Crackmes: simpledata's simpledatas_keygenme_1

Website: https://crackmes.one/

Author of the crackme: simpledata

Language: C/C++

Level: 1

Index

Introduction

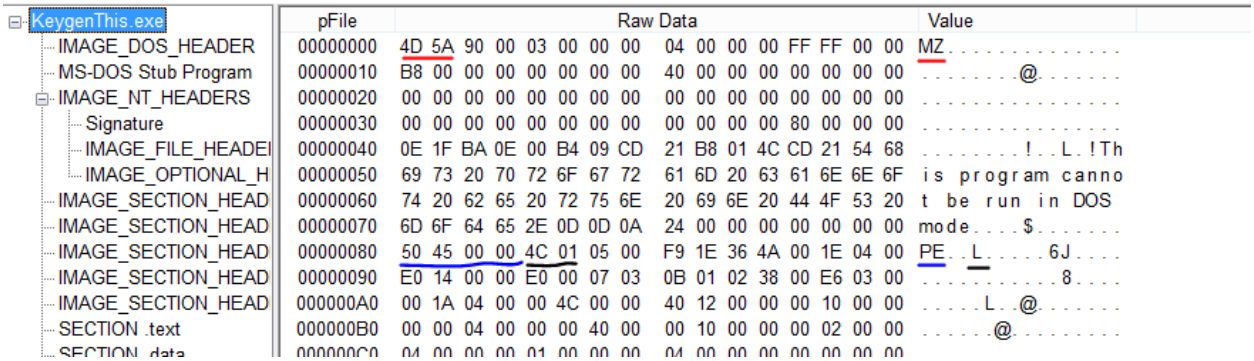
This is an easy challenge. We will see some data management and a code structure. No advanced skills are required. The purpose of this walkthrough is to show the procedure followed on analyzing this binary and reversing it to, finally, build a keygen. It is out of scope to teach how to use *IDA*. If you do not know how to follow up on *IDA* at some point, head to your browser and search for detailed information on getting how to do what you need.

Tools

This crackme is intended to be run on windows, so we will analyze it on a windows VM. In my particular case, it is a 8.1 windows. The tools used in this walkthrough are going to be *PEview*, *IDA* and *x64dbg*.

Walkthrough

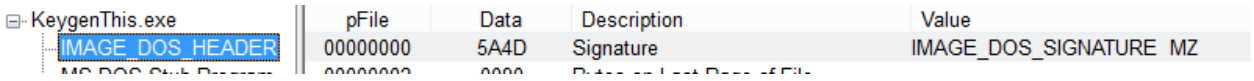
First of all, lets see what kind of program are we facing. We can open it on *PEview* to get that task done.



PEview opens shows as its hex output and parses its sections in order to serve us its information. At the very beginning, we can see its “magic number”, 5A4D (little endian), MZ. As stated on Wikipedia:

> *The DOS MZ executable format is the executable file format used for .EXE files in DOS.*
>
> *The file can be identified by the ASCII string "MZ" (hexadecimal: 4D 5A) at the beginning of the file (the "magic number"). "MZ" are the initials of Mark Zbikowski, one of leading developers of MS-DOS.*
>
> *(https://en.wikipedia.org/wiki/DOS_MZ_executable)*

As said before, this is parsed by *PEview* (head to the next section in *PEview* as in the following image).

