UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS no. 368

Forensic analysis of the Mac OS X keychain

Robert Vežnaver

Zagreb, June 2012

I would like to thank:
Jelena - for the support
Šime - for the laughs
Hrvoje - for the Mac
Marko - for the Tekken

Contents

1. Introduction

The keychain is a password management system for Apple's Mac OS X. It is used to store passwords, private keys, certificates, and secure notes (White, 2012).

Mac OS X's keychain works analogous to a physical keychain. Its main purpose is to keep multiple keys in one place so the user does not have to worry about multiple objects, but only one. The keychain itself is not important, but it gains utility by keeping all the keys together (Kissell, 2010). However, unlike a physical keychain, if someone steals a user's digital keychain, they can not use it without knowing the password for accessing the contents of the keychain. Also unlike a physical keychain, a user is not required to go through all the keys to find the right one. The operating system and application accessing the keychain automatically locate the right key at any given time.

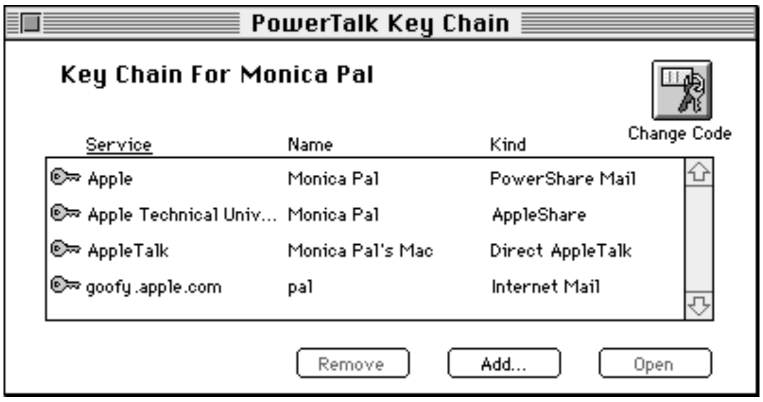
Each user on Mac OS X has his or hers own keychain by default. It automatically created alongside the user account and its password matches the user's login password. However, the keychain password is independent from the account password sharing neither location nor format. Furthermore, every Mac OS X installation has its own system keychain which all users may access.

The purpose of this thesis is to provide an analysis of the current state of the Mac OS X keychain, its usage, and file format. Since the keychain is supposed to be both secure and user friendly, the analysis will start from the top (the user's perspective), going through topics such as managing keychains through the command line (the system administrator's perspective) and using the keychain API (the developer's perspective), down to the file format, cryptographic functions and standards used to create a keychain.

2. History

To truly grasp the fundamentals of the keychain's design and structure it is important to know the historical context and environment of the time it was written in. System engineering, both as an art and science is constrained by available resources and technical limitation, which were considerably different back in 1993 when the keychain was first introduced.

2.1. From PowerTalk to iOS


PowerTalk Key Chain

Key Chain For Monica Pal

Service	Name	Kind
Apple	Monica Pal	PowerShare Mail
Apple Technical Univ...	Monica Pal	AppleShare
AppleTalk	Monica Pal's Mac	Direct AppleTalk
goofy.apple.com	pal	Internet Mail

Figure 2.1: PowerTalk Key Chain

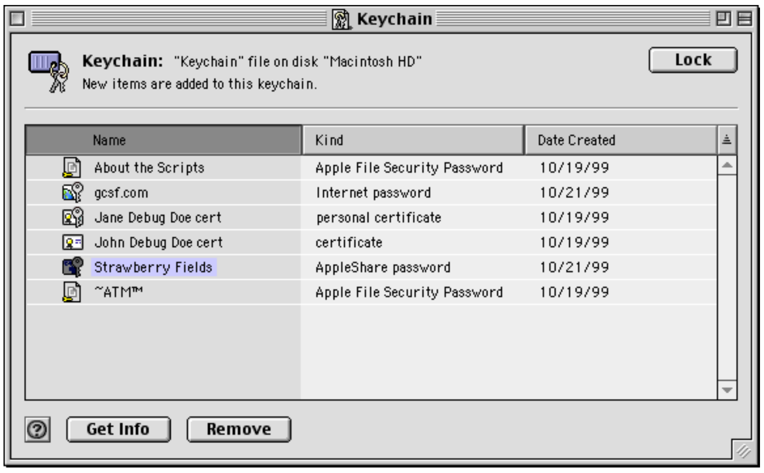
The concept of a digital keychain was first introduced in Apple's PowerTalk. PowerTalk was the later name of the Apple Open Collaboration Environment (AOCE) that was released in 1993 within the System 7 Pro bundle. AOCE was created to solve a series of issues related to the electronic mail and delivery systems of that time (Mori, 2009).

AOCE was an entirely proprietary system based upon AppleTalk. Rather than just delivering text messages, PowerTalk was conceived as a client side system for choosing, sending, and interacting with various things, including files, mail, fax machines, and printers. PowerTalk mirrored the rest of the elegant and richly engineered point-and-click Mac desktop environment, and promised to be the next great Mac advantage (Dilger, 2006).

Since there were some serious security concerns regarding sending text messages over phone lines, all communications could be secured using RSA encryption and digital signing, even on the local network. AOCE mutual authentication services and the Apple Secure Data Stream Protocol (ASDSP) enabled people and processes to verify the identity of the person on the other side of the communications session and to encrypt the data that is exchanged (Pal, 1993).

To solve the problem of remembering various usernames and passwords for different service PowerTalk could access, Apple introduced the PowerTalk Key Chain. Instead of remembering every single username and password for every system they used, the Key Chain enabled users to save confidential credentials in a single secure place and they had to remember only a single password for accessing the Key Chain. AOCE would retrieve the credentials for a particular service on demand. For users in large organizations with hundreds of servers where passwords change frequently the Key Chain was a necessary fact of life (Engst, 1995).

Although PowerTalk brought forth some interesting concepts and ideas the product itself failed. The system requirements were too high for the Macs of that era, it was impossible to use it alongside QuickDRAW GX (another Apple technology at the time), and the user interface was not thought out well. As the market was moving towards SMTP-based internet mail, by 1996 Apple had given up on PowerTalk and the AOCE.



Keychain: "Keychain" file on disk "Macintosh HD"
New items are added to this keychain.

Name	Kind	Date Created
About the Scripts	Apple File Security Password	10/19/99
gcsf.com	Internet password	10/21/99
Jane Debug Doe cert	personal certificate	10/19/99
John Debug Doe cert	certificate	10/19/99
Strawberry Fields	AppleShare password	10/21/99
~ATM™	Apple File Security Password	10/19/99

Figure 2.2: Mac OS 9 keychain

It was not until 1999 and the release of Mac OS 8.6 that the keychain resurfaced on the Mac. Aside from being tightly integrated into the whole operating system, the new keychain could store textual data which was then encrypted in the keychain file. In addition, the user could lock and unlock a keychain. If the keychain was locked the user was prompted for a password to unlock it. Once unlocked, applications specified in the access control list for a credential could use the credential. After a specified idle timeout interval the keychain would lock itself. This behaviour is the same throughout the whole Mac OS X family as well.

Not only is the keychain so interwoven in Mac OS X's Security Framework, it is also part of another operating system made by Apple - iOS. In iOS 4, Apple introduced the Data Protection API. The Data Protection API was designed to make it as simple as possible for application developers to sufficiently protect sensitive user data stored in files and keychain items in case the user's device is lost. All the developer has to do is indicate which files or items in the keychain may contain sensitive data and when that data must be accessible (Miller et al., 2012).

However, unlike OS X's keychain, the iOS keychain is implemented as a SQLite database stored on the file system, while its security is provided by a different key hierarchy that runs parallel to the key hierarchy used to protect files. There is only one database; the securityd daemon determines which keychain items each process or app can access (App, 2012b). In addition, every iOS device has an AES and SHA-1 engine implemented in hardware to reduce cryptographic operation overhead, so while the concept is still the same, the iOS keychain is implemented quite differently.

2.2. Apple's Open Source strategy

Darwin is an open source POSIX-compliant operating system built by Apple and it forms the core set of components upon which Mac OS X and iOS are based. Darwin's kernel is called XNU, a hybrid kernel which combines the Mach 3 microkernel, elements of BSD, and a device driver API named I/O Kit. Darwin's heritage draws from NeXT's NeXTSTEP operating system and its development began out of need to replace the old and venerable "Classic" Mac OS. It is the direct result of Apple acquiring NeXT along with all of its products line in 1999.

Apple's Open Source efforts began in 2000 when Darwin 1.0 was released. When Darwin 1.0 was first announced, it was released in a bootable binary form of an ASR image (only available for PowerPC). The source was available from a CVS repository to which external developers could in time gain commit access (Braun, 2006). However, all changes made by external developers went on a branch which was merged with the main branch only after an internal Apple change control process. Furthermore, building the source was very difficult because Apple had a proprietary build system which was too large and complex for an average person to use. A system was devised with which some Darwin projects could be built but only a handful of people have ever been able to build a full Darwin release.

Even with all these setbacks it was still possible to contribute to Darwin. However, as the developer base grew so did the pressure on Apple's internal change control process of approving commits. As time went on, and Mac OS X was about to get released, Apple wanted to keep some things secret. Step by step projects were removed from the CVS repository, and after a while the repository was scrapped altogether. Instead, Apple decided to place compressed archives of projects on a web page. This is still the case today and the open source portion of the code of every Mac OS X release may be found at *opensource.apple.com*.

Despite Apple's decreasing involvement with Darwin's Open Source, significant progress was made in one area: building individual projects. Sometime in 2003 the DarwinBuild project was introduced enabling external developers to build Darwin projects as close as Apple did. This was all thanks to the OpenDarwin project in which the community tried to build a standalone Darwin operating system. Since the open source part of Mac OS X was more and more intertwined with closed source parts, and those open source parts which were eventually released were never meant to be built without the close source parts the task of building an OpenDarwin release required more and more heavy patching and the whole project was eventually abandoned in 2006.

