Keychain Analysis with Mac OS X Memory Forensics

Kyeongsik Lee[1], *Hyungjoon Koo*[2]
Defense Cyber Warfare Technology Center, Agency for Defense Development, Sonpa P.O Box 132, Seoul, Republic of Korea
Center for Information Security Technologies (CIST), Korea University, Anam-Dong, Seongbuk-Gu, Seoul, Republic of Korea

Abstract

User credentials are often regarded as one of significant digital evidence during an investigation process. Users tend to save their credentials in various devices for the ease of use such as messenger accounts, e-mail accounts, websites form, calendar, contacts and so forth. In particular, Mac OS X gets more information as it begins to interact with diverse smart devices like iPhone and iPad. Mac OS X maintains its own password management system called a Keychain, which stores sensitive data including application users account, keys, certificates, encrypted volume passwords with providing protection features. The core of this mechanism takes triple-DES in CBC mode. However, examiners have had difficulty in further investigation but performing simple keyword search because the structure of a Keychain remains unknown. This paper proposes how to analyze a Keychain file with a digital forensic perspective. We present the method to obtain master key from dumped memory image and to demystify a Keychain format from acquired disk image, thereby eventually reveal user credentials. The result of our experiment shows all user credentials in a Keychain. This technique helps investigators not only to extend the range of evidence examination but also to preserve integrity and reliability.

Keywords: Digital forensics, Memory forensics, Keychain, Apple Database, Mac OS X.

1. Introduction

As of January 2013, statistics shows that the iOS, Apple mobile operating system, accounts for 60% market share on smartphones [1]. The number of Mac OS X, Apple desktop operating system, has also increased as its many features have interacted with iOS operating system gradually. [2].

Mac OS X holds the password management mechanism called a Keychain for the purpose of user credential protection such as E-Mail client and messenger software in use. A Keychain is the file which maintains the space to store encrypted user accounts, public/private key pairs, certificates, encrypted volume passwords and security notes [3]. Apple explicitly states that a Keychain takes the 3DES block cipher algorithm for encryption and decryption. However, implementation details and inside logic have not revealed yet [4]. The way digital investigators often use today for useful information extraction is simple file carving and/or retrieval technique from artifact acquirement in memory and raw disk image.

With regard to a Keychain file analysis, a couple of methodologies have been introduced. However, mostly it covered merely the extraction of signature-based data due to the lack of full interpretation of a Keychain file structure. But, traditional methodology had limitations in that it was unlikely to extract entire user information. Moreover, it only worked in a live system which actual keychain resided in with root privilege.

This paper suggests how to extract the master key to decrypt a Keychain from the acquired memory and/or disk image during an investigation process, whose targets are mainly Mac OS X Lion (version 10.7) and Mountain Lion (version 10.8). Moreover, the technique will be proposed to extract encrypted area and to decrypt the information which users creates through the structure of a Keychain file format analysis, irrespective of operating systems.

2. Related Works

So far little has been known for a Keychain in Mac OS since the research on Mac artifacts has not relatively

[1]*Corresponding author.* Fax: +82 2 403 3512 (15826).
E-mail addresses: n0fate@add.re.kr (K, Lee).

[2]E-mail addresses: kevinkoo001@gmail.com (H. Koo)

widely performed as much as that of Microsoft Windows OS from a digital forensic perspective. Although a Keychain analysis has been highlighted as one of significant artifacts in Mac OS forensic analysis, it should be performed in a live system or available on the other Mac OS system. There are two mainstreams to access a Keychain: one is with the tool Apple officially provides, and the other is with the enhanced tool to help analysis in effective manner.

Apple offers a console-based tool called '*security*' and a GUI-based application called '*Keychain Access*' in order to handle a Keychain, the integrated system for password management in Mac OS. The *security* command provides a variety of built-in features which allow users to dump, add, and find Keychain elements and to create, delete, lock, and unlock a Keychain file itself [5]. However, this tool does not reveal actual data in plaintext thus it might not come in useful for the purpose of investigation. *Keychain Access* also allows to use mostly features in *security* and to obtain each decrypted data from encrypted blob with user password [6]. Still the latter has the same limitations that examiners should input user password to decrypt each blob.

In 2004, Matt Johnston released the tool "*exetractkeychain*", written in python for a Keychain analysis [7]. This was based on secuirty-177, the source code opened to the public by Apple. This tool introduced a Keychain analysis for the first time. It generated a master key with user password from the beginning and then decrypted a wrapped DB key in a Keychain file with that key. Next, it dumped user data area with *security* command. Eventually, decrypted data could be obtained with a decrypted DB key. Yet it chiefly relied on built-in system command, *security* as well as it helped to extract only partial information such as e-mail client accounts, messengers and so forth. In addition, it might not maintain system integrity because of the limitation of root privilege requirement ia live system.

In October 2011, Juuso Salonen wrote "*keychaindump*" tool, which employed more enhanced technique compared to the previous ones [8]. It targeted Mac OS X Lion or later because Apple adapted to store the master key of a Keychain in memory to promote user convenience. Here is a brief procedure of the tool. When a user executed *keychaindump* with root privilege, at first it checked MALLOC_TINY of heap space from *security server* process area in memory with built-in system command, *vmmap.* This led to extract master key candidates from the space. With choosing a correct master key among them, user credentials were disclosed at last. This technique basically took a signature-based analysis of a Keychain file before pulling a master key and user data in sequence. Nonetheless it was essential to have root privilege, and was unlikely to draw all stored user information as well. Thus it had the same drawbacks with Matt's tool.

In this paper, we propose a brand new technique not only to extract useful evidence in a forensically sound manner, but also to preserve integrity and reliability. Besides our tool is platform-independent. We start to explain by introducing the overall procedure for a Keychain analysis.

3. Procedure for Keychain Analysis

Figure 1 illustrates the entire procedure for Keychain analysis. We have a premise that by any means an investigator already obtains a Keychain file from disk image and acquires memory image. Once attained, a master key from dumped memory should be extracted and checked its validation with file signature. Once *Secure Server Daemon* process is discovered from memory image in linear format, then *Virtual Memory Map* can be extracted from that process. Since a master key in use resides in a Keychain, we need to pull MALLOC_TINY area allocated within heap from *virtual memory map*, resulting in the extraction of multiple master key candidates. This is feasible because the data structure of a master key contains a master key and its fixed-length, 0x18.