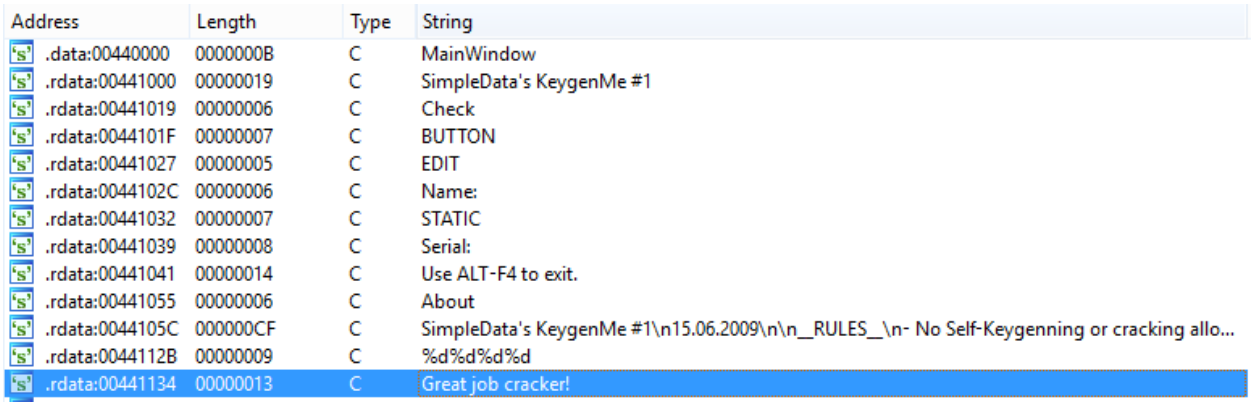


Next, we have its signature, 00004550, which stands for PE (https://en.wikipedia.org/wiki/Portable_Executable).

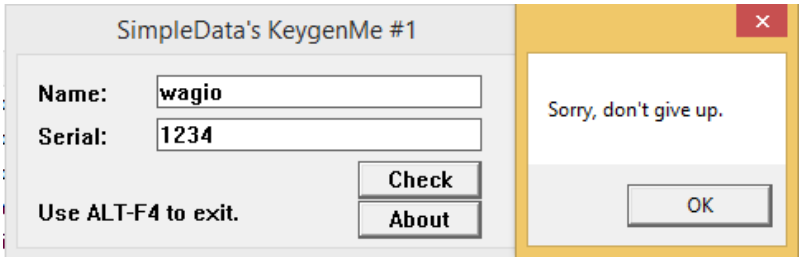
Finally, we should check its *IMAGE_FILE_HEADER*, which tells us its meant to be run on 32 bits architectures.

There is much more information that can be extracted from here, but it is out of the scope of this walkthrough, so this is going to be enough.

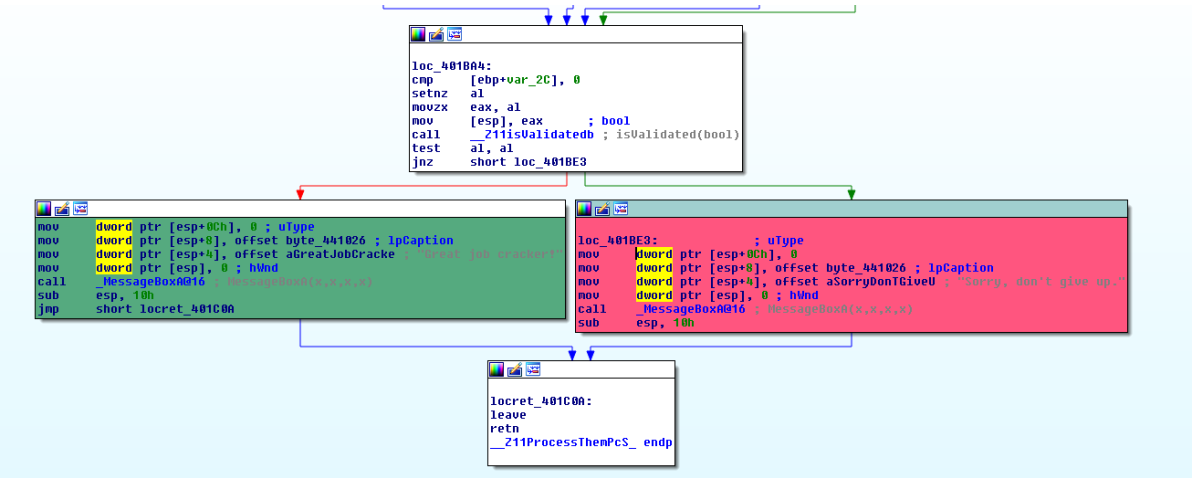
Now lets head to IDA. Starting from *main*, we notice a bunch of things that provides its graphical window and such things. It does not seem to be interesting, so lets see its strings' section. After looking on them, we can highlight the ones which we should track.



