

BinaryNewbie Small Keygenme

<https://crackmes.one/crackme/5e83f7f433c5d4439bb2e059>

Crackme writeup by @H0l3Bl4ck <https://twitter.com/H0l3Bl4ck>

crackmes.one user **b1h0** <https://crackmes.one/user/b1h0>

Date: 22/abr/2020

You can download **Small Keygenme** from this [link](#).

The author informs us of this within a **README**:

```
Hello buddy !!!!  
An easy keygenme challenge.  
Read the [SPECs]  
Have fun.  
Binary Newbie.  
  
[SPECs]  
- Assembled with nasm;  
- Stripped;  
- 64-bits elf file, tested in ubuntu 18.04 LTS;  
- Any kind of patch is forbidden;  
- Obfuscation: 0;  
- anti-debug/anti-disassembly tricks: 0;  
- Goals - Understand how it works, write a keygen, and finally, write a solution;  
- Target - Beginners.  
  
PS: If you have any doubt or any question about the challenge, feel free to call me in Dis
```



Ghidra's static analysis

We have here an executable in **ELF** format compiled for **64 bits**.

About little-crackme	
i	Project File Name: little-crackme
	Last Modified: Mon Apr 20 14:32:46 CEST 2020
	Readonly: false
	Program Name: little-crackme
	Language ID: x86:LE:64:default (2.9)
	Compiler ID: gcc
	Processor: x86
	Endian: Little
	Address Size: 64
	Minimum Address: 00400000
	Maximum Address: _elfSectionHeaders::0000013f
	# of Bytes: 1096
	# of Memory Blocks: 6
	# of Instructions: 143
	# of Defined Data: 12
	# of Functions: 7
	# of Symbols: 7
	# of Data Types: 16
	# of Data Type Categories: 2
	Analyzed: true
	Created With Ghidra Version: 9.1.1
	Date Created: Mon Apr 20 14:32:20 CEST 2020
	ELF File Type: executable
	ELF Original Image Base: 0x400000
	ELF Prelinked: false
	Executable Format: Executable and Linking Format (ELF)

We see that it uses the **SYSCALL** and to identify the functions we can start from [this website](#): <https://filippo.io/linux-syscall-table/> that offers us the reference for **64 bits**.

%rax	Name	Entry point	Implementation
0	read	sys_read	fs/read_write.c
%rdi		%rsi	%rdx
unsigned int fd		char __user * buf	size_t count
1	write	sys_write	fs/read_write.c
%rdi		%rsi	%rdx
unsigned int fd		const char __user * buf	size_t count

Analyzing the entry point, we identify the first function that prints the texts on the screen and rename it to clarify the code.

```
*****  
*          FUNCTION          *  
*****  
undefined[16] __stdcall FUN_PRINT_STRING(void)  
undefined[16]    RDX:8, RAX:8  <RETURN>  
    FUN_PRINT_STRING  
XREF[5]:   entry:004001d0(c), entry:004001dc(c),  
              entry:0040023c(c), entry:0040024f(c)  
  
004000c3 48 83 c4 08      ADD    RSP,0x8  
004000c7 b8 01 00 00 00    MOV    EAX,0x1  
004000cc 48 89 c7      MOV    RDI,RAX  
004000cf 5e              POP    RSI  
004000d0 5a              POP    RDX  
004000d1 0f 05      SYSCALL  
004000d3 48 83 ec 18    SUB    RSP,0x18  
004000d7 c3              RET
```

Next, we identify the function expected to enter the serial number.

```

***** FUNCTION *****
undefined[16] __stdcall FUN_GET_SERIAL(void)
undefined[16]           RDX:8,RAX:8 <RETURN>
FUN_GET_SERIAL                                     XREF[1]: entry:004001f4(c)
004000d8 48 83 c4 08    ADD    RSP,0x8
004000dc 48 29 c0    SUB    RAX,RAX
004000df 48 89 c7    MOV    RDI,RAX
004000e2 5e            POP    RSI
004000e3 5a            POP    RDX
004000e4 0f 05          SYSCALL
004000e6 48 83 e8 01    SUB    RAX,0x1
004000ea 80 3c 06 0a    CMP    byte ptr [RSI + RAX*0x1],0xa
004000ee 74 02          JZ     LAB_004000f2
004000f0 eb 07          JMP    LAB_004000f9

LAB_004000f2                               XREF[1]: 004000ee(j)
004000f2 48 c7 c0 ff ff ff ff    MOV    RAX,-0x1

LAB_004000f9                               XREF[1]: 004000f0(j)
004000f9 48 83 ec 18    SUB    RSP,0x18
004000fd c3            RET

