Reverse Engineering a book cover - writeup

FEBRUARY 3, 2017

This writeup is really related to hacking and to be more precise cracking a **real** book cover. I bought this book some time ago as a pre-order.

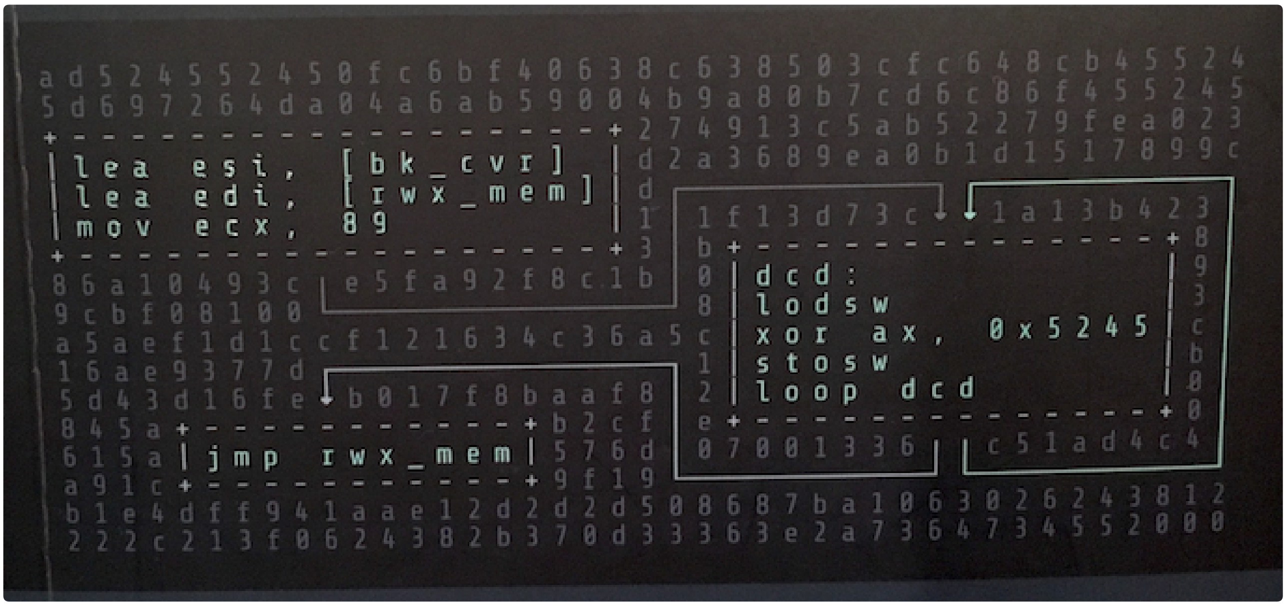
UPDATE 02/03/2017

Gynvael mentioned my solution on his livestream (starts around **10m:40s**): https://youtu.be/wJxWBeHWnGQ?t=640

The full title of this polish book is "Praktyczna inżynieria wsteczna" which we can translate to "Applied Reverse Engineering". The editorial board was Gynvael Coldwind and Mateusz Jurczyk. A lot of great people actually helped to create this book by writing articles, check it out.

Full details about that book: PWN bookstore

Book cover (take into account that the original book cover you can find on the Internet is different [without an assembly code] than this final one):



Recovering a code

Tools/Helps:

- x86 instructions - List of Assembly instructions on x86
- Kali Linux 32bit - Linux distributions with variety of tools
- EDB - x86 debugger (Pre-installed on Kali Linux)
- NASM - Assembly compiler (Pre-installed on Kali Linux)

Before we start analysing anything we need to copy the hex bytes and assembly code from a book cover. I recreated those bytes manually in a text editor without using any kind of OCR tool, it was the fastest method since we need to do that only once. No need for automation here.

