small python script to convert ASCII to UTF-16 strings and print them in hexadecimal[14]. Opening the memory image file in `hexedit` allowed me to search for these byte strings easily.

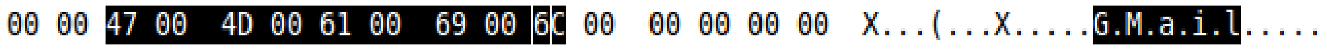
Even after the program has exited, some information can be revealed from within the memory dump. For example, it can be determined what sites a user has passwords for as the metadata about each password is stored in the clear.

12 About QEMU. http://wiki.qemu.org/Main_Page
13 C. Neilson, "Memory Analysis Project Setup," University of Denver. http://web.cs.du.edu/~mitchell/forensics/projects/memory/Setup.pdf
14 See Appendix IV.



```
22 FE 2B 09  00 00 53 00  75 00 70 00  65 00 72 00  20 00 53 00  65 00 63 00  72 00 65 00  ".+...S.u.p.e.r. .S.e.c.r.e.
74 00 20 00  57 00 65 00  62 00 73 00  69 00 74 00  65 00 00 00  C8 CC 2B 09  48 00 00 00  t. .W.e.b.s.i.t.e.....+.H...
```

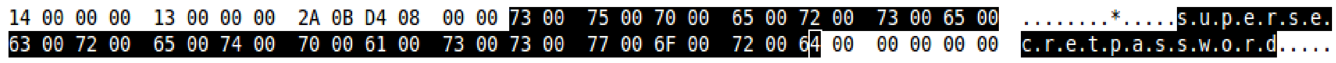
Illustration 1: Website information is visible



```
00 00 47 00  4D 00 61 00  69 00 6C 00  00 00 00 00  X...(...X.....G.M.a.i.l.....
```

Illustration 2: More visible website information

A memory dump with a password edit window open, but the password hidden behind a series of asterisks, even reveals the secret password, though this is not terribly surprising. The source shows that the field is loaded as soon as the edit window is opened.



```
14 00 00 00  13 00 00 00  2A 0B D4 08  00 00 73 00  75 00 70 00  65 00 72 00  73 00 65 00  ........*.....s.u.p.e.r.s.e.
63 00 72 00  65 00 74 00  70 00 61 00  73 00 73 00  77 00 6F 00  72 00 64 00  00 00 00 00  c.r.e.t.p.a.s.s.w.o.r.d.....
```

Illustration 3: Passwords visible when edit window open, but password hidden

I must acknowledge that nothing in my proof of concept is particularly glaring as far as a security risk goes. The authors of KeePassX have done a good job of ensuring that particularly sensitive data (keys & passwords) spends as little time as possible unencrypted in RAM.

VI. Improvements Made

I have written 3 separate patches for the Linux port of KeePassX, specifically using the Ubuntu and Debian patching tools. The full text of the patches is included in the appendices. It is my intent to submit these patches for inclusion both in the Debian and Ubuntu builds of KeePassX as well as upstream after cleaning them up somewhat. These patches specifically target two security concerns and one user interface concern.

The first patch (`01-mlockall.patch`) calls `mlockall` and requests that all currently used memory (`MCL_CURRENT`) and future (`MCL_FUTURE`) memory be locked into RAM. This has the benefit of protecting the entire process space from being written to disk by a swap-out. Unfortunately, because KeePassX is a GUI program, this exceeds the default limit of 64 kilobytes for locked memory. On a 64-bit system, opening a 1.2 MB KeePass Database results in a process with a resident size of approximately 4 MB, so this is not just an opportunity to squeeze a little bit more out via optimization.

Fortunately, since Linux 2.6.9 introduced capabilities, we can provide the process with the `CAP_IPC_LOCK` capability, which allows it to lock an unlimited amount of its memory. `CAP_IPC_LOCK` is granted in a Debian `postinst` script with the command: `/sbin/setcap cap_ipc_lock+ep /usr/bin/keepassx`. Given a sufficiently large KeePass database on a system with a very limited amount of memory, it might be possible to create a denial of service situation. However, since KeePassX is not a network-connected service, is used by a user from a GUI, and is (or at least should be) almost exclusively used by the owner of a system on a single-user system, this risk is probably very minimal. It is, however, a trade-off for the benefits of having sensitive data not written to disk.