The PureDarwin project launched in 2007 to continue where OpenDarwin left off, but the community involvement with the project is minimal. Some technologies Apple developed are released entirely as Open Source like Bonjour (for zeroconf networking), WebKit (web browser engine), and even low level technologies such as Grand Central Dispatch which has been ported to FreeBSD, Linux, and Solaris. However, the difficulty of building Mac OS X parts above the kernel level has only increased throughout the years and releases. One such part (or project) is the security subsystem of Mac OS X used to reverse engineer and gain valuable insight into the keychain file format.

3. Overview of the Mac OS X Keychain

Although the keychain's primary role is to store sensitive data securely, it would be rash and unjust to analyse the keychain from only one perspective. First of all, the keychain was and is built primarily with the average end user in mind. The kind of user who doesn't know what constitutes a "good" or "bad" password, let alone anything about encryption algorithms and keys. Even top-notch cryptographic algorithms won't help a system to be more secure if the user provides an empty string as a password (which is quite a common occurrence when speaking about the average user).

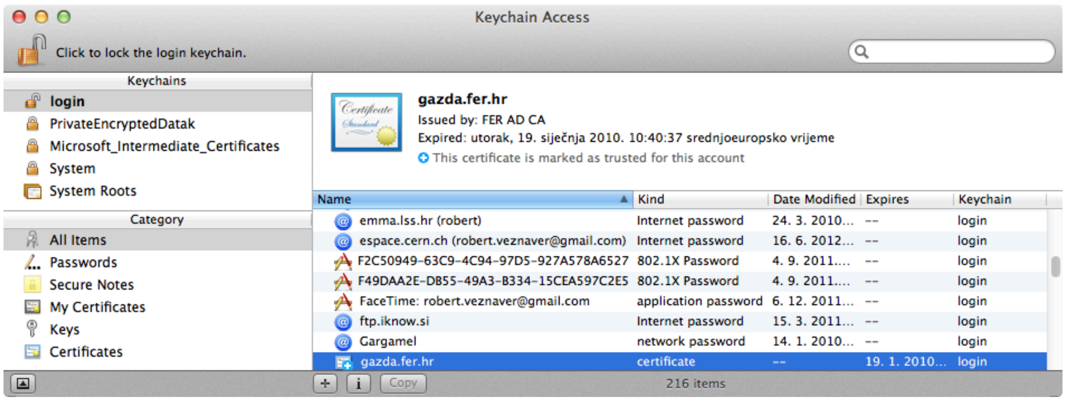
Secondly, the keychain is built to be portable so users may user it over the network or even put it on a USB stick and carry it around at all times (App, 2010). This allows users to have their information with them wherever they ago. It also allows system administrators to copy them from one Mac to another in a homogenous network environment and have their credentials propagated over network directories. The keychain may even be repaired to a certain extent if damaged by a disk error or such.

Lastly, the keychain services API provides a handful of high-level functions that handle all of the keychain operations most applications will ever need to perform (App, 2012c). By making a single call to this API, an application can store login information on a keychain where the application can retrieve the information—also with a single call—when needed. This allows programmers to worry less about the security aspects and to focus more on the core structure of the application.

The following chapters provides several perspectives on the keychain, going from what the average users experiences, through what a system administrator should know, to what a developer should do to use and access data inside the keychain. Section by section the keychain's security layers are analysed one at a time and best security practices are given at the end of each section.

3.1. The end user's perspective

Albert Einstein once said that any darn fool can make something complex, it takes a genius to make something simple. The Mac OS X keychain is built in such a way that it makes its complexity invisible to the end user. However, since artificial intelligence doesn't exist yet, the keychain needs additional user input from time to time, albeit very minimal. To be more precise, most Mac users don't even know how to access or what exactly a keychain is, but every one of them had used it at least once (Kissell, 2010).



gazda.fer.hr
Issued by: FER AD CA
Expired: utorak, 19. siječnja 2010. 10:40:37 srednjoeuropsko vrijeme
This certificate is marked as trusted for this account

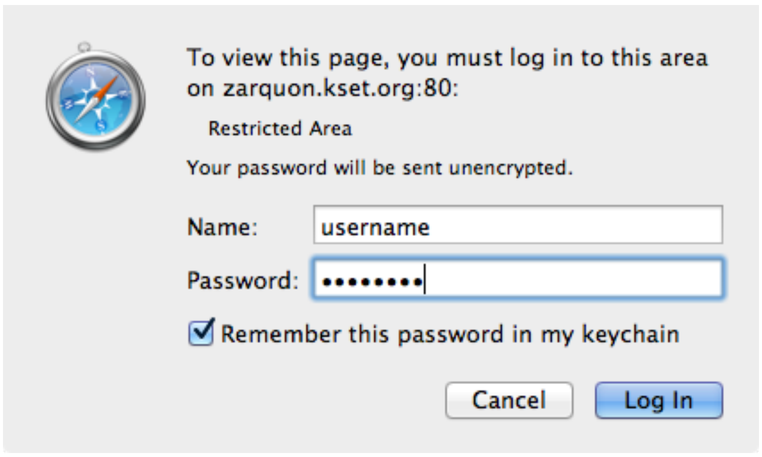
Name	Kind	Date Modified	Expires	Keychain
emma.lss.hr (robert)	Internet password	24. 3. 2010...	--	login
espace.cern.ch (robert.veznaver@gmail.com)	Internet password	16. 6. 2012...	--	login
F2C50949-63C9-4C94-97D5-927A578A6527	802.1X Password	4. 9. 2011....	--	login
F49DAA2E-DB55-49A3-B334-15CEA597C2E5	802.1X Password	4. 9. 2011....	--	login
FaceTime: robert.veznaver@gmail.com	application password	6. 12. 2011...	--	login
ftp.iknow.si	Internet password	15. 3. 2011...	--	login
Gargamel	network password	14. 1. 2010...	--	login
gazda.fer.hr	certificate	--	19. 1. 2010...	login

Figure 3.1: Mac OS X Lion login keychain

By default, every user on Mac OS X has a keychain called login. This keychain is automatically created alongside the user account and its password matches the user's login password. This is the keychain where most, if not all of the user's confidential data for accessing various resources is stored. In addition, each Mac has a keychain called System that holds passwords needed by the operating system even when no user is logged in. The System keychain may be modified only by administrators.

Whenever a user tries to access a resource, such as a password-protected web site, the authentication dialog box contains a check box that says Remember this password in my keychain (or something similar). If the user checks the check box, the credentials typed in are store in the login keychain, and the next time the user attempts to access the same resource, Mac OS X supplies the password automatically, with no user interaction required.

However, this automatic entry of passwords happens only when the keychain is unlocked. By default, the login keychain is unlocked automatically when the user logs in. If the keychain happens to be locked at the time an application or service wants to access something inside it, an alert appears on-screen asking the user to type in his keychain password to unlock it. If the user unlocks the keychain, the


To view this page, you must log in to this area on zarquon.kset.org:80:

Restricted Area

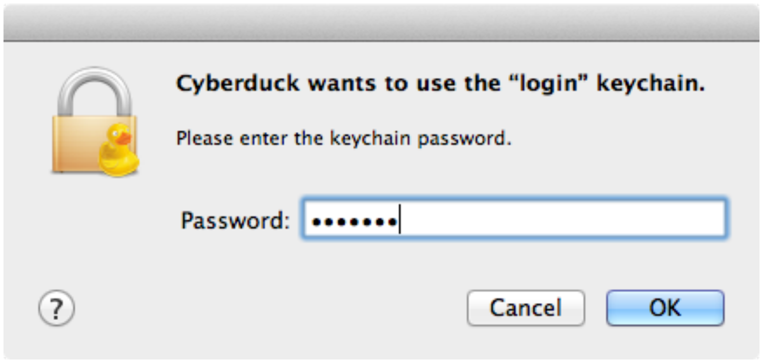
Your password will be sent unencrypted.

Name: username

Password: ••••••••

☑ Remember this password in my keychain

Figure 3.2: Safari authentication prompt


Cyberduck wants to use the "login" keychain.

Please enter the keychain password.

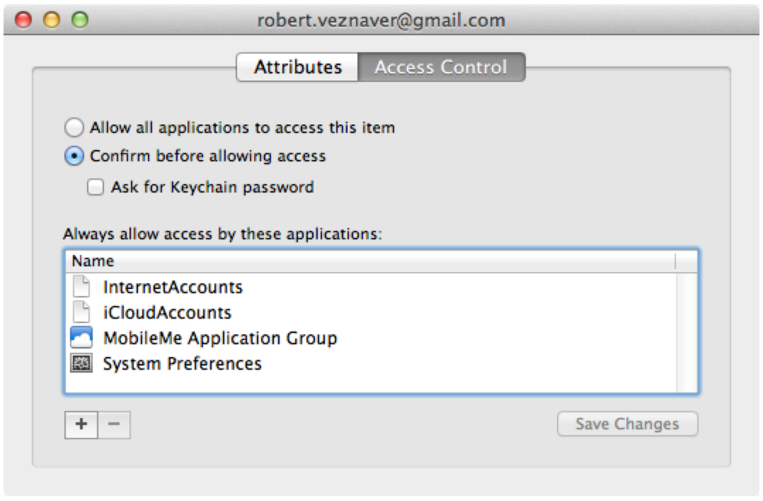
Password: •••••••

Figure 3.3: Cyberduck asking for acces to the login keychain

password is delivered and the keychain remains unlocked until the user logs out, restarts or shuts down the machine, or manually locks the keychain again.

Although the locking mechanism seems simple enough, it is only the top layer of the keychain security model. Each password or other item in the keychain has its own security settings. When a user stores a password (or any other resource) in the keychain and grants access to a particular application, that application is added to the access control list for that particular password. Applications in the access control list of an item are automatically allowed to access that item (provided that the keychain is unlocked).

Applications which require access to a certain item, but are not in that item's access control list will ask the user to grant them access (this is the default behaviour). The prompt may be additionally secured by asking for the keychain password as well. This option, as well as the option to allow all applications access to an item (which is highly insecure and not recommended) may be configured directly in the Keychain Access application on Mac OS X.