```

We continue to identify small pieces of code and functions and give them names. In this case, the exit to the system.

```

***** FUNCTION *****
undefined __stdcall FUN_SYS_EXIT(void)
undefined           AL:1 <RETURN>
FUN_SYS_EXIT                                     XREF[2]: entry:00400243(c), entry:00400256(c)
004001b8 48 83 c4 08    ADD    RSP,0x8
004001bc b8 3c 00 00 00    MOV    EAX,0x3c
004001c1 5f            POP    RDI
004001c2 0f 05          SYSCALL

```

So, after having renamed some functions and commenting on some operations, we have the **entry point** (which is the **main function** of the program) as follows, in the absence of finishing analyzing some functions.

```

entry                                         XREF[2]: Entry Point(*), 00400018(*)
004001c4 e8 e7 fe ff ff    CALL   FUN_INIT_REGISTERS
004001c9 6a 26            PUSH   0x26
004001cb 68 5c 02 60 00    PUSH   s_Welcome_to_this_little_challenge_0060025c
004001d0 e8 ee fe ff ff    CALL   FUN_PRINT_STRING
004001d5 6a 20            PUSH   0x20
004001d7 68 82 02 60 00    PUSH   s_Developed_by_Binary_Newbie_!!!_00600282
004001dc e8 e2 fe ff ff    CALL   FUN_PRINT_STRING
004001e1 6a 11            PUSH   0x11
004001e3 68 a2 02 60 00    PUSH   s_Enter_a_serial:_006002a2
004001e8 e8 d6 fe ff ff    CALL   FUN_PRINT_STRING
004001ed 6a 10            PUSH   0x10
004001ef 68 d0 02 60 00    PUSH   DAT_SERIAL_INPUT
004001f4 e8 d1 fe ff ff    CALL   FUN_GET_SERIAL
004001f9 84 c0            TEST  AL,AL
004001fb 78 38            JS    LAB_NOT_VALID
004001fd 66 29 ff          SUB   DI,DI
00400200 66 83 f8 0f          CMP   AX,0xf
00400204 66 0f 45 c7          CMOVNZ AX,DI
00400208 84 c0            TEST  AL,AL
0040020a 74 29            JZ    LAB_NOT_VALID
0040020c 50            PUSH   RAX
0040020d 68 d0 02 60 00    PUSH   DAT_SERIAL_INPUT
00400212 68 e0 02 60 00    PUSH   DAT_SERIAL_TRANSFORMED
00400217 e8 e2 fe ff ff    CALL   FUN_TRANSFORM
0040021c a8 01            TEST  AL,0x1
0040021e 75 15            JNZ   LAB_NOT_VALID
00400220 52            PUSH   RDX
00400221 68 e0 02 60 00    PUSH   DAT_SERIAL_TRANSFORMED
00400226 e8 2e ff ff ff    CALL   _DEAD_BABE_
0040022b a8 01            TEST  AL,0x1
0040022d 75 06            JNZ   LAB_NOT_VALID
0040022f 68 48 02 40 00    PUSH   0x400248
00400234 c3            RET

```

I will now focus on 2 functions that are what have to give us the solution.

The first one I will call **FUN_TRANSFORM**, which without knowing exactly what it does yet, seems to carry out some kind of calculation or transformation of the string.

I will call the second function **DEAD_BABE** for a specific operation it does.

And finally, note a curious thing, and that is that the code has a jump to the **LAB_NOT_VALID** tag that shows us the message that the serial is not valid, but there is **no jump** to the message that indicates that the serial is correct .

To hide the jump to the "Valid !!!" message, note that after the conditional jump there is a **PUSH** in the stack of the value **0x400248**. This is the address where the code to display this message starts and when it finds the **RET** afterwards it forces a salo to this address.

```

00400212 68 e0 02 60 00    PUSH DAT_SERIAL_TRANSFORMED
00400217 e8 e2 fe ff ff    CALL FUN_TRANSFORM
0040021c a8 01             TEST AL,0x1
0040021e 75 15             JNZ LAB_NOT_VALID
00400220 52               PUSH RDX
00400221 68 e0 02 60 00    PUSH DAT_SERIAL_TRANSFORMED
00400226 e8 2c ff ff ff    CALL _DEAD_BABE_
0040022b a8 01             TEST AL,0x1
0040022d 75 06             JNZ LAB_NOT_VALID
0040022f 68 48 02 40 00    PUSH 0x400248
00400234 c3               RET

LAB_NOT_VALID
00400235 6a 0f             PUSH 0xf
00400237 68 b3 02 60 00    PUSH s_Not_valid_!!!_006002b3
0040023c e8 82 fe ff ff    CALL FUN_PRINT_STRING
00400241 6a 01             PUSH 0x1
00400243 e8 70 ff ff ff    CALL FUN_SYS_EXIT
00400248 6a 0b             PUSH 0xb
0040024a 68 c2 02 60 00    PUSH s_Valid_!!!_006002c2
0040024f e8 6f fe ff ff    CALL FUN_PRINT_STRING
00400254 6a 00             PUSH 0x0
00400256 e8 5d ff ff ff    CALL FUN_SYS_EXIT
.....
```

