Reversing Basics – A Practical Approach

Author: Amit Malik (DouBle_Zer0)

E-Mail: m.amit30@gmail.com

Note: Keep Out of Reach of Children/Danger-Software Poison.

Download EXE/Crackme: https://sites.google.com/site/hacking1now/crackmes

Introduction:

Reverse engineering is a very important skill for information security researchers. In this tutorial I will explore the basic concepts of reverse engineering by reversing a simple crackme.

The crackme used in this tutorial is from binary auditing course. I will use static approach to solve the problem as it clearly demonstrates the power of reverse engineering. A little bit knowledge of Assembly and Disassemblers, Debuggers is required to understand this material. I will use IDA Disassembler as it is the most powerful disassembler exists in the market, Hexrays provide a demo version of IDA and I think demo version is enough for solving this exercise but I am using version 5.1.

Reverse Engineering A Crackme:

Reverse engineering can be easy and can be difficult, it depends on your target. But the basic steps are:

1) Detect packer/encryptor -> if present, then first unpack/decrypt the file and fix imports etc.
2) Static Analysis -> Understand the application logic without executing it in live environment.
3) Dynamic Analysis -> Execute the binary and monitor the application activities.

Above steps are just basic of reverse engineering, overall process is based on reverser goals. For eg: for AV researchers and crackers basic steps are same but process is different.

Some Important Terms:

API(Application Programming Interface): In windows world, code sharing is the core of communication and trust. A user application can't directly control hardware or can't directly communicate with windows kernel. So how would application work if application can't talk with the kernel ?. Windows provide various DLLs(Dynamic Link Library) and these DLLs exports various functions to provide services to user applications and we call them API. So the understanding of APIs is necessary.

Eg: if you are using printf() function in your code and the linker links the function call to the printf() function in msvcrt.dll, so the printf() function is an API : Read PE file format if you want to know how these things work.

The second important thing is that every function/API mostly returns to EAX register. For eg: lets say we are using strlen() to calculate the length of the string, strlen() will return the value into EAX register.

The situation is something like : int a = strlen(const char *); then asm is mov dword ptr[ebp-2c],EAX where [ebp-2c] is a.

Ok its time to fire up our crackme.

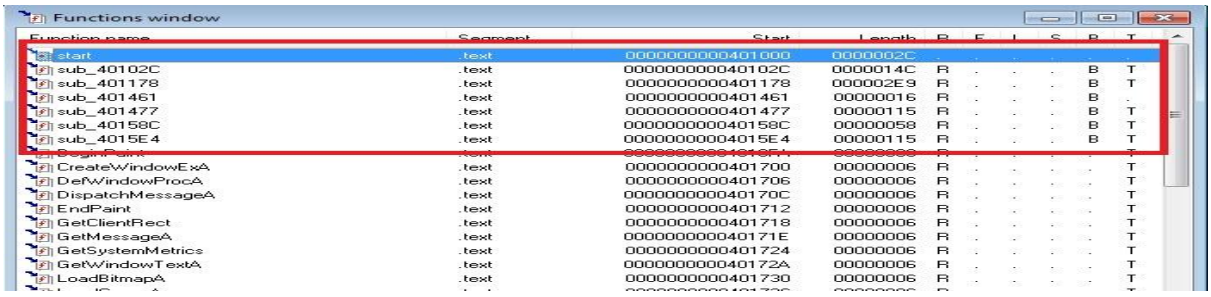
Our tasks are:

1) Remove splash screen -> i am leaving this task for you ☺
2) Find Hard Coded Password -> we will work on this.
3) Write a keygen -> we will work on this.

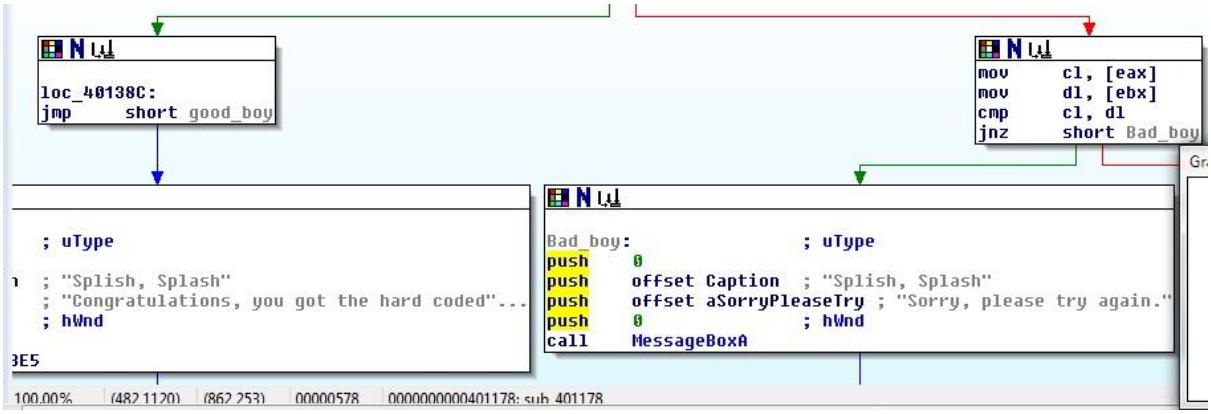
Load file into IDA Pro. (if you don't know how to operate with IDA then first read "THE IDA PRO BOOK" excellent guide for IDA usage)