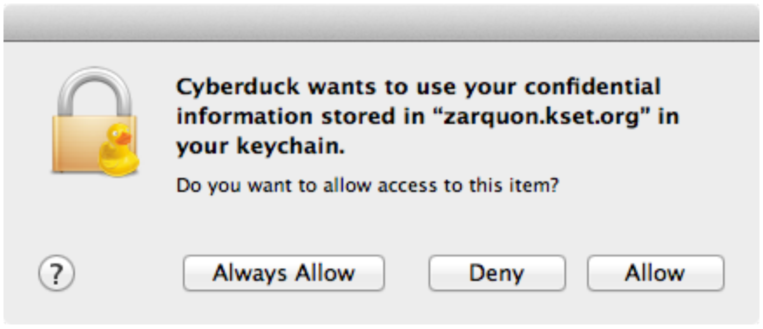

robert.veznaver@gmail.com

Allow all applications to access this item
[✓ Confirm before allowing access]
Ask for Keychain password

Always allow access by these applications:

Name
InternetAccounts
iCloudAccounts
MobileMe Application Group
System Preferences

Figure 3.4: Access control list for a keychain item

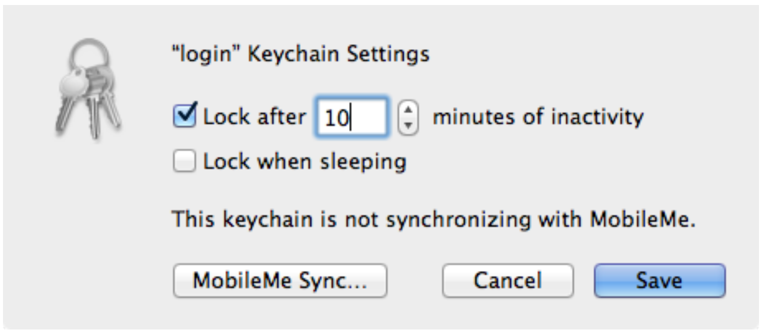

Cyberduck wants to use your confidential information stored in "zarquon.kset.org" in your keychain.

Do you want to allow access to this item?

Figure 3.5: Cyberduck asking for access to a keychain item

There are several security implications one must consider at the user level. First and foremost is the possibility to have an empty string as a password. Since it is possible for a user to not have an account password, their login keychain doesn't have one either. Although Mac OS X warns the user when creating a user account or keychain without a password, the possibility still exists. A similar problem is the creation of keychains with weak passwords. Apple mitigates this problem by providing a progress bar in the New Password dialog window which changes colour from red to green as the typed password gets stronger. Text below the progress bar tells the user if the password is considered weak, fair, good, or excellent.

The issue of timeout poses a possible security risk as well (Edge et al., 2008, 2010). After unlocking, the login keychain by default doesn't lock itself after a timeout. It has to be explicitly enabled in Keychain Access. Also, it is important to note that the timeout doesn't take into consideration keyboard and mouse activity, only the keychain activity, or rather the lack of it. So if a user remains logged in and goes away from the machine for a while, another malicious user may gain access to every


"login" Keychain Settings

☑ Lock after 10 minutes of inactivity

☐ Lock when sleeping

This keychain is not synchronizing with MobileMe.

Figure 3.6: Setting the keychain timeout interval in Keychain Access

website, network share, or any other resource which has saved something in the keychain. However, the malicious user may not retrieve the items himself, but he may use the items indirectly through applications which are allowed to use items in the keychain. For example, a malicious user may access a webmail system through Safari if the password is saved, but cannot access the password through Keychain Access. There is a rather important distinction to be made here between regular applications and Keychain Access because Keychain Access is one of the few applications which allow the user to see the password in plaintext. If Keychain Access is in the item's access control list it is possible to decrypt it without entering the keychain password.

The last issue which may arise from the user perspective stems from a fundamental design flaw in the keychain file format itself. Although the keychain is described as an encrypted file it is in fact only partially encrypted. All items have fully visible names. Passwords have fully visible corresponding account names and the location where the user needs to provide it. For example, an e-mail password usually has the server name as the item name, the account name is the account name on that server, and the location is an URL to that server. So an attacker knows everything he needs to know about a mail account to initiate an attack on the server just by opening Keychain Access or by reading the keychain in a simple text editor. Only the most sensitive data, the password, is encrypted.

The issues outlined in this section prove that no matter how strong the cryptographic algorithm and implementation, the first step towards a more secure system lies within people themselves. Only through constant education of the importance of using strong passwords, what constitutes a strong password, and the importance of frequent password changes can this problem be mitigated. As long as man creates technology it will always be prone to human error, the best we can do is try to minimise its impact on the system as a whole.

3.2. The system administrator's perspective

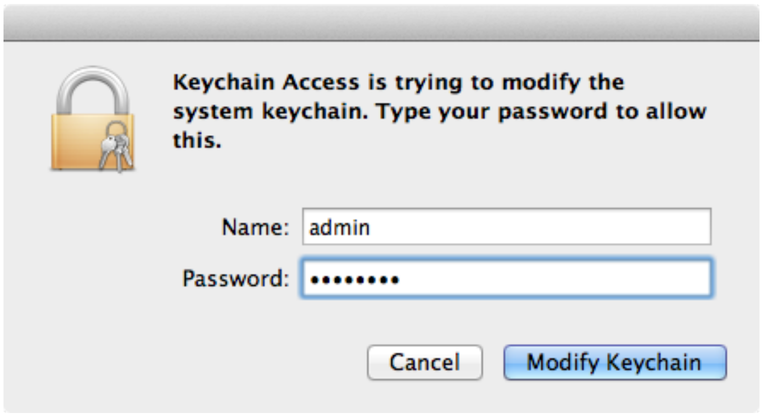
Administering keychains doesn't require much work, but besides knowing how to manage keychains through the graphical user interface, a system administrator should know how to use the command line, where keychains are normally stored, what common issues may arise, and how to repair a damaged keychain. This information is crucial when preparing a forensic analysis of a Mac because it enables the analyst to quickly and efficiently extract the keychain, especially if the keychain is extracted off a damaged hard drive.

There are several locations where keychains reside:

- */Users/<username>/Library/Keychain/login.keychain* - This is the default location for the login keychain. As a default, the password for this keychain matches the user's account password, so this keychain is automatically unlocked and available when the user logs in. If the user's account password does not match the keychain's password, it will not automatically unlock during login (White, 2012).

  Users can create additional keychains if they wish to segregate their authentication assets. For example, they can keep their default login keychain for trivial items, and then create a more secure keychain that does not automatically unlock for more important items.

- */Library/Keychain/FileVaultMaster.keychain* - This keychain is encrypted with the FileVault master password. FileVault is Apple's system which encrypts files on a Mac. FileVault is superseded by FileVault 2 which offers full disk encryption, but this keychain is still used for creating and deploying a recovery key. The recovery key is used for unlocking encrypted disks. This keychain is mainly used when deploying FileVault on a large scale to multiple Macs.
- */Library/Keychain/System.keychain* - This keychain maintains authentication assets that are not user specific. Examples of items stored here include Wi-Fi wireless network passwords, 802.1X network passwords, and local Kerberos support items. Although all users benefit from this keychain, only administrative users can make changes to it.
- */System/Library/Keychains/* - There are several keychain files in this folder that store root certificates used to help identify trusted network services. Once again, all users benefit from these keychains, but only administrative users can make changes to them.


Keychain Access is trying to modify the system keychain. Type your password to allow this.

Name: admin

Password: ••••••••

Figure 3.7: Authentication dialog for modifying the System keychain

On the inside, the System keychain is no different than an ordinary keychain. However, there are some differences in accessing this particular keychain. Unlike other keychains, only administrative users may access this keychain and they access it with their own password. This is because the master key for the keychain is saved in the */var/db/SystemKey* file, and this file may only be read by the root user. So when Keychain Access tries to unlock this keychain the authorization dialog is not from the keychain handling component of the Security Framework, but from the component that handles file access and escalating user privileges.

This special method of access is also evident by the fact that the System keychain has, apart from Keychain Access in the graphical user interface, a special command for accessing it called **systemkeychain**. The **systemkeychain** can be used to create a system keychain, make it possible for a keychain to unlock another keychain, or test unlocking a keychain (App, 2012e). It may even create a System keychain with a password like a regular keychain, although the documentation clearly states that this is generally used only for testing purposes and using it is not recommended.

So, if a system administrator would like to copy the System keychain to different machines the */var/db/SystemKey* file has to be copied as well because there is no other way to input a password for decrypting the keychain. This is also a security risk since the master key written in the file may be easily read if the analysis is done whilst ignoring file permissions. It is best if nothing of real value is kept in this keychain, unless the whole filesystem is encrypted.

All other keychains may be managed by the **security** tool. This tool is a simple command line interface which lets users administer keychains, manipulate keys and certificates, and do just about anything the Security Framework is capable of from the command line (App, 2012d). The following are some examples in using the **security** command.

