keygen-me-1

by Nine Inch Nulls

Tags: reversing

Rating: 0

```
—–> ./activate
Usage: ./activate <PRODUCT_KEY>
[andrei@jacky 01:35:47] ~/Documents/pico
—–> ./activate AAAA
Please Provide a VALID 16 byte Product Key.
[andrei@jacky 01:35:51] ~/Documents/pico
—–> ./activate AAAABBBBCCCCDDDD
INVALID Product Key.
```

the executable wants a (exactly) 16 character code. inspecting the main we can clearly see we have to pass the validate_key function, then it will call print_flag.

this function has basically three blocks:

1.

```
[0x8048771]                                 |
| (fcn) sym.validate_key 172                |
|   sym.validate_key (char *serial);        |
| ; var int keysum @ ebp-0x14               |
| ; var signed int counter @ ebp-0x10       |
| ; var size_t serial_len @ ebp-0xc         |
| ; var int dunno @ ebp-0x4                 |
| ; arg char *serial @ ebp+0x8              |
| ; CALL XREF from main (0x8048899)         |
| push ebp                                  |
| ebp = esp                                 |
| push ebx                                  |
| esp -= 0x14                               |
| esp -= 0xc                                |
| ; const char *s                           |
| push dword [serial]                       |
| ; size_t strlen(const char *s)            |
| sym.imp.strlen ();[ga]                    |
| esp += 0x10                               |
| dword [serial_len] = eax                  |
| dword [keysum] = 0                        |
| dword [counter] = 0                         |
| goto 0x80487c9;[gb]                         |
```

function initialization; we can start by identifying the variables and renaming them (already done here) to understand the logic more easily. Since the name of the function is validate_key, we can assume the key we provide (here, serial) is passed to the function as an argument (`sym.validate_key (char *serial)` and `arg char *serial @ ebp+0x8`). Also, it is pushed on the stack right before calling strlen, and after that the valur returned in eax is stored in a local variable (`dword [serial_len] = eax`).

Moving on we have a while() block:

2.a - while head)

```
| ; CODE XREF from sym.validate_key (0x8048797) |
| eax = dword [serial_len]                      |
| eax -= 1                                      |
| var = eax - dword [counter]                   |
| if (var > 0) goto 0x8048799;[ge]              |
```

2.b - while body)

```
; CODE XREF from sym.validate_key (0x80487d2)
edx = dword [counter]
; [0x8:4]=-1
eax = dword [serial]
eax += edx
eax = byte [eax]
eax = al
esp -= 0xc
push eax
sym.ord ();[gd]
esp += 0x10
eax = al
edx = [eax + 1]
eax = dword [counter]
eax += 1
eax = eax * edx
dword [keysum] += eax
dword [counter] += 1
```

at the end of the while body we can see a variable is incremented before returning to the while head (`dword [counter] += 1`) (you can't see the arrows indicated the program flow here but trust me lol) so i assumed it's a counter.

this is the first part of our key validation. it takes every character of our serial, calls ord() on it, then sums 1 to this value, gets the counter, sums 1 to it, multiplies ((ord(serial[counter]) + 1) * (counter + 1)) and adds this result into another local variable, that i renamed keusym. So basically this is what the second block does:

```
int calculate_keysum(char *str) {

        int keysum = 0;
        int i;

        for (i = 0; i < strlen(str) - 1; i++) {
                keysum += (ord(str[i]) + 1) * (i + 1);
        }
        return keysum;

}
```

with ord() being the same function you find also in python, which returns the int corresponding to its char representation (ex ord('4') = 0x04):

```
int ord(char c) {
       if (c < 48 || c > 57) {
               puts("invalid char\n");
               exit(1);
       }
       return c - 48;
}
```

and here comes the third block:

3.

```
0x80487d4 [gf]                    |
| ecx = dword [keysum]            |
| edx = 0x38e38e39                |
| eax = ecx                       |
| eax = eax * edx                 |
| ebx = edx                       |
| ebx >>>= 3                      |
| eax = ebx                       |
| eax <<<= 3                      |
| eax += ebx                      |
| eax <<<= 2                      |
| ecx -= eax                      |
| ebx = ecx                       |
| eax = dword [serial_len]        |
| edx = [eax - 1]                 |
| ; [0x8:4]=-1                    |
| ; 8                             |
| eax = dword [serial]            |
| eax += edx                      |
| eax = byte [eax]                |
| eax = al                        |
| esp -= 0xc                      |
| push eax                        |
| sym.ord ();[gd]                 |
| esp += 0x10                     |
| eax = al                        |
| var = ebx - eax                 |
| al = e                          |
| ebx = dword [dunno]             |
| leave                           |
| return                          |
```

this is kind of a bit tricky to follow just commenting and renaming variables on r2, so first i reported the steps in C:

```
int validate_key(int key, char *str) {

        int c = 0x38e38e39;
        int len = 16;

        unsigned int ra, rb, rc, rd; // registers

        rc = key;
        rd = c;
        ra = key * c;
        rb = ((long long)key * c) >> 32; // get high portion of mul into ebx
        rb = rb >> 3;
        ra = rb;
        ra = ra << 3;
        ra = ra + rb;
        ra = ra << 2;
        rc = rc - ra;
        rb = rc;
        ra = len;
        rd = ra - 1;
        ra = str[rd];
        ra = ord(ra);

        printf("ebx: 0x%08x\neax: 0x%08x\n", rb, ra);
        printf("result: 0x%08x\n", rb - ra);
        return rb - ra;
        // must return val != 0

}
```

the `mul` operation is a bit tricky because, first of all it has the destination operand implicit being eax, and second, if the result is greater than 32 bits, it memorizes the high portion into edx. this was not evident from the asm code. also notice the (long long) casting: in C we have to cast one of the operands (or declare them as long long) otherwise it would truncate the result to 32 bits, and the next bit shifting by 32 would return zero.

at this point we can simplify this code into something like this, knowing that a right / left bit shifting by n is nothing but a division / product by (2 times n):

```
int validate_key_simpl(int key, char *str) {

        int c = 0x38e38e39;
        int ra, rb;

        rb = ((long long)key * c) / pow(2, 35);
        rb = key - (rb * 36);
        ra = ord(str[15]);

        printf("ebx: 0x%08x\neax: 0x%08x\n", rb, ra);
        printf("result: 0x%08x\n", rb - ra);
        return rb - ra;

}
```

i don't really understand what these calculation do, i tried to come up with a formula or something, but i got nothing :(

however we can see that it has many solutions by executing this C code searching for a 0 as the return value.

```
—-> ./solver 6666999966669990
key: 0x00000414
ebx: 0x00000000
eax: 0x00000000
```

this is the one i got the flag with

```
xnand@pico-2018-shell-3:/problems/keygen-me-1_1_8eb35cc7858ff1d2f55d30e5428f30a7$ ./
activate 6666999966669990
Product Activated Successfully: picoCTF{k3yg3n5_4r3_s0_s1mp13_3718231394}
```

Original writeup.

Comments

XC33 – sabato 10 novembre 2018 03:51:56

Hi there I love your explanation so much but I have doubts in this part

rb = key - (rb * 36);
ra = ord(str[15]);

May I know how did you derive that "36" and "str[15]"? I assume the str[15] is from your input "6666999966669990" which is 16 digits minus 1. Much help is appreciated, thank you.