Go with default options and you will see IDA is processing the file, you can start the analysis now. One of the most important thing is to look on the Import and Export function tabs to get a compact view that how many and what api is our target application using. Now run the application independently, I mean like a normal application not under debugger and feed some garbage value and note the messages that we get.

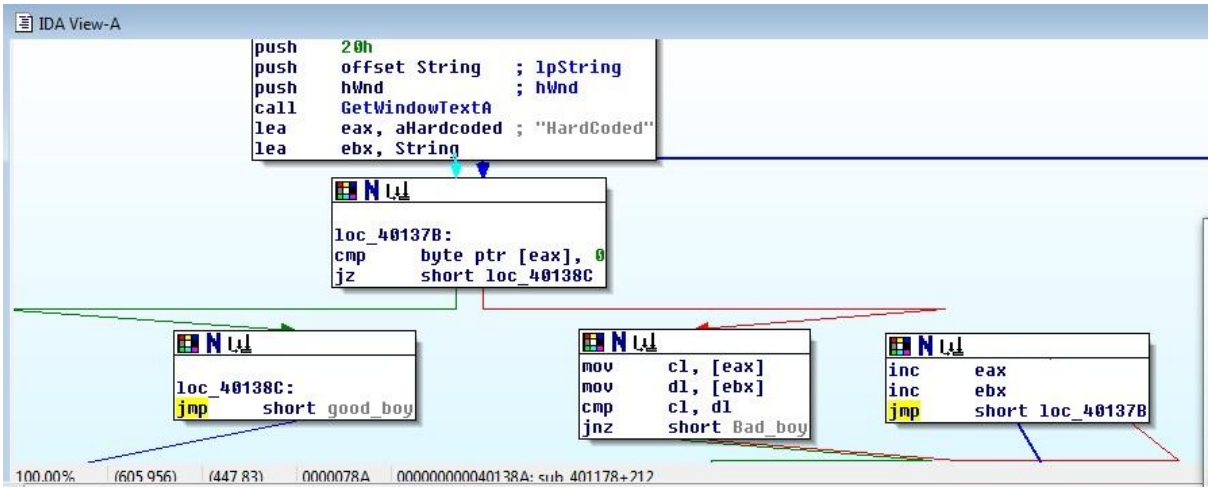
As you can see in the picture that our crackme is popping up a message box on invalid input. The String "Sorry, please try again" is important or you can say that this string will save a lot of work, situation may vary with target to target but for this crackme this string can be the starting point. But as we can see that IDA is showing the starting function and we don't have any string that can match with the error message i.e "Sorry, Please try again". Now we have two approaches one is trace the call from start function to the function that is containing our magic string. For eg. Go into call **sub_40102c** and do the same within this function and another approach is to go to function tab in IDA and analyse each function independently. Generally we use the combination of both to manage the analysis time.



As you can see in the picture that the function name starting with sub_* are user defined functions, we will open each function and look for our magic string i.e “Sorry, Please try again”. Continue with this process we can find our magic string in function **sub_401178.**



As we can see in the picture that we have now clear targets, now we can backtrace and can find out the starting point of string matching.



In the first box we can see that application is calling a API **GetWindowTextA**. If you don’t know the api functionality then in this case you can search on msdn win api reference guide. The guide will provide you the parameter meanings, structure and expected return values etc.

GetWindowText Function

Copies the text of the specified window's title bar (if it has one) into a buffer. If the specified window is a control, the text of the control is copied. Howe
cannot retrieve the text of a control in another application.

Syntax

```
int WINAPI GetWindowText(
  __in   HWND hWnd,
  __out  LPTSTR lpString,
  __in   int nMaxCount
);
```

Parameters

hWnd [in]

Type: **HWND**

A handle to the window or control containing the text.

lpString [out]

Type: **LPTSTR**

The buffer that will receive the text. If the string is as long or longer than the buffer, the string is truncated and terminated with a null character.

Now compare this format with the format that is displayed by IDA, we can say that **PUSH OFFSET String** will receive the data entered by user. Now notice the next two statements **LEA EAX, aHardcoded and LEA EBX, String**, that means the address of a hardcoded string is moved into EAX and the address of the string that is entered by user is moved into EBX. Now we can say that the aHardcoded contain our hardcoded password because application is matching this string with the user entered string.