After analyzing the **FUN_TRANSFORM** function, we see that what it actually does is check that all the digits entered in the serial are valid values for a hexadecimal number, and it stores its value in decimal (one per byte) in a new array.

So we will change the name of the function and call it **FUN_HEXA_CONVERT**.

I leave here the complete function in two screenshots, and with your comments, which I think are more than enough to understand what it does.

Listing: little-crackme

```

***** FUNCTION *****


```

Listing: little-crackme

```

LAB_STORE_CHAR XREF[2]: 0040012f(j), 0040013d(j)
0040011b 88 04 0f MOV byte ptr [RDI + RCX*0x1],AL
                                         Store decimal value in byte
0040011e 80 c1 01 ADD CL,0x1
                                         inc char counter
00400121 eb e5 JMP LAB_NEXT_CHAR

LAB_CHAR_GREATER_NUMBER XREF[1]: 00400117(j)
00400123 3c 41 CMP AL,0x41
                                         AL == "A"
00400125 72 18 JC LAB_END_LOOP
00400127 3c 46 CMP AL,0x46
                                         AL == "F"
00400129 77 06 JA LAB_CHAR_LOWERCASE
0040012b 2c 41 SUB AL,0x41
                                         get the true numerical value
0040012d 04 0a ADD AL,0xa
                                         adding the decimal value 10
0040012f eb ea JMP LAB_STORE_CHAR
                                         loop to next char

LAB_CHAR_LOWERCASE XREF[1]: 00400117(j)
00400131 3c 61 CMP AL,0x61
                                         AL == "a"
00400133 72 0a JC LAB_END_LOOP
00400135 3c 66 CMP AL,0x66
                                         AL == "f"
00400137 77 06 JA LAB_END_LOOP
00400139 2c 61 SUB AL,0x61
                                         get the true numerical value
0040013b 04 0a ADD AL,0xa
                                         adding the decimal value 10
0040013d eb dc JMP LAB_STORE_CHAR
                                         loop to next char

LAB_END_LOOP XREF[5]: 0040010a(j), 00400113(j), 00400125(j), 00400133(j), 00400137(j)
0040013f 48 83 ec 20 SUB RSP,0x20
00400143 80 e9 01 SUB CL,0x1
00400146 66 41 b8 01 00 MOV R8W,0x1
                                         move 0x1 t0 lower 16 bits of R8
0040014b 48 31 c0 XOR RAX,RAX
0040014e 66 39 d1 CMP CX,DX
00400151 66 41 0f 45 c0 CMOVNZ AX,R8W
                                         if CX == DX then AX = R8W
00400156 c3 RET

```

We will now analyze the function that we have named as **DEAD_BABE** which will give us the solution to crackme.

We have here a function that goes through all the values and performs some operations to check certain values. Depending on whether the value is **odd** or **even**, it will do different operations. If all the values pass the validation, the serial number will be good.

Knowing that the valid digits are hexadecimal values we have that we can only test the values **0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F**.

```

***** FUNCTION *****
undefined _DEAD_BABE_()
AL:1 <RETURN>
DEAD_BABE_ XREF[1]: entry:00400226(c)
00400157 48 83 c4 08 ADD RSP,0x8
0040015b 5e POP RSI
0040015c 5a POP RDX
0040015d 48 31 db XOR RBX,RBX

LAB_LOOP_SERIAL_VALUES XREF[1]: 0040018a(j)
00400160 38 d3 CMP BL,DL Counter
00400162 77 3c JA LAB_END_LOOP_D_B
00400164 66 0f b6 04 1e MOVZX AX,byte ptr [RSI + RBX*0x1] get current byte
00400169 a8 01 TEST AL,0x1 check lowest bit
0040016b 75 1f JNZ LAB_VALUE_IS_ODD jump if ODD
0040016d 41 88 c1 MOV R9B,AL store value
00400170 35 ad de 00 00 XOR EAX,0xdead 2 is unique valid value
00400175 05 be ba 00 00 ADD EAX,0xbabe
0040017a c1 e8 04 SHR EAX,0x4
0040017d 44 01 c8 ADD EAX,R9D
00400180 3d 98 19 00 00 CMP EAX,0x1998
00400185 75 19 JNZ LAB_END_LOOP_D_B

LAB_NEXT_VALUE XREF[1]: 0040019e(j)
00400187 80 c3 01 ADD BL,0x1
0040018a eb d4 JMP LAB_LOOP_SERIAL_VALUES

LAB_VALUE_IS_ODD XREF[1]: 0040016b(j)
0040018c 83 f0 1a XOR EAX,0x1a 5,7,d,f are valid values
0040018f 83 c8 0a OR EAX,0xa
00400192 35 87 19 00 00 XOR EAX,0x1987
00400197 3d 98 19 00 00 CMP EAX,0x1998

