Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software

Daniel Plakosh
Scott Hissam
Kurt Wallnau

March 1999

COTS-Based Systems Initiative

Technical Note
CMU/SEI-99-TN-010

Contents

List of Figures

Abstract

We were recently involved with a project that faced an interesting and not uncommon dilemma. The project needed to programmatically extract private keys and digital certificates from the Netscape Communicator v4.5 database. Netscape documentation was inadequate for us to figure out how to do this. As it turns out, this inadequacy was intentional—Netscape was concerned that releasing this information might possibly violate export control laws concerning encryption technology. Since our interest was in building a system and not exporting cryptographic technology, we decided to further investigate how to achieve our objectives even without support from Netscape. We restricted ourselves to the use of Netscape-provided code and documentation, and to information available on the Web. Our objective was to build our system, and to provide feedback to Netscape on how to engineer their product to provide the capability that we (and others) need, while not making the product vulnerable or expose the vendor to violations of export control laws. This paper describes our experiences peering "into the black box."

1 Introduction

The use of commercial off-the-shelf (COTS) software products can reduce the time and cost of developing software, assuming that developers know how to make full use of the product. COTS product vendors often supply only user-level documentation. In most cases, this level of documentation is adequate, but in some instances the developer may need information about the internal operation of a product, its performance characteristics, and perhaps internal data formats. COTS software vendors are often reluctant to release such information because it may have proprietary value. Nevertheless, it is sometimes necessary for the developer to probe into a COTS product to obtain needed functionality or understanding in order to effectively use the product.

Such was the case in one of our projects. We needed to programmatically extract private keys and certificates from the Netscape Communicator (version 4.5) internal databases. The Netscape certificate database (cert7.db) and key database (key3.db) contain certificates and private keys that are ultimately used to provide authentication and secure communication. Netscape does *not,* however, make the format of their key database (key3.db) and certificate database (cert7.db) publicly available because releasing this information could *possibly* violate the International Trade and Export Regulations (ITAR) regarding key management in cryptographic systems.

This report describes what we did to gain insight into Netscape's Communicator databases, the internal formats of the databases, and the password and encryption schemes used in the key3.db database. Note that we did *not* disassemble any Netscape software products. We limited ourselves to documentation and other resources provided by Netscape and to resources that we could obtain from the Web. The results of our work can *not* be used in any manner to subvert or crack the standard encryption algorithms used by Netscape Corporation in the protection of certificate and key material stored in the Communicator's databases.

The rest of this report is organized of as follows: In Section 2, we describe the database used by Netscape. Section 3 describes the record formats of the certificate database. In Section 4 we describe the key database record formats and the encryption algorithm used to encrypt private keys. Finally, we present our summary in Section 5.

2 Database

The first step in decoding these databases was to determine the type of database system that Netscape used to store information. If Netscape used a proprietary database, this step was going to be difficult. We recalled that Netscape released some initial source code of their Mozilla browser. Although the released source code did not contain support for security, we suspected that Netscape used the same database to store more than just security-related items. If this suspicion held true, we could take advantage of our knowledge of this implementation detail to gain programmatic access to the Netscape databases.

We downloaded the Mozilla source, unzipped it and discovered a directory named "dbm." After a closer investigation, we discovered that the files in the dbm directory were the source code files for the Berkeley DB 1.85 database. Next, we built a library from the source for the Berkeley DB 1.85. We wrote a simple test program called "`DBDump`" (see Figure 1) to open a database, dump all records, and access keys in binary form.

The Berkeley DB 1.85 database supports three different types of databases files:

- `DB_HASH`- allows arbitrary key/data pairs to be stored in data files
- `DB_BTREE` - allows arbitrary key/data pairs to be stored in a sorted, balanced binary tree
- `DB_RECNO` – allows both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in `DB_HASH` and `DB_BTREE`. For `DB_RECNO,` the key will consist of a record (line) number

The test program executed successfully on both the key (key3.db) and certificate (cert7.db) databases. Thus, we determined that the Berkley DB 1.85 was the database system Netscape used to create, access and modify the databases. Figure 2 shows the output from the "`DBDump`" program when given a key3.db file as input. Both the certificate and key databases are in the DB_HASH format.

```
//---------------------------------------------------------------------------
#pragma hdrstop
#include <condefs.h>
#include <stdio.h>
#include <ctype.h>
#include "mcom_db.h"
//---------------------------------------------------------------------------
USELIB("..\lib\dbm\dbmlib.lib");
//---------------------------------------------------------------------------
void dumphex(unsigned char *dptr,int size);
//----------------------------------------------------------------------------
#pragma argsused
int main(int argc, char **argv) {
 static HASHINFO   hash_info = {16*1024,0,0,0,0,0};
 DB              * db;
 int               status,record=R_FIRST,cnt=0;
 DBT               key,data;

 if (argc!=2) {
     fprintf(stderr,"%s <filename>",argv[0]);
     return(-1);
 }
 if ((db=dbopen(argv[1],O_RDONLY,0644,DB_HASH,&hash_info))==NULL) {
     fprintf(stderr,"Database open error\n");
     return(-1);
 }
 while ((status=(*db->seq)(db,&key,&data,record))==0) {
     printf("Record %d\nKey Data: (%d bytes)\n",++cnt,key.size);
     dumphex((unsigned char *)key.data,key.size);
     printf("Record Data: (%d bytes)\n",data.size);
     dumphex((unsigned char *)data.data,data.size);
     printf("\n\n");
     record=R_NEXT;
 }
 db->close(db);
 if (status<0) {
     fprintf(stderr,"Database sequence error");
     return(-1);
 }
 return(0);
}
//---------------------------------------------------------------------------
void dumphex(unsigned char *dptr,int size) {
 int cnt,counter=0;

 while(size>0) {
  (size>16)? cnt=16 :cnt=size;
  printf("%08lx   ",counter);
  for (int i=0;i<cnt;i++) printf("%02x ",dptr[counter+i]);
  for (int i=0;i<16-cnt;i++) printf("   ");
  printf(" ");
  for (int i=0;i<cnt;i++)
      (isprint(dptr[counter+i])) ? printf("%c",dptr[counter+i]):printf(".");
  printf("\n");
  counter+=16;
  size-=16;
 }
 return;
}
//---------------------------------------------------------------------------
```