My second patch (`02-overwrite-before-free.patch`) makes an effort to ensure memory space that contains sensitive data is overwritten before it is returned to the system. This is most important for objects allocated on the heap, but I also made an effort to do it for particularly sensitive data structures on the stack (for example, the strengthened key that is used to encrypt the entire database). Specifically, the following sensitive allocations are addressed: the `FinalKey` and the raw file buffer in `Kdb3Database::loadReal`, the `FinalKey` and buffers for storage of unencrypted and encrypted data in `Kdb3Database::save`, the key schedule for the implementation of RC4 used to encrypt session data in RAM, and the encrypted data in the `SecString` class (overwritten on destruction).

The third patch is less about the security of KeePassX and more about the false sense of security that the current "strength meter" gives to the end user. By giving each character 8 bits of "strength" (entropy), they cause users to vastly overestimate the strength of their password. User-generated passwords without dictionary or composition rules are valued at 1-2 bits of entropy per character.[15] My patch implements the rules from NIST publication 800-63, but continues to display a full strength bar for 128 bits of entropy. Under the NIST guidelines, it would require a 112 character long password to

15 NIST publication 800-63, Appendix A, p 63.

reach 128 bits of entropy, which is unlikely to be commonly used by users, but is a substantial difference from the 16 character password that the original KeePassX suggests is sufficient.

VII. Suggested Improvements

A large improvement could be made by extending the `QLineEdit` class to provide a secure version guaranteed not to copy the contents you provide it so that a developer could pass it an address that is locked in RAM without needing to `mlock` the entire process space. There is no such guarantee with the current `QLineEdit`, even if it is likely the case.

Additionally, there is no reason to store the unencrypted version of the user's password in the text field until the user requests to see the password. If the user requests that it be displayed, or asks to have the password copied to the clipboard, the application could decrypt the RC4 encrypted password at that time. This would result in many operations never decrypting the password, further reducing the risk of the password being compromised.

A thorough analysis of KeePassX could result in modifications to reduce the need to lock the entire process space, however this would likely require writing an allocator to create objects in the pages locked into RAM to avoid exceeding the 16-page (64-kbyte) limit. This would be a major effort to re-engineer the manner in which sensitive data is handled in KeePassX.

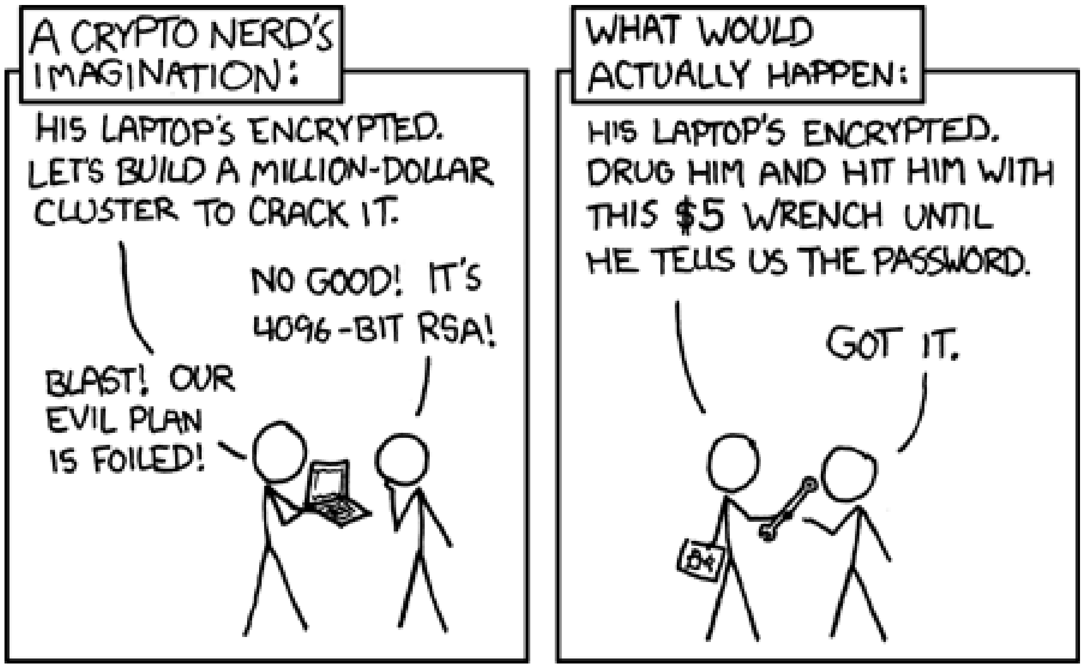
The strength meter could be further improved by examining the character set being used by the user to adjust the relative per-character entropy. While running a full dictionary check is probably impractical, it's clear that “!22pSzzk” is stronger than “agklnnup” by virtue of the larger key space.