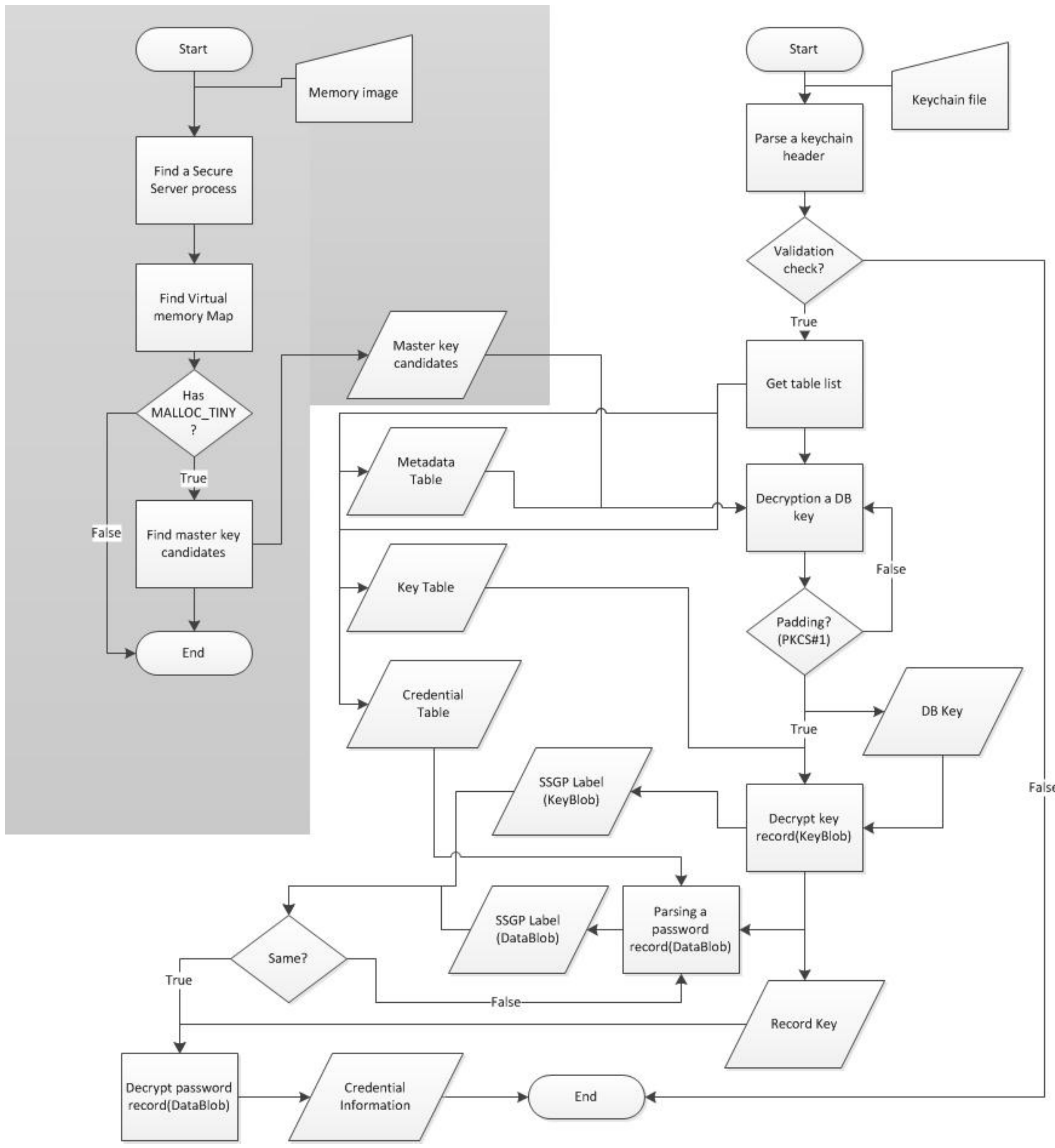


Figure 1. Procedure for Keychain Analysis
(Master key candidates extraction on the left and Keychain analysis on the right)

With master key candidates, it is possible to analyze a Keychain. A Keychain header enables to verify the file, to parse its schema and to extract inner table lists. Now it is time to make an attempt to get a database key, 24 bytes in size for Triple DES symmetric cryptography. We have to learn an appropriate master key from the candidates by checking if there is a valid padding (in the form of PKCS#1) at the end of the excerpt. Then this process repeatedly should be done until the correct master key is found. Once discovered, the database key needs to be decrypted. This can be done with Metadata Table called CSSM_DL_DB_RECORD_METADATA in table lists. After the database key is decrypted, it is feasible to have a series of key records, KeyBlob, from Key Table such as CSSM_DL_DB_RECORD_SYMMETRIC_KEY. Again, the key records can be decrypted by the database key.

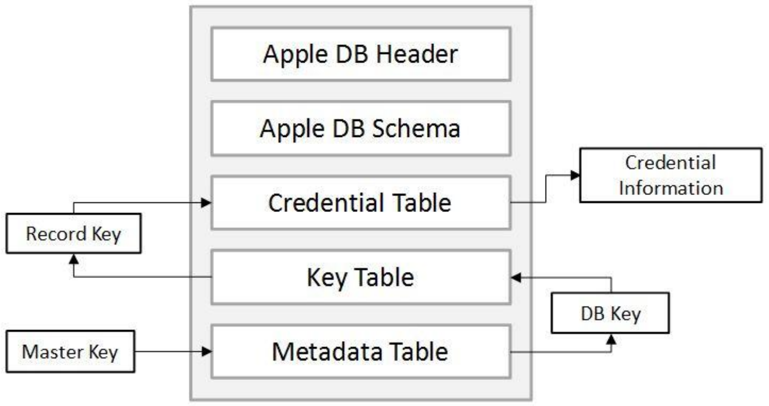


Figure 2. Keychain structure

Each Keyblob in Key Table consists of a record key and a corresponding SSGP label. This label identifies encrypted data, DataBlob, by matching the equivalent of the value in the Credential Table. Ultimately, it is obtainable to gain user credential including stored passwords, WiFi key, and other diverse information in a Keychain.

The most significant part along the process above is to extract the master key and ultimate user credential by thoroughly interpreting a Keychain structure. At the following section, we discuss the methodology of logical analysis and real test results in details.

4. Keychain Analysis with Memory Forensic Techniques

This section demonstrates a Keychain analysis with memory forensic techniques.