Figure 1: DBDump.c Code

```
Record 1
Key Data: (7 bytes)
00000000   56 65 72 73 69 6f 6e                             Version
Record Data: (1 bytes)
00000000   03                                               .


Record 2
Key Data: (11 bytes)
00000000   67 6c 6f 62 61 6c 2d 73 61 6c 74                 global-salt
Record Data: (16 bytes)
00000000   d4 b4 e9 b8 d2 6c 78 ad b9 28 e0 52 36 48 3b b7  .....lx..(.R6H;.


Record 3
Key Data: (14 bytes)
00000000   70 61 73 73 77 6f 72 64 2d 63 68 65 63 6b        password-check
Record Data: (48 bytes)
00000000   03 10 01 ea f1 02 3f c8 d9 3c 3b 86 b8 53 3f 2d  ......?..<;..S?-
00000010   0d 52 6c 00 0b 2a 86 48 86 f7 0d 01 0c 05 01 03  .Rl..*.H........
00000020   5e ed a0 c0 65 d1 39 0f e3 7a 37 ed 99 76 7b 1c  ^...e.9..z7..v{.


Record 4
Key Data: (65 bytes)
00000000   00 b1 e0 ad 39 e7 09 41 b9 d3 21 90 9b 0f 95 78  ....9..A..!....x
00000010   e6 fd ef d3 62 34 51 4d 79 02 83 17 9f 4f 09 68  ....b4QMy....O.h
00000020   5c 81 a2 e6 2d b1 f7 bb e6 69 ba 39 a5 f4 17 0b  \...-....i.9....
00000030   a9 a9 ea b0 4c 7f ff 55 a5 46 a7 67 10 3a 1f e1  ....L..U.F.g.:..
00000040   7b                                               {
Record Data: (436 bytes)
00000000   03 08 23 47 eb a8 ce fc 4b c0 6b 53 63 6f 74 74  ..#G....K.kScott
00000010   20 41 20 48 69 73 73 61 6d 27 73 20 56 65 72 69   A Hissam’s Veri
00000020   53 69 67 6e 2c 20 49 6e 63 2e 20 49 44 00 30 82  Sign, Inc. ID.0.
00000030   01 82 30 1c 06 0b 2a 86 48 86 f7 0d 01 0c 05 01  ..0...*.H.......
00000040   03 30 0d 04 08 47 eb a8 ce fc 4b c0 6b 02 01 01  .0...G....K.k...
00000050   04 82 01 60 bf 3e 52 71 3e 07 94 73 25 f2 28 8d  ...‘.>Rq>..s%.(.
00000060   06 d6 1e f8 b3 ec fa 59 17 06 ec f9 8f 92 19 fe  .......Y........
00000070   4c ff c3 81 f8 be f0 12 a2 dd 6a d3 17 da 56 5a  L.........j...VZ
00000080   b4 65 8b e8 5d 6f 4b ae 6f 5f 39 dc 1f ef bf 56  .e..]oK.o_9....V
00000090   6e 79 d5 b4 2b 9a 6e 20 98 4d 66 98 79 4c 85 98  ny..+.n .Mf.yL..
000000a0   31 1d 4b e3 de ef c3 07 54 76 86 50 a8 22 9e 94  1.K.....Tv.P."..
000000b0   c8 cb f9 f4 46 9e 52 26 f8 20 8c 51 e8 52 6e 95  ....F.R&. .Q.Rn.
000000c0   16 ca 9d 4d e6 7e 90 69 96 1e 1e df cc 67 fe ab  ...M.~.i.....g..
000000d0   96 5a d7 88 26 1a a9 cc 52 f6 97 0f 28 fc 52 96  .Z..&...R...(.R.
000000e0   de fb fd f7 87 01 ae 71 e0 88 1b c6 7d 01 c8 83  .......q....}...
000000f0   27 40 36 a3 46 23 dd 64 86 64 f7 64 73 46 04 30  ’@6.F#.d.d.dsF.0
00000100   3a 96 71 33 7e 98 f1 be 18 b9 8b 10 da ff fa 32  :.q3~..........2
00000110   ac 03 18 37 da 87 32 5f eb f7 ed 0d 37 b2 1b 97  ...7..2_....7...
00000120   35 d6 38 f2 f8 cc 4e 2d 00 e2 43 f1 6f 02 b2 fd  5.8...N-..C.o...
00000130   94 53 9d 7b 78 00 4d fb 4d 47 63 6e b9 65 92 4c  .S.{x.M.MGcn.e.L
00000140   03 c2 a6 9f 20 59 80 d5 a0 d4 b2 79 51 6e 31 b6  .... Y.....yQn1.
00000150   20 d4 a9 43 80 31 ce c6 93 0c b0 1e 2f 13 3f c3   ..C.1....../.?.
00000160   c0 e0 7b 16 89 76 88 dd 38 d6 8f 2b 5f 6f 50 1d  ..{..v..8..+_oP.
00000170   f7 48 d9 2e 89 c2 04 1f 78 6b ac 85 97 55 0f 71  .H......xk...U.q
00000180   be 5d d2 c7 c8 22 41 b6 c9 a0 c9 81 cd 93 55 83  .]..."A.......U.
00000190   d2 9d e3 00 63 72 4f 79 d4 e9 ad 1d 1e cd 79 3f  ....crOy......y?
000001a0   89 9a 66 e4 f6 a2 1d ec a0 3e 61 35 81 cc b8 83  ..f......>a5....
000001b0   5c df 87 24                                      \..$
```

Figure 2: Output of DBDump (key3.db File as Input)

3 Certificate Database

The next step was to determine the format of the data and access keys for each database record.