Listing keychains: **security** list-keychains

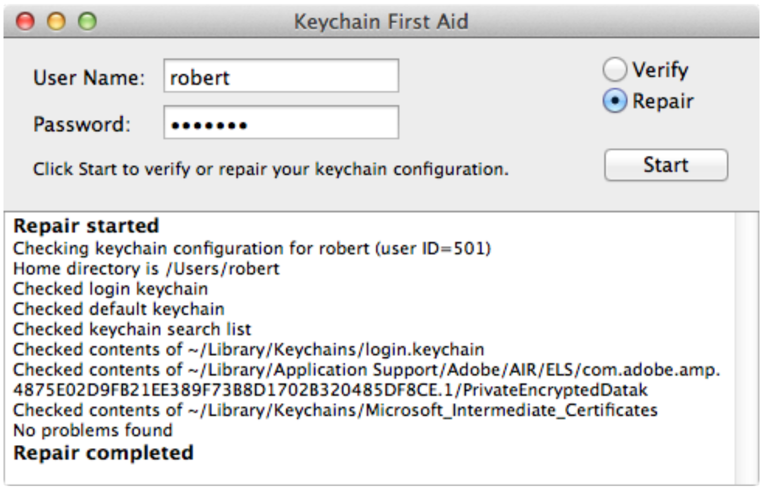
```
"/Users/robert/Library/Keychains/login.keychain"
"/Library/Keychains/System.keychain"
```

Dumping authorization control lists for items (only one item shown):
security dump-keychain -a /Users/robert/Library/Keychains/login.keychain

```
keychain: "/Users/robert/Library/Keychains/login.keychain"
class: "genp"
attributes:
  0x00000007 <blob>="robert.veznaver@gmail.com"
  0x00000008 <blob>=<NULL>
  "acct"<blob>="281859764"
  "cdat"<timedate>=0x32303131313031323231343532355A00
                  "20111012214525Z\000"
  "crtr"<uint32>=<NULL>
  "cusi"<sint32>=<NULL>
  "desc"<blob>=<NULL>
  "gena"<blob>=<NULL>
  "icmt"<blob>=<NULL>
  "invi"<sint32>=<NULL>
  "mdat"<timedate>=0x32303132303531323232333234385A00
                  "20120512223248Z\000"
  "nega"<sint32>=<NULL>
  "prot"<blob>=<NULL>
  "scrp"<sint32>=<NULL>
  "svce"<blob>="iCloud"
  "type"<uint32>=<NULL>
access: 3 entries
  entry 0:
    authorizations (1): encrypt
    don't-require-password
    description: iCloud
    applications: <null>
  entry 1:
    authorizations (6): decrypt derive export_clear
                        export_wrapped mac sign
    don't-require-password
    description: iCloud
    applications (4):
      0: 0x67726F75703A2F2F496E7465726E65744163636F756E747300
         "group://InternetAccounts\000"
      1: 0x67726F75703A2F2F69436C6F75644163636F756E747300
         "group://iCloudAccounts\000"
      2: 0x67726F75703A2F2F646F742D6D616300
         "group://dot-mac\000"
      3: /Applications/System Preferences.app (OK)
  entry 2:
    authorizations (1): change_acl
    don’t-require-password
    description: iCloud
    applications (0):
```

Dumping raw (encrypted) data for items (only one item shown):
security dump-keychain -r /Users/robert/Library/Keychains/login.keychain

```
keychain: "/Users/robert/Library/Keychains/login.keychain"
class: "genp"
attributes:
  0x00000007 <blob>="robert.veznaver@gmail.com"
  0x00000008 <blob>=<NULL>
  "acct"<blob>="281859764"
  "cdat"<timedate>=0x32303131313031323231343532355A00
                   "20111012214525Z\000"
  "crtr"<uint32>=<NULL>
  "cusi"<sint32>=<NULL>
  "desc"<blob>=<NULL>
  "gena"<blob>=<NULL>
  "icmt"<blob>=<NULL>
  "invi"<sint32>=<NULL>
  "mdat"<timedate>=0x32303132303531323232333234385A00
                   "20120512223248Z\000"
  "nega"<sint32>=<NULL>
  "prot"<blob>=<NULL>
  "scrp"<sint32>=<NULL>
  "svce"<blob>="iCloud"
  "type"<uint32>=<NULL>
raw data:
0x73736770B0C823DA24A664D3DB14EF6A3138AD78AB4724BF5272D6234
F326E0B584C8591BB7016642237055E0CE1F6E6A7B377C82DF204D20372
0F59BC701F478212E7186D07E08828E9DE63
"ssgp\260\310#\332$\246$\323\333\024\357\33218\255x\253G$
\277Rr\326#O2n\013\230L\205\241\273p\026\324"7\005^\014
\341\366\344\254\263w\310-\362\004\322\003r\017Y\264p\037M
\202\022\347\030m\007\344\210\351\336c"
```



User Name: robert
Password: •••••••
Verify [✓ Repair]
Click Start to verify or repair your keychain configuration.

```
Repair started
Checking keychain configuration for robert (user ID=501)
Home directory is /Users/robert
Checked login keychain
Checked default keychain
Checked keychain search list
Checked contents of ~/Library/Keychains/login.keychain
Checked contents of ~/Library/Application Support/Adobe/AIR/ELS/com.adobe.amp.
4875E02D9FB21EE389F73B8D1702B320485DF8CE.1/PrivateEncryptedDatak
Checked contents of ~/Library/Keychains/Microsoft_Intermediate_Certificates
No problems found
Repair completed
```

Figure 3.8: Keychain First Aid utility repairing keychains

A damaged keychain may be verified and repaired by the Keychain First Aid utility in the Keychain Access application. The utility is meant to fix certain problems that can corrupt a keychain, but with certain setups in the past it has caused more damage than it repairs. It is advisable to verify keychains using the utility and try to repair them manually (von Stauber, 2004).

The main problem with administering the keychain is that as its size grows, its performance deteriorates significantly. The main cause of this is how updates are handled. When updating any data, a separate temporary file is created with a full copy of the original keychain, and then copied over. As the file size increases, copying the original file contents becomes very slow (Agi, 2008).

3.3. The developer's perspective

In Mac OS X, Keychain Services and other security APIs are built on the open source Common Data Security Architecture (CDSA) and its programming interface, Common Security Services Manager (CSSM).

CDSA is an open source security architecture adopted as a technical standard by the Open Group. It consists of a cryptographic framework and several layers of security services. Apple uses its own CDSA implementation. CDSA helps in the implementation of security features such as encryption, fine-grained access permissions and user authentication, and secure data storage (Singh, 2006). In Mac OS X 10.7 Lion CDSA is deprecated in favour of Apple's own CommonCrypto system.

The Mac OS X Keychain Services API provides functions to perform most of the operations needed by applications, including creating, deleting, and modifying keychains and keychain items, controlling access to keychain items, finding keychain items, and retrieving attributes and data from items. However, the underlying CSSM API provides more capabilities that might be of interest to specialty applications, such as applications designed to administer the security of a computer or network. The Keychain Services API includes a number of functions that return or create CSSM structures so that a developer can move freely back and forth between Keychain Services and CSSM (App, 2012c).

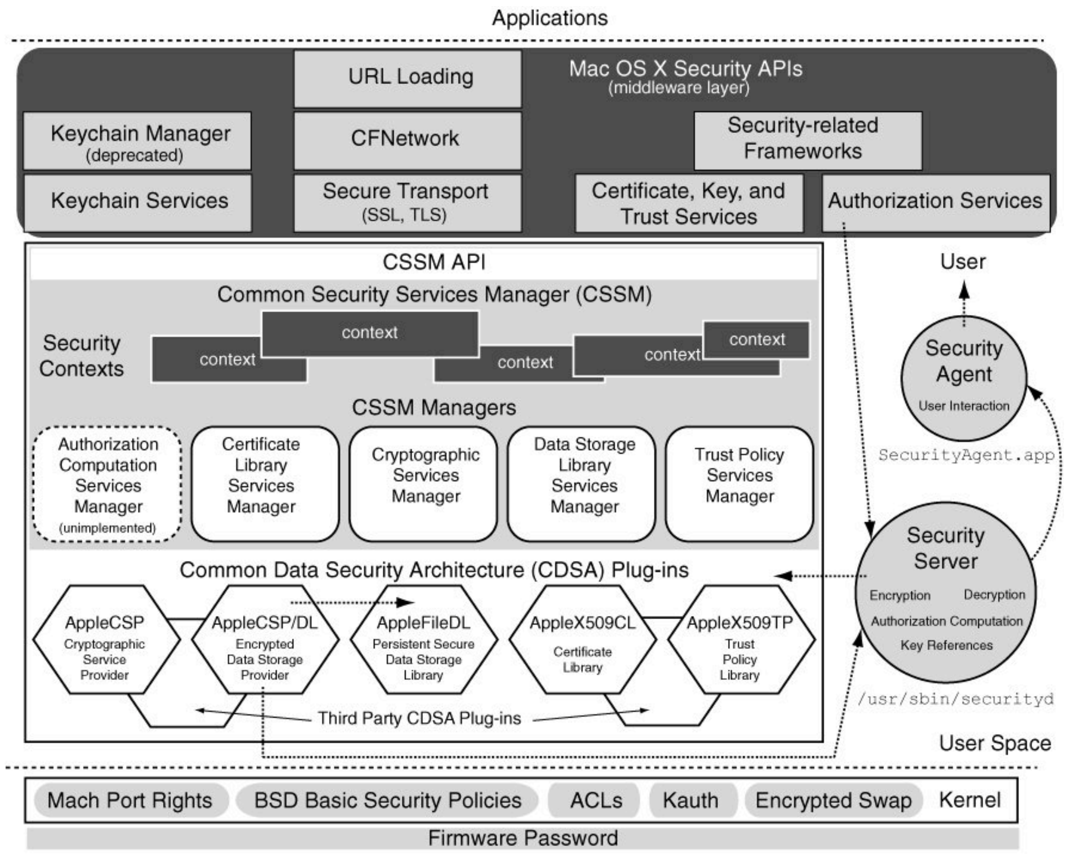


Figure 3.9: Mac OS X 10.4 security architecture (Singh, 2006)