4.1. Extracting master key candidates from Physical Memory Image

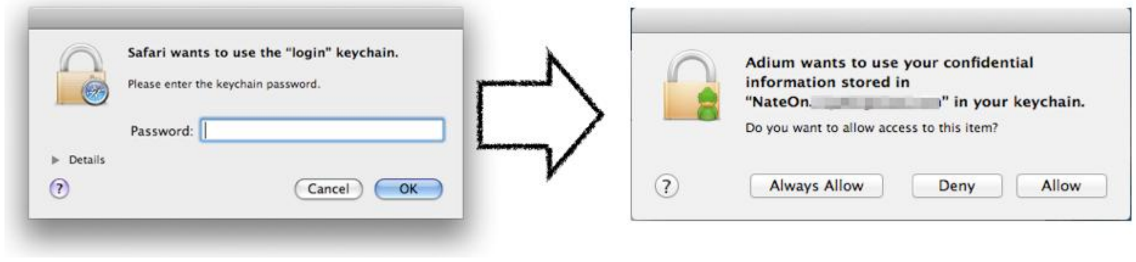
The following summary shows how to extract master key candidates from obtained physical memory image.

(1) Finding a *Security Server* process
(2) Finding MALLOC_TINY area in virtual memory map
(3) Extracting master key candidates

We introduced the process of extracting master key candidates in previous study, so this paper focuses on different part from it.

4.1.1. Finding a *Security Server* Process

Mac OS X system controls security issues in a *Security Server* process. This process takes the request by *Security Agent* and responds its result back on client-server architecture [9]. Especially it maintains diverse data regarding with security in memory space. One of them is a master key related to a Keychain. This key has been generated, based upon user password. When application or user needs the privilege to access stored user credential, OS X Lion or later makes Mac users better, allowing them to access a Keychain simply by clicking “Allow” button instead of typing a password every single time in the past [10]. In other words, the state of a Keychain file is unlocked, which means no password is required, potentially resulting in a master key extraction through a *Security Server* process analysis in memory.


Safari wants to use the "login" keychain.

Please enter the keychain password.

Password:

Adium wants to use your confidential information stored in "NateOn.[redacted]" in your keychain.

Do you want to allow access to this item?

Figure 3. Keychain Access Control Change

The structure of a *Security Server* process can be traced with the kernel symbols. Using *volafox*, Mac OS X memory forensic toolkit, it allows to extract the structure of a *Security Server* process, called '*securityd'* in the '*kernproc*' symbol pointing to a *proc* structure of BSD system [11][12][13].

4.1.2. Finding MALLOC_TINY Area

According to aforementioned *keychaindump*, a *Security Server* stores the master key of a Keychain in MALLOC_TINY area (1MB in size) from heap space in memory. To begin with, the virtual memory space of a *security server* process should be dumped in order to retain the specific area. The Mach, one of Mac OS X kernel components, provides virtual memory management. Each *task* structure in Mach contains the pointer of virtual memory map, or *vmmap*, which represents *task* address space. This means that it is probable to obtain the regions of virtual memory in the process through *vmmap* structure analysis. [14] However, currently *vmmap* structure does not explicably indicate MALLOC_TINY area. Hence we assume that MALLOC_TINY would be a single megabyte(1,024KB) in size. The following summarizes the steps to find MALLOC_TINY area in order.

(1) Finding Task structure from a *security server* process
(2) Getting the pointer of *vmmap* structure consisting of *vm_map_entry* in the form of doubly linked list and *pmap* (physical memory map structure) pointers
(3) Getting the regions of virtual memory by *vm_map_entry* analysis respectively
(4) Defining MALLOC_TINY area if the size of virtual memory region were 1MB to be exact.
(5) Dumping MALLOC_TINY area since the value of CR3 register indicates *pmap* structure.

As a result, we were successfully able to get the regions of virtual memory including MALLOC_TINY area in the experiment.

4.1.3. Extracting master key Candidates

Now master key candidates can be extracted from the obtained MALLOC_TINY area. We are aware that a *Security Server* process keeps this key in the allocated heap space. In MALLOC_TYNY area, when the key length 0x18 is found as 8 bytes pointer move on, the following 24 bytes is defined as one of master key candidates [15]. Interestingly enough, DB key can be obtained as well with the same fashion.

4.2. Keychain Analysis

Keychain files are located in the following.

(1) /Library/Keychains/System.keychain
(2) ~/Library/Keychains/login.keychain

Mac OS X creates a Keychain file while installation process. Each user in Mac OS X is able to create or delete a Keychain file in need. The keychain file at the location (1) keeps certificates for Mac OS X application and security update validation. The other one at the location (2) holds user-generated credential such as user accounts for installed applications, Wi-Fi password or wireless network key, encrypted volume password. Each user owns a *login.keychain* file separately. While Keychain files have the same file format, the record column in the file varies depending on master key and record type. This paper deals with a *login.keychain* file only which contains user credential.

Once a Keychain is obtained, now we have the following encrypted area in the file:

(1) $E_1=E_{KMS}(K_{DB})$,
(2) $E_2=(s_1||E_{KDB}(r_1), s_2||E_{KDB}(r_2), \dots, s_n||E_{KDB}(r_n))$,
(3) $E_3=(s_1||E_{r1}(p_1), s_2||E_{r2}(p_2), \dots, s_3||E_{rn}(p_n))$

where

E: Encryption with Triple DES inCBC mode
K_{MS}: Master Key
K_{DB}: DB Key
$R=\{r|$ record keys per each user credential$\}=\{r_1,r_2, \dots, r_n\}$
$S=\{s|$ SSGP label$\}=\{s_1,s_2, \dots, s_n\}$
$P=\{p|$ user credential in plaintext$\}=\{p_1,p_2, \dots, p_n\}$

The following summarizes how to decrypt necessary keys from each blob structure.

(1) $E_1=E_{KMS}(K_{DB})$
E_1 is stored within DbBlob record in the table from a Keychain.
By decrypting E_1 with extracted master key K_{MS}, we can get K_{DB}.
(2) $E_2=(s_1||E_{KDB}(r_1), s_2||E_{KDB}(r_2), \dots, s_n||E_{KDB}(r_n))$
E_2 is stored within KeyBlob record in the table from a Keychain.
By decrypting E_2 with extracted K_{DB}, we can get partially encrypted "$r_1||s_1, r_2||s_2, \dots, r_n||s_n$".
(3) $E_3=(s_1||E_{r1}(p_1), s_2||E_{r2}(p_2), \dots, s_3||E_{rn}(p_n))$
E_3 is stored within DataBlob record in the table from a Keychain.
By decrypting E_3 with extracted $r_1,r_2, \dots, r_n$, we can eventually get "$p_1||s_1, p_2||s_2, \dots, p_n||s_n$"
(4) Lastly, we can obtain "$p_1,p_2, \dots, p_n$" by matching each SSGP label in (2) and (3).