Decoding the certificate database was much easier than expected. We searched the Web and newsgroups using most of the available search engines for information describing Netscape's certificate database. Combinations of the keywords such as *cert7.db, decode, ASN.1, DER, certificate-database, format, specification, certificate, security*, and *Netscape* were used as input into the search engines.

It turned out that some information describing the content and format of the Netscape certificate database was available on the Internet. All records in the certificate database have a common header that describes the type of record. This information was described in some detail at the following Web sites (note that one of these sites was overseas, thus calling into question whether export control laws are material insofar as Netscape's product are concerned):

- http://www.drh-cosultancy.demon.co.uk/cert7.html
- http://www.columbia.edu/~ariel/good-certs/
- http://www.netscape.com/eng/security/downloadcert.html

The information at these Web sites did not describe every field of the header or every field of each record. We then obtained a copy of the Netscape Security Services (NSS) library from Netscape. It turned out that Netscape documented, to a certain extent, the exact format of the common header as well as the format for each possible type of record in the database. The common header as shown in Figure 3 has the following fields:

1. a Version field that indicates the database version (currently 7)
2. a Type field that indicates the *type* of record
3. a Flags field (always zero)

```
typedef struct
 {
  unsigned char Version;
  unsigned char Type;
  unsigned char Flags;
 } DBHeader;
```

Figure 3: Certificate Database Record Type Header

Using some of the NSS header files, we determined the list of possible record *types* (the Type field in Figure 3) in the certificate database as shown in Figure 4. Some of this information was also defined in the Internet resources that we located.

```
// Record Types
#define CERT7VERSION        0
#define CERT7CERTIFICATE    1
#define CERT7NICKNAME       2
#define CERT7SUBJECT        3
#define CERT7REVOCATION     4
#define CERT7KEYREVOCATION  5
#define CERT7SMIMEPROFILE   6
#define CERT7CONTENTVERSION 7
```

Figure 4: Certificate Database Record Types

Then we focused on determining the format of each record. This task was simple thanks to Netscape's NSS header files. Figure 5 shows the C structures that define the format of each record type in the database. These structures were derived using Netscape's header files that document the byte offsets of fields within a record and hexadecimal dumps from the "DBDump" tool described earlier. Records in the certificate database are in big endian format, so all fields that are of the type "unsigned short" must be byte swapped. Most of the important information contained within a record is distinguished encoding rules (DER) encoded.

Two records that are always in the database are the `CERT7VERSION` and `CERT7CONTENTVERSION` records. These records have the access key `"\0Version\0"` and `"\7ContentVersion\0"` respectively and may be used to identify a certificate database.

Now that we had determined the record formats for the certificate database, a tool to browse the database was constructed. This tool (shown in Figure 6) displays to the user a listing of each record in the database. The user can then select a particular record and the tool will display the key index for the record as well as its contents. Record fields that are DER encoded can be displayed in abstract syntax notation one (ASN.1) or Hex/ASCII format. Additionally, the tool allows the user to save a certificate to a file in DER format.

```
#define CERTIFICATEHEADERFIXEDSIZE 10

// Flags for Object Signing, E-mail and SSL
#define CERT7DB_VALID_PEER        (1<<0)
#define CERT7DB_TRUSTED           (1<<1)
#define CERT7DB_SEND_WARN         (1<<2)
#define CERT7DB_VALID_CA          (1<<3)
#define CERT7DB_TRUSTED_CA        (1<<4)
#define CERT7DB_NS_TRUSTED_CA     (1<<5)
#define CERT7DB_USER              (1<<6)
#define CERT7DB_TRUSTED_CLIENT_CA (1<<7)
#define CERT7DB_INVISIBLE_CA      (1<<8)
#define CERT7DB_GOVT_APPROVED_CA  (1<<9)
#define CERT7DB_PROTECTED_OS_CA   (1<<10)

typedef struct
 {
    unsigned short SSLFlags;
    unsigned short EMailFlags;
    unsigned short ObjectSigningFlags;
    unsigned short DERCertificateLength;
    unsigned short NickNameLength;
    unsigned char *DERCertificate;
    char          *Nickname;
 }CertificateHeader;

#define NICKNAMEHEADERFIXEDSIZE 2
typedef struct
 {
   unsigned short NickNameDERLength;
   unsigned char *NicknameDER;
 }  NickNameHeader;

#define SUBJECTHEADERFIXEDSIZE 6
typedef struct
 {
  unsigned short   NumberOfCertificates;
  unsigned short   NicknameLength;
  unsigned short   EmailAddressLength;
  char *           NickName;
  char *           EMailAddress;
  unsigned short * CertificateKeyLength;
  unsigned short * KeyIDLength;
  unsigned char  * CertificateKeys;
  unsigned char  * KeyIDs;
 }SubjectHeader;

#define MIMEHEADERFIXEDSIZE 6
typedef struct
{
  unsigned short  DERSubjectNameLength;
  unsigned short  MineOptionsLength;
  unsigned short  OptionsDateLen;
  unsigned char * DERSubjectName;
  unsigned char * MimeOptions;
  unsigned char * OptionsDate;
} MimeHeader;


#define REVOCATIONHEADERFIXEDSIZE 4
typedef struct
{
 unsigned short  DERCertificateLength;
 unsigned short  URLLength;
 unsigned char  *DERCertificate;
 char           *URL;
} RevocationHeader;

#define CERTVERSIONHEADERFIXEDSIZE  0
typedef struct
{
 // Contains just the common header
} CertVersionHeader;

#define  CERTCONTENTVERSIONHEADERFIXEDSIZE 1
typedef struct
{
 unsigned char ContentVersion;
} CertContentVersionHeader;
```

