Beating an SEH/VEH based CrackMe :

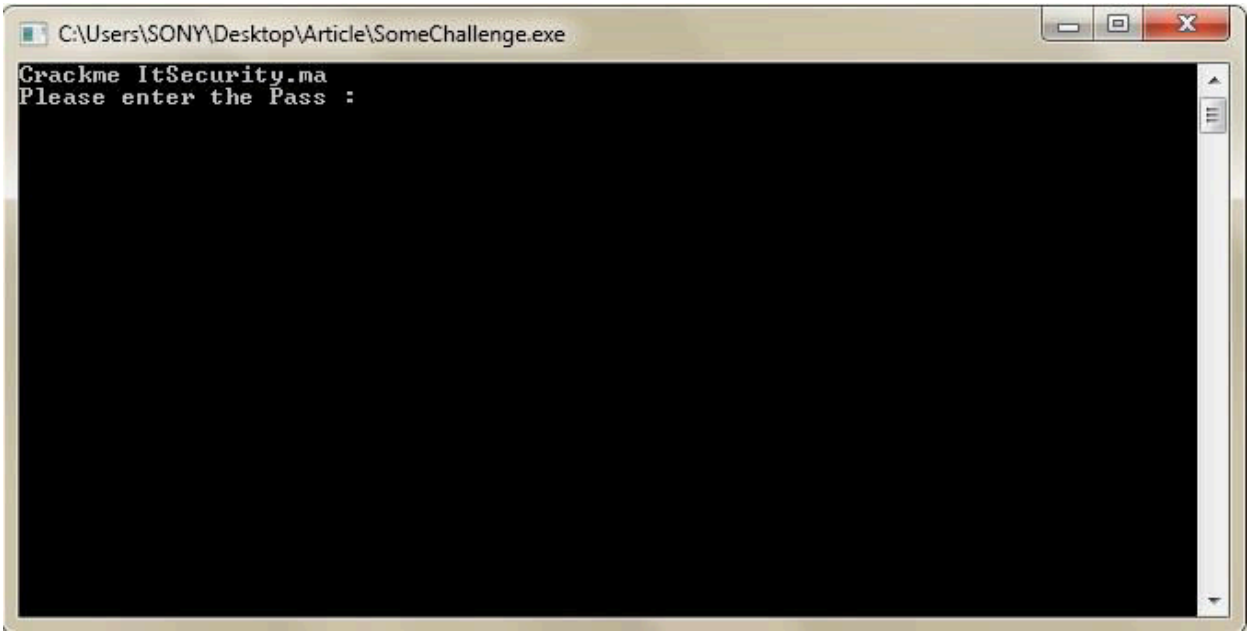
Deep Analysis Is The Key To Beat A Crackme :

By : Souhail Hammou
(Dark-Puzzle)

http://www.itsecurity.ma/

Introduction :
In this paper , I will try to show how to beat an advanced crackme that is using an interesting way to calculate the length and it's generating exceptions to be dealed with in order to return values into 32-bit registers such as EAX register , the key to beat a crackme is deep analysis through what it dœs under the hood especially when it's using mixed methods to confuse,stop or slow the reverser.
Let's Start :

This Crackme was taken from a very popular challenge website that I will not mention , I edited the strings printed in the interface in memory not to spot the website . I was also the 16th person to validate it (Validation rate 1%).
Let's start by opening the CrackMe and see what it's waited from us to do !!



It asks us politely to type a pass or to Crack it I guess.
Open your mind and carry on .
Now we need to take a quick look on what routines are exactly dealing with the user input . Let's switch to Immunity and take a quick look.

You can see that it is taking a user input then calling an adresse sepcified by EBX register after that it's deciding weither printing the success or fail message.
We are now interested in what's directly going after getting the user input using scanf so let's see what EBX holds and step into that call.

EBX isn't taking us farther but just below this code a little bit.
The instructions which EBX will take us to are the ones responsible for checking the user input and deciding weither it's right or not.
The responsible routine is a little bit long and it's splitted into 4 main parts each part ends with a JE (Jump If Equal) instruction.
So let's take care of each part alone :

1st Part – Checking the length :

Here are the instructions :

```
00CC107F      90                NOP
00CC1080      BE DD657A22       MOV ESI,227A65DD
00CC1085    . 31ED              XOR EBP,EBP
00CC1087    . BB BEBAADDE       MOV EBX,DEADBABE
00CC108C    . 01DE              ADD ESI,EBX
00CC108E    . F9                STC
00CC108F    . BF D9697A22       MOV EDI,227A69D9
00CC1094    . FD                STD
00CC1095    . B9 00040000       MOV ECX,400
00CC109A    . D6                SALC
00CC109B    . 01DF              ADD EDI,EBX
00CC109D    . F3:AE             REPE SCAS BYTE PTR ES:[EDI]
00CC109F    . FC                CLD
00CC10A0    . 83E9 0F           SUB ECX,0F
00CC10A3    . 74 03             JE SHORT SomeChal.012810A8
```