VIII. Conclusion

With the ever-growing set of online services, users are now responsible for managing more accounts and more passwords than ever before. For users to be able to use passwords with more than the most basic amount of security, it seems inescapable that they will need some manner to record them. Password management programs like KeePassX fill that void by providing a secured storage mechanism to allow users to avoid a sticky note on the side of their monitor.

My changes to KeePassX are documented in the Appendices as well as available from my launchpad PPA at https://launchpad.net/~matir/+archive/ppa. I will be submitting my changes upstream and making them available on my personal website after feedback from this course.

KeePassX is a robust and feature-rich program that has thwarted my efforts to discover any major weaknesses. When used on a single-user system, particularly with encrypted swap, it is highly unlikely that any sensitive data will be leaked. The most likely way for a determined adversary to gain access to a user's accounts will be through service weaknesses, predictable password reset options, or the ever-popular $5 wrench[16]. It's highly unlikely that KeePassX will be the source of a compromise, and with continued development, that risk will be mitigated even further.


A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

16 XKCD by Randall Munroe: http://xkcd.com/538/ (Licensed under CC-BY-NC 2.5)

Appendix I: 01-mlockall.patch

```
## Description: Lock all memory
## Origin/Author: David Tomaschik <david@tuxteam.com>
--- a/src/lib/SecString.cpp
+++ b/src/lib/SecString.cpp
@@ -92,15 +92,12 @@

 void SecString::generateSessionKey(){
      sessionkey = new quint8[32];
-     if (!lockPage(sessionkey, 32))
-           qDebug("Failed to lock session key page");
      randomize(sessionkey, 32);
      RC4.setKey(sessionkey, 32);
 }

 void SecString::deleteSessionKey() {
      overwrite(sessionkey, 32);
-     unlockPage(sessionkey, 32);
      delete[] sessionkey;
 }

--- a/src/lib/tools.cpp
+++ b/src/lib/tools.cpp
@@ -219,24 +219,12 @@
      return true;
 }

-bool lockPage(void* addr, int len){
-#if defined(Q_WS_X11) || defined(Q_WS_MAC)
-     return (mlock(addr, len)==0);
-#elif defined(Q_WS_WIN)
-     return VirtualLock(addr, len);
-#else
-     return false;
-#endif
+bool lockAll(){
+    return (mlockall(MCL_CURRENT|MCL_FUTURE)==0);
 }

-bool unlockPage(void* addr, int len){
-#if defined(Q_WS_X11) || defined(Q_WS_MAC)
-     return (munlock(addr, len)==0);
-#elif defined(Q_WS_WIN)
-     return VirtualUnlock(addr, len);
-#else
-     return false;
-#endif
+bool unlockAll(){
+    return (munlockall()==0);
 }

 bool syncFile(QFile* file) {
--- a/src/lib/tools.h
+++ b/src/lib/tools.h
@@ -44,8 +44,8 @@
 QString makePathRelative(const QString& Abs,const QString& Cur);
 QString getImageFile(const QString& name);
 bool createKeyFile(const QString& filename,QString* err, int length=32, bool
Hex=true);
-bool lockPage(void* addr, int len);
-bool unlockPage(void* addr, int len);
+bool lockAll();
+bool unlockAll();
 bool syncFile(QFile* file);
 void installTranslator();
 bool isTranslationActive();
--- a/src/main.cpp
+++ b/src/main.cpp
@@ -49,6 +49,9 @@

 int main(int argc, char **argv)
 {
+    // Lock all RAM as early as possible
+    lockAll();
+
      setlocale(LC_CTYPE, "");

 #if defined(Q_WS_X11) && defined(AUTOTYPE)
```