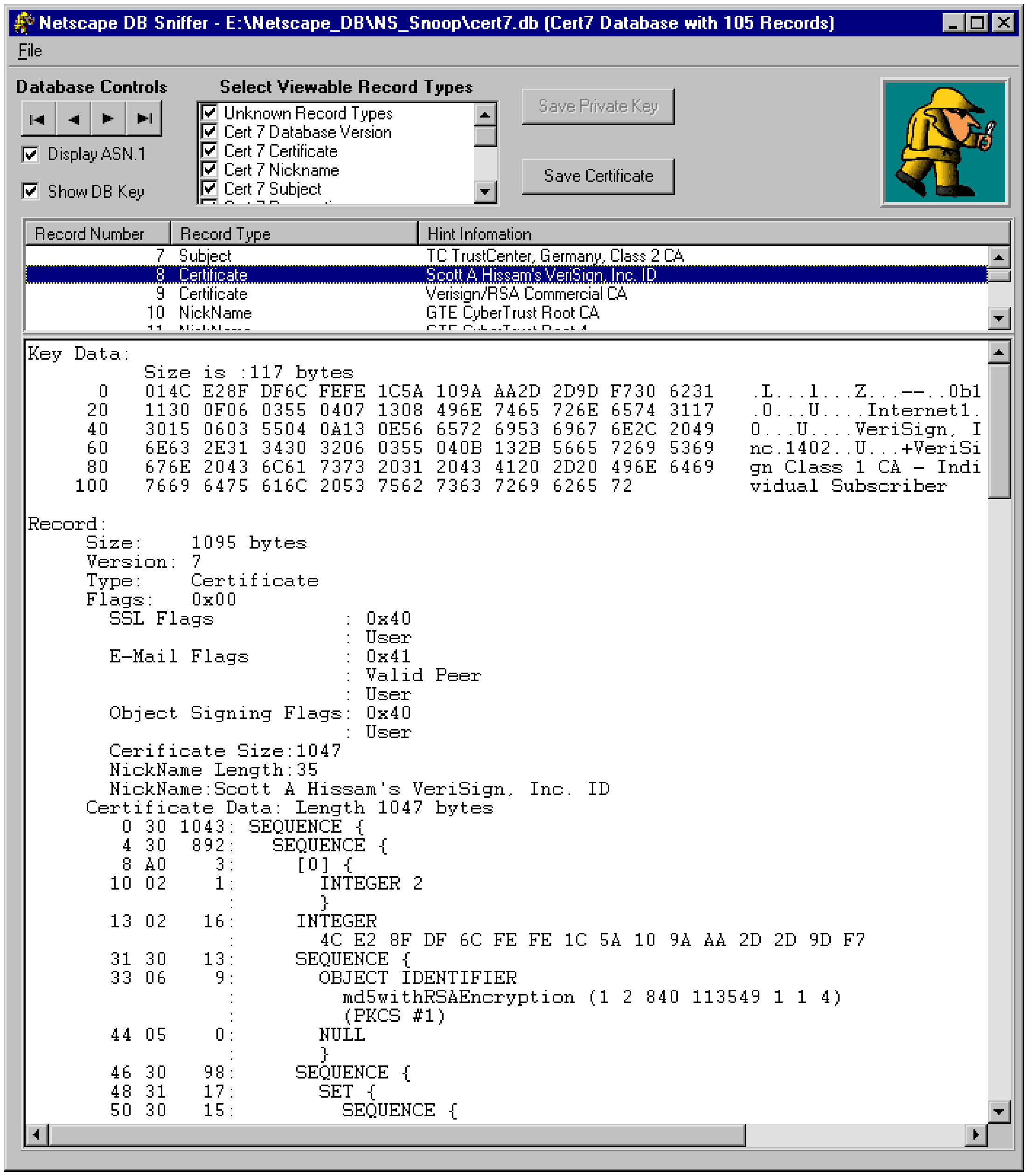
Figure 5: Certificate Database Record Formats

The database key information shown in Figure 6 at the beginning of the record content is used by the database to quickly retrieve a record. A record is typically retrieved using the key information as shown in the code fragment below:

```
DBT key, data;
key.data=(void *)”Version”;
key.size=strlen(“Version”)+1;
if ((db->get)(db,&key,&data,0)==RET_SUCCESS) DisplayRecord(&data);
```

In the above example the key and data variable are the type DBT (data base thang [sic]) as described in the Berkley 1.85 documentation.

The exact details of how Netscape selects keys for each particular type of record are unknown. In some cases the database key appears to contain DER encoded information while in other cases the key appears to be just a string. Additional information regarding database index keys will be discussed in the next section.



Record Number	Record Type	Hint Infomation
7	Subject	TC TrustCenter, Germany, Class 2 CA
8	Certificate	Scott A Hissam's VeriSign, Inc. ID
9	Certificate	Verisign/RSA Commercial CA
10	NickName	GTE CyberTrust Root CA

```
Key Data:
          Size is :117 bytes
       0  014C E28F DF6C FEFE 1C5A 109A AA2D 2D9D F730 6231   .L...l...Z...--..0b1
      20  1130 0F06 0355 0407 1308 496E 7465 726E 6574 3117   .0...U....Internet1.
      40  3015 0603 5504 0A13 0E56 6572 6953 6967 6E2C 2049   0...U....VeriSign, I
      60  6E63 2E31 3430 3206 0355 040B 132B 5665 7269 5369   nc.1402..U...+VeriSi
      80  676E 2043 6C61 7373 2031 2043 4120 2D20 496E 6469   gn Class 1 CA - Indi
     100  7669 6475 616C 2053 7562 7363 7269 6265 72          vidual Subscriber

Record:
     Size:     1095 bytes
     Version:  7
     Type:     Certificate
     Flags:    0x00
       SSL Flags             : 0x40
                             : User
       E-Mail Flags          : 0x41
                             : Valid Peer
                             : User
       Object Signing Flags: 0x40
                             : User
       Cerificate Size:1047
       NickName Length:35
       NickName:Scott A Hissam's VeriSign, Inc. ID
     Certificate Data: Length 1047 bytes
        0 30 1043: SEQUENCE {
        4 30  892:   SEQUENCE {
        8 A0    3:     [0] {
       10 02    1:       INTEGER 2
                 :       }
       13 02   16:     INTEGER
                 :       4C E2 8F DF 6C FE FE 1C 5A 10 9A AA 2D 2D 9D F7
       31 30   13:     SEQUENCE {
       33 06    9:       OBJECT IDENTIFIER
                 :         md5withRSAEncryption (1 2 840 113549 1 1 4)
                 :         (PKCS #1)
       44 05    0:       NULL
                 :       }
       46 30   98:     SEQUENCE {
       48 31   17:       SET {
       50 30   15:         SEQUENCE {
```