We can see that DEADBABE will be added to 227A65DD which will make ESI holding the memory address that specifies the user-input, then the next instruction will try to set the CarryFlag which is already set , the next instruction that may attract your attention is at address 00CC109D this is the address that will actually calculate the input string length . How did I know it ? I will explain.
You can see that the value 400 is moved to ECX , you can also remark that 227A69D9 is moved to EDI then EBX is added to it , the result will be stored at EDI for sure. Before the ADD instruction we have a VERY important instruction which is SALC , this instruction will Set the AL value to FF if the CF is set or to 00 if the CF is cleared . In our case CF is set , so the value of AL will be FF , this value is very important because the SCASB instruction will try to find all bytes that aren't matching AL starting at ES:[(E)DI] . In addition, here we have the REPE instruction that is accompaigned with the SCASB instruction so it will try to use the ECX register to specify a search « array » , you can clearly see that ECX register was set to 400. Now , go and check what EDI is holding after the ADD instruction you will see that it's holding the value 00CC2497 . Follow this value in dump and you will find yourself in front of a bunch of «FF » , you see now that ECX holds the value 400 , this means that the search array will go to zero in other words and in theory the search will end when ECX will hold the value 00000000 , which make us figure out that the instruction will search for the first value that is different from « FF » from 00CC2497 until (00CC2497 – 400) = 00CC2097 and if no different values from FF were found ECX will just hold 00000000 . When following 00CC2097 in dump you will find what follows :

Here, the REPE SCASB instruction will stop in the last highlighted NULL byte in blue « 00 » because it is different from « FF » here ECX will hold the length from 00012097 until the value before the null byte. In my case here (input 123456) ECX will hold the value 9 because we should begin the counting from 0 then 1 then 2 until reaching 9 means reaching 000120A0.

Now that we know how the length is calculated we should figure out what length this crackme needs. In this phase we don't care about if the serial is right or not because we just want to get throught the first condition in a right way.
You can see in the last two lines that we will substract 0F from ECX then Jump if ZF=1 or not jump if ZF=0 , in other words if the ECX = 00000000 after the substruction the ZF will be set if not it will still equal 0.
So basically after the REPE SCASB instruction ECX should hold 0F which equals 15 in decimal . So we just need to insert a string with 12 charachter length and he jump will be taken .

2nd Part – First 4 bytes of the flag :

```
00CC10A5    . 83CD 01          OR EBP,1
00CC10A8    > AD               LODS DWORD PTR DS:[ESI]
00CC10A9    . BA 5930645A            MOV EDX,5A643059
00CC10AE    . 31D0             XOR EAX,EDX
00CC10B0    . 3D 0E030816            CMP EAX,1608030E
```

As the conditional jump was taken you will fall directly into the second instruction which is LODS DWORD PTR DS:[ESI], this instruction will basically load the DWORD DS:[ESI] value into EAX register this value should be the first 4 characters that we wrote in our flag in decimal and also converted to little endian so if the first 4 characters that you entered were 1234 then EAX should hold after this instruction 34333231. After that we see that a DWORD is moved to EDX then EAX is Xored with it , this is almost the same case that I coded in CrackMe#3 at Hackathon Challenge . The right value of EAX after xoring it with EDX should be 1608030E so the first DWORD of our flag is 1608030E Xored with EDX . Which will give you that value : XOR 1608030E, 5A643059 = 4C6C3357 you will just have to convert it to big endian and you will have the first 4-bytes of the flag : 57336C4C which is « W3lL » in ASCII.
Now just type W3lL and type 8 random characters after it and you will see that ZF will be set after the compare and the jump will be taken.