```

As the calculation is different for odd and even we separate them and do the manual test with the operations. Let's see an example with the number 2 that is even:

We see that the original value is saved in the **R9B** register, subsequently an **XOR** is made with the value **0xdead**, the result is added the value **0xbabe**, they are moved 4 bits to the right and the previously stored value is added. If the result of the operation is **0x1998** then the value is valid.

0040016d 41 88 c1	MOV R9B,AL	; R9B = 2
00400170 35 ad de 00 00	XOR EAX,0xdead	; 2 XOR 0xdead = 0xdeaf
00400175 05 be ba 00 00	ADD EAX,0xbabe	; EAX = EAX + 0xbabe = 0x1
0040017a c1 e8 04	SHR EAX,0x4	; 0x1996D shr 4 = 0x1996
0040017d 44 01 c8	ADD EAX,R9D	; EAX = 0x1996 + 2 = 0x199
00400180 3d 98 19 00 00	CMP EAX,0x1998	
00400185 75 19	JNZ LAB_END_LOOP_D_B	

As we can see **2 meets the condition**. If we do the same operations with the rest of the even numbers, none of them comply, therefore, in this section we can determine that the only valid number is 2.

We are going to do the same with the odd numbers.

```

    Listing: little-crackme
    little-crackme x

    LAB_NEXT_VALUE
    00400187 80 c3 01      ADD    BL,0x1
    0040018a eb d4        JMP    LAB_LOOP_SERIAL_VALUES

    LAB_VALUE_IS_ODD
    0040018c 83 f0 1a      XOR    EAX,0x1a
    0040018f 83 c8 0a      OR     EAX,0xa
    00400192 35 87 19 00 00 XOR    EAX,0x1987
    00400197 3d 98 19 00 00 CMP    EAX,0x1998
    0040019c 75 02        JNZ    LAB_END_LOOP_D_B
    0040019e eb e7        JMP    LAB_NEXT_VALUE

    LAB_END_LOOP_D_B
    004001a0 48 83 ec 18    SUB    RSP,0x18
    004001a4 80 eb 01      SUB    BL,0x1
    004001a7 66 41 b8 01 00 MOV    R8W,0x1
    004001ac 48 31 c0      XOR    RAX,RAX
    004001af 66 39 d3      CMP    BX,DX
    004001b2 66 41 0f 45 c0 CMOVNZ AX,R8W
    004001b7 c3            RET

    LAB_VALUE_IS_ODD
    0040018c 83 f0 1a      XOR    EAX,0x1a
    0040018f 83 c8 0a      OR     EAX,0xa
    00400192 35 87 19 00 00 XOR    EAX,0x1987
    00400197 3d 98 19 00 00 CMP    EAX,0x1998
    0040019c 75 02        JNZ    LAB_END_LOOP_D_B
    0040019e eb e7        JMP    LAB_NEXT_VALUE
  
```

LAB_VALUE_IS_ODD
 0040018c 83 f0 1a XOR EAX,0x1a ; EAX = 5 XOR 0x1a = 0x1f
 0040018f 83 c8 0a OR EAX,0xa ; 0x1f OR 0xa = 0x1f
 00400192 35 87 19 00 00 XOR EAX,0x1987 ; 0x1f XOR 0x1987 = 0x1998
 00400197 3d 98 19 00 00 CMP EAX,0x1998
 0040019c 75 02 JNZ LAB_END_LOOP_D_B
 0040019e eb e7 JMP LAB_NEXT_VALUE

Here the operations are different. An **XOR** of the value is made with **0x1a**, followed by an **OR** with **0xa** and again an **XOR** with **0x1998**. If the result is **0x1998** then that value (digit) is valid. As there are not many, you just have to do the operations with this and we obtain that the valid values are: **5,7,d, and f**.