Most applications will use the functions *SecKeychainAddGenericPassword* and *SecKeychainFindGenericPassword* for adding a generic password into a keychain. The following example shows how a typical application might use Keychain Services functions to get and set passwords for generic items.

```
#include <CoreFoundation/CoreFoundation.h>
#include <Security/Security.h>
#include <CoreServices/CoreServices.h>

//Call SecKeychainAddGenericPassword to add
// a new password to the keychain:
```

```
OSStatus StorePasswordKeychain (void* password, UInt32
   passwordLength)
{
 OSStatus status;
 status = SecKeychainAddGenericPassword (
               NULL,            // default keychain
               10,              // length of service name
               "SurfWriter",    // service name
               10,              // length of account name
               "MyUserAcct",    // account name
               passwordLength,  // length of password
               password,        // pointer to password data
               NULL             // the item reference
    );
    return (status);
 }

//Call SecKeychainFindGenericPassword to get
// a password from the keychain:
OSStatus GetPasswordKeychain (void *passwordData, UInt32 *
   passwordLength, SecKeychainItemRef *itemRef)
{
  OSStatus status1 ;

  status1 = SecKeychainFindGenericPassword (
                NULL,           // default keychain
                10,             // length of service name
                "SurfWriter",   // service name
                10,             // length of account name
                "MyUserAcct",   // account name
                passwordLength, // length of password
                passwordData,   // pointer to password data
                itemRef         // the item reference
  );
  return (status1);
}

//Call SecKeychainItemModifyAttributesAndData to change
// the password for an item already in the keychain:
OSStatus ChangePasswordKeychain (SecKeychainItemRef itemRef)
{
  OSStatus status;
  void * password = "myNewP4sSw0rD";
  UInt32 passwordLength = strlen(password);

  status = SecKeychainItemModifyAttributesAndData (
                itemRef,         // the item reference
                NULL,            // no change to attributes
                passwordLength,  // length of password
                password         // pointer to password data
  );
  return (status);
}

int main (int argc, const char * argv[]) {
  OSStatus status;
  OSStatus status1;

  void * myPassword = "myP4sSw0rD";
  UInt32 myPasswordLength = strlen(myPassword);

  void *passwordData = nil; // will be allocated and filled in by
                            // SecKeychainFindGenericPassword
  SecKeychainItemRef itemRef = nil;
  UInt32 passwordLength = nil;

  status1 = GetPasswordKeychain (&passwordData,&passwordLength,&
     itemRef);
  //Call SecKeychainFindGenericPassword
  if (status1 == noErr)       //If call was successful, authenticate
                              // user and continue.
  {
    //Free the data allocated by SecKeychainFindGenericPassword:
    status = SecKeychainItemFreeContent (
                NULL,           //No attribute data to release
                passwordData    //Release data buffer allocated by
                                // SecKeychainFindGenericPassword
    );
  }

  if (status1 == errSecItemNotFound) { //Is password on keychain?
  /*
  If password is not on keychain, display dialog to prompt user for
  name and password. Authenticate user. If unsuccessful, prompt user
  again for name and password. If successful, ask user whether to
     store
  new password on keychain; if no,return. If yes, store password:
  */
    status = StorePasswordKeychain (myPassword,myPasswordLength);
    //Call SecKeychainAddGenericPassword
    return (status);
  }

  /*
  If password is on keychain, authenticate user.
  If authentication succeeds, return.
  If authentication fails, prompt user for new user name
  and password and authenticate again.
  If unsuccessful, prompt again.
  If successful, ask whether to update keychain with new info.
  If no, return.
  If yes, store new information:
  */
  status = ChangePasswordKeychain (itemRef);
  //Call SecKeychainItemModifyAttributesAndData

  if (itemRef) CFRelease(itemRef);
  return (status);

}
```

Using the *SecKeychainUnlock* function it is possible to create a small bruteforce attack program. Albeit very slow, the following example demonstrates how to attack the keychain with a dictionary file. It should be noted that if the keychain is already unlocked the function will return with no error, the same as if the password was correct. This may be avoided by locking the keychain beforehand, but it is assumed that the keychain under attack is locked.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <Security/SecKeychain.h>

int main(void)
{
  OSStatus err;

  //A reference to the keychain to unlock. NULL specifies the
  // default keychain.
  SecKeychainRef keychain = NULL;

  //An unsigned 32-bit integer representing the length of the
  // password buffer.
  UInt32 passwordLength = 0;

  //A buffer containing the password for the keychain.
  const void *password;

  //Value indicating whether the password parameter is used.
  // If no password is provided, the Unlock Keychain dialog
  // box will appear.
  Boolean usePassword = TRUE;

  // Define an example dictionary of strings
  char dict[3][10]= { "one", "two", "three" };
  int x = 0;

  // Loop through the dictionary
  for (x=0; x<3; x++) {
    // Load the password
    password = dict[x];
    passwordLength = strlen(dict[x]);
    printf("pass:␣%s␣|␣len:␣%d\n", (char *) password, passwordLength
       );

    // Try unlocking the keychain
    err = SecKeychainUnlock(keychain, passwordLength, password,
       usePassword);

    // No error - success!
    if (!err) {
      printf ("Found␣password:␣%s\n", (char *) password);
      exit(0);
    }

  }

  // If there is no matching password in dictionary
  printf ("Password␣not␣found.\n");
  return 0;
}
```

4. Reverse engineering the Keychain

Reverse engineering can roughly be defined as taking a final product, dissecting it to understand its functionality, obtain design and other useful information. And Software Reverse Engineering is to analyse system's code, documentation and behavior to create system abstractions and design information. And understanding internal system complexities. While having access only to external behavior. And it is the first phase of reengineering process. There are three different of approaches to the system analysis employing reverse engineering (Ali, 2005).

White Box analysis consists of analysing and understanding program's code without running it. This "static analysis" approach is used to produce not only the potential program but also to obtain information regarding characteristics of the code. Black box analysis involves checking program behavior by inputs. It is usually done before white box analysis. It is often used to specify areas for white box analysis. Third approach is Gray-Box Analysis. In which black and white box analysis is used together. Some parts of the system treated as a black box while others looked at more closely using white box analysis.

4.1. High level analysis

The Mac OS X and iOS security implementation includes a daemon called the Security Server that implements several security protocols, such as access to keychain items and root certificate trust management. Mac OS X also includes a separate per-user agent, called the Security Agent, that is used by the Security Server to display a user interface (App, 2012a).

The Security Server (securityd) is a daemon running in Mac OS X and iOS that implements several security protocols, such as encryption, decryption, and (in Mac OS X) authorization computation. In Mac OS X and iOS, the Security Server listens for messages from various security APIs and performs cryptographic services on their behalf.

Because developers generally use references to keys rather than using the keys themselves, the Security Server can keep those keys in a separate address space from the client process, thus reducing the risk of accidental disclosure. As an added advantage, whenever Apple introduces new authentication or encryption technology, existing software that uses the Mac OS X security APIs can transparently support it without code changes, provided that the software does not need to import or export keys directly.

The Security Server has no public API. Instead, the developer's code calls APIs such as the Keychain Services API, the Certificate, Key, and Trust services API, and the Authorization Services API (only on Mac OS X), which in turn communicate with the Security Server.

The Security Agent is a separate process that provides the user interface for the Security Server in Mac OS X. When the Security Server requires the user to authenticate, the Security Agent displays a dialog requesting a user name and password. The advantages of performing this action in a separate process are twofold. First, an application can obtain authorization without ever having access to the user's credentials (username and password, for example). Second, it enables Apple to add new forms of authentication without requiring every application to understand them.

The Security Agent runs with restricted permissions so that the user must be physically present, using the graphical user interface, in order to be authenticated. Because the graphical user interface elements can not be used through a command-line interface, this restriction makes it much more difficult for a malicious user to breach an application's security.

Keychains contain three kinds of objects: keys, access control lists (ACLs), and other items. Storage and access of objects in keychains is done by the AppleCSPDL module, via Keychain Services and the CSSM. Each key in the keychain has an ACL, processed by the Security Server, which determines which applications can access it and how they can access it.

An application is identified by a signature (hash) of the invariant parts of the app binary. This signature is also used in the Mac App Store and the new GateKeeper technology in Mac OS X Mountain Lion. In most cases application access to keys must be re-established after a software update. The signature provides reasonable assurance that the application is the same as when the user gave it permission to access a key. The only way to modify ACLs is by user direction through a Security Agent, and the Security Server will only accept ACL changes from an Agent that it has started.

Keychain items seen in Keychain Access are not actually keys. Each item (password, secure note, etc.) is stored in the keychain, and encrypted with its own key, which is also stored in the keychain. ACLs are applied to per-item encryption keys, but they are made to look as if they are properties of the items the keys protect.

A master signing key is used to sign the per-item keys and their ACLs. Per-item keys and the master signing key are themselves encrypted with a master key. ACLs can not be encrypted, since they are needed to determine whether access to encrypted keys should be permitted. The master key is encrypted with the keychain password. The master key and master signing key are also stored in the keychain, making the keychain file completely portable, requiring only the password to unlock its contents.

When an application desires to access a keychain items the following happens (von Stauber, 2004):

1. The application makes a request with Keychain Services, through the CSSM, which calls on the AppleCSPDL.

2. If the default keychain is locked, AppleCSPDL retrieves the encrypted master key and hands it to the Security Server.

3. The Security Server has a Security Agent prompt the user for the keychain password, which is passed back to the Security Server.

4. The Security Server uses the password to decrypt the master key, then caches it in memory (a keychain is unlocked when the Security Server has its decrypted master key cached in memory).

5. Once the keychain is unlocked, the AppleCSPDL retrieves the desired item, the item's encryption key, and the key's ACL, handing them to the Security Server.

6. The Security Server verifies the signature on the key and ACL with the master signing key.

7. If the signature checks out, the Security Server processes the ACL, resulting in denial, permission, or another prompt through the Security Agent.

8. If access is permitted, the Security Server decrypts the keychain item and hands it to the AppleCSPDL, which passes it back up the software stack to the application.

There are several important things to note in the item granting process. First of all, the application process never sees the user's keychain password, nor any of the decrypted keys in the keychain. This is to ensure that the application will only gain access to those items it was explicitly allowed. Secondly, locking the keychain means to have the Security Server cache a copy of the keychain's decrypted master key. Unlocking simply erases the master key from memory. This has been a security problem until the introduction of full ASLR (Address space layout randomization) in Mac OS X 10.7 Lion.

4.2. Building a custom Security Server

The critical part of encrypting and decrypting the keyhcain lies within the Security Server. Apple has published a debug version of the Security Framework until Mac OS X 10.5, but the debug information doesn't contain any interesting data regarding the keychain file format, which may be concluded from reading the source code of the Security Framework.

The most direct way of getting to the keychain file format is to rebuild Mac OS X's Security Server with custom options to output the necessary information. However, building the source code is more difficult than it appears. It is impossible to build any security project from Mac OS X 10.5 onwards without heavy patching. The PureDarwin project, although based on Mac OS X 10.5, doesn't even use the Security Framework because there is a lot of proprietary code needed for compiling and using the framework that is not published.

It is also interesting to note that not all versions of Xcode, Apple's integrated developer environment, will build every project. The exact version of Xcode which works with a specified Mac OS X release has to be used. Even minor version differences result in compilation failure. Using the method of trial and error, and comparing the dates of Mac OS X releases with Xcode releases it was found that the most recent version of Mac OS X on which the Security Server may be built is Mac OS X 10.4.3 with Xcode 2.2. This Mac OS X version (build number 8F46) may be built only on a PowerPC, so a Mac mini with a G4 processor was borrowed and used for compiling the source code. However, although the Security Server may be built, it is not possible to build the whole Security Framework, but the Security Server was enough for the scope of this thesis.

The Darwin Build Scripts are a collection of tools that assist compilation of the many projects contained in Darwin, the open source base of Apple's Mac OS X operating system. Apple publishes the sources of these projects in an archive format (.tar.gz) and publishes them on *opensource.apple.com*. These tools provide the proper build environment as well as help to resolve any necessary dependencies prior to building (Dar, 2005). Darwin Build Scripts are part of the DarwinBuild project which appeared in OpenDarwin and is currently hosted at *darwinbuild.macosforge.org*.

Building such an old version of Mac OS X required an old version of Xcode and DarwinBuild as well. Xcode 2.2 was obtained from *developer.apple.com* and DarwinBuild 0.7.2 was obtained from the old OpenDarwin web site archives hosted at the Internet Archive (*archive.org*) since the OpenDarwin project and web site shut down in 2006.

After compiling and installing Darwin Build Scripts, the command **darwinbuild** is used to initialize the build directory. It is strongly recommended that builds be performed on a UFS or case-sensitive HFSX filesystem. On Mac OS X, a disk image was used. It is important that the volume's file ownership is honored. Since the OpenDarwin website is no longer accessible, the .plist files which contain the necessary information (dependencies, file list and versions) for **darwinbuild** to build the project were downloaded from *darwinbuild.macports.org*.

To initialize the build directory the following commands are used:

```
hdiutil create -size 4g -type UDIF -fs HFSX -volname Builds \
        -uid 0 -gid 0 -attach Builds.dmg
sudo vsdbutil -a /Volumes/Builds
sudo -s
cd /Volumes/Builds
mkdir Build8F46
cd Build8F46
darwinbuild -init 8F46
```

The **hdiutil** command crates a 4GB UDIF disk image with the HFSX filesystem and permissions set to the root user. The image is called Builds.dmg and is mounted automatically by default in */Volumes/Builds*. The **vsdbutil** command re-enables permissions on the mount point in case the operating system decides to ignore permissions (which is the default behavious on Mac OS X for mounting disk images). After logging in as the root user and moving to the directory where the image is mounted the **darwinbuild** command is issued to create the build environment.

After initialization, the build directory contains the following directories:

- *.build* - private data for the DarwinBuild system
- *Headers* - resulting header files from previous builds

- *Logs* - logs of previous build attempts
- *Roots* - finished products of previous successful builds
- *Sources* - sources downloaded from Apple
- *Symbols* - debug symbol versions of previous build products

When using the darwinbuild script, it is necessary that the current working directory is the build directory, or alternatively, that the DARWIN_BUILDROOT environment variable is set to the absolute path of the destination directory. Since an old Darwin-Build version was used, the .plist files (*8A428.plist*, *8B15.plist*, *8C46.plist*, and *8F46.plist*) downloaded from the new website are put into the *.build* directory and the **darwin-build -init 8F46** command is run again. It is necessary to download all .plist files up to the build which is supposed to be initialized and which are from the same major version of Mac OS X.

The **darwinxref** tool allows querying which source version of a Darwin project is present in a particular Mac OS X build. It also stores information about what dependencies a particular project has, and what files the project produces. As each Mac OS X release is made available, Apple publishes a property list file containing the project names and versions in that release. These property lists are read by the darwinxref tool to seed its internal database.

At the minimum, the build environment consists of the creation of SRCROOT, OBJROOT, SYMROOT, and DSTROOT environment variables. These variables contain an absolute path to a directory which must exist prior to invoking the Makefile. User configurable environment variables, such as the target architecture, can be set in the build plist file that is loaded via loadIndex. The darwinbuild script creates the aforementioned directories, sets the environment variables, and issues the appropriate make command.

Within the plist file, the RC_ARCHS variable indicates which target architectures should be included in the build. To build only for the PowerPC, this variable should be set to "ppc" with the command **darwinxref -b 8A428 edit** (fG!, 2012). The lowest build number in a major version of Mac OS X should be used to edit environment variables.

To build the Security Server project (securityd), the darwinbuild script can be used in the following manner: **darwinbuild securityd**. The darwinxref tool is consulted to find the version that corresponds to the build specified when the build directory was initialized. It is necessary to run the darwinbuild tool as root so that projects can set the proper ownership and permissions on the build results. Darwin-

Build first looks in the Sources directory for a directory containing the sources to be built or a .tar.gz archive containing the sources. If neither is found, darwinbuild will attempt to download the sources from Apple.

If it does not already exist, a BuildRoot directory will be created. This is where the build will actually take place. During the build, DarwinBuild will change the root directory to BuildRoot. DarwinBuild is capable of copying the required tools, libraries and headers from the Roots directory into the BuildRoot prior to building. If a necessary dependency is not found in the Roots directory, it will be downloaded from Apple. The build output will be written to the console, and additionally logged into a file in the Logs directory.

When the build succeeds, the finished product will be copied out of the Build-Root directory and into the Roots directory. After the copy, darwinbuild traverses the directory and records all files found in the darwinxref database. This makes it possible to query which project a file is produced by. When a Mach-O executable, library, or bundle is found during the traversal, the dynamic library load commands are recorded in the darwinxref database. This makes it possible to query which additional projects are required to run an executable produced by the project. Additionally, any products containing debug symbols will be placed into the Symbols directory (Dar, 2005).

Shortened output of **darwinbuild securityd**:

```
*** Fetching Sources ...
Found securityd-25481 in /Volumes/Builds/Build8F46/Sources
*** Copying Sources ...
*** Installing Roots ...
*** Installing Headers ...
*** Mounting special filesystems ...
devfs appears to exist ...
volfs appears to be mounted ...

BUILDING securityd-25481~20 on Mon Jun 18 15:57:24 PDT 2012

Build configuration:
    Build host:       robert-veznavers-mac-mini.local
    Build tool:       xcodebuild
    Build action:     install
    Build number:     8F46
    cc version:       gcc version 3.3 20030304 (Apple, build 1809)
    cctools version: version cctools-590.obj, GNU assembler v. 1.38
    xcode version:    Component versions: DevToolsCore-660.0;
```

```
DevToolsSupport-651.0

Build parameters:
SRCROOT: /SourceCache/securityd/securityd-25481
OBJROOT: /private/var/tmp/securityd/securityd-25481.obj
SYMROOT: /private/var/tmp/securityd/securityd-25481.sym
DSTROOT: /private/var/tmp/securityd/securityd-25481.root
RC_ProjectName: securityd
RC_ProjectSourceVersion: 25481
RC_ProjectNameAndSourceVersion: securityd-25481
RC_ProjectBuildVersion: 20
INSTALLED_PRODUCT_ASIDES: YES
MACOSX_DEPLOYMENT_TARGET: 10.4
RC_ARCHS: ppc
RC_OS: macos
RC_PRIVATE: /private
RC_RELEASE: Tiger
UNAME_RELEASE: 8.0
UNAME_SYSNAME: Darwin

Environment variables:
CHROOTED=YES
DARWINBUILD_BUILD=8F46
DARWINXREF_DB_FILE=/Volumes/Builds/Build8F46/.build/xref.db
DARWIN_BUILDROOT=/Volumes/Builds/Build8F46
DSTROOT=/private/var/tmp/securityd/securityd-25481.root
GROUP=wheel
HOME=/var/root
INSTALLED_PRODUCT_ASIDES=YES
LOGNAME=root
MACOSX_DEPLOYMENT_TARGET=10.4
OBJROOT=/private/var/tmp/securityd/securityd-25481.obj
PATH=/bin:/sbin:/usr/bin:/usr/sbin/:/usr/local/bin:/usr/local/sbin
RC_ARCHS=ppc
RC_ProjectBuildVersion=20
RC_ProjectName=securityd
RC_ProjectNameAndSourceVersion=securityd-25481
RC_ProjectSourceVersion=25481
RC_RELEASE=Tiger
SRCROOT=/SourceCache/securityd/securityd-25481

Installed Roots:
CF                   -> d678b88c3287d1149aa1a35f69e353e11ee93e59
```

```
Libsystem            -> dac3cf9561d42f174b1de070dc8c782f6add4b4e
OpenSSL              -> b4793e414afc7616cd4eced70e0ba94d1ca78c5b
Security             -> 9f82bd6f6138fda7feb4773f16e77d84547c556b
SecurityTokend       -> 43448396f8cf309179648aa77187649efc16c330
SmartCardServices    -> 2fd8aaa1451fe16136fdc9590a811b889968eda8
libsecurity_agent    -> 64f63a8df4421bd19ff5999554140304ee8a87f8
libsecurity_cdsa_client -> 1d34e6f8a1e757ac91d0dc2a9d120442ab535ded
libsecurity_cdsa_utilities ->
   e19aeab31e470650fa44ac844b9e1612cb759170
libsecurity_utilities -> 9b43347581af9d197f6186f641fef6b051a3af98
libsecurityd         -> 26f00e414ee3d0ed93826bdb6f4e948bc35e4206
xnu                  -> c651cd7d2d8d102b4e297effb5fa951989fc2ba1

xcodebuild install  "SRCROOT=/SourceCache/securityd/securityd-25481"
"OBJROOT=/private/var/tmp/securityd/securityd-25481.obj"
"SYMROOT=/private/var/tmp/securityd/securityd-25481.sym"
"DSTROOT=/private/var/tmp/securityd/securityd-25481.root"
"RC_ProjectName=securityd" "RC_ProjectSourceVersion=25481"
"RC_ProjectNameAndSourceVersion=securityd-25481"
"RC_ProjectBuildVersion=20" "INSTALLED_PRODUCT_ASIDES=YES"
"MACOSX_DEPLOYMENT_TARGET=10.4" "NEXT_ROOT=" "RC_ARCHS=ppc"
"RC_CFLAGS=-pipe␣-no-cpp-precomp␣-arch␣ppc" "RC_JASPER=YES"
"RC_NONARCH_CFLAGS=-pipe␣-no-cpp-precomp" "RC_OS=macos"
"RC_PRIVATE=/private" "RC_RELEASE=Tiger"
"RC_USE_GCC35_FOR_PPC64=YES" "RC_XBS=YES" "RC_ppc=YES"
"SEPARATE_STRIP=YES" "UNAME_RELEASE=8.0" "UNAME_SYSNAME=Darwin"

Build log begins here:

=== BUILDING AGGREGATE TARGET mig WITH THE DEFAULT CONFIGURATION
(Default) NOT USING PER-CONFIGURATION BUILD DIRECTORIES ===

Checking Dependencies...