Figure 6: Database Browsing Tool

4 Key Database

Decoding the key database was significantly more difficult than was the case for the certificate database. This difficulty was mainly due to the lack of documentation available, and the fact the private key record in the data are encrypted with a password. Unlike the certificate database, the Netscape NSS does not provide *any* information describing the format of this database or the encryption used.

In trying to decode this database, we first dumped all of the records in the database. We discovered that there are only four different types of records in the key database and only two records contained the common header mentioned in Section 3. Records that use the common header have the record types shown in Figure 7.

```
//Record Types
#define PRIVATEKEY     8
#define PASSWORDCHECK 16
```

Figure 7: Key Record Types

The other two records which do not contain the common header are the `Version` record and the `Global Salt`[1] record. These records can be easily identified by their access keys, “`Version`" and `"global-salt"` respectively. The key database can be identified by the existence of the version record. Additionally, if the key database contains any private key records it will also contain a password check record, which can be accessed using “password-check” for the database access key.

As in the certificate database, records in the key database are in big endian format. The key database record formats shown in Figure 8 were actually easy to determine. However, determining how to use this information to decrypt a private key was a different story. Determining the role of each record in the decryption of a private key was going to be a challenge.

We started this task by first dumping a private key record header and data (ASN.1 encoded) as shown in Figure 9. The software used to decode the ASN.1 encoded information was written by Peter Gutmann and may be download from his Web site at

http://www.cs.auckland.ac.nz/~pgut001/

[1] A string of random bits concatenated with a key or password to foil pre-computation attacks.

Decoding the ASN.1 key data revealed the object identifier (OID)[2] of (06 0B 2A 86 48 86 F7 0D 01 0C 05 01 03) that has description string of

`pkcs-12-PBEWithSha1AndTripleDESCBC`

indicating the specific encryption technique used to encrypt the private key. This OID description specifies password-based encryption (PBE) with secure hash version one (SHA1) and the Triple Data Encryption Standard (DES) in cipher block chaining mode (CBC). The OCTET String and the integer contained in the sequence following the OID are the salt and iterator value for the PBE scheme. Finally, the last OCTET STRING is the encrypted private key.

```
typedef struct
{
 unsigned char GlobalSalt[16];
}GlobalSaltHeader;

typedef struct
{
 // Contains just the common header
} KeyVersionHeader;

#define KEYPASSCHKFIXEDSIZE  18
typedef struct
{
 unsigned char  Salt[16];
 unsigned short CryptAlgLength;
 unsigned char *AlgInfo;
 unsigned char *EncryptedAccessKey; //  "password-check" Encrypted 16 bytes
} PasswordCheckHeader;

#define KEYHEADERFIXEDSIZE  8
typedef struct
{
 unsigned char   Salt[8];
 char *          NickName;
 unsigned char * KeyInfoDER;
}KeyHeader;
```

Figure 8: Private Key Database Record Formats

[2] A concept defined by the ASN.1 specification.

```
Record:
     Size:    436 bytes
     Version: 3
     Type:    Private Key
     Flags: 0x23
     Initial Vector:  47 EB A8 CE FC 4B C0 6B
     Key Name:Scott A Hissam’s VeriSign, Inc. ID
     Name Length:34
     Encrypted ASN.1 Private Key
        0 30  386: SEQUENCE {
        4 30   28:   SEQUENCE {
        6 06   11:     OBJECT IDENTIFIER
                 :       pkcs-12-PBEWithSha1AndTripleDESCBC (1 2 840 113549 1 12 5 1
3)
                 :       (PKCS #12 OID PBEID (1 2 840 113549 1 12 5 1).  Deprecated,
use the incompatible but similar (1 2 840 113549 1 12 1 3) or (1 2 840 113549 1 12 1
4) instead)
       19 30   13:     SEQUENCE {
       21 04    8:       OCTET STRING
                 :         47 EB A8 CE FC 4B C0 6B
       31 02    1:       INTEGER 1
                 :       }
                 :     }
       34 04  352:   OCTET STRING
                 :     BF 3E 52 71 3E 07 94 73 25 F2 28 8D 06 D6 1E F8
                 :     B3 EC FA 59 17 06 EC F9 8F 92 19 FE 4C FF C3 81
                 :     F8 BE F0 12 A2 DD 6A D3 17 DA 56 5A B4 65 8B E8
                 :     5D 6F 4B AE 6F 5F 39 DC 1F EF BF 56 6E 79 D5 B4
                 :     2B 9A 6E 20 98 4D 66 98 79 4C 85 98 31 1D 4B E3
                 :     DE EF C3 07 54 76 86 50 A8 22 9E 94 C8 CB F9 F4
                 :     46 9E 52 26 F8 20 8C 51 E8 52 6E 95 16 CA 9D 4D
                 :     E6 7E 90 69 96 1E 1E DF CC 67 FE AB 96 5A D7 88
                 :     26 1A A9 CC 52 F6 97 0F 28 FC 52 96 DE FB FD F7
                 :     87 01 AE 71 E0 88 1B C6 7D 01 C8 83 27 40 36 A3
                 :     46 23 DD 64 86 64 F7 64 73 46 04 30 3A 96 71 33
                 :     7E 98 F1 BE 18 B9 8B 10 DA FF FA 32 AC 03 18 37
                 :     DA 87 32 5F EB F7 ED 0D 37 B2 1B 97 35 D6 38 F2
                 :     F8 CC 4E 2D 00 E2 43 F1 6F 02 B2 FD 94 53 9D 7B
                 :     78 00 4D FB 4D 47 63 6E B9 65 92 4C 03 C2 A6 9F
                 :     20 59 80 D5 A0 D4 B2 79 51 6E 31 B6 20 D4 A9 43
                 :     80 31 CE C6 93 0C B0 1E 2F 13 3F C3 C0 E0 7B 16
                 :     89 76 88 DD 38 D6 8F 2B 5F 6F 50 1D F7 48 D9 2E
                 :     89 C2 04 1F 78 6B AC 85 97 55 0F 71 BE 5D D2 C7
                 :     C8 22 41 B6 C9 A0 C9 81 CD 93 55 83 D2 9D E3 00
                 :     63 72 4F 79 D4 E9 AD 1D 1E CD 79 3F 89 9A 66 E4
                 :     F6 A2 1D EC A0 3E 61 35 81 CC B8 83 5C DF 87 24
                 :   }
```