Appendix II: 02-overwrite-before-free.patch

```
## Description: Try to overwrite sensitive memory before giving it up
## Origin/Author: David Tomaschik <david@tuxteam.com>
--- a/src/Kdb3Database.cpp
+++ b/src/Kdb3Database.cpp
@@ -512,8 +512,10 @@
 }

 #define LOAD_RETURN_CLEANUP \
+    memset(FinalKey,0,sizeof(FinalKey)); \
      delete File; \
      File = NULL; \
+    memset(buffer,0,sizeof(buffer)); \
      delete[] buffer; \
      return false;

@@ -548,7 +550,8 @@
      quint8 FinalRandomSeed[16];
      quint8 ContentsHash[32];
      quint8 EncryptionIV[16];
-
+     quint8 FinalKey[32];
+
      total_size=File->size();
      char* buffer = new char[total_size];
      File->read(buffer,total_size);
@@ -593,8 +596,6 @@
      MasterKey.unlock();


KeyTransform::transform(*RawMasterKey,*MasterKey,TransfRandomSeed,KeyTransfRounds)
;

-     quint8 FinalKey[32];
-
      SHA256 sha;
      sha.update(FinalRandomSeed,16);
      sha.update(*MasterKey,32);
@@ -632,6 +633,7 @@

      if(memcmp(ContentsHash, FinalKey, 32) != 0){
            if(PotentialEncodingIssueLatin1){
+             memset(buffer,0,sizeof(buffer));
                  delete[] buffer;
                  delete File;
                  File = NULL;
@@ -642,6 +644,7 @@
                  return loadReal(filename, readOnly, true); // second try
            }
            if(PotentialEncodingIssueUTF8){
+             memset(buffer,0,sizeof(buffer));
                  delete[] buffer;
                  delete File;
                  File = NULL;
@@ -740,7 +743,8 @@
            LOAD_RETURN_CLEANUP
      }

-     delete [] buffer;
+    memset(buffer,0,sizeof(buffer));
+     delete[] buffer;

      hasV4IconMetaStream = false;
      for(int i=0;i<Entries.size();i++){
@@ -1483,7 +1487,9 @@
            CTwofish twofish;
            if(twofish.init(FinalKey, 32, EncryptionIV) == false){
                  UNEXP_ERROR
-                 delete [] buffer;
+            memset(buffer,0,sizeof(buffer));
+            memset(FinalKey,0,sizeof(FinalKey));
+                 delete[] buffer;
                  return false;
            }
            EncryptedPartSize = (unsigned
long)twofish.padEncrypt((quint8*)buffer+DB_HEADER_SIZE,
@@ -1491,7 +1497,9 @@
      }
      if((EncryptedPartSize > (0xFFFFFFE - 202)) || (!EncryptedPartSize &&
Groups.size())){
            UNEXP_ERROR
-           delete [] buffer;
+        memset(buffer,0,sizeof(buffer));
+        memset(FinalKey,0,sizeof(FinalKey));
+           delete[] buffer;
            return false;
      }

@@ -1499,11 +1507,15 @@

      if (!saveFileTransactional(buffer, size)) {
            error=decodeFileError(File->error());
-           delete [] buffer;
+        memset(buffer,0,sizeof(buffer));
+        memset(FinalKey,0,sizeof(FinalKey));
+           delete[] buffer;
            return false;
      }
-
-     delete [] buffer;
+
+    memset(buffer,0,sizeof(buffer));
+    memset(FinalKey,0,sizeof(FinalKey));
+     delete[] buffer;
      //if(SearchGroupID!=-1)Groups.push_back(SearchGroup);
      return true;
 }
--- a/src/crypto/arcfour.cpp
+++ b/src/crypto/arcfour.cpp
@@ -28,7 +28,7 @@
      quint32 w;

      for(w = 0; w < 256; ++w)
-           S[w] = static_cast<quint8>(w); // Fill linearly
```

```
+           S[w] = static_cast<quint8>(w); // Identity permutation

      const quint8 btBufDep = static_cast<quint8>((length & 0xFF) << 1);

@@ -55,4 +55,11 @@
            t = S[i] + S[j]; // Generate random byte
            dst[w] = src[w] ^ S[t]; // XOR with PT
      }
+
+    // Overwrite temporary value
+    t = 0;
+
+    // Overwrite key schedule
+    for(w = 0; w < 256 ; ++w)
+        S[w] = 0;
 }
--- a/src/lib/SecString.cpp
+++ b/src/lib/SecString.cpp
@@ -34,6 +34,7 @@
 }

 SecString::~SecString(){
+    overwrite((unsigned char *)crypt.data(), crypt.length());
      lock();
 }

@@ -108,10 +109,8 @@
 }

 SecData::~SecData(){
-     if (!locked){
-           for (int i=0; i<length; i++)
-                 data[i] = 0;
-     }
+     for (int i=0; i<length; i++)
+           data[i] = 0;
      delete[] data;
 }
```