ExternalBuildToolExecution startup

=== BUILDING TOOL TARGET securityd WITH THE DEFAULT CONFIGURATION
(Default) NOT USING PER-CONFIGURATION BUILD DIRECTORIES ===

** BUILD SUCCEEDED **

EXIT STATUS: 0
40755 0 0 0 ./private
```

```
40755 0 0 0 ./private/etc
100644 0 80 17305 ./private/etc/authorization
40755 0 0 0 ./private/etc/mach_init.d
100644 0 0 420 ./private/etc/mach_init.d/securityd-installCD.plist
100644 0 0 390 ./private/etc/mach_init.d/securityd.plist
40755 0 0 0 ./private/var
40755 0 0 0 ./private/var/db
100644 0 80 3196 ./private/var/db/CodeEquivalenceCandidates
40755 0 0 0 ./usr
40755 0 0 0 ./usr/sbin
100555 0 0 876100 ./usr/sbin/securityd
100555 0 0 2110328 ./usr/sbin/securityd_debug
40755 0 0 0 ./usr/share
40755 0 0 0 ./usr/share/man
40755 0 0 0 ./usr/share/man/man1
100644 0 0 1514 ./usr/share/man/man1/securityd.1
```

4.3. Finding the Master Key

The interesting parts of the keychain are "blobs" (Johnston, 2004). There are two types of blobs: database blobs and key blobs. Both blobs (defined as DbBlob and KeyBlob) are subclasses of CommonBlob. Each blob starts with the magic hex string 0xfade0711 (ssb, 2004). There's only one DbBlob (at the end of the file), and that contains the file encryption key (amongst other things), encrypted with the master key. The master key is derived purely from the user's password, and a salt, also found in the DbBlob. PKCS #5 v2.0 PBKDF2 is used for deriving the master key.

To verify the DbBlob format, the method in **securityd** which handles database blob encoding (dbc, 2004) was modified to output the master key, signing key, IV, and blob signature.

```
//
// Encode a database blob from the core.
//
DbBlob *DatabaseCryptoCore::encodeCore(const DbBlob &blobTemplate,
    const CssmData &publicAcl, const CssmData &privateAcl) const
{
    assert(isValid());     // must have secrets to work from

    secdebug ("SS", "===␣Encoding␣database␣blob␣===");
    // make a new IV
```

Table 4.1: DbBlob format

Offset from start	Name	Description
0	0xfade0711	magic number
4	version	0x00000100 for 10.0.x, 0x00000101 for 10.1.x and on
8	crypto-offset	offset of the encryption and signing key
12	total len	length of blob in bytes
16	signature	master signing key
32	sequence	
36	idletimeout	how long until the keychain should lock itself
40	lockonsleep flag	whether the keychain has autolock enabled
44	salt	salt for encryption
64	iv	initialization vector for encryption
72	blob signature	signature of the blob, 20 bytes

```
uint8 iv[8];
Server::active().random(iv);

// build the encrypted section blob
CssmData &encryptionBits = *mEncryptionKey;
CssmData &signingBits = *mSigningKey;
CssmData incrypt[3];
incrypt[0] = encryptionBits;

secdebug ("SS", "==␣encryptionBits.length:␣%zu␣==",
   encryptionBits.length());
secdebug ("SS", "==␣encryptionBits.data:␣%s␣==", encryptionBits.
   toHex().c_str());

incrypt[1] = signingBits;

secdebug ("SS", "==␣signingBits.length:␣%zu␣==", signingBits.
   length());
secdebug ("SS", "==␣signingBits.data:␣%s␣==", signingBits.toHex
   ().c_str());

incrypt[2] = privateAcl;

secdebug ("SS", "==␣privateAcl.length:␣%zu␣==", privateAcl.
   length());
```

```
  secdebug ("SS", "==␣privateAcl.data:␣%s␣==", privateAcl.toHex().
     c_str());

  CssmData cryptoBlob, remData;
  Encrypt cryptor(Server::csp(), CSSM_ALGID_3DES_3KEY_EDE);
  cryptor.mode(CSSM_ALGMODE_CBCPadIV8);
  cryptor.padding(CSSM_PADDING_PKCS1);
  cryptor.key(mMasterKey);
  CssmData ivd(iv, sizeof(iv)); cryptor.initVector(ivd);

  secdebug ("SS", "==␣ivd.length:␣%zu␣==", ivd.length());
  secdebug ("SS", "==␣ivd.data:␣%s␣==", ivd.toHex().c_str());

  cryptor.encrypt(incrypt, 3, &cryptoBlob, 1, remData);

  // allocate the final DbBlob, uh, blob
  size_t length = sizeof(DbBlob) + publicAcl.length() + cryptoBlob
     .length();
  DbBlob *blob = Allocator::standard().malloc<DbBlob>(length);

  // assemble the DbBlob
  memset(blob, 0x7d, sizeof(DbBlob)); // deterministically fill
     any alignment gaps
  blob->initialize();
  blob->randomSignature = blobTemplate.randomSignature;
  blob->sequence = blobTemplate.sequence;
  blob->params = blobTemplate.params;
memcpy(blob->salt, mSalt, sizeof(blob->salt));
  memcpy(blob->iv, iv, sizeof(iv));
  memcpy(blob->publicAclBlob(), publicAcl, publicAcl.length());
  blob->startCryptoBlob = sizeof(DbBlob) + publicAcl.length();
  memcpy(blob->cryptoBlob(), cryptoBlob, cryptoBlob.length());
  blob->totalLength = blob->startCryptoBlob + cryptoBlob.length();

  // sign the blob
  CssmData signChunk[] = {
  CssmData(blob->data(), fieldOffsetOf(&DbBlob::blobSignature)),
  CssmData(blob->publicAclBlob(), publicAcl.length() + cryptoBlob.
     length())
};

  secdebug ("SS", "==␣signChunk[0].length:␣%zu␣==", signChunk[0].
     length());
```

```
    secdebug ("SS", "==␣signChunk[0].data:␣%s␣==", signChunk[0].
        toHex().c_str());
    secdebug ("SS", "==␣signChunk[1].length:␣%zu␣==", signChunk[1].
        length());
    secdebug ("SS", "==␣signChunk[1].data:␣%s␣==", signChunk[1].
        toHex().c_str());

    CssmData signature(blob->blobSignature, sizeof(blob->
        blobSignature));

    secdebug ("SS", "==␣signature.length:␣%zu␣==", signature.length
        ());
    secdebug ("SS", "==␣signature.data:␣%s␣==", signature.toHex().
        c_str());

    GenerateMac signer(Server::csp(), CSSM_ALGID_SHA1HMAC_LEGACY);
    signer.key(mSigningKey);
    signer.sign(signChunk, 2, signature);
    assert(signature.length() == sizeof(blob->blobSignature));

    // all done. Clean up
    Server::csp()->allocator().free(cryptoBlob);
    return blob;
}
```

After building the custom Security Server, the binary was copied to */usr/sbin/* and the file *securityd.plist* was added to */etc/mach_init.d/*. This file is used to enable debug symbols in the binary and log them into */debug/securityd.log*.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple␣Computer//DTD␣PLIST␣1.0//EN" "http:
    //www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>Command</key>
        <string>/usr/bin/env DEBUGDEST=/debug/securityd.log
            DEBUGSCOPE=all /usr/sbin/securityd</string>
        <key>OnDemand</key>
        <false/>
        <key>ServiceName</key>
        <string>com.apple.securityd</string>
</dict>
</plist>
```

4.4. HMAC-SHA-1

HMAC is a mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function, e.g., MD5, SHA-1, in combination with a secret shared key. The cryptographic strength of HMAC depends on the properties of the underlying hash function (Krawczyk et al., 1997).

The definition of HMAC requires a cryptographic hash function, which is denote by H, and a secret key K. It is assumed H to be a cryptographic hash function where data is hashed by iterating a basic compression function on blocks of data. B is the byte-length of such blocks (B=64 for all the above mentioned examples of hash functions), and L is the byte-length of hash outputs (L=16 for MD5, L=20 for SHA-1). The authentication key K can be of any length up to B, the block length of the hash function. Applications that use keys longer than B bytes will first hash the key using H and then use the resultant L byte string as the actual key to HMAC. In any case the minimal recommended length for K is L bytes (as the hash output length).

Two fixed and different strings ipad and opad are defined as follows:
ipad = the byte 0x36 repeated B times
opad = the byte 0x5C repeated B times.

To compute HMAC over the data "text": **H(K XOR opad, H(K XOR ipad, text))**.
Namely,

1. Append zeros to the end of K to create a B byte string (e.g., if K is of length 20 bytes and B=64, then K will be appended with 44 zero bytes 0x00).
2. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with ipad.
3. Append the stream of data 'text' to the B byte string resulting from step (2).
4. Apply H to the stream generated in step (3)
5. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with opad.
6. Append the H result from step (4) to the B byte string resulting from step (5).
7. Apply H to the stream generated in step (6) and output the result.

The Mac OS X keychain uses HMAC with SHA-1.

4.5. PBKDF2

PBKDF2 (Password-Based Key Derivation Function 2) applies a pseudorandom function to the input password or passphrase along with a salt value and iterates the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. This technique is known as key stretching and the added complexity of computation makes it more difficult to crack. PBKDF2 is part of PKCS #5: Password-Based Cryptography Specification Version 2.0 (Kaliski, 2000).