According to the source code in Apple Open Source website, a Keychain follows Apple Database format [16]. The data in the format adopts big-endian representation.

A Keychain consists of Apple Database Header and Apple Database Schema which includes Database schema, table offsets, and tables in designated offsets. After understanding basic structure of Apple Database Header and Schema, we are quickly able to grasp how to decrypt database key, record keys and user credential in order.

4.2.1. The Basic Structure of a Keychain

Figure 4 describes the structure of Apple Database Header and Schema at the beginning of a Keychain file.

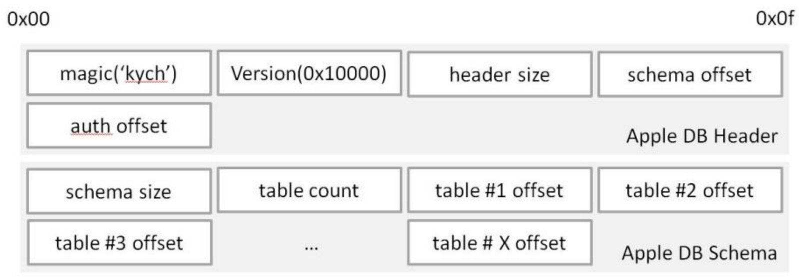


Figure 4. Apple DB Header and Schema

There are five 4-Bytes fields in Apple Database Header: *magic number*, *version*, *header size*, *schema offset*, and *auth offset*. Mac OS X does validation check with the magic number, *kych*, and version information. The *schema offset* points to the starting address of Apple Database Schema. The field, *auth offset*, is not currently in use [17].

Apple Database Schema contains table information. The field, *table count*, literally informs the number of tables. The location of each table can be calculated by adding each *table offset* to the starting address of Apple Database Schema. The Apple Open Source site publishes that Schema management table comes first and user-defined one comes next for the initial launch of Apple Database [18][19].

Table 1. Table Types in Apple Database Schema

Name Space	Record Type (prefix 'CSSM_DL_DB_')	Value	Description
Schema Management	SCHEMA_INFO	0x00000000	Schema information
	SCHEMA_INDEXES	0x00000001	Schema indexes
	SCHEMA_ATTRIBUTES	0x00000002	Schema attributes
	SCHEMA_PARSING_MODULE	0x00000003	Schema parsing module
Open Group Application	RECORD_ANY	0x0000000A	Temporary table type.
	RECORD_CERT	0x0000000B	Certificates
	RECORD_CRL	0x0000000C	Certificate Revocation List
	RECORD_POLICY	0x0000000D	Policy
	RECORD_GENERIC	0x0000000E	Generic information
	RECORD_PUBLIC_KEY	0x0000000F	Public key
	RECORD_PRIVATE_KEY	0x00000010	Private key
	RECORD_SYMMETRIC_KEY	0x00000011	Symmetric key
	RECORD_ALL_KEY	0x00000012	Temporary table type
Industry at Large Applications	RECORD_GENERIC_PASSWORD	0x80000000	User credential
	RECORD_INTERNET_PASSWORD	0x80000001	User credential on the Internet in particular
	RECORD_APPLESHARE_PASSWORD	0x80000002	(Depreciated)
	RECORD_USER_TRUST	0x80000003	User-defined certificates
	RECORD_X509_CRL	0x80000004	X.509 Certificate Revocation List
	RECORD_UNLOCK_REFERRAL	0x80000005	Unlock referral
	RECORD_EXTENDED_ATTRIBUTE	0x80000006	Extended attribute for database management
	RECORD_X509_CERTIFICATE	0x80001000	X.509 Certificates
	RECORD_METADATA	0x80008000	Metadata information

The tables in Apple Database Schema are classified into a couple of Name Spaces: Schema Management, Open Group Application and Industry at Large Applications (**Table 1**). The Schema Management table lies at the-first-four-table-offsets in Apple Database Schema. The record type is defined in the table header. The structure of this table and header looks like **Figure 5**.

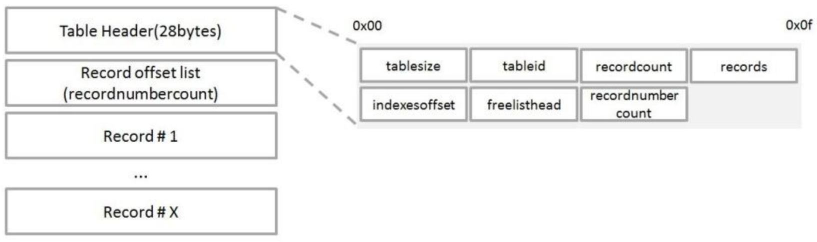


Figure 5. Table and Table Header Structure in Apple Database Schema

The *tableid* field in Table Header represents the record type of each table. The *records* field indicates the first record offset, and the *record offset list* depends on value of *recordnumbercount* field in Table Header. Additionally, each record comprises Record Header and corresponding Record Data. Record Header contains basic information including starting address of record data, column list on record and so on, which varies depending on table structure.

4.2.2. Decrypting database key with master key in DbBlob

Let us look into the table, named CSSM_DB_DL_RECORD_METADATA.(**Figure 6**) As we expected, this table maintains Table Header and records, and each record contains Record Header and Record Data. We call the area DbBlob which contains database key in the table.

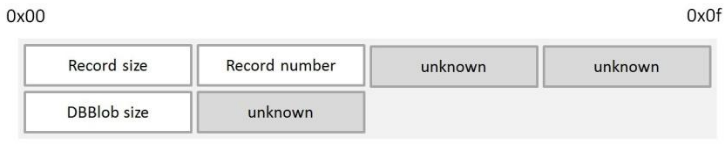


Figure 6. Record Header in METADATA table

We make use of identified three fields: *Record size*, *record number* and *DbBlob size* from Record Header. In Record Data, it contains a salt (20 bytes) for master key generation, encrypted DB Key, and DB Key IV (8 bytes) [20]. The size of encrypted DB key varies, which is determined by *startCryptoBlob* and *totalLength*. This paper does not cover how to generate master key, and we assume that it can be selected from master key candidates in memory image. **Figure 7** illustrates DbBlob structure in details.