Appendix III – 03-entropy.patch

```
## Description: Use the NIST 800-63 estimation for password entropy
## Origin/Author: David Tomaschik <david@tuxteam.com>
Index: keepassx-0.4.3/src/dialogs/EditEntryDlg.cpp
===================================================================
--- keepassx-0.4.3.orig/src/dialogs/EditEntryDlg.cpp 2009-08-31 12:44:21.000000000
-0400
+++ keepassx-0.4.3/src/dialogs/EditEntryDlg.cpp2010-12-02 21:08:41.140217296 -0500
@@ -101,11 +101,7 @@

     // MX-COMMENT: This call is not needed. Both Passwords fields will always
have the same value
     OnPasswordwLostFocus();
-      int bits=(Password.length()*8);
-      Label_Bits->setText(tr("%1 Bit").arg(QString::number(bits)));
-      if(bits>128)
-            bits=128;
-      Progress_Quali->setValue(100*bits/128);
+    updateEntropy();
      Edit_Attachment->setText(entry->binaryDesc());
      Edit_Comment->setPlainText(entry->comment());
      InitGroupComboBox();
@@ -259,10 +255,7 @@
 void CEditEntryDlg::OnPasswordTextChanged()
 {
      Edit_Password_w->setText(QString());
-      int bits=(Edit_Password->text().length()*8);
-      Label_Bits->setText(QString::number(bits)+" Bit");
-      if(bits>128)bits=128;
-      Progress_Quali->setValue(100*bits/128);
+    updateEntropy();
 }

 void CEditEntryDlg::OnPasswordwTextChanged()
@@ -448,3 +441,21 @@
      }
 #endif
 }
+
+void CEditEntryDlg::updateEntropy(){
+    // This is probably not the best place to do this,
+    // but we have access to the UI elements here.
+    int len=Edit_Password->text().size();
+    int entropy=0;
+
+    // Calculations based on NIST 800-63
+    if(len<=8)
+        entropy=len*2+2; // First character is 4 bits, all others are 2.
+    else if(len <= 20)
+        entropy=18+1.5*(len-8); // 18 bits for first 8 chars, then 1.5 bpc
+    else
+        entropy=16+len; // 36 bits for first 20 chars, then 1bpc
+
+    Label_Bits->setText(tr("%1 Bit").arg(QString::number(entropy)));
+    Progress_Quali->setValue( (entropy >= 128)?(100):(100*entropy/128) );
+}
```

```
Index: keepassx-0.4.3/src/dialogs/EditEntryDlg.h
===================================================================
--- keepassx-0.4.3.orig/src/dialogs/EditEntryDlg.h   2009-03-18 08:09:20.000000000
-0400
+++ keepassx-0.4.3/src/dialogs/EditEntryDlg.h  2010-12-02 21:08:41.140217296 -0500
@@ -63,6 +63,7 @@
      private:
            virtual void paintEvent(QPaintEvent*);
            virtual void resizeEvent(QResizeEvent *);
+        virtual void updateEntropy();

            int IconIndex;
            bool pNewEntry;
```

Appendix IV – Python utf16 Converter

```
#!/usr/bin/python
import sys

print ''.join(["%02X"%ord(x) for x in sys.argv[1].encode("UTF-16")]).strip()
```