Rocket®

MultiValue University 2015

D3 Security – Field Encryption

Lab Guide

Developed by
D3 MVU Team

Lab Overview

Abstract

The purpose of this lab is to demonstrate one way in which dynamic field level encryption may be implemented using callr and callx triggers that use the **`%encrypt()`** and **`%decrypt()`** FlashBASIC functions. The API for these two functions is presented, which enables the programmer to encrypt and decrypt an arbitrary string, followed by a sample program demonstrating their use, and finally sample callr and callx triggers which are used to implement dynamic field level encryption.

This lab used a controlled environment at MV University. You can do this exercise in your environment if you download a special account – FLE. This can be found at our github site:

https://github.com/RocketSoftware/multivalue-lab

About the Lab Environment

The lab environment uses the following:

Desirable Prerequisites:

- Some BASIC programming skills
- Ability to edit items (the Update Processor is used in the examples)
- D3 10.1 Linux or D3 9.2 Windows
- FLE Account (Field Level Encryption)

Lab Overview

- Time estimate: 60 minutes
- There are four sections to this lab:
  - Section 1: Overview of the FlashBASIC encryption API
  - Section 2: Using the **`%encrypt()`** and **`%decrypt()`** functions
  - Section 3: Using a **`callx`** trigger to encrypt a field
  - Section 4: Using a **`callr`** trigger to decrypt a field

Exercise 1: Overview of the FlashBASIC encryption API

Purpose of the Exercise

This exercise will show you the FlashBASIC encryption/decryption API.

After this exercise you will be familiar with the:

- crypto.inc FlashBASIC include
- **`%encrypt()`** and **`%decrypt()`** API

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3.

a. Do one of the following:

- On Linux, from the shell, enter **`d3`**.
- On Windows, Telnet to localhost.

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: FLE
```

__ 2. The %eecrypt() and %encrypt() API.

The function declarations are identical:

```
code = %decrypt(params, inputString, outputString, &outputStringLength)
code = %encrypt(params, inputString, outputString, &outputStringLength)
```

Input arguments:

params: Structure (dynamic array) used to provide the function meta-data. See the "Crypto parameter structure" step below.

inputString: The text that will be either encrypted or decrypted.

Output arguments:

outputString: The encrypted or decrypted text.

outputStringLength: Number of characters used in the outputString buffer.

These functions also return a result code. See the "Result code" step below.

__ 3. Review the dm,bp,includes crypto.inc FlashBASIC include item.

There are multiple methods to review. For example:

- **ed dm,bp,includes crypto.inc**, or
- **u dm,bp,includes crypto.inc**, or
- **ct dm,bp,includes crypto.inc**

There are two sections of interest, each described in its own step below.

- Result codes
- Crypto parameter structure

__ 4. Result Code.

The functions return one of the following result codes:

- Crypto$Result$Success: The call was successful. outputString and outputStringLength are valid.
- **Crypto$Result$Failed**: The call failed. The cause is unknown.
- **Crypto$Result$InvalidKey**: The length of the key provided in the parameter structure is not 16 bytes (AES128).
- **Crypto$Result$OutputBufferTooSmall**: The buffer provided to hold the **outputString** was too small. When encrypting, the **outputString** buffer must be at least twice the size of the **inputString** buffer. When decrypting, the **outputString** buffer must be at least the same size of the **inputString** buffer.
- **Crypto$Result$NoEncryption**: The encryption libraries are not loaded.

__ 5. Crypto parameter structure.

The crypto parameter structure is a dynamic array. It must be properly populated and passed in as the first argument to the functions.

- **Crypto$P$Algorithm**: The encryption algorithm to use. For example, **Crypto$Algorithm$AES128**.
- Crypto$P$InputLength: The length of the inputString buffer.
- **Crypto$P$OutputLength**: The number of characters available in the **outputString** buffer. When encrypting, the **outputString** buffer should be at least twice the size of the **inputString** buffer. This is to allow for the worst case scenario in which each input character is encrypted to a reserved character, which requires escaping it into two characters. When decrypting, the **outputString** buffer should be at least the same size as the **inputString** buffer. This is to allow for the worst case scenario in which each input character

is not decrypted from a reserved character, meaning that the **outputString** buffer will have as many characters as the **inputString** buffer, not fewer.

- **Crypto$P$ClearKey**: The key used to encrypt or decrypt the **inputString**. For AES128 it must be 16 bytes long.

Exercise 1 summary: Familiarity with the crypto.inc include item and the **%encrypt()** and **%decrypt()** API.

End of Exercise 1

Exercise 2: Using the %Encrypt() and %Decrypt() functions

Purpose of the Exercise

This exercise will demonstrate how to encrypt an arbitrary string using the FlashBASIC **`%encrypt()`** function and decrypt it using **`%decrypt()`**.

After this exercise you will be able to:

- Encrypt an arbitrary string using the FlashBASIC **`%encrypt()`** function
- Decrypt the encrypted string using the FlashBASIC **`%decrypt()`** function

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3.

a. Do one of the following:
   - On Linux, from the shell, enter **`d3`**
   - On Windows Telnet to localhost

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: FLE
```