Figure 9: Private Key Record Header And Key

We needed to find a document that described the PBEWithSha1AndTripleDESCBC password-based encryption technique. An initial search of the Web did not reveal any additional information about the OID. However, we located documentation that described the password-based encryption technique for a similar OID called PBEWithSha1And3-KeyTripleDESCBC in the RSA laboratories PKCS#12 Personal Information Exchange Standard [RSA 97]. We thought there was a good chance that both object identifiers used the same password-based encryption technique.

We performed a Web search for an encryption package that supported the hashing function SHA1 and Triple DES CBC encryption. This resulted in the discovery of a package called SSLEAY that contains cryptographic libraries and certificate support software. Additionally, we located a software package that enhanced the certificate support software in SSLEAY by adding support for the PKCS12 standard [RSA 97]. This was fantastic because we found all of the software needed to decrypt a Netscape private key record on the Web.

We examined the source code from the downloaded software and incorporated into our browsing tool the portions that were needed to decrypt a private key. We then attempted to decrypt a private key using the code extracted from the implementation of the PKCS12 standard. This attempt ended in failure.

Because of our failed attempt, we decided to take a closer look at the Netscape NSS software. Upon examination, we noticed the function call `SECKEY_ChangeKeyDBPasswordAlg`. This API call appeared to change the password-based encryption algorithm used to encrypt the database. This was a guess because the NSS documentation only describes the higher level API calls necessary for using SSL and NSPR, it does not include (other than undocumented C header files) any documentation describing the lower level APIs. Examination of the header files yielded two password-based encryption algorithm identifiers that were of particular interest:

1. `SEC_OID_PKCS12_PBE_WITH_SHA1_AND_TRIPLE_DES_CBC`

2. `SEC_OID_PKCS12_V2_PBE_WITH_SHA1_AND_3KEY_TRIPLE_DES_CBC`

The first algorithm identifier appeared to be the same as the OID that we were unable to find any information about, while the second algorithm appeared to be the same as the OID that we had obtained documentation as well was an implementation. Possibly, our assumption that both OID's were compatible was incorrect.

We then proceeded to write a program to change the database encryption algorithm to SEC_OID_PKCS12_V2_PBE_WITH_SHA1_AND_3KEY_TRIPLE_DES_CBC. After much trial and error in trying to figure out the semantics of Netscape's undocumented interface, we were successful using code shown in Figure 10. This exercise turned out to be very informative. We learned that the global salt record was used in combination with the password (exact details were not known at this time) and that, contrary to what we had thought, the two OID's were *not* compatible.

Next, we tried to decrypt a private key record in the converted base database. Initially we were unsuccessful, but after some trial and error, with different password formats (unicode or non-unicode), we discovered that we could decrypt a private key. The output from the NSS API call `SECKEY_HashPassword` needed to be the input password to the PBE PKCS12 decryption software that we obtained from the Web. After further trial and error (really a wild guess), we determined that the `SECKEY_HashPassword` actually performs the hashing function shown in Figure 11. This was determined by first noticing that all password were always 20 bytes long, indicating that the user input password and salt were most likely being used as input to SHA-1 (since SHA1 is a hashing function that always returns a twenty-byte digest).

On our first attempt, we used the `SHA1_Update` call in the hash function shown in Figure 11 to concatenate salt onto the password; this failed, however. Next we changed the order (salt then password); this worked.[3]

Almost incidentally, we also determined that key databases do not always contain a "Global Salt" record, which is reason for the `HaveGlobalSalt` flag in the password hashing function, which explains the "if" statement in the hash function. The hashed password, however, is *always* 20 bytes in length.

```
#include <stdio.h>
#include <string.h>
#include <secitem.h>
#include <key.h>
int main (int argc, char **argv)
{
 SECKEYKeyDBHandle *Handle;
 SECItem           *st;
 char               passwd[512];
 if (argc!=2) {
  printf("usage: changedb <database file>\n");
  return -1;
 }
 if ((Handle=SECKEY_OpenKeyDBFilename(argv[1],0))==NULL) {
  printf("database open error\n");
  return -1;
  }
 printf("Enter Password:");
 fgets(passwd,sizeof(passwd),stdin);
 if (strlen(passwd)) passwd[strlen(passwd)-1]='\0';
 st=SECKEY_HashPassword(passwd,Handle->global_salt );
 if (SECKEY_CheckKeyDBPassword(Handle,st)!=SECSuccess) {
   printf("Incorrect Password\n");
   SECKEY_CloseKeyDB(Handle);
   return -1;
  }
  // Original Database format was SEC_OID_PKCS12_PBE_WITH_SHA1_AND_TRIPLE_DES_CBC
 if  (SECKEY_ChangeKeyDBPasswordAlg(Handle,st, st,
       SEC_OID_PKCS12_V2_PBE_WITH_SHA1_AND_3KEY_TRIPLE_DES_CBC)==SECSuccess)
       printf("Database Format Change Success\n");
 else printf("Database Format Change Falied\n");

 SECKEY_CloseKeyDB(Handle);
 return 0;
}
```

