Startsida
Anslag
Diskussioner
Omdömen
Personer
Moduler
Uppgifter
Media Gallery
Video Recording

Hidden Encryption Assignment

Skicka uppgift

Förfallodatum Måndag före 21:00 **Poäng** 100 **Lämnar in** en filuppladdning **Filtyper** zip
Tillgänglig 6 maj kl 18:00–18 maj kl 21:00 12 dagar

Introduction

In this lab you will gain insights into the problems of hidden encryption and plausible deniability. You will also get first-hand experience with tools and methods for symmetric-key encryption and decryption, and for cryptographic hashing.

The lab is about encrypting information and storing it in the file system on your computer. If you have sensitive information that you want to protect, you can put the information in a file and encrypt it. Most operating systems have easy-to-use tools for this.

There is a problem with this approach, though. The encrypted files are still visible in the computer. If an attacker gets access to a computer and finds an encrypted file, the attacker might be able to use methods of social engineering, threats, and so on, to get the decryption key from the computer's owner and decrypt the information.

If the encrypted information instead is hidden somehow, it might be more difficult for the attacker to detect that it exists, and the computer's owner could be able to deny the presence of encrypted information – this is called *plausible deniability*.

In this lab assignment, you will extract encrypted information hidden in files, and create files with hidden encrypted information in them.

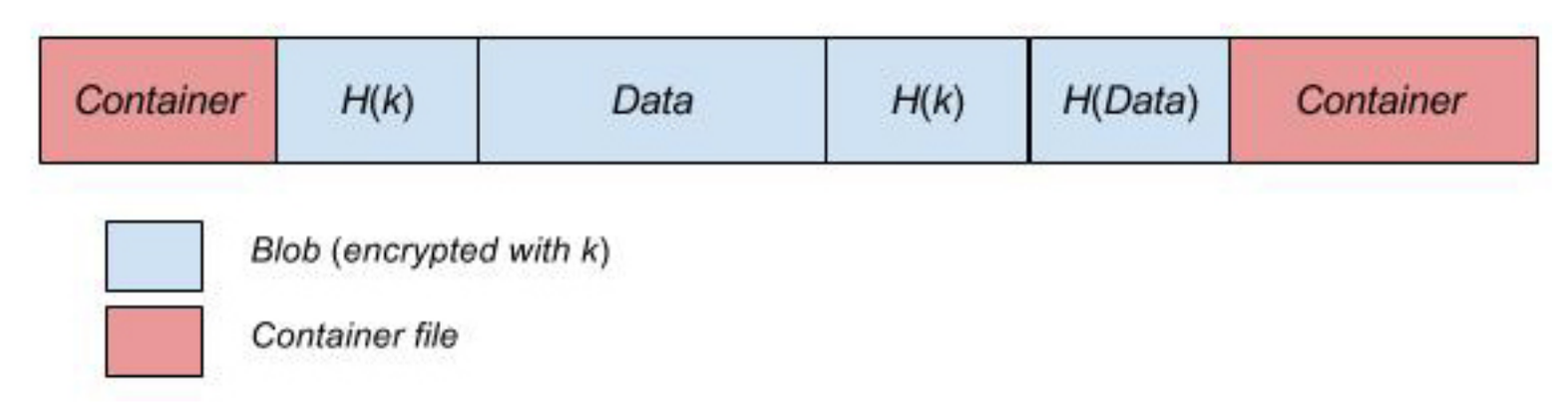
Hidden Encryption

The idea behind the way in which encrypted information is hidden in this assignment is that the encrypted information will be a "blob" of data that can placed anywhere inside a file. To the attacker, the encrypted information appears as random data. For someone that knows what to look for, the data is possible to locate and decrypt. The data can then be stored in a suitable place in the file system: as data embedded somewhere in a binary file, such as an executable file, JPEG image, audio data, or in some unused parts of the file system (unallocated space on a disk drive, for instance).

The format of the the data "blob" is shown in the figure below (in blue). The entire blob is encrypted with a secret key *k*. The first element in the blob is the MD5 hash of the secret key, *H*(*k*). Then comes the information you want to hide, *Data*, followed by *H*(*k*) again. Last is another hash: the MD5 hash of Data, *H*(*Data*).

The basic principle here is that *H*(*k*), the MD5 hash of the secret key, marks the start and the end of the hidden information. The terminating hash *H*(*Data*) is for integrity protection.

The blob is placed somewhere inside a *container* file (the red parts). In other words, the blob is inserted at some position inside a file.



The idea behind this scheme is that in order to locate and decrypt the hidden data in a container file, the following steps would be taken:

- Compute *H*(*k*), the MD5 hash of the secret key *k*.
- Scan the file, by decrypting with *k* and searching for *H*(*k*) in the decrypted data.
- When *H*(*k*) is found, it indicates the start of the blob.
- Decrypt the succeeding blocks until the next *H*(*k*) is found.
- Take the plaintext between the two occurrences of *H*(*k*) as the secret information, *Data*.
- Decrypt the block after the second *H*(*k*). Call this value *H'*.
- Compute the MD5 hash of the data, *H*(*Data*).
- Verify that *H*(*Data*) equals *H'*, in which case the operation has been successful and *Data* is the hidden information.