MOV CL,[EAX] ; hardcoded string [one char each time]

MOV DL,[EBX ; string entered by user [one char each time]

CMP CL,DL ; compare

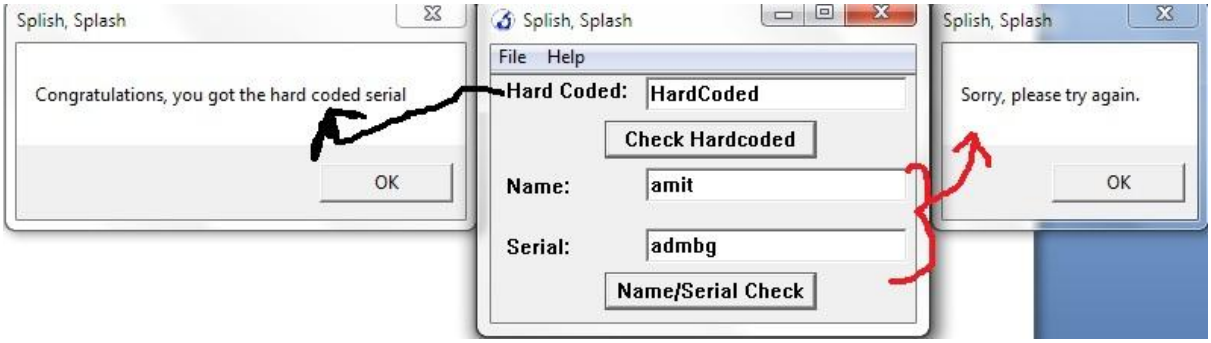
JN Z SHORT_BAD_BOY ; if no match call bad_boy

INC EAX ; else increase eax,ebx

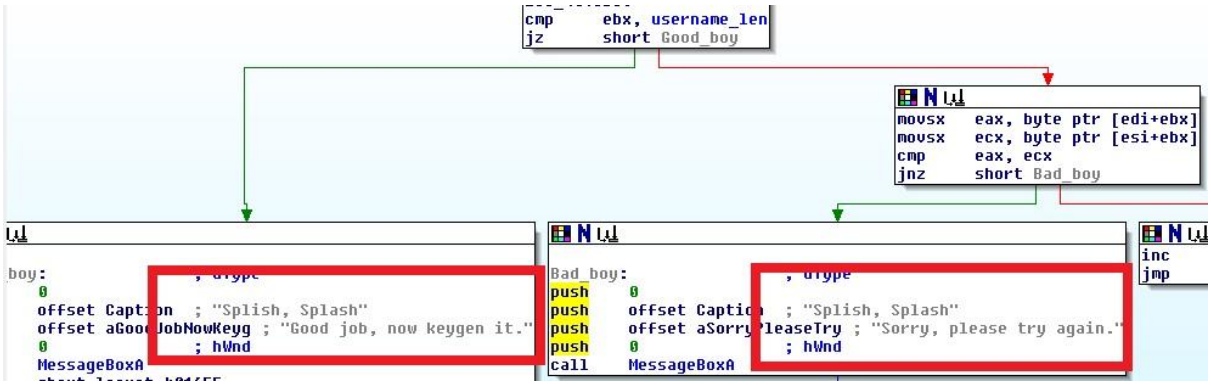
INC EBX

JMP @Loop ; jump back to loop

The above loop is for string matching, so now we are sure that the aHardcoded contain our harcoded password and that is **HardCoded.**



Now we have to find out the solution of second challenge. But if we look into the current function we have only solution for hardcoded one so it means we have to jump to another function to find out the solution for second challenge. Continue with the same process [jumping to functions from function tab for our magic string] we can say that the function **sub_4015e4** is the next target for analysis.



Now we will backtrace to find out the origin of these message boxes and then figure out that what value will invoke good_boy message box. Starting point of this function is the origin of these message boxes because at the beginning application is calling two **GetWindowTextA** and we know the purpose of this API from our previous challenge. So application is expecting Name and Serial from user. If you look at the code then we can say that if we don't fill any values into the fields then we get a message box like "Please enter username or please enter a Serial". Now what if we enter a garbage value to the fields? Then we will enter into a simple computation and we have to reverse that logic.