__ 2. Review the FLE,bp, test program.

There are multiple methods to review. For example:

- ed FLE,bp, test, or
- u FLE,bp, test, or
- ct FLE,bp, test

Each section of interest is described in its own step below.

__ 3. Get input.

This part of the program interactively gathers user input. The plain text string and the 16 character key are collected.

__ 4. Prepare the **`%encrypt()`** input.

The crypto parameter structure described in the previous exercise is populated.

Notice that the output buffer is allocated at twice the size of the input buffer. This is to allow for the worst case scenario in which each input character is encrypted to a reserved character, which requires escaping it into two characters.

```
encryptedTextBufferSize = 2 * plainTextLength

Char encryptedText[ encryptedTextBufferSize ]
```

__ 5. Call **%encrypt()**, check the result, and extract the encrypted text.

The **%encrypt()** function is called.

Notice that **encryptedTextLength** argument is passed in with the "&" prefix. This passes the address of **encryptedTextLength** to **%encrypt()**, which allows it to be set. **encryptedText**, on the other hand, was declared as a char array, so its value is already an address.

```
Char encryptedText[ encryptedTextBufferSize ]
result = %encrypt( cryptoParams, plainText, encryptedText,
&encryptedTextLength )
```

Also, notice how the encrypted text is extracted. This is necessary so that anything beyond the end of the actual string is eliminated.

```
encryptedText = encryptedText[ 1, encryptedTextLength ]
```

__ 6. Display the plain text and encrypted text as hexadecimal strings.

The **mx** conversion is used to convert the strings to hexadecimal.

The hexadecimal strings are then displayed.

__ 7. Prepare the **%decrypt()** input.

Like the **%encrypt()** function, the crypto parameters structure is populated. The **outputString** buffer is allocated at the same size as the **inputString** buffer. This is to allow for the worst case scenario in which each input character is not decrypted from a reserved character, meaning that the **outputString** buffer will have as many characters as the **inputString** buffer, not fewer.

__ 8. Call **%decrypt()**, check the result, and extract the decrypted text.

Like the **%encrypt()** function, notice how the arguments are passed and the decrypted text is extracted.

__ 9. Display the decrypted text as a hexadecimal string and compare it to the original plain text.

This step demonstrates (verifies) that encrypting and decrypting produces the same string as the original plain text.

__ 10. Run the test program.

a. At the TCL prompt, enter **`test`**.

b. Enter a string.

c. Enter a 16 character key.

The plain text, encrypted text, and decrypted text are displayed as hexadecimal strings. If the original plain text and decrypted text match, and they should, the test program indicates that encryption and decryption succeeded.

Exercise 2 summary: Using **`%encrypt()`** and **`%decrypt()`** to encrypt and decrypt and arbitrary string.

End of Exercise 2

Exercise 3: Using a callx trigger to encrypt a field

Purpose of the Exercise

This exercise will demonstrate dynamically encrypting field data when it's written to a file by using a callx trigger and the %encrypt() function.