HEX bytes:

```
ad524552450fc6bf40638c638503cfc648cb455245d697264da04a6ab59004b9a80b7cd6c86f
455245274913c5ab52279fea023d2a3689ea0b1d1517899cd11f13d73c1a13b4233b886a1049
3ce5fa92f8c1b099cbf0810083a5aef1d1ccf121634c36a5cc16ae9377d1b5d43d16feb017f8
baaf820845ab2cfe0615a576d07001336c51ad4c4a91c9f19b1e4dff941aae12d2d2d508687b
a1063026243812222c213f0624382b370d33363e2a7364734552000
```

How do I know that bytes from a book cover are actually hexadecimal pairs? There are a few indicators that confirm my theory:

- Only valid HEX characters within a range **A-F** and **0-9**.

- If you split the whole string into pairs of 2-bytes each you will get HEX encoded values like `0xAD`, `0x52` ...
- A length of this string is actually 359 bytes and it doesn't divide by 2 (359 % 2 !== 0) which means 358th is the last valid pair of 2-bytes and the next value is a **single** character. Nevertheless it doesn't matter here, look at 3 last bytes which are `000`, last valid HEX is `0x00` and one additional character is just a **padding** value to make a book cover look right.

Assembly code:

The full code that is ready to compile is here: [Pseudo code from a book cover ported to NASM · GitHub]

```
    lea esi, [bk_cvr]
    lea edi, [rwx_mem]
    mov ecx, 89

dcd:
    lodsw
    xor ax, 0x5245
    stosw
    loop dcd

    jmp rwx_mem
```

At first when I looked at this snippet I thought it's just a random assembly code but then I tried to expand those acronyms like `[bk_cvr]` and `[rwx_mem]`.

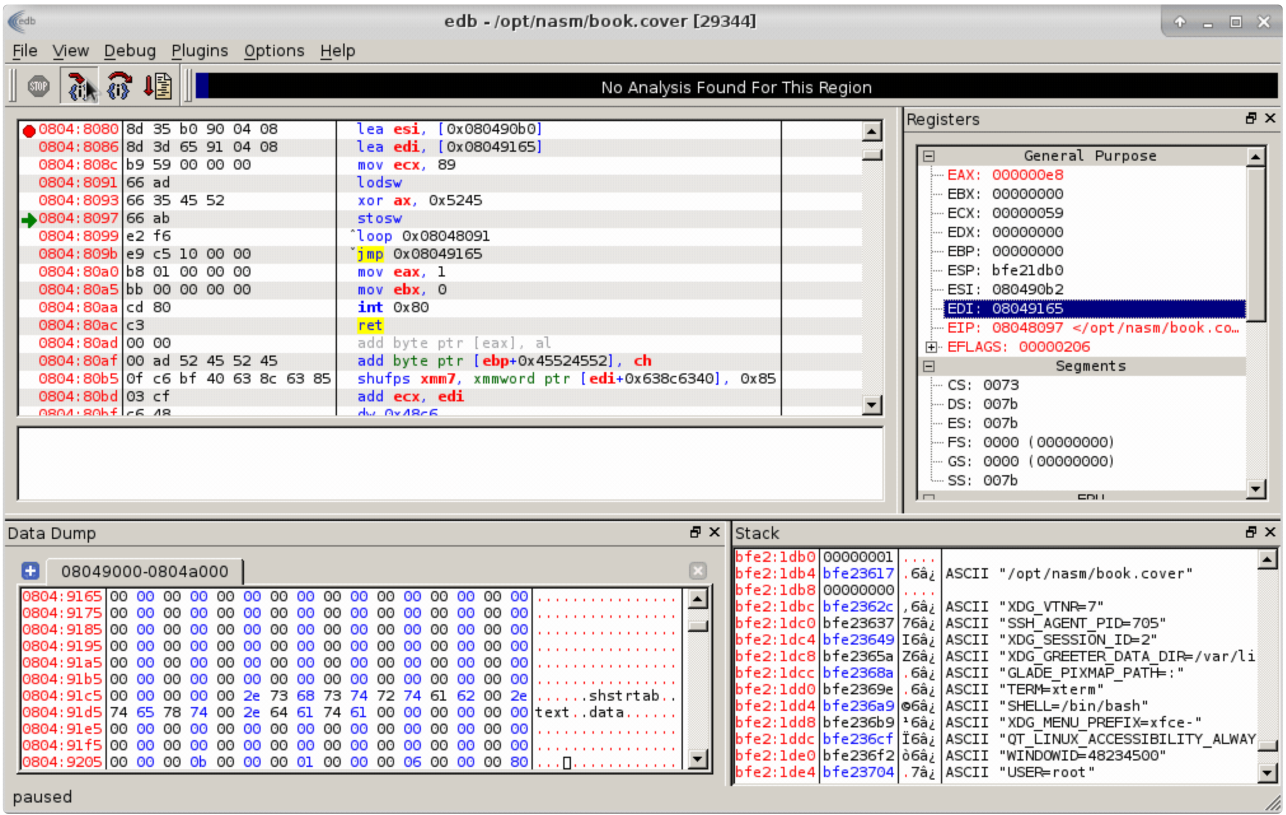
- **bk_cvr** - it expands to **book cover**
- **rwx_mem** - here is a **memory** with access rights to **r**ead, **w**rite, e**x**ecute (rwx).