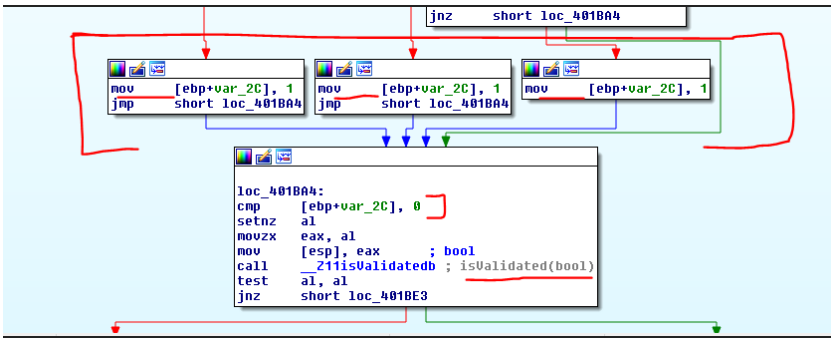
Running the executable confirms us those are the strings we need to look for.



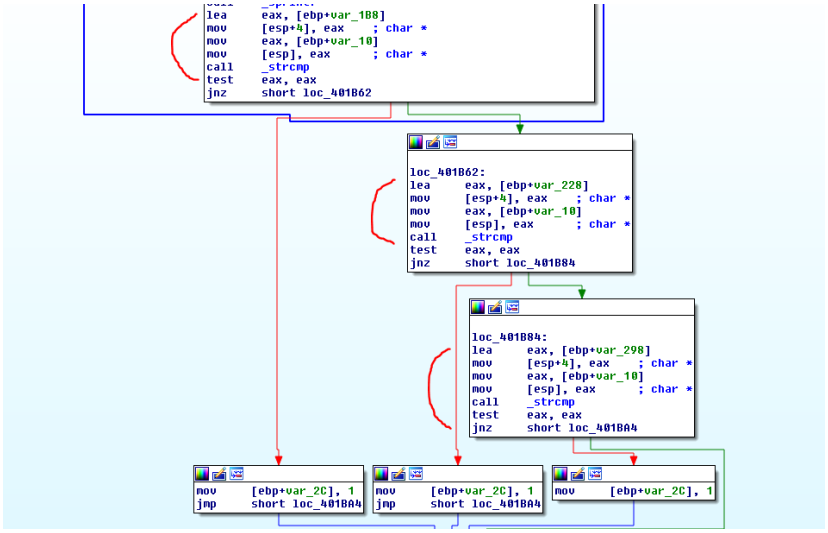
If we head to the point these strings are being used, we see it is doing a validation and deciding whether to congratulate or ask us to try harder. (Green section is the nice one, as opposed to the red one; their colors have been manually set up).



Going up a little bit, it seems to be checking whether our input is validated or not. Moreover, seems that there are three different correct ways.



Going above, we can note it is comparing three different strings with *var10*.



If we were to traduce this snippet back to source code, I would say it is an *if* block, something like:

If (var_10==var_1B8 || var_10==var_228 ||var_10==var_298)

{

Var_2C = True;

}

If (isValidated(Var_2C))

{/ Good block (green one)*/}*

*Else {/*Bad block (red one)*/}*

If we are not mistaken, all we need know is what are those strings storing. We are going to focus on each one at a time. Lets start with the one stored at *var1B8*.

First solution

If we check references to *var_1B8*, we can see one above, at address 0x00401AAE.

```
mov     eax, [ebp+var_C]
mov     [esp], eax      ; char *
call    __Z15CharArrayLengthPc ; CharArrayLength(char *)
mov     edx, eax
mov     eax, [ebp+var_13C]
mov     [esp+14h], eax
mov     eax, [ebp+var_140]
mov     [esp+10h], eax
mov     dword ptr [esp+0Ch], 6Eh
mov     [esp+8], edx
mov     dword ptr [esp+4], offset aDDDD ; "%d%d%d%d"
lea     eax, [ebp+var_1B8]
mov     [esp], eax      ; char *
call    _sprintf
```

It is not being initialized, but just being passed as a parameter into a function. In fact, we see this function (sprint) is taking five arguments. In order to get some light here, lets search for documentation about this function. As stated here, this function is storing its parameters onto the first one after formatting them as stated by its second parameter. Following the calling convention used here, we know it is performing the following call (the last argument pushed is the first parameter of the function):

Sprintf(var_1B8, "%d%d%d%d", edx, 0x6E, var_140, var_13C)

Given that, we must cut it into pieces until we know which is the value of each parameter.