mov username_len, eax ; move the value of eax (username length – returned by API) into a variable

xor ecx, ecx ; clear out ECX register

xor ebx, ebx ; clear out EBX register

xor edx, edx ; clear out EDX register

lea esi, username_stor ; move the address of username (a[] = "amit" then ESI = a[0]) into ESI

lea edi, user_gene ; destination buffer where we want to store username after computation (like b[] = a[0] ^ 2; it is just a example)

mov ecx, 0Ah ; move the value 10 (decimal) into ecx

loop:

movsx eax, byte ptr [esi+ebx] ;ESI=address of username and EBX=0 -> look above

cdq ;nothing special in this application used to convert dword into quadword

idiv ecx ; divide the value of EAX with ECX (EAX/10 –> as ECX = 10) -> look above, now idiv instruction store the quotient into EAX and Remainder into EDX so we can say (EAX%10 = EDX)

xor edx, ebx ; xor the value of EDX with EBX (notice that EBX will play as a counter)(EDX=EDX^EBX)

add edx, 2 ; ADD two into EDX (EDX = EDX+2)

cmp dl, 0Ah ; now compare DL (EDX) with 10

jl short loc_401646 ;if EDX < 10 jump to loc_401646

sub dl, 0Ah ; else subtract 10 from DL(EDX)

loc_401646:

mov [edi+ebx], dl ; move the value of DL(EDX) into EDI (EDI is our destination) -> look above

inc ebx ;increment EBX (notice that EBX work as a pointer or you can say like i in a loop ☺)

cmp ebx, username_len ;compare the value of EBX with length of username

jnz short loop_username ; jump if EBX != username_length to the (loop) -> look above

so now we can generate a pseudo code of these instructions

let say a[] = “amit” is our source string

and b[] is our destination

then

c = 10 ; ECX

i = 0 ; EBX

loop:

b[i] = a[i] % c;

b[i] = b[i] ^ i;

b[i] = b[i]+2;

if (b[i] > 10)

    b[i] = b[i]-10;

I++;

If(i != strlen(a))

    Goto loop;

So this is the logic of username computation and we have our computed value into b[].

Now Serial Computation:

xor ecx, ecx ; clear out ECX

xor ebx, ebx ; clear out EBX

xor edx, edx ; clear out EDX

lea esi, serial_stor ;move address of user entered serial into ESI

lea edi, serial_gene ; target buffer means store serial after computation (similar to username process)

mov ecx, 0Ah ; move 10 into ECX

loc_401669:

movsx eax, byte ptr [esi+ebx] ; move the value of serial into EAX (one char ☺)

cdq ; nothing special for this application

idiv ecx ; divide EAX with 10;

mov [edi+ebx], dl ; move the remainder into EDI (our destination)

inc ebx ; increment pointer (means i)

cmp ebx, serial_len ; comare it with the length of serial

jnz short loc_401669

now the result of this computation is stored into another array let say c[];

in the next few instructions we have a loop in the application that will match the value of c[] with b[].

Notice that b[] hold the result of username computation where c[] hold the result of serial computation.

as we can see in above code that application is only dividing the serial number entered by user with 10 and comparing the result with the result of username computation, so we can say that if x is the result of username computation then x * 11 is the value of serial.

So now we can develop the keygen:

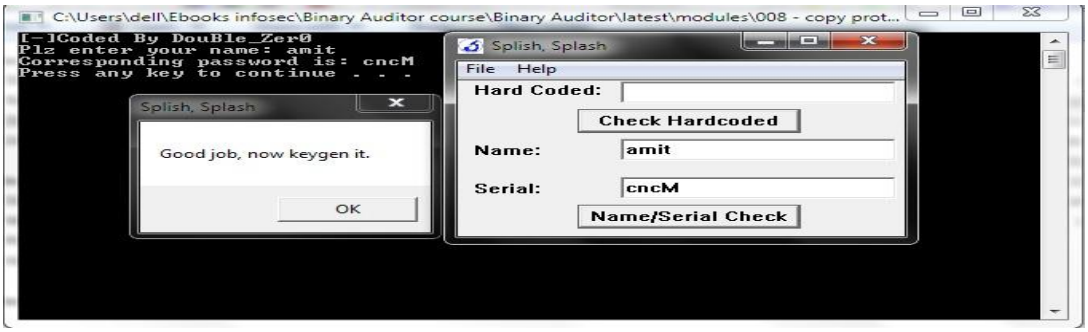
```
/*
Author: DouBle_Zer0
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
   char a[10];
   char b[10];
   int c;
   int i;
   int d;
```

```
    printf("[-]Coded By DouBle_Zer0\n");
    printf("Plz enter your name: ");
    scanf("%s",a);
    d = strlen(a) - 1;

    for(i=0;i<=d;i++)
    {
                    c = a[i] % 10;
                    c = c ^ i;
                    c = c + 2;
                    if (c > 10)
                    {
                        c = c - 10;
                        }
                        b[i] = c * 11;
    }
   b[i] = 0;
   printf("Corresponding password is: %s\n",b);
   system("pause");
   exit(0);
}
```

And the Output is:



References:

1. http://www.binary-auditing.com/
2. [IDA PRO] http://www.hex-rays.com/index.shtml
3. [Challenge EXE, keygen source] https://sites.google.com/site/hacking1now/crackmes
4. [Video Tutorial] http://vimeo.com/18821178