This assembly fragment executes from the top to the bottom:

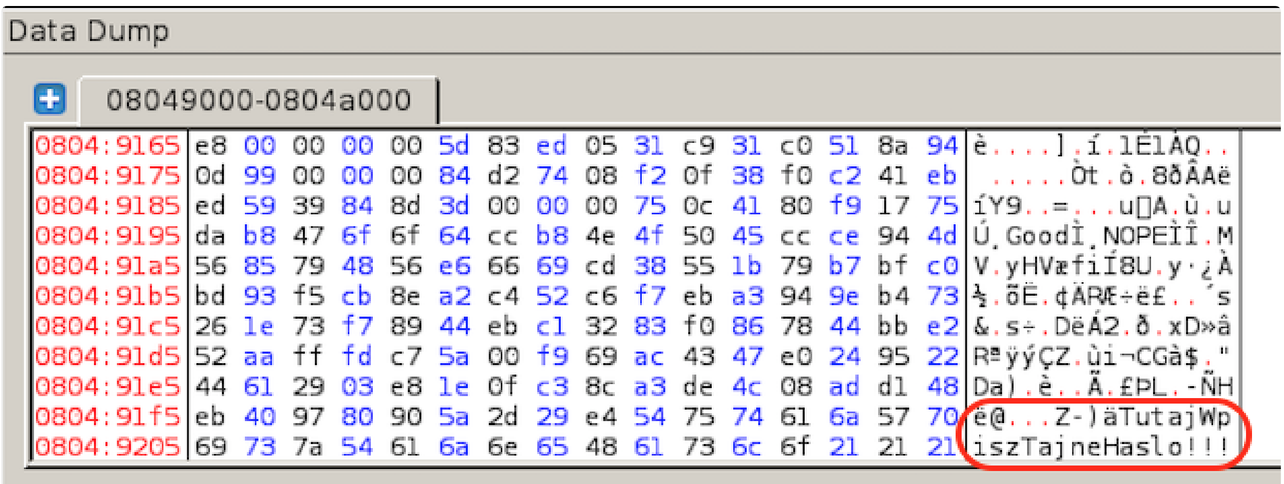
1. Assign to ESI register an address to HEX bytes from a book cover.
2. Assign to EDI register an address of the allocated memory space where we can put our decoded bytes.
3. Set the value of ECX register to 89 (decimal).
4. Set a label called "dcd" so we can reference to it from a loop or JMP directives (like with a label for goto directive).
5. lodsw - Load word at address defined by ESI (hex character from a book cover) into AX register. After a first iteration EAX = 0x000052ad.
6. XOR value in AX register with 0x5245 (AX = 0x52ad XOR 0x5245 = 0x00e8).

7. stosw - store the XOR result (AX = 0xe8) at the memory address from EDI register (rwx_mem).
8. Jump to “dcd” label and repeat the whole process 89 times (ECX), decreasing ECX every next loop iteration.
9. After the whole process finished we can jump to the address where “rwx_mem” is pointing to. It’s going to be an actual code that parses.