The function is defined as **DK = PBKDF2 (PRF, S, c, dkLen)** where:

- *PRF* is the underlying pseudorandom function (hLen denotes the length in octets of the pseudorandom function output)
- *P* is the password, an octet string
- *S* is the salt, an octet string
- *c* is the iteration count, a positive integer
- *dkLen* intended length in octets of the derived key, a positive integer, at most $(2^{32} - 1)$ * hLen
- *DK* is the derived key, a dkLen-octet string

The algorithm is as follows:

1. If dkLen > $(2^{32} - 1)$ * hLen, output "derived key too long" and stop.
2. Let l be the number of hLen-octet blocks in the derived key, rounding up, and let r be the number of octets in the last block:
   l = CEIL (dkLen / hLen), r = dkLen - (l - 1) * hLen
   Here, CEIL (x) is the "ceiling" function, i.e. the smallest integer greater than, or equal to, x.
3. For each block of the derived key apply the function F defined below to the password P, the salt S, the iteration count c, and the block index to compute the block:
   T_1 = F (P, S, c, 1) ,
   T_2 = F (P, S, c, 2) ,
   ...
   T_l = F (P, S, c, l) ,
   where the function F is defined as the exclusive-or sum of the first c iterates of

the underlying pseudorandom function PRF applied to the password P and the concatenation of the salt S and the block index i:
F (P, S, c, i) = U_1 XOR U_2 XPR ... XOR U_c
where
U_1 = PRF (P, S || INT (i)),
U_2 = PRF (P, U_1),
...
U_c = PRF (P, U_{c-1}).
Here, INT (i) is a four-octet encoding of the integer i, most significant octet first.

4. Concatenate the blocks and extract the first dkLen octets to produce a derived key DK:
DK = T_1 || T_2 || ... || T_l <0..r-1>

5. Output the derived key DK.

The Mac OS X keychain uses the HMAC-SHA-1 function with 1000 iterations, a salt length of 20 octets and intended length of 24 octets (or 192 bits). In other words; **DK = PBKDF2 (HMAC-SHA-1, S, 1000, 24)**.

4.6. Triple DES

Triple DES was the answer to many of the shortcomings of DES. Since it is based on the DES algorithm, it is very easy to modify existing software to use Triple DES. It also has the advantage of proven reliability and a longer key length that eliminates many of the shortcut attacks that can be used to reduce the amount of time it takes to break DES (Tro, 2000).

Triple DES is simply another mode of DES operation. It takes three 64-bit keys, for an overall key length of 192 bits. The procedure for encryption is exactly the same as regular DES, but it is repeated three times. The data is encrypted with the first key, decrypted with the second key, and finally encrypted again with the third key.

Consequently, Triple DES runs three times slower than standard DES, but is much more secure if used properly. The procedure for decrypting something is the same as the procedure for encryption, except it is executed in reverse. Like DES, data is encrypted and decrypted in 64-bit chunks.

Note that although the input key for DES is 64 bits long, the actual key used by DES is only 56 bits in length. The least significant (right-most) bit in each byte is a parity bit, and should be set so that there are always an odd number of 1s in every byte. These parity bits are ignored, so only the seven most significant bits of each byte are used, resulting in a key length of 56 bits. This means that the effective key strength for Triple DES is actually 168 bits because each of the three keys contains 8 parity bits that are not used during the encryption process.

The keychain uses Triple DES CBC mode where each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, a pseudo-random initialization vector is used. The key for encrypting the file key is derived from PBKDF2.

4.7. CMS

The Cryptographic Message Syntax describes an encapsulation syntax for data protection. It supports digital signatures, message authentication codes, and encryption. The syntax allows multiple encapsulation, so one encapsulation envelope can be nested inside another. Likewise, one party can digitally sign some previously encapsulated data. It also allows arbitrary attributes, such as signing time, to be signed along with the message content, and provides for other attributes such as countersignatures to be associated with a signature (Housley, 1999).

Although there is only reference to the PKCS#1 v1.5 standard of key padding in the code and although that standard is used to wrap keys in key blobs, the Mac OS X keychain uses the CMS padding for wrapping the file key. This was found out thanks to building the custom Security Server and comparing the plaintext output with its encrypted version.

Since the file key is generated from the SHA-1 function which has an output length of 160 bits, and 192 bits are needed for a Triple DES key, the file key is padded with the value x04040404 making it 192 bits long. This file key is then encrypted with the master key derived from the user's password and saved into the keychain file.

5. Conclusion

The Mac OS X keychain has been consisent in its design from 2000. Although in its time it was using the latest cryptographic algorithms, things are different today.

The Mac OS X keychain uses Triple DES as its encryption algorithm which is quite secure, but is growing older and has been superseded by newer encryption algorithms with longer key lengths. The US government has deprecated the use of Triple DES and has set AES as its new standard (Agi, 2008).

The usage of SHA-1 is also worrying. In 2005 three Chinese cryptographers showed that SHA-1 is not collision-free. That is, they developed an algorithm for finding collisions faster than brute force (Schneier, 2005). It is only a matter of time when this hash function will be deemed completely insecure.

Apple knows this, and it may be one of the reason why the iOS keychain uses newer cryptographic algorithms, however, the OS X keychain is still the same. The community around the Open Source password cracker John the Ripper has developed support for cracking the Mac OS X keychain, and the only thing which is truly slowing down the cracking progress is PBKDF2. There are plans to develop OpenCL support for PBKDF2 so the cracker may utilize the power of the GPU.

Lastly, the keychain was built in a time when processing power was more expensive and it was deemed to time consuming to encrypt everything. Since today modern processors have on-chip support for AES and full disk encryption is gaining traction thanks to SSD disks, it seems that is high time for the keychain to update its design and make itself fully encrypted.

Bibliography

securityd/securityd-25481/src/dbcrypto.cpp, 2004. URL `opensource.apple.com`.

libsecurityd/libsecurityd-22/lib/ssblob.h, 2004. URL `opensource.apple.com`.

History of OS X Keychain Integration in 1Password. AgileBits, 2008.

Muhammad Raza Ali. Why teach reverse engineering? *ACM SIGSOFT Software Engineering Notes*, 30(4), July 2005.

Mac OS X Security Configuration For Mac OS X Version 10.6 Snow Leopard. Apple Inc, 2010.

Security Overview. Apple Inc, 2012a.

iOS Security. Apple Inc, 2012b.

Keychain Services Programming Guide. Apple Inc, 2012c.

security Mac OS X Manual Page. Apple Inc, 2012d.

systemkeychain Mac OS X Manual Page. Apple Inc, 2012e.

Rob Braun. A brief history of apple's open source efforts. *DaemonNews*, 2006.

Darwin Build Scripts Manual. DarwinBuild, 2005.

Daniel Eran Dilger. 1990-1995: Apple vs. microsoft in the enterprise. *RoughlyDrafted*, 2006.

Charles Edge, William Barker, and Zack Smith. *Foundations of Mac OS X Leopard Security.* Apress, 2008.

Charles Edge, William Barker, Beau Huner, and Gene Sullivan. *Enterprise Mac Security.* Apress, 2010.

Adam C. Engst. Powertalk to the rescue? *TidBITS*, (279), 1995.

fG! How to compile gdb for ios! *Reverse Engineering Mac OS X*, 2012.

R. Housley. Cryptographic Message Syntax. RFC 2630 (Proposed Standard), June 1999. URL `http://www.ietf.org/rfc/rfc2630.txt`. Obsoleted by RFCs 3369, 3370.

Matt Johnston. *extractkeychain.py*, 2004.

B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. URL `http://www.ietf.org/rfc/rfc2898.txt`.

Joe Kissell. *Mac Security Bible*. Wiley Publishing, 2010.

H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997. URL `http://www.ietf.org/rfc/rfc2104.txt`. Updated by RFC 6151.

Charlie Miller, Dion Blazakis, Dino Dai Zovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philipp Weinmann. *iOS Hacker's Handbook*. Wiley Publishing, 2012.

Riccardo Mori. Remember powertalk? *System Folder*, 2009.

Monica Pal. Messaging service access modules. *MacTech*, 10(2), 1993.

Bruce Schneier. Cryptanalysis of sha-1. *Schneier on Security*, February 2005.

Amit Singh. *Mac OS X Internals: A Systems Approach*. Addison Wesley Professional, 2006.

Triple DES Encryption. Tropical Software, 2000.

Leon Towns von Stauber. *Mac OS X Security Framework*, 2004.

Kevin M. White. *Apple Pro Training Series: OS X Lion Support Essentials*. Peachpit Press, 2012.

Forensic analysis of the Mac OS X keychain

Abstract

Keychain is Apple Inc.'s password management system in Mac OS X. A Keychain can contain various types of data: passwords (for Websites, FTP servers, SSH accounts, network shares, wireless networks, groupware applications, encrypted disk images), private keys, certificates, and secure notes. By analyzing and decrypting the keychain file it is possible, in most cases, to gain access to every secret information a user has, and as such it represents a critical component of the operating system.

Research the Mac OS X keychain. Describe the keychain file format. Analyze used cryptographic functions. Describe possible vulnerabilities and attacks on the keychain file. Propose future improvements for security hardening.

Keywords: Mac OS X, keychain, security, cryptography, forensics

Forenzička analiza keychain datoteke Mac OS X-a

Sažetak

Keychain je Apple-ov sustav za upravljanje lozinkama na Mac OS X-u. Keychain datoteka može sadržavati različite tipove podataka kao što su: lozinke (za internetske stranice, FTP poslužitelje, SSH korisničke račune, mrežne resurse, bežične mreže, poslovne aplikacije, kriptirane preslike sadržaja), privatni ključeve, certifikati i sigurne zabilješke. Analizom i dekripcijom keychain datoteke može se, u većini slučajeva, doći do svih tajnih podataka korisnika i kao takav predstavlja kritičku komponentu operacijskog sustava.

Proučiti keychain sustav Mac OS X-a. Opisati zapis keychain datoteke. Analizirati korištene kriptografske funkcije. Opisati moguće ranjivosti i napade na keychain datoteku. Predložiti buduće postupke pojačane zaštite.

Ključne riječi: Mac OS X, keychain, sigurnost, kriptografija, forenzika