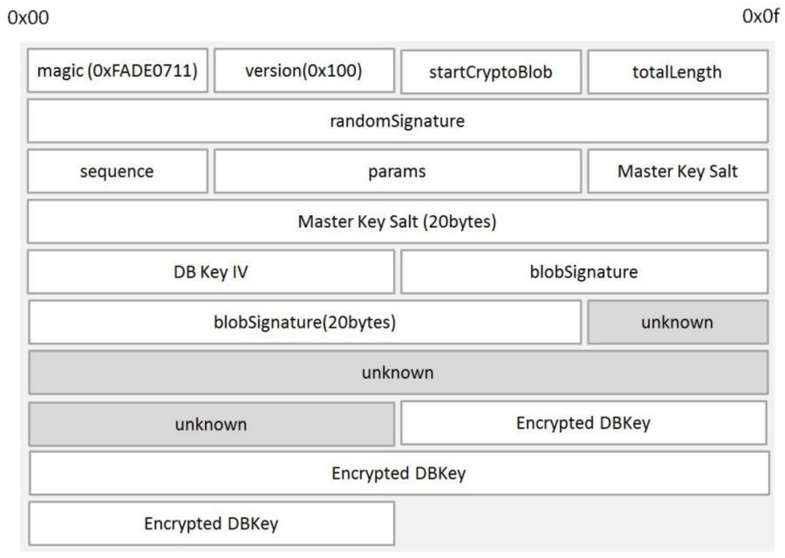


Figure 7. DbBlob Structure

All Blob structures have the first 16 bytes in common and the rest part often varies due to different structures from each table. KeyBlob and DataBlob in the following section have similar structure as well.

Database key encryption applies symmetric key algorithm, 3DES block cipher with CBC mode and PKCS#1 padding technique [21]. Putting database key IV and master key together, we can obtain decrypted database key, which will use at the decryption process of record keys in KeyBlob.

4.2.3. Decrypting record keys with database key in KeyBlob

Once decrypted database key, KeyBlob decryption should be done to attain user credential. KeyBlob is the terminology called by Apple, indicating a chunk of encrypted record keys with database key. What we need is to extract KeyBlob area in a specific table which stores identified record types.

The tables associated with record key are CSSM_DB_DL_PUBLIC_KEY, CSSM_DB_DL_PRIVATE_KEY, and CSSM_DB_DL_SYMMETRIC_KEY. As the name indicates, the first two tables are for asymmetric cryptography and the last table is for symmetric one, which decrypts all password tables in practice. Thus we take a

CSSM_DB_DL_SYMMETRIC_KEY table for this time.

The records in the table contain the offset information pointing to the location of elements in Record Header and actual data at designated offset in Record Data or KeyBlob. We concentrate on two things: record keys to decrypt user credential and *SSGP Label* fields to recognize which encrypted user credential matches with which record key. **Figure 8** shows the structure of record header (0x84 bytes in size).

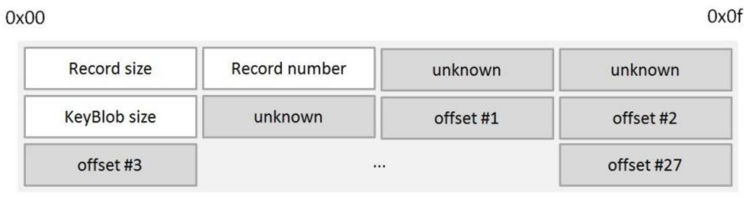


Figure 8. Record Header in SYMMETRIC_KEY table

The very following section by Record Header locates KeyBlob in **Figure 9**. As stated above, the first 16 bytes represents common Blob fields, including StartCryptoBlob and totalLength. This value helps to determine the exact encrypted Key Record range by subtracting startCryptoBlob from totalLength. The size of this range should be a multiple of eight bytes because it is 3DES block cipher output. The next field represents *initial vector* at offset 0x10, followed by common Blob fields. *SSGP Label* is followed by encrypted area with the signature "*ssgp*". If this string is found, the next 16 bytes is identified as *SSGP Label*.

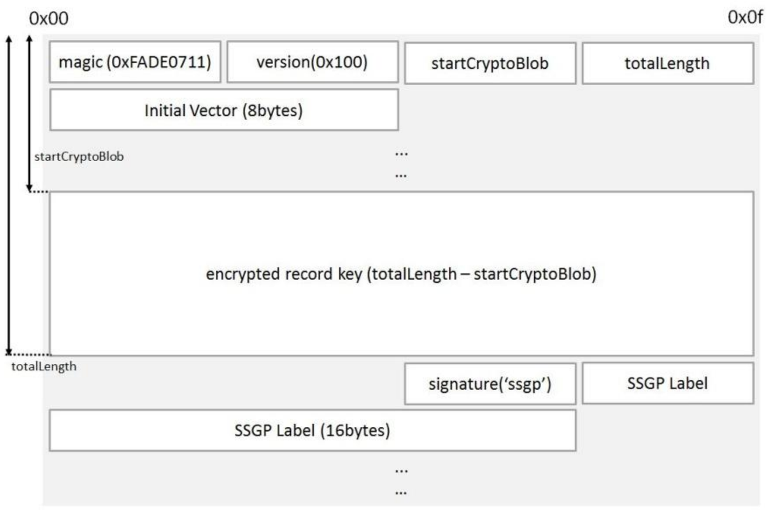


Figure 9. KeyBlob Structure

It is now feasible to decrypt record keys with Database Key and given initial vector in KeyBlob. The cryptography basically uses the same algorithm as database key encryption, 3DES with CBC Mode and PKCS #1 padding. Note that KeyBlob has encrypted twice in the **Figure 10** [22].

Hence we need to decrypt key record area in KeyBlob twice: both use the same DB Key but different IV. The first decryption process employs the fixed IV, magicCmsIv (0x4adda22c79e82105). And the second one employs the extracted IV from KeyBlob structure. Make sure that the input of the second decryption takes the reversed order of the octets from the first output. Eventually the record key (24 Bytes in size) is returned. The following section shows how to obtain user credential with *SSGP label* and the record keys from KeyBlob.