Instructions

Preliminaries

You will get a challenge with personalized data that you should work with. We use a separate assignment in Canvas for this: Hidden Encryption Challenge. Your individual challenge is available for you there (you don't need to do anything to get the challenge).

Task 1

In your personalized challenge, you can find two files for the first task: a data file with binary data, called "task1.data", and a file called "task1.key", which is a text file with an encryption key represented as a hexadecimal string.

The data file contains a blob encrypted with AES-128-ECB. Your job is to locate the blob, extract and verify the encrypted information from the blob, and submit it as the result of this lab. Store the extracted and decrypted information in a file called "file1.data". You can find more details below for how to submit.

Task 2

In this task, your job is to create a blob of data. The blob should be encrypted with AES-128-ECB.

In your challenge, you can find three files for this task: a data file with binary data, called "task2.data"; a file called "task2.key", a text file with an encryption key represented as a numeric string; and "task2.offset", a text file that defines the offset in bytes at which the blob should be placed in the output file.

Create a blob that contains the data in "task2.data", using the key in "task2.key". The blob *must* conform to the format as illustrated in the figure above. Store the blob inside a container file, which should be padded with random data to a total size of 2048 bytes. Call the resulting file "file2.data".

Task 3

Task 3 is similar to Task 1. The difference is that for Task 3, the blob is encrypted with AES-128-CTR. To solve this task, you should write a program called Hiddec that that takes a container file as input, extracts a blob from it, and produces the decrypted data from the blob as output. Hiddec should support at least two encryption algorithms: AES-128-ECB and AES-128-CTR.

The program is compiled in the following way:

```
$ javac Hiddec.java
```

The program should take the following arguments:

```
--key=KEY
```

Use KEY for encryption key, given as a hexadecimal string.

```
--ctr=CTR
```

Use CTR as the initial value for the counter in AES-128-CTR mode. Implies that AES-128-CTR mode should be used for encryption (otherwise AES-128-ECB).

```
--input=INPUT
```

Input file. The file INPUT is the container.

```
--output=OUTPUT
```

Output file. OUTPUT is where the decrypted data should be stored.

For instance, if you want to extract encrypted data from a container in a file called "container.data" and store the output in a file called "plain.txt", you would issue the following command (the container is in ECB format, so there is no "--ctr" argument):

```
$ java Hiddec --key=92d4ab32eac2d8a0042342e0fdbe80f5 --input=container.data --output=plain.txt
```

Use your program to extract and verify the encrypted information from the blob, and submit it as the result from this lab. Store the extracted and decrypted information in a file called "file3.data".

Task 4

Task 4 is similar to task 2. The difference is that for task 4, the blob should be encrypted with AES-128-CTR. To solve this task, you should write a program called Hidenc that that creates a blob and embeds it into a container file. Hidenc should support at least two encryption algorithms: AES-128-ECB and AES-128-CTR. The program is compiled in the following way:

```
$ javac Hidenc.java
```

The program should take the following arguments:

```
--key=KEY
```

Use KEY for encryption key, given as a hexadecimal string.

```
--ctr=CTR
```

Use CTR as the initial value for the counter in AES-128-CTR mode. Implies that AES-128-CTR mode should be used for encryption (otherwise AES-128-ECB)

```
--offset=NUM
```

Place the blob in the file at an offset of NUM bytes into the container file. If no offset is given, Hidenc generates it by random.

```
--input=INPUT
```

Input file. The data from file INPUT should be used as the data portion of the blob.

```
--output=OUTPUT
```

Output file. OUTPUT is the container where the final result is stored.

```
--template=TEMPLATE
```

Use the file TEMPLATE as a template for the container in which the blob should be stored. The output file should have exactly the same size the the template. In other words, parts of the data in the template should be overwritten with the blob. If no template is given, Hidenc generates a container with random data. Only one of --template and --size can be specified.

```
--size=SIZE
```

The total size of the output file should be SIZE bytes. This implies that the container file should be generated automatically. In other words, there is no template. Only one of --size and --template can be specified.

First create a blob that contains the data in "task4.data", using the encryption key in "task4.key", the initial counter in "task4.ctr", at the offset specified in "task4.offset". The blob *must* conform to the format as illustrated in the figure above. Store the blob inside a container file, which should be padded with random data to a total size of 2048 bytes. Call this file "file4.data".

Simplifications and Assumptions

- The smallest data unit is an AES block. That is, the hidden information is aligned on block boundaries, and container files consist of an integer number of AES blocks. In other words, you need not worry about padding.
- For CTR mode, the initial counter value is given. This is the value used for the first block in the blob. Then the counter is increased by one for each block. So if blocks are numbered starting from zero, the counter for block number i is $c+i$, where c is the initial counter.