Thus we can conclude that any **16 digit** serial number containing the characters **2, 5, 7, d or f** will be valid.

Let's check it with 3 different random combinations to see what happens ...

- 252575d2f777f55d

```

Parrot Terminal
Archivo Editar Ver Buscar Terminal Ayuda
[b1h0@parrot]~[~/mnt/programacionvmrev/CrackMe/crackmes.one/BinaryNewbie-Small_Keygenme]
└─$ ./little-crackme
Welcome to this little challenge !!!
Developed by Binary Newbie !!!
Enter a serial: 252575d2f777f55d
Valid !!!
  
```

- 7df522ff575dd2df

```

Parrot Terminal
Archivo Editar Ver Buscar Terminal Ayuda
[b1h0@parrot]~[~/mnt/programacionvmrev/CrackMe/crackmes.one/BinaryNewbie-Small_Keygenme]
└─$ ./little-crackme
Welcome to this little challenge !!!
Developed by Binary Newbie !!!
Enter a serial: 7df522ff575dd2df
Valid !!!
  
```

- d5f7f52ff55dd275

```
[b1h0@parrot]~[/mnt/programacionvmrev/CrackMe/crackmes.one/BinaryNewbie-Small_Keygenme]
└─$ ./little-crackme
Welcome to this little challenge !!!
Developed by Binary Newbie !!!
Enter a serial: d5f7f52ff55dd275
Valid !!!
```

Serial generator

We are going to create a simple serial number generator. Each time we invoke it it will give us a random number that matches the specifications.

Keep in mind that the letters can be uppercase or lowercase, and have the same value, but we can mix them to make it appear that the serial number is different, although in reality it is the same.

[Download source code](#)

```
/*
 * little-crackme-keygen.c
 * Serial generator for BinaryNewbie's Small Keygenme
 * https://crackmes.one/crackme/5e83f7f433c5d4439bb2e059
 * Author: Gabriel Martí
 * Twitter: @H0l3Bl4ck
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

char *serialgen(unsigned int length, char *charset) {
    char *serial_string = NULL;
    int sz = strlen(charset);
    if (length) {
        srand(time(NULL));
        serial_string = malloc(sizeof(char) * (length+1));
        if (serial_string) {
            for (int n = 0; n < length; n++) {
                int key = rand() % sz;
                serial_string[n] = charset[key];
            }
            serial_string[length] = '\0';
        }
    }
    return serial_string;
}

int main(int argc, char **argv) {
    char *valid_digits = "257dfDF";
```

```
    printf("Serial generator for BinaryNewbie's Small Keygenme\n");
    printf("Serial: %s\n", serialgen(16, valid_digits));
    return 0;
}
```

We're going to try it

The screenshot shows a terminal window titled "little-crackme-keygen.c". The code is displayed on the left, and the output is on the right. The output shows three different serial numbers generated by the program.

```
little-crackme-keygen.c x
1 /*
2  * little-crackme-keygen.c
3  * Serial generator for BinaryNewbie's Small Keygenme
4  * https://www.binarynewbie.com/
5  * Author: Gabriel Marti
6  * Twitter: @H0l3Bl4ck
7  */
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <string.h>
11 #include <time.h>
12
13 char *serialgen(unsigned int length) {
14     char *serial_string = NULL;
15     int sz = strlen(charset);
16     if (length) {
17         srand(time(NULL));
18         serial_string = malloc(sizeof(char) * length);
19         if (serial_string) {
20             for (int n = 0; n < length; n++) {
21                 int key = rand() % sz;
22                 serial_string[n] = charset[key];
23             }
24             serial_string[length] = '\0';
25         }
26     }
27     return serial_string;
28 }
```

```
C:\DATA\WorkFiles\Workspace1\Little-crackme-keygen\Release>Little-crackme-keygen.exe
Serial generator for BinaryNewbie's Small Keygenme
Serial: 7DDd5FF5DF57d5ff

C:\DATA\WorkFiles\Workspace1\Little-crackme-keygen\Release>Little-crackme-keygen.exe
Serial generator for BinaryNewbie's Small Keygenme
Serial: 5fff2D5F572FDff5

C:\DATA\WorkFiles\Workspace1\Little-crackme-keygen\Release>Little-crackme-keygen.exe
Serial generator for BinaryNewbie's Small Keygenme
Serial: D5d2fDF22F57755
```

You can [download the keygen Windows binary from here](#)

That's all folks!