Starting from the beginning, *edx* is storing the return value from *CharArrayLength*. The only parameter this function is taking is *var_C*. Above, at the beginning (address 0x401A12), we note it is storing arg_0. What is arg_0? We can check refs to *ProcessThem* and see which is passed on this parameter. There is a single call to *ProcessThem*.

```
loc_4018E3:
lea     eax, [ebp+lParam]
mov     [esp+0Ch], eax  ; lParam
mov     dword ptr [esp+8], 55h ; wParam
mov     dword ptr [esp+4], 0Dh ; Msg
mov     eax, ds:_hUsernameEdit
mov     [esp], eax      ; hWnd
call    _SendMessageA@16 ; SendMessageA(x,x,x,x)
sub     esp, 10h
lea     eax, [ebp+var_E8]
mov     [esp+0Ch], eax  ; lParam
mov     dword ptr [esp+8], 55h ; wParam
mov     dword ptr [esp+4], 0Dh ; Msg
mov     eax, ds:_hSerialEdit
mov     [esp], eax      ; hWnd
call    _SendMessageA@16 ; SendMessageA(x,x,x,x)
sub     esp, 10h
lea     eax, [ebp+var_E8]
mov     [esp+4], eax    ; char *
lea     eax, [ebp+lParam]
mov     [esp], eax      ; char *
call    __Z11ProcessThemPcS_ ; ProcessThem(char *,char *)
mov     [ebp+var_F4], 0
jmp     loc_4019FC
```

Note the parameters we are receiving have also been used on two different calls preceding our function. These calls are for SendMessageA. We can suppose *lParam* is receiving our name and *var_E8* is storing the serial. We could confirm that by debugging the program. Hence, we know our first parameter is the name we have provided to the program at runtime and the second one is the serial.

Once we know that, we back to our function and rename *var_C* to *NAME*, since it is not changed anymore.

Now recalling the call to *CharArrayLength*, we can confirm *edx* equals NAME.length().

Sprintf(var_1B8, "%d%d%d%d", NAME.length, 0x6E, var_140, var_13C)

The second parameter is 0x6E. Since this is hardcoded, we do not need any further analysis, lets just update our call. The third one is *var_140*. As we did before, we search for refs to *var_140* on the code and we see it is once initialized (at 0x00401A40) and never changed.

```
mov     eax, [ebp+NAME]
movsx   eax, byte ptr [eax]
mov     [ebp+var_140], eax
mov     [ebp+var_29C], 0
```

Var_140 is storing something from our NAME, but what exactly? The key is on the instruction *movsx*. Searching over *Intel SW developer's manual*, we read the following information.

MOVSX/MOVSXD—Move with Sign-Extension

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F BE /r	MOVSX *r16, r/m8*	RM	Valid	Valid	Move byte to word with sign-extension.
0F BE /r	MOVSX *r32, r/m8*	RM	Valid	Valid	Move byte to doubleword with sign-extension.
REX.W + 0F BE /r	MOVSX *r64, r/m8*	RM	Valid	N.E.	Move byte to quadword with sign-extension.

As *eax* is a 32 bit register, we should read the second row of the given table. It is moving a single byte with sign-extension. In this case, as it is an array of characters, this is equal to just *move a byte to doubleword adding zeros*. So, if *eax* is '*wagio*', *var_140* is '*w*'. Lets rename *var_140* to *NAME_FIRST_CHR* and update the function's call.

Sprintf(var_1B8, "%d%d%d%d", NAME.length, 0x6E, NAME_FIRST_CHR, var_13C)

Last but not least, we need to analyze var_13C. Seeking for refs to this variable, we see it is being modified at the beginning and at address 0x00401A6D. At the beginning, it is just being initialized to 0. At the second address, however, it is being changed inside a loop. Note this block is jumping back to loc_401A50.