3rd part – Second 4-bytes of the flag (SEH) :

The 2 first parts were fun , now m0re . Let's see the instructions :

```
00CC10B7    . 83CD 01          OR EBP,1
00CC10BA    > 8B1D 8F20CC00    MOV EBX,DWORD PTR DS:[CC208F]
00CC10C0    . 81EB 00100000          SUB EBX,1000
00CC10C6    . 53                     PUSH EBX
00CC10C7    . 64:FF35 000000>        PUSH DWORD PTR FS:[0]
00CC10CE    . 64:8925 000000>        MOV DWORD PTR FS:[0],ESP
00CC10D5    . AD                     LODS DWORD PTR DS:[ESI]
00CC10D6    . BB 65425F49            MOV EBX,495F4265
00CC10DB    . CD 01            INT 1
00CC10DD    . 31D8             XOR EAX,EBX
00CC10DF    . 3D D786D318            CMP EAX,18D386D7
00CC10E4    . 74 03                  JE SHORT SomeChal.00CC10E9
```

Like the last part, we will fall directly into the second instruction which will move a DWORD from memory to EBX register , after that a substruction of 1000 will be done to EBX which will carry now 00CC1530 . This adress is the new adresse of the exception handler which will be set in a while , EBX will be pushed then the new exception handler will be completely created when moving ESP into DWORD PTR FS:[0] . After that the second 4 bytes of the user-input will be placed into EAX register in little endian format , then a value that will xor EAX is moved into EBX.

Here where the TRAP is : the « INT 1 » instruction.
We can see here that when we will step over this instruction using « F8 » the EIP will just hold directly the adresse 00CC10DF , so we don't have to step over this instructions but let run normally the crackme as it was executed outside a debugger . Basically the INT 01 instruction is called single-step break it will run after each instruction if the TrapFlag is set . Nevertheless, here it's invoked directly inside the code and the TF is cleared which will generate an exception and never set the TF. Let me explain to you what is exactly happening when the « INT 1 » is passed through in normal execution and not by single stepping through it , keep in mind that this INT instruction will generate an exception that will be handeled by the SEH newly created . Basically when we will trigger this interrupt the processor will go into the 1st location in the Interrupt Vector Table which starts in memory location 0x00 and ends at 0x3FF simply because interrupts can take a number which is between 0 and 255. After that the IP will be saved and also the CS , this basically will store 4 bytes (IP = 2 bytes & CS = 2 bytes) , before the interrupt will hand back the flow of execution to the program normally it will return using an « iret » instruction . Here the IMPORTANT PART that the CS:IP and all FLAGS are restored again.

So basically when the instruction PUSH EBX at 00CC10C6 is executed it will indicate the current SE Handler which means the instructions that will deal with an exception , the exception here is triggered by the « INT 1 » instruction and the execution flow is moved directly into 00CC1530 , after returning the exception will be handeled and the execution flow will continue normally . The only thing you need to do is just set a breakpoint on the instruction after the « INT 1 » instruction because the EIP will be incremented by 2 and it will skip that instruction. After we will return from the Exception handling routines we will see that EAX will hold a return value that is ADDed to the previous value that was holded by EAX.

Now let's work on finding that god damn second part of the validation flag. Pretend that I didn't say that the return value stored in EAX isn't added to its previous value so here you can just see after stepping over the « INT 1 » that the value of EAX will change. So we need to figure out if the EAX holds an adresse that have been moved , added or substracted to it. In order to do it let's rerun our Crackme inside a debugger for sure . Now we will enter this input for example : W3lL11119876 the DWORD that will be treated in this part is 31313131 (111 in ASCII) so let's step over the LODSD instruction and you will see that EAX is filled now with 31313131. As I said previously , you have to set a bp at 00CC10DD then step over it using <shift + F8> BUT we don't want to do that now because this will make the value of EAX change and we will need to figure out what arithmetic operation is done when the value that is returned by interrupt will be Moved , added , substracted , multiplicated by the current value of EAX. So here what I've done is that I went and edited the value of EAX just before executing the interrupt to NULL , EAX = 00000000 So I will not need to bruteforce each arithmetic operation if it's an ADD so EAX will hold a value if it's a multiplication EAX will still hold 0 , division either 0 or an exception … etc
So , after executing the Interrupt I realized that EAX holds the value 21486553 , let's covert this to big endian and to ASCII cause it's printable =) … we will finally have 53654821 = SeH!
If you want to be more sure if the operation is an addition just go and change EAX to 00000001 and you will get 21486554 which is in big endian + ASCII : TeH! .
Okey so now after we knew what is the value returned by the interrupt we must know what is the right value that EAX should hold before the XOR instruction. That's simple , we see that EAX is compared to 18D386D7 after being Xored and it's Xored with 495F4265 , so just before the XOR and just after « INT 1 » EAX should hold :
518CC4B2 (Xoring 18D386D7 with 495F4265) . Okey now we found what value EAX

should hold just after the « INT 1 » instruction and we know that after the interrupt 21486553 is added to EAX register . Sooo the right value of EAX after the LODSD instruction is 518CC4B2 – 21486553 = 30445F5F int big endian 5F5F4430 and in ASCII : __D0 . So now the 8 first characters of the flag are W3lL__D0 . Let's try to rerun the crackme and enter this serial : W3lL__D09876 . By stepping throught instructions until the Jump if equal in this part (don't forget the bp) , you will see that the ZF will be set and the jump will be taken simply because the comparison went true and those 4 bytes are the correct ones.