Environment

This assignment is essentially a programming exercise, and you should use Java to solve it. You can use any Java development environment of your choice; however, it is a requirement that the code you finally submit runs on the the course virtual machine. Furthermore, your submitted code should be possible to execute directly in a Linux shell, as described above in section "Implementation", *so it should not depend on any IDE tool such NetBeans or Eclipse.*

Even though the assignment is designed as a programming assignment, you may be able to address some parts of it using command line tools, in particular the openssl command line tool. It is a toolkit that can be used for a wide range of cryptographic operations. For instance, in order to encrypt the data in the file "plain.txt" and store the output in "cipher.bin" with DES-CBC, you would issue the command:

```
$ openssl enc -e -des-cbc -in plain.txt -out cipher.bin -K feed1234beef9812 -iv ab9876ba -nopad
```

The "-K" option gives the encryption key as a hexadecimal string, and the "-iv" option specifies an initialization vector. To learn more about encryption with openssl, run "man enc", and about openssl in general, type "man openssl".

You might also find a hex editor useful. It is a tool that allows you to view and edit binary files. Two hex editors, ghex and bless, are installed on the lab image. Handy command line tools for viewing the content of a file in hexadecimal are xxd and hexdump.

Another tool you may consider is "dd". It is a versatile tool that can be very useful when it comes to working with files and I/O devices in Linux. You can use it to copy data between files with a high degree of control. For instance, in order to copy the first 82 bytes from file "infile" and put them at position 784 of file "outfile" (overwriting the current content at that position), you could do something like:

```
$ dd if=infile of=outfile bs=1 count=82 seek=784
```

Hints and Tips

Before you start, think through how you want to approach this assignment. On the one hand, it would be possible to solve the first two tasks without any programming; you could use openssl together with an hex editor instead, although it would be a bit tricky and require attention to details. On the other hand, the last two require you to write the two programs Hidenc and Hiddec. If you decide to write programs, you can use the same programs to solve the first two tasks, so then it is probably wise to go straight for that solution, and not spending time on solving the first two tasks manually.

AES is available in JCA/JCE, the standard security framework for Java. We do not have any learning activities about JCA/JCE in this course, but there is plenty of material online. You can find some useful links here: JCA and JCE.

Files

You will be provided with the following files in your challenge:

- task1.data - a container file with hidden encrypted data for Task 1
- task1.key - the encryption key for the encrypted data in task1.data
- task2.data - data to hide for Task 2
- task2.key - The encryption key for hiding the data in task2.data
- task2.offset - the offset (in bytes) at which the blob should be placed in the output file for Task 2
- task3.data - a container file with hidden encrypted data for Task 3
- task3.key - the encryption key for the encrypted data in task3.data
- task3.ctr - the initial counter value for the encrypted data in task3.data
- task4.data -data to hide for Task 4
- task4.key - The encryption key for hiding the data in task4.data
- task4.ctr - the initial counter value for hiding the data in task4.data
- task4.offset - the offset (in bytes) at which the blob should be placed in the output file for Task 4
- INSTRUCTIONS - a file with brief instructions explaining the content of the challenge
- Makefile – configuration file for "make" that you should use to create your submission

Submission

For submission, you will turn in the following files:

- file1.data – the data hidden in task1.data
- file2.data – a container file containing the data in task2.data, encrypted with the key in task2.key
- file3.data – the data hidden in task3.data
- file4.data – a container file containing the data in task4.data, encrypted with the key in task4.key, using the value in task4.ctr as initial counter value
- Hiddec.java – Java source code for your implementation of Hiddec
- Hidenc.java – Java source code for your implementation of Hidenc

To submit your solution, use the config file for "make" included in the challenge:

```
$ make hidden.zip
```

This will create a zip archive called "hidden.zip". To submit, upload exactly as is.

If you decide not to do all tasks, you can still create your submission using the "make" config file. It will take whatever submission files it finds in the directory and create the zip archive. You will see a warning message for any missing file.

Important: the challenge contain authentication files, which are used to verify that your submission really is for your challenge (and not for someone else's). The "make" program will automatically include those files with your submission. Therefore, you must run "make" in the same directory where you extracted your challenge from the zip archive.

Testing

Besides verifying that you have met your challenge and submitted the correct data files, your implementation will be tested for functionality using different data files and blobs. Furthermore, your implementation will be tested for basic input validation and error handling. If the user gives incorrect parameters, or if the input data does not match the parameters, this should be detected and an informative error message should be generated. (Java exceptions do not count as informative error messages.) Additional test cases include:

- Hiddec does not extract incorrectly formatted blobs:
  - No terminating *H(k)*
  - No trailing MD5 sum of data *H(data)*
- Hidenc does not attempt to put a blob past the end of a given container file

Requirements and Points

For this assignment, you can get a maximum of 100 points. The grading scale is as follows:

- 40 points:
  - Task 1 and 2 correctly solved
- 100 points:
  - Task 3 and 4 correctly solved
  - Working implementations of Hidenc and Hiddec

Note that there can be deduction in points depending on the quality of your solution. If you aim for 100 points, for instance, but we assess that you solution only solves half of the challenges in a satisfactory manner, you might get 70 points.