After this exercise you will be able to:

Encrypt field data as it's written to a file

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3.

a. Do one of the following:

- On Linux, from the shell, enter **d3**
- On Windows Telnet to localhost

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: FLE
```

__ 2. Review the FLE,bp, encrypt program.

- There are multiple methods to review. For example:
  - **ed FLE,bp, encrypt**, or
  - **u FLE,bp, encrypt**, or
  - **ct FLE,bp, encrypt**

  Each section is described in its own step below.

__ 3. Main

This is the main part of the program.

A flag (part of the data in the item) is checked to see if the attribute to be encrypted is already encrypted. If it is, it returns without doing anything. This is important. Since the encryption algorithm is symmetric, calling **%encrypt()** again with the encrypted data will decrypt it.

The Crypto helper subroutine is called to encrypt the data.

If the Crypto call is successful, the attribute in the item is updated with the encrypted data and a flag is set indicating that it is currently encrypted.

__ 4. Prepare the **%encrypt()** input.

This section of code is in the inline helper subroutine "Crypto".

The crypto parameter structure described in an earlier exercise is populated.

Notice that the output buffer is allocated at twice the size of the input buffer. This is to allow for the worst case scenario in which each input character is encrypted to a reserved character, which requires escaping it into two characters.

```
encryptedTextBufferSize = 2 * plainTextLength

Char encryptedText[ encryptedTextBufferSize ]
```

__ 5. Call **%encrypt()**, check the result, and extract the encrypted text.

This section of code is in the inline helper subroutine "Crypto".

The **%encrypt()** function is called.

Notice that **encryptedTextLength** argument is passed in with the "&" prefix. This passes the address of **encryptedTextLength** to **%encrypt()**, which allows it to be set. **encryptedText**, on the other hand was declared as a char array, so its value is already an address.

```
Char encryptedText[ encryptedTextLength ]
result = %encrypt( cryptoParams, plainText, encryptedText,
&encryptedTextLength )
```

Also, notice how the encrypted text is extracted. This is necessary so that anything beyond the end of the actual string is eliminated.

```
string = encryptedText[ 1, encryptedTextLength ]
```

__ 6. Add the **callx** trigger to the file defining item.

This step can be accomplished using your preferred method of editing items. The use of the Update Processor is described here.

a. From TCL, enter:

   ud secretdata

b. Press <Enter> until the cursor is on the line titled **correlative**. If you pass it, use **^B** to go back up a line.

c. Enter the following.

   callx FLE,bp, encrypt

d. Press **^XF** to save the file defining item and exit.

e. Compile and catalog the encrypt program by entering the following command:
compile-catalog FLE,bp, encrypt (o
Also, on D3 Windows, if the system is not configured to always reload subroutines, you will need to exit the line and log on again if the program has been run and then recompiled.

__ 7. Create an item.

This step can be accomplished using your preferred method of editing items. The use of the Update Processor is described here.

a. From TCL, enter:
u secretdata *<item-id>*
where ***<item-id>*** is an item-id of your choice.

b. Type something into each attribute, pressing <Enter> after each entry. If the Update Processor is not being used, make sure only attribute 1-3 have data.

c. Press **^XF** to save the item and exit.

__ 8. Display the item that was just created.

a. From TCL, enter:
ct secretdata
Notice that the phone number does not appear. This is the piece of data that was encrypted and now appears as garbage in attribute 3.
Also notice the number 1 in attribute 4. This is the flag maintained by trigger programs indicating whether or not the phone number is currently encrypted.
The **callx** dynamically encrypted the data as it was written to storage.

Exercise 3 summary: Dynamically encrypting field data when it's written to a file by using a **callx** trigger and the **%encrypt()** function.

End of Exercise 3

Exercise 4: Using a callr trigger to decrypt a field

Purpose of the Exercise

This exercise will demonstrate dynamically decrypting field data when it's read from a file by using a **callr** trigger and the **%decrypt()** function.

After this exercise you will be able to:

- Decrypt field data as it's read from a file

Exercise Instructions

Perform the following steps:

__ 1. Log in to D3.

a. Do one of the following:

- On Linux, from the shell, enter **d3**
- On Windows Telnet to localhost

b. Respond to the prompts as shown below:

```
user id: dm
master dictionary: FLE
```

__ 2. Review the FLE,bp, decrypt program.

There are multiple methods to review. For example:

- **ed FLE,bp, decrypt**, or
- **u FLE,bp, decrypt**, or
- **ct FLE,bp, decrypt**

Each section is described in its own step below.

__ 3. Main

This is the main part of the program.

A flag (part of the data in the item) is checked to see if the attribute to be encrypted is already encrypted. If it is not encrypted, it returns without doing anything. This is important. Since the decryption algorithm is symmetric, calling **%decrypt()** again with the decrypted data will encrypt it.

The Crypto helper subroutine is called to decrypt the data.

If the Crypto call is successful, the attribute in the item is updated with the decrypted data and a flag is set indicating that it is currently decrypted.

__ 4. Prepare the **%decrypt()** input.

This section of code is in the inline helper subroutine "Crypto".

The crypto parameter structure described in an earlier exercise is populated.

The **outputString** buffer is allocated at the same size as the **inputString** buffer.

This is to allow for the worst case scenario in which each input character is not decrypted from a reserved character, meaning that the **outputString** buffer will have as many characters as the **inputString** buffer, not fewer.

```
decryptedTextBufferSize = encryptedTextLength

Char decryptedText[ decryptedTextBufferSize ]
```

__ 5. Call %decrypt(), check the result, and extract the decrypted text.

This section of code is in the inline helper subroutine "Crypto".

The **%decrypt()** function is called.

Notice that **decryptedTextLength** argument is passed in with the "&" prefix. This passes the address of **decryptedTextLength** to **%decrypt()**, which allows it to be set. **decryptedText**, on the other hand was declared as a char array, so its value is already an address.

```
Char decryptedText[ decryptedTextBufferSize ]
result = %decrypt( cryptoParams, encryptedText, decryptedText,
&decryptedTextLength )
```

Also, notice how the decrypted text is extracted. This is necessary so that anything beyond the end of the actual string is eliminated.

```
string = decryptedText[ 1, decryptedTextLength ]
```

__ 6. Add the **callr** trigger to the file defining item.

This step can be accomplished using your preferred method of editing items. The use of the Update Processor is described here.

a. From TCL, enter:

   ud secretdata

b. Press **<Enter>** until the cursor is on the line titled **correlative**. If you pass it, use **^B** to go back up a line.

c. The **callx** trigger from the previous exercise should already be present. Press **^V** to insert space for another value.

d. Enter the following:

```
callr FLE,bp, decrypt
```

e. Press **^XF** to save the file defining item and exit.

f. Compile and catalog the decrypt program by entering the following command:

```
compile-catalog FLE,bp, decrypt (o
```

Note that typically the trigger program (decrypt) needs to be compiled (Flash compiled on D3 Windows), but this has already been done in the FLE demo account. Also, on D3 Windows, if the system is not configured to always reload subroutines, you will need to exit the line and log on again if the program has been run and then recompiled.

__ 7. Optionally, create an item.

This step may be skipped. The item from the previous exercise may be used. Skip ahead to the "Display the item that was just created" step.

This step can be accomplished using your preferred method of editing items. The use of the Update Processor is described here.

a. From TCL, enter:

```
u secretdata <item-id>
```

where ***<item-id>*** is an item-id of your choice.

b. Type something into each attribute, pressing <Enter> after each entry.

c. Press **^XF** to save the item and exit.

__ 8. Display the item that was just created.

a. From TCL, enter:

```
ct secretdata
```

Notice that the phone number now appears in plain text. This is the piece of data that was encrypted by the callx trigger using **%encrypt()**. It has now been decrypted by the callr trigger using **%decrypt()**.

Also notice the number 0 in attribute 4. This is the flag maintained by trigger programs indicating whether or not the phone number is currently encrypted. The **callr** dynamically decrypted the data as it was read. This is in memory only. The image in storage is still encrypted.

Exercise 4 summary: Dynamically decrypting field data when it's read from a file by using a **`callr`** trigger and the **`%decrypt()`** function.

End of Exercise 4