4th part – The last 4 bytes of the flag (VEH) :

Here are the instructions :

```
00CC10E6    . 83CD 01          OR EBP,1
00CC10E9    > 8B1D 9320ED00    MOV EBX,DWORD PTR DS:[ED2093]
00CC10EF    . 81EB 00100000          SUB EBX,1000
00CC10F5    . 53                     PUSH EBX
00CC10F6    . 6A 01            PUSH 1
00CC10F8    . E8 151F0000      CALL 00CC3012 ; Add VEH
00CC10FD    . A3 9720ED00            MOV DWORD PTR DS:[ED2097],EAX
00CC1102    . 54                     PUSH ESP
00CC1103    . 5F                     POP EDI
00CC1104    . AD                     LODS DWORD PTR DS:[ESI]
00CC1105    . BB 53664074      MOV EBX,74406653
00CC110A    . 895D 00          MOV DWORD PTR SS:[EBP],EBX
00CC110D    . 31D8             XOR EAX,EBX
00CC110F    . 3D F332768E      CMP EAX,8E7632F3
00CC1114    . 74 03            JE SHORT SomeChal.00ED1119
00CC1116    . 83CD 01          OR EBP,1
00CC1119    > FF35 9720ED00          PUSH DWORD PTR DS:[ED2097]
00CC111F    . E8 F41E0000      CALL 00CC3018 ; Remove VEH
00CC1124    . 64:8F05 000000>        POP DWORD PTR FS:[0]
00CC112B    . 83C4 04          ADD ESP,4
00CC112E    . C3                     RETN
```

We can see from a general view that these instructions are building a Vectored Exception Handler (VEH) which will deal with an exception executing a routine present at the instruction pointed by EBX , pushing a second Nonzero argument indicates that the VEH is inserted into the very head of the list then it's Removed after executing a bunch of instructions that will check how is the last DWORD of the user-input is correct , those instructions are containing an exception at adresse 00CC110A.
But first what is a Vectored Exception Handler . According to MSDN :

– Vectored Exception Handling is new as of Windows XP.
– All information about VEH are stored in the Heap.
– Vectored exception handlers are explicitly added by your code, rather than as a byproduct of try/catch statements.
– Handlers aren't tied to a specific function nor are they tied to a stack frame.

So basically to be sure that an excpetion is trigerred and dealed with we have to put a breakpoint on the first instruction that is executed by the VEH which will be the EBX register pushed adresse for sure. While running the code we will see that the last DWORD is loaded in little endian format again into EAX register then a value is moved to EBX which is the value that we will use for Xoring. But just after this we have a MOV instruction which will move EBX to the current DWORD in the memory location pointed by EBP , while stopping in that instruction you will see

that EBP is holding the value 00000001 so an exception should be triggered as it's impossible to move EBX to that location . If you put a bp on the pushed EBX in the stack you will see that the execution flow will be taken by the instructions at 00CC1960 (pushed EBX as an arg to create the VEH) . Those routines will handle this exception and return also a value to EAX register which will be added as happened in the previous part of checking the flag.
So we will need to figure out what is that added value again , all we need to do is to change the value of EAX register after the LODSD instruction to 00000000 then put a breakpoint on 00CC110D and press « F9 » so we don't skip that instruction as happened last time. Now all we have to do is look at what EAX is holding : it's holding D9150F32 . So after the handling the exception this value (D9150F32) will be added to EAX register , now we need to figure out what should be the right value of EAX just after handling the exception means :
(D9150F32+ LastFlagDwordLittleEndian)
You will just have to XOR 8E7632F3 with EBX , and you will have this value : FA3654A0 .
So the right last DWORD of the flag in little endian should be :
FA3654A0 – D9150F32 =2121456E –> Big Endian = 6E452121 –> ASCII =nE!!
So the last 4 characters of the flag are : nE!! …

5 – Regrouping the 3 parts :

So the complete flag to validate the challenge is : W3lL__D0nE!!
Now just try to provide the flag to the Crackme and you will see that :

Finally , this was a really GOOD crackme that I actually enjoyed discovering and cracking because it uses many handlers to deal with exceptions then return some values that will be added and also uses a very interesting method to check for the length .

See you soon guys …
Regards,

Souhail Hammou (Dark-Puzzle) from www.itsecurity.ma

Follow me : http://www.facebook.com/dark.puzzle.sec
: http://twitter.com/dark_puzzle
Download The CrackMe : http://www.mediafire.com/?5661n2z4w44664f