Figure 10: Code to Change the DB Encryption Algorithm

```
Unsigned char HashPassword[20];
void __fastcall TForm1::SetHashPassword(char *Password)
{
  SHA_CTX c;
  SHA1_Init(&c);
  if (HaveGlobalSalt) SHA1_Update(&c,GlobalSalt, 16);
  SHA1_Update(&c, (unsigned char *)Password,strlen(Password));
  SHA1_Final(HashPasswd,&c);
}
```

Figure 11: Password and Global Salt Hash Function

At this point, our tool could decrypt all of the records in private key database that had been converted to use the `SEC_OID_PKCS12_V2_PBE_WITH_SHA1_AND_3KEY_TRIPLE_DES_CBC` encryption algorithm. However, requiring a database conversion was unsatisfactory to us—we were too close to stop here. So we needed to determine the details of the PBEWithSha1AndTripleDESCBC encryption algorithm. An exhaustive search of the Web

[3] Sometimes, good clean living pays off.

was performed and the following information was discovered about this *uncommon* OID (note again the overseas addresses in one of the sources):

- Personal Information Exchange Syntax and Protocol Standard Version 0.020 27, January 1997 Microsoft Corporation
- a PFX software program (pfx-012.tar.gz) written by Dr. Stephen Henson shenson@drh-consultancy.demon.co.uk
- PKCS #1 RSA Cryptography Specifications Version 2.0
- RFC 2104 HMAC: Keyed-Hashing for Message Authentication
- the TLS Protocol Version 1.0

Using the above resources and *still more* trial and error, this time to figure out the semantics of the above terse documentation, we finally were able to decrypt the private key information in the database without using NSS to change to the database password encryption algorithms. The private keys were decrypted as follows:

1. The user input password and global salt (if present) are used to generate a hash password using the `SetHashPassword` method shown in Figure 11.

2. The ”Key” and the “Initial Value” for Triple Des Cipher are generated by calling the `BEPGetKeyIV` method shown in Figure 12 using the `HashPassword` for the password value, salt and iterator from the ASN.1 object. A 24-byte key and 16-byte *initial value* are returned.

3. Next, the decrypt function shown in Figure 13 is called using the *initial value* and key generated in step 2 and the encrypted data portion of ASN.1 object. If decryption is successful, a pointer to decrypted data as well as its length is returned.

This software was then incorporated into our browsing tool. This tool now had the capability to examine and decrypt all the records in Netscape’s certificate and key databases.

Next, we investigated Netscape’s password check record. After some trial and error, we determined that this record contained a sixteen-byte salt, an encryption algorithm OID, and sixteen bytes of encrypted data. When the encrypted data is decrypted correctly, the plain text turns out to be the string “password-check.” This is how Netscape determines if a password is correct without decrypting a private key record.

Now our database-browsing tool was robust enough to allow us to easily examine the Netscape databases. We investigated how Netscape uses database keys to link certificates in the certificate database to the private key database. We studied a certificate and private key record that was known to match and noticed that Netscape included an octet string (see Figure 14), the certificate record which was the database access key to obtain the private key record from the private database (see Figure 15). Additional information about Netscape’s use of database access keys can be determined through studying database records using the browsing tool. Such information is beyond the scope of this report.

```
void __fastcall TForm1::PBEGetKeyIV(unsigned char *Password,
                                    unsigned char *Salt,
                                    int            SaltLength,
                                    int            Iterator,
                                    unsigned char *Key,
                                    unsigned char *IV)

{
       unsigned char Digest[20],
                  SecondDigest[20],
                  DK[40];
    SHA_CTX c;
    HMAC_SHA1_CTX   hmac_ctx;
    memset (SecondDigest, 0, 20);
    memcpy (SecondDigest, Salt, SaltLength);
    SHA1_Init(&c);
       SHA1_Update(&c, Password,20);
    SHA1_Update(&c, Salt,SaltLength);
    SHA1_Final(Digest,&c);
       for (int i = 1; i < Iterator; i++)
      {
        SHA1_Init(&c);
               SHA1_Update(&c,Digest, 20);
               SHA1_Final(Digest,&c);
         }

    for (int i = 0; i < 2; i++)
    {
     HMAC_SHA1_Init(&hmac_ctx, Digest,20);
     HMAC_SHA1_Update(&hmac_ctx, SecondDigest, 20);
     HMAC_SHA1_Update(&hmac_ctx, Salt, SaltLength);
     HMAC_SHA1_Final(&hmac_ctx, &DK[i*20], NULL);
     HMAC_SHA1_Init(&hmac_ctx, Digest,20);
     HMAC_SHA1_Update(&hmac_ctx, SecondDigest, 20);
     HMAC_SHA1_Final(&hmac_ctx, SecondDigest, NULL);
    }
  memcpy (Key, DK,24);
  memcpy (IV, DK + 32, 8);
}
```

Figure 12: Key and IV Generation

```
unsigned char * __fastcall TForm1::TrippleDESDecrypt(unsigned char *CryptData,
                                                     int CryptDataLen,
                                                     unsigned char *Key,
                                                     unsigned char *IV,
                                                     int  *DecryptDataLen)
{
 DES_EDE3_CBC_Type cipher_ctx;
 unsigned char *DecryptData;
 int tmp;
 if ((DecryptData = (unsigned char *)malloc (CryptDataLen + 8))==NULL)
    {
     *DecryptDataLen=0;
     return(NULL);
    }
 DES_EDE_3_CBC_Init(&cipher_ctx, Key, IV,DECRYPT);
 DES_EDE_3_CBC_Update(&cipher_ctx,DecryptData,DecryptDataLen,CryptData,
                  CryptDataLen);
 if (!DES_EDE_3_CBC_Final(&cipher_ctx, DecryptData+*DecryptDataLen,&tmp))
    {
     free(DecryptData);
     *DecryptDataLen=0;
     return(NULL);
    }
 (*DecryptDataLen)+=tmp;

 return(DecryptData);
}
```