Decoding HEX bytes



After all bytes were decoded and copied we can find an actual string that says we need to put a password there. Mentioned string is marked by a red square on a picture bellow:



You should notice the sentence which wasn't visible before, `TutajWpiszTajneHaslo!!!`. It's a Polish sentence which translates into English as `PutYourSecretPasswordHere!!!`. I'm going to refer to this as a **hidden message**.

Bellow is the code where we jumped to from the instruction `jmp rwx_mem`:

0804:9165	e8 00 00 00 00	call 0x0804916a
0804:916a	5d	pop ebp
0804:916b	83 ed 05	sub ebp, 5
0804:916e	31 c9	xor ecx, ecx
0804:9170	31 c0	xor eax, eax
0804:9172	51	push ecx
0804:9173	8a 94 0d 99 00 00 00	mov dl, byte ptr [ebp+ecx+153]
0804:917a	84 d2	test dl, dl
0804:917c	74 08	jz 0x08049186
0804:917e	f2 0f 38 f0 c2	crc32 rax, dl
0804:9183	41	inc ecx
0804:9184	eb ed	jmp 0x08049173
0804:9186	59	pop ecx
0804:9187	39 84 8d 3d 00 00 00	cmp dword ptr [ebp+ecx*4+61], eax
0804:918e	75 0c	jnz 0x0804919c
0804:9190	41	inc ecx
0804:9191	80 f9 17	cmp cl, 23
0804:9194	75 da	jnz 0x08049170

Mentioned code is pretty straightforward and what it does is a simple iteration over our hidden message to generate the CRC32 checksum. We need to be aware of that the assembly instruction `CRC32C` is actually not a standard CRC32 function which is commonly used. There is a special instruction set called SSE4 which introduced this function in the Intel processors - SSE4 - Wikipedia. It's **CRC32C** which does return a different result than a standard function.

Within this loop which repeats exactly 23 times (the string length of `TutajWpiszTajneHaslo!!!`) we calculate CRC32C checksum:

- **1st** iteration: CRC32C is calculated using `TutajWpiszTajneHaslo!!!`.
- **2nd** iteration: CRC32C is calculated using `utajWpiszTajneHaslo!!!`.
- **3rd** iteration: CRC32C is calculated using `tajWpiszTajneHaslo!!!`.

Every next iteration CRC32C checksum calculation starts at the **n-th** letter. At the end this code will calculate a checksum **only from the laster letter** - that's an important information.

Instruction `cmp dword ptr [ebp+ecx*4+61], eax` is checking if generated CRC32C value is the same as one stored at the address `ebp+ecx*4+61` (it's an array of checksums). If it does match then we start another loop iteration, if not then code "stops".

The obvious thing is that I didn't have any checksum that was matching those checksums referenced within an array, even after the first loop iteration my generated CRC32C was invalid. At first I thought maybe this XOR decoding routine broke something and that's the reason checksum doesn't match…

There is an array that stores 23 different CRC32C checksums that are compared to those generated from `TutajWpiszTajneHaslo!!!`. After some checks I realised that code is actually valid and XOR decoding procedure also works fine.

Magic checksums

If you modify mentioned hidden message your CRC32C checksum will change. Here comes a tricky part, if we have an array with 23 valid checksums we need to generate a **new** hidden message that will match exactly those values.

Not only the checksum of the whole hidden message needs to be valid but also every its character. It means that finding the collision in CRC32C will not resolve our issue, we need to **brute force** that. We know exactly what is a value of CRC32C checksum in every step of our loop.

Pseudocode for mentioned brute-force script:

We iterate from the last character to the first one, in a reverse order.

1. Dump a list of all valid checksums (array) from the memory.
2. Iterate from the last CRC32C value in that array.
3. Generate checksum for every character in a range 0-255.
4. If generated checksum does match then move to the next valid checksum and repeat that process.
5. At the end we're going to receive a valid string that matches to all checksums.

I dumped the memory after XOR operation directly from EDB to a file named `decoded.dump.bin`.