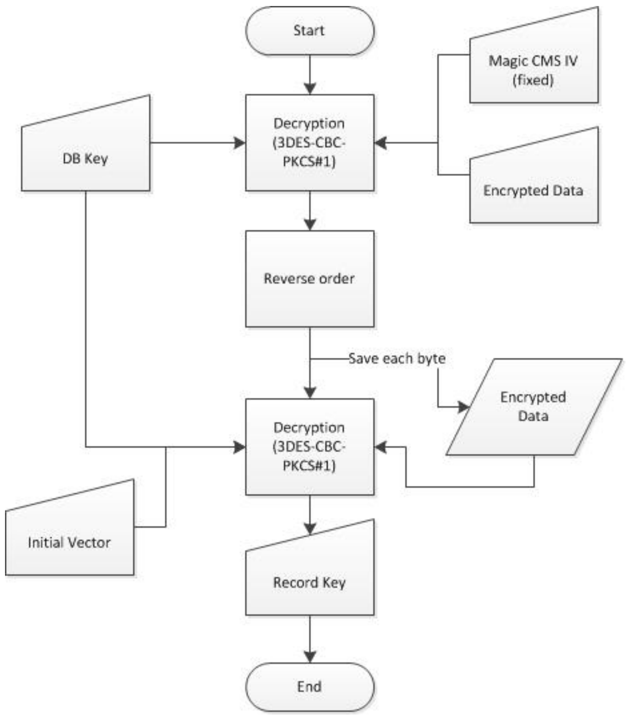
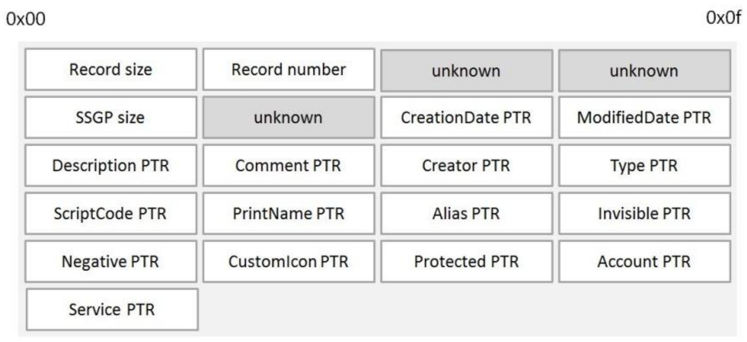


Figure 10. KeyBlob Decryption Process

4.2.4. Decrypting User Credential with record key in DataBlob

This section explains the final step to extract user credential with aforementioned *SSGP label* and decrypted record key. A Keychain manages user passwords in three tables: CSSM_DL_DB_GENERIC_PASSWORD, CSSM_DL_DB_INTERNET_PASSWORD and CSSM_DL_DB_APPLESHARE_PASSWORD. The last table AppleShare passwords, are no longer in use unless application takes lower-level API because they are stored as Internet password items [23].

Although the structure of these tables are similar to that of KeyBlob table, they store more columns to provide more information. The Schema namespace in KeychainCore class defines these columns [24]. This paper predominately focuses on Generic Password table and Internet Password table, which holds most user credential. (**Figure 11 and 12**)



0x00			0x0f
Record size	Record number	unknown	unknown
SSGP size	unknown	CreationDate PTR	ModifiedDate PTR
Description PTR	Comment PTR	Creator PTR	Type PTR
ScriptCode PTR	PrintName PTR	Alias PTR	Invisible PTR
Negative PTR	CustomIcon PTR	Protected PTR	Account PTR
Service PTR			

Figure 11. Record Header in Generic Password table

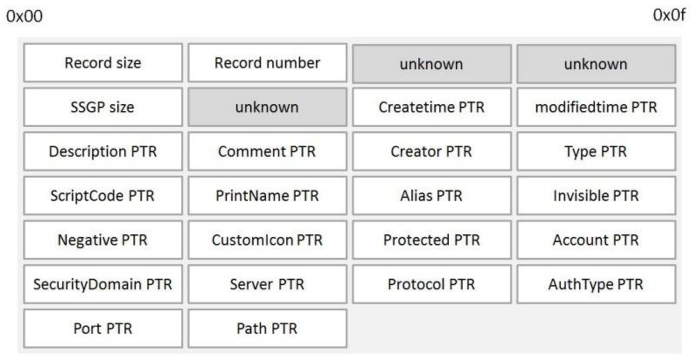


Figure 12. Record Header in Internet Password table

Each record header of two tables has actual data offsets or pointers on pre-defined column. They have many columns in common, but the finding shows that Internet Password table has more. Note that actual data is stored at the offsets subtracted by one respectively. The **Table 2** describes useful columns in the table to contain user credential.

Table 2. Useful Columns on Password Tables

Record Type (prefix 'CSSM_DL_DB')	Data	Description
Common Columns in PASSWORD table	Create Date	Creation date and time (GMT +0)
	Modified Date	Last modified date and time (GMT +0)
	Description	Description for the record
	Comment	Additional description for the record
	Creator	The object to create the record
	Type	Record type
	PrintName	Keychain name shown by *Keychain Access* tool
	Alias	Record alias
	Account	Account name (e.g. UserName)
Additional Columns in GENERIC_PASSWORD table	Service	Service name to create a Keychain record
Additional Columns in INTERNET_PASSWORD table	Server	Destination domain or IP address which connect to.
	Protocol	Protocol type (e.g SSH, HTTP, SSL)
	AuthType	Authentication type (e.g Basic, Non)
	Port	Port number (Web-form always has the value of 0x00)
	Path	Application path for the record

DataBlob is the terminology also called by Apple, indicating a chunk of encrypted user credential with record key and fields defined in Record Header. **Figure 13** illustrates SSGP structure, which directly follows by Record Header. The size of SSGP comes from Record Header.

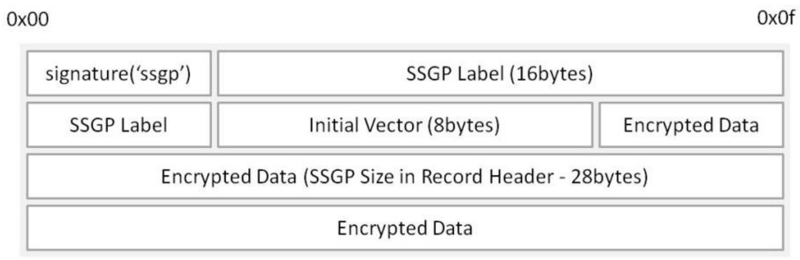
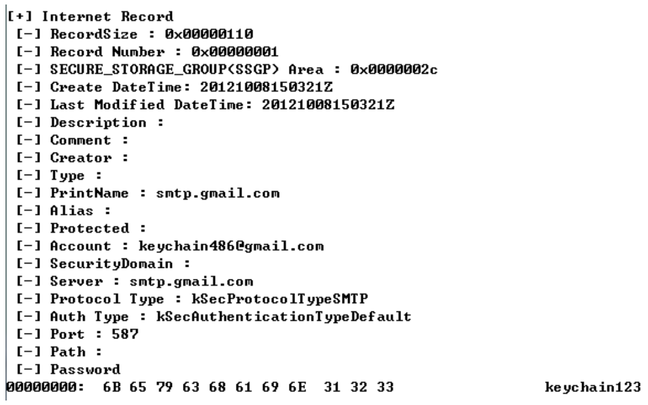


Figure 13. SSGP Structure in DataBlob

SSGP Label, 16 bytes in length, can be used as an identifier by matching the corresponding record key.