Figure 13: Triple DES Decrypt Function

```
470 03   75:       BIT STRING 0 unused bits, encapsulates {
      473 30   72:           SEQUENCE {
      475 02   65:             INTEGER
                 :               00 B1 E0 AD 39 E7 09 41 B9 D3 21 90 9B 0F 95 78
                 :               E6 FD EF D3 62 34 51 4D 79 02 83 17 9F 4F 09 68
                 :               5C 81 A2 E6 2D B1 F7 BB E6 69 BA 39 A5 F4 17 0B
                 :               A9 A9 EA B0 4C 7F FF 55 A5 46 A7 67 10 3A 1F E1
                 :               7B
      542 02    3:             INTEGER 65537
                 :             }
                 :           }
```

Figure 14: Certificate Fragment of Database Access Key

```
Key Data:
          Size is :65 bytes
       0   00B1 E0AD 39E7 0941 B9D3 2190 9B0F 9578 E6FD EFD3   ....9..A..!....x....
      20   6234 514D 7902 8317 9F4F 0968 5C81 A2E6 2DB1 F7BB   b4QMy....O.h\...-...
      40   E669 BA39 A5F4 170B A9A9 EAB0 4C7F FF55 A546 A767   .i.9........L..U.F.g
      60   103A 1FE1 7B
```

Figure 15: Private Key Record Access Key

5 Summary

Netscape’s certificate database is straightforward and easy to decode. The key database was somewhat difficult to decode because of the difficulty in obtaining information about the obsolete PFX format that is used to encrypt the private key data. This PFX specification defined the uncommon PBEWithSha1AndTripleDESCBC OID. The ability to decode the key and certificate databases stems from Netscape’s use of standards such as ASN.1 and PKCS. Knowledge of these standards allowed us to more easily interpret information within Netscape databases. While the use of Netscape’s NSS provided some information, we believe that the information provided in this document could have been determined without NSS. However, if Netscape did not use standards in the development of the databases, records, and encryption schemes, this task would have been nearly impossible.

The major lessons to be learned from this case study are the following:

1. If you need to peer inside a product (a black box), you must know what you are looking for. In this case study deep and detailed knowledge of computer security was necessary. Without this knowledge it is doubtful that progress could have been made.
2. For good and sufficient reasons, vendors such as Netscape will make use of standards in building their products (for example, ASN.1). Knowledge of these standards is also crucial for developers who want to peer inside a product. From a vendor’s perspective, this shows the use of standards to be a two-edged sword.
3. A significant degree of systems expertise is needed by developers who will peer inside a product. Programs must be written, raw data dumps must be interpreted, networks “sniffed,” and so forth in order to crack the puzzle. Moreover, strong problem solving skills and perseverance are needed since there is rarely just one puzzle to be cracked.

All of this tends to support the observation that building systems from commercial software product often requires more, rather than less, technical sophistication on the part of software developers.

References

Dierks 98	Dierks, T. & Allen, C. “The TLS Protocol Version 1.0,” internet draft <draft-ieft-tls-protocol-05.txt> [online]. Available FTP: <URL: ftp://ds.internic.net/internet-drafts/draft-ietf-tls-protocol-05.txt> (November 12, 1997).
Krawczyk 97	Krawczyk, H.; Bellare, M.; & Canetti, R. “HMAC: Keyed-Hashing for Message Authentication,” request for comments <rfc2104.txt> [online]. Available WWW: <URL: http://www.ietf.org/rfc/rfc2104.txt> (February 1997).
Microsoft 97	Microsoft Corporation. *PFX: Personal Information Exchange Syntax and Protocol Standard, Version 0.020*. Microsoft Developers Network (MSDN) Library. Seattle, Wa.: Microsoft Corporation, January 1997.
RSA 98	RSA Laboratories. *PKCS #1 RSA Cryptography Specification Version 2.0* [online]. Available FTP: <URL: ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-1v2.asc> (September 1998).
RSA 97	RSA Laboratories. *PKCS #12 Personal Information Exchange Syntax Standard Version 1.0*, draft [online]. Available WWW: <URL: http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-12.html> (April 30, 1997).
RSA 88	RSA Laboratories. *PKCS #5: Password-Based Cryptography Standard Version 2.0*, second draft [online]. Available WWW: <URL: http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-5.html> (December 10, 1988).

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (LEAVE BLANK)		**2.** REPORT DATE March 1999	**3.** REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Into the Black Box: A Case Study in Obtaining Visibility into Commercial Software			**5.** FUNDING NUMBERS C — F19628-95-C-0003
6. AUTHOR(S) Daniel Plakosh, Scott Hissam, Kurt Wallnau			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			**8.** PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-99-TN-010
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116			**10.** SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			**12.B** DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

We were recently involved with a project that faced an interesting and not uncommon dilemma. The project needed to programmatically extract private keys and digital certificates from the Netscape Communicator v4.5 database. Netscape documentation was inadequate for us to figure out how to do this. As it turns out, this inadequacy was intentional—Netscape was concerned that releasing this information might possibly violate export control laws concerning encryption technology. Since our interest was in building a system and not exporting cryptographic technology, we decided to further investigate how to achieve our objectives even without support from Netscape. We restricted ourselves to the use of Netscape-provided code and documentation, and to information available on the Web. Our objective was to build our system, and to provide feedback to Netscape on how to engineer their product to provide the capability that we (and others) need, while not making the product vulnerable or expose the vendor to violations of export control laws. This paper describes our experiences peering "into the black box."

14. SUBJECT TERMS commercial off-the-shelf (COTS), component integration, netscape, security			**15.** NUMBER OF PAGES 19 pp.
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18.** SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	**19.** SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	**20.** LIMITATION OF ABSTRACT UL