PHP snippet that extracts valid checksums from a memory dump:

```
<?php
$file = file_get_contents('decoded.dump.bin');
$offset = 0x19E;
```

```
$crc32Table = array();
$fakePassword = 'TutajWpiszTajneHaslo!!!';
$numOfCrcItems = strlen($fakePassword);
$wSz = 4;
function get4byteVal($str) {
    $res = '';
    for ($i = 3; $i >= 0; $i--) {
        $res .= sprintf('%02x', ord($str[$i]));
    }
    return sprintf('0x%s', $res);
}
for ($i = 0; $i < $numOfCrcItems; $i++) {
    $seek = $offset + ($wSz * $i);
    $crc32Table[] = get4byteVal(substr($file, $seek, $wSz));
}
print_r($crc32Table);
```

As a result the list of checksums is:

```
Array
(
    [0] => 0x564d94ce
    [1] => 0x56487985
    [2] => 0xcd6966e6
    [3] => 0x791b5538
    [4] => 0xbdc0bfb7
    [5] => 0x8ecbf593
    [6] => 0xc652c4a2
    [7] => 0x94a3ebf7
    [8] => 0x2673b49e
    [9] => 0x89f7731e
    [10] => 0x32c1eb44
    [11] => 0x7886f083
    [12] => 0x52e2bb44
    [13] => 0xc7fdffaa
    [14] => 0x69f9005a
    [15] => 0xe04743ac
    [16] => 0x44229524
    [17] => 0xe8032961
    [18] => 0x8cc30f1e
    [19] => 0x084cdea3
    [20] => 0xeb48d1ad
    [21] => 0x90809740
    [22] => 0xe4292d5a
)
```

Brute-force

We have everything we need to know, how to find valid checksums and write brute-force code. I struggled here for a while because every software implementation of CRC32C I did find wasn't generating a valid result. I tried C, Python and Node.JS libraries, without a success. That was the main reason why I decided to write this brute-force script in a pure assembly. I can use there built-in hardware support for CRC32C instruction without a need to actually use any external libraries.

I was using NASM for that, here is a code:

```
; compile: nasm -f elf poc.decode.flag.asm; ld -m elf_i386 -s -o
poc.decode.flag poc.decode.flag.o
; author: @radekk
; -------------------------------------------
bits 32

section .data
    string times 25 db 0
    sizeOfString equ $-string
    crc32table dd 0x564d94ce, 0x56487985, 0xcd6966e6, 0x791b5538,
0xbdc0bfb7, 0x8ecbf593, 0xc652c4a2, 0x94a3ebf7, 0x2673b49e, 0x89f7731e,
0x32c1eb44, 0x7886f083, 0x52e2bb44, 0xc7fdffaa, 0x69f9005a, 0xe04743ac,
0x44229524, 0xe8032961, 0x8cc30f1e, 0x084cdea3, 0xeb48d1ad, 0x90809740,
0xe4292d5a

section    .text
    global _start

_start:
    xor esi, esi
    mov ebx, 22         ; number of crc32 hashes (index)
    call _brute

    call _print
    call _close
    ret

_brute:
    mov ecx, 255        ; max ascii code for character

_loopA:
    xor edx, edx
    xor eax, eax
    crc32 eax, cl
    cmp esi, 1
    je _crc32
    jmp _cmp

_crc32:
    inc edx
    crc32 eax, byte [string + ebx + edx]
```

```
        cmp byte [string + ebx + edx + 1], 0
        jne _crc32

    _cmp:
        cmp eax, [crc32table + ebx * 4]
        je _found

        loop _loopA
        ret

    _found:
        mov esi, 1
        mov byte [string + ebx], cl
        dec ebx
        jmp _brute

    _print:
        mov edx, sizeOfString
        mov ecx, string
        mov ebx, 1
        mov eax, 4
        int 0x80
        jmp _close

    _close:
        mov eax, 1          ;system call number (sys_exit)
        mov ebx, 0
        int 0x80            ;call kernel
        ret
```

As a result it prints out a new valid hidden message:

```
root@kali:/opt/nasm $> ./poc.decode.flag; echo ""
coldwind.pl/piwflag1337
```

Our flag is: `coldwind.pl/piwflag1337`

This is an actual URL address http://coldwind.pl/piwflag1337 where details for winners were described.

← All posts

Radek
Blogging about applications insecurity

Tweet Share