Since the encryption also uses 3DES-CBC-PKCS#1 algorithm, we can acquire decrypted user credential with the IV from DataBlob and obtained record keys in **4.2.3**. The user data in the PASSWORD tables can be obtained through this process. In **Figure 14**, the output shows one of the records in Internet Password table.



```
[+] Internet Record
 [-] RecordSize : 0x00000110
 [-] Record Number : 0x00000001
 [-] SECURE_STORAGE_GROUP(SSGP) Area : 0x0000002c
 [-] Create DateTime: 20121008150321Z
 [-] Last Modified DateTime: 20121008150321Z
 [-] Description :
 [-] Comment :
 [-] Creator :
 [-] Type :
 [-] PrintName : smtp.gmail.com
 [-] Alias :
 [-] Protected :
 [-] Account : keychain486@gmail.com
 [-] SecurityDomain :
 [-] Server : smtp.gmail.com
 [-] Protocol Type : kSecProtocolTypeSMTP
 [-] Auth Type : kSecAuthenticationTypeDefault
 [-] Port : 587
 [-] Path :
 [-] Password
00000000:  6B 65 79 63 68 61 69 6E  31 32 33                keychain123
```

Figure 14. Decrypted User Password from *login.keychain* file

5. Implementation

5.1. The *Keychaindump* module in the *volafox* project for extracting master key candidates

In order to extract master key candidates from physical memory, we wrote *keychaindump* module in the *volafox* utility known as Mac OS X memory forensic toolkit. As we mentioned earlier, master key candidates can be acquired if two parameters are provided like this:

# volafox –i [memory_image] –o keychaindump

The *volafox* has been written in Python by Kyeongsik Lee as a cross platform open source project. It is now available at "http://code.google.com/p/volafox".

5.2. *Chain Breaker*, Tool for Keychain Forensics

Chain Breaker is the tool to extract user credential in a Keychain file with master key and/or DB key candidates, extracted from *volafox keychaindump* module. The **Table 3** enumerates the features of this tool.

Table 3. Features of *Chain Breaker*

Category	Description
Target	A Keychain file for each login user
Features	[database key Decryption] - database key decryption with master key candidates - database key decryption with user password
	[Generic Password Decryption] - SSH account including password - Wireless AP SSID and password - Application UserName and password - Email / Calendar related ID and password - Messenger(eg. Adium, etc) ID and password - FaceTime / iCloud / iMessage - Encrypted Volume password - RSA Public/Private Key Pair

	[Internet Password Decryption] - Google Chrome SafeStorage Key - Safari Webform Auto Fill ID and password - SSH account including password - HTTPS/HTTP/GitHub/svn related to account including password
	[Apple Share Password Decryption] - Not Applicable

This open-source tool is currently available at github site, "https://github.com/n0fate/chainbreaker". We wrote it in Python, supporting cross platform. We will re-implement this tool in GUI (using QT) for the user-friendly purpose. While *Chain Breaker* merely provides passwords generated by user, later on it supports decryption for all information stored in a Keychain.

6. Experiment

We conducted a series of experiments to prove what we have presented so far. The five different version of target Mac OS were: Mac OS X Lion (version 10.7.5) and Mountain Lion (version 10.8.2) on physical machines, and Snow Leopard (10.6.3), Lion (10.7.5), and Mountain Lion (10.8.2) on virtual machines. In order to gather master key candidates, each memory had to be dumped while user logged in respectively.

Using Mac Memory Reader or Inception tool, we were able to dump memory image on physical machine. However, Inception did not fully support memory dump due to IOKit stack bug [25]. Therefore we used Mac Memory Reader by Hajime Inoue to dump physical memory [26]. Then, a Keychain file, *login.keychain* was extracted on a live state. In case of virtual machine, we dumped parallels memory image (which has "mem" file extension) under suspended state and extracted a *login.keychain* file.

Table 4. Experiment: Test Set

Targets	Version (Darwin Version)	Memory Size (bytes)	Number of passwords stored in a Keychain file
Physical Machine (Macbook Pro)	Lion (10.7.5)	8,498,995,200	25(Generic: 13,Internet: 12)
	Mountain Lion (10.8.2)	8,498,995,200	47(Generic: 30,Internet: 17)
Virtual Machine (VMWare)	Snow Leopard (10.6.3)	1,073,741,824	2(Generic: 2,Internet: 0)
	Lion (10.7.5)	2,147,483,648	18(Generic: 12,Internet: 6)
	Mountain Lion (10.8.2)	2,147,483,648	18(Generic: 12,Internet: 6)

The experiment comprised two phases: extracting master key candidates from dumped memory and decrypting user credential from an extracted Keychain file. **Table 4 and 5** shows the test set on the experiment and its results.

Table 5. Experiment: Result of extracting master key candidates with *volafox*

Targets	Version (Darwin Version)	Master/DB key candidates	Master key Validation (count)	DB Key Validation (count)
Physical Machine (Macbook Pro)	Lion (10.7.5)	12	True (1)	True (1)
	Mountain Lion (10.8.2)	8	True (1)	True (2)
Virtual Machine (VMWare)	Snow Leopard (10.6.3)	0	False	False
	Lion (10.7.5)	7	True (1)	True (1)
	Mountain Lion (10.8.2)	8	True (1)	True (1)

As expected, we could extract master key and DB key candidates from the image except Mac OS X Snow Leopard. This is normal because Mac OS X has begun to store master key from the Lion / ML or later. With these candidates, we successfully performed the decryption of user credential using *Chain Breaker*. Since it was unlikely to extract master key candidates from Snow Leopard, we entered a valid password and finally obtained user credential as well. Obviously, **Table 6** displays the end results to decrypt passwords and keys found in a Keychain.

Table 6. Experiment: Result of decrypting user's credential with *Chain Breaker*

Targets	Version (Darwin Version)	Test Sets	User credential found in a Keychain file
Physical Machine (Macbook Pro)	Lion (10.7.5)	25(Generic: 13,Internet: 12)	25(Generic: 13,Internet: 12)
	Mountain Lion (10.8.2)	47(Generic: 30,Internet: 17)	47(Generic: 30,Internet: 17)
Virtual Machine (VMWare)	Snow Leopard (10.6.3)	2(Generic: 2,Internet: 0)	2(Generic: 2,Internet: 0)
	Lion (10.7.5)	18(Generic: 12,Internet: 6)	18(Generic: 12,Internet: 6)
	Mountain Lion (10.8.2)	18(Generic: 12,Internet: 6)	18(Generic: 12,Internet: 6)

7. Further Research

The *keychaindump* module from *volafox* shows multiple master key candidates, which might lead false master keys to make an attempt. This happens because we do not have accurate extraction process. Although it provides small number of candidate keys, it is necessary to organize this part for further analysis.