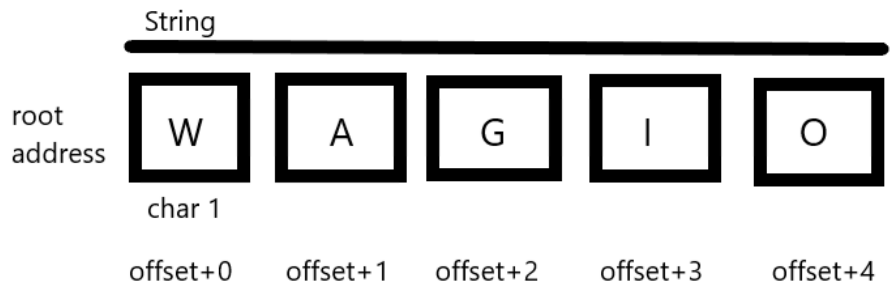
LOOP

```
.text:00401A50 loc_401A50:                             ; CODE XREF: ProcessThem(char *,char *)+71↓j
.text:00401A50                 mov     eax, [ebp+var_29C]
.text:00401A56                 cmp     eax, [ebp+var_30]
.text:00401A59                 jge     short loc_401A79
.text:00401A5B                 mov     eax, [ebp+NAME]
.text:00401A5E                 add     eax, [ebp+var_29C]
.text:00401A64                 movsx   edx, byte ptr [eax]
.text:00401A67                 lea     eax, [ebp+var_13C]
.text:00401A6D                 add     [eax], edx
.text:00401A6F                 lea     eax, [ebp+var_29C]
.text:00401A75                 inc     dword ptr [eax]
.text:00401A77                 jmp     short loc_401A50
.text:00401A79 ; ---------------------------------------------------------------------------
.text:00401A79
.text:00401A79 loc_401A79:                             ; CODE XREF: ProcessThem(char *,char *)+53↑j
```

I have highlighted two different parts of the block. The red one seems to be the condition, while the blue one must be the block inside the condition. As there is an unconditional jump at the end of the block, we will suppose it is a while loop (it could also be a for loop). Hence, the condition would be:

While (var_29C < var_30){ […] }

Lets go towards the first three lines inside the blue block. We have seen *movsx* before. The difference here is that instruction at 0x00401A5E. If that instruction was not there, *edx* would be the first character of NAME. What is going on here is a common ay of accessing elements in a structure. When getting elements from an structure, the standard way to achieve it is to get that element using an offset from the structure's root. Strings are structures, arrays of characters. If we wanted to get a character from a string, say it is NAME, we would use an offset (var_29C) on its start (*ebp + NAME*).

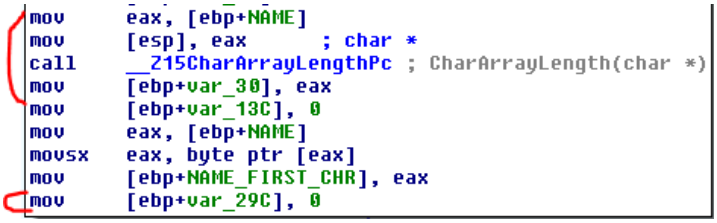


Next, *edx* is added to the value *var_13C* is pointing to. Finally, *var29C* is incremented by one. Lets update our loop:

```
While (var_29C < var_30)
{
        Var_13C += NAME.charAt(var_29C);
        Var_29C ++;
}
```

Recall that *var_13C* was initialized to 0 before entering this loop. Lets get what *var_30* and *var_29C* mean and it is done.

Easy task: *var_30* equals NAME.length and *var_29C* equals 0 (before entering the loop).



Then:

```
Var_29C = 0;
While (var_29C < NAME.length)
{
        Var_13C += NAME.charAt(var_29C);
        Var_29C ++;
}
```

If we want to read it as a for loop:

```
For (int i=0; i<NAME.length; i++)
{
        Var_13C += NAME.charAt(i);
}
```