Chain Breaker has been designed to mostly extract user credential, since it is most frequently used information. We have a plan to add features to parse other information as well, such as Apple-issued certificate. Although our tool might not work for new Mac OS X version, we believe to improve both features and performances. Based on a demystified Keychain structure including table structure and records, *Chain Breaker* will support additional features in the future.

8. Conclusions

Historical digital forensics has been mainly focused on Microsoft windows systems with high market share. Yet, even today, digital forensics in Mac OS X has been conducted in a limited scope other than memory forensics such as disk forensics technology with common API and/or existing technique reuse. Additionally, it is necessary to have advanced analysis of Mac OS X artifacts.

Mac OS X as well as other OS maintains a variety of artifacts. In particular, application accounts – IDs and passwords - could be major artifacts in practice in order to gather significant evidence such as e-mail, individual notes (e.g. evernote) and so on. A Keychain, the Password Management System on Mac OS X, holds user credential as integrated management system. Because this system encrypts all information based on user password, application needed password lively whenever getting access to a Keychain at all times in Snow Leopard or earlier. Change has been made since Mac OS X Lion and ML or later for user convenience, and this allows application to gain access to a Keychain with one mouse click instead of typing user password. Technically speaking, this means the master key of a Keychain is loaded into somewhere in memory so that Mac OS X can use it repetitively when needed.

Based on this idea, this paper suggested a brand new method to extract master key with memory forensics and thereby to decrypt user credential in a Keychain file with disk forensics. This allows investigators to perform forensic analysis on a separate Mac OS X.

At first, we extracted master key candidates from dumped memory image and choose one. Using *keychaindump* module from *volafox* helped to extract master key candidates. Then database key was decrypted with master key, and subsequently each record key was decrypted with database key. Using *Chain Breaker* helped to learn record key. Finally, user credential was decrypted with record key in sequence. We wrote *keychaindump* module and *Chain Breaker* to achieve our goal. As a result, it was feasible to decrypt user credential in a Keychain.

By utilizing the technique of this paper, it is practicable to acquire external evidence from other services because it allows to investigate the accounts user registered. Taking cloud service boosting into account, we have no doubt that our technique will assist investigators to examine further evidence.

Acknowledgments

This research was supported by Agency for Defense Development (ADD).

References

[1] Desktop Top Operating System Share Trend. April 2012 to February 2013. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=0.

[2] Mobile/Tablet Top Operating System Share Trend. April 2012 to February 2013. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1.

[3] Keychain Access. Security Overview. https://developer.apple.com/library/mac/documentation/security/conceptual/security_overview/Security_Overview.pdf. 2012.

[4] OS X Security. Apple Technical White Paper. http://training.apple.com/pdf/wp_osx_security.pdf. 2012.

[5] security(1) OS X Manual Page, Mac Developer Library. http://developer.apple.com/library/mac/#documentation/Darwin/Reference/ManPages/man1/security.1.html.

[6] Keychain Access, Wikipedia. http://en.wikipedia.org/wiki/Keychain_Access.

[7] Matt Johnston. extractkeychain. http://www.ucc.gu.uwa.edu.au/~matt/src/extractkeychain-0.1; 2004

[8] Juuso Salonen. Breaking into the OS X keychain. http://juusosalonen.com/post/30923743427/breaking-into-the-os-x-keychain; 2012

[9] Security Server and Security Agent, Mac Developer Library. https://developer.apple.com/library/mac/#documentation/Security/Conceptual/Security_Overview/Architecture/Architecture.html.

[10] 6 things that can be done to secure a Mac. http://www.sas.upenn.edu/~jasonrw/6ssym.html.

[11] volafox – Mac OS X Memory Analysis Toolkit. http://code.google.com/p/volafox.

[12] Kyeongsik Lee, Sangjin Lee. Research on Mac OS X Physical Memory Analysis. Korea Institute of Information Security & Cryptology; 21; 4:S89-100.

[13] volafox: Support New OS!! Mountain Lion xD(Korea Language). n0fate's forensic space. http://forensic.n0fate.com/2012/08/volafox-support-new-os-mountain-lion-xd.html.

[14] Jonathan Levin. Chapter 12: Commit to Memory: Mach Virtual memory. Mac OS X and iOS Internals: To the Apple's Core. Wrox. 2012. P. 447-465

[15] volafox: decrypting the keychain file using volafox(Korea Language). n0fate's forensic space. http://forensic.n0fate.com/2012/09/volafox-decrypting-keychain-file-using.html.

[16] AppleDatabase.h. Apple Open Source. http://www.opensource.apple.com/source/libsecurity_filedb/libsecurity_filedb-55016/lib/AppleDatabase.h.

[17] DbVersion::open(). AppleDatabase.cpp. http://www.opensource.apple.com/source/libsecurity_filedb/libsecurity_filedb-55016/lib/AppleDatabase.cpp.

[18] AppleFileDL Record Types. Apple Open Source. http://www.opensource.apple.com/source/libsecurity_cssm/libsecurity_cssm-6/lib/cssmapple.h.

[19] CSSM DB Record Types. Apple Open Source. http://www.opensource.apple.com/source/libsecurity_cssm/libsecurity_cssm-6/lib/cssmtype.h

[20] ssblob.h, Apple Open Source. http://www.opensource.apple.com/source/libsecurityd/libsecurityd-36988/lib/ssblob.h.

[21] BLOBFORMAT, Apple Open Source, http://www.opensource.apple.com/source/securityd/securityd-55137.1/doc/BLOBFORMAT.

[22] wrapKeyCms.cpp, Apple Open Source, http://www.opensource.apple.com/source/Security/Security-28/AppleCSP/AppleCSP/wrapKeyCms.cpp.

[23] Keychain Services Programming Guide, Apple Developer, 2012. P. 18-19.

[24] Schema.m4, Apple Open Source, http://www.opensource.apple.com/source/Security/Security-55179.1/include/security_cdsa_utilities/Schema.m4.

[25] Inception – Known bugs, http://www.breaknenter.org/projects/inception/#Known_bugs.

[26] Hajime Inoue, Frank Adelstein, Robert A. Joyce, Visualization in testing a volatile memory forensic tool. Digital Investigation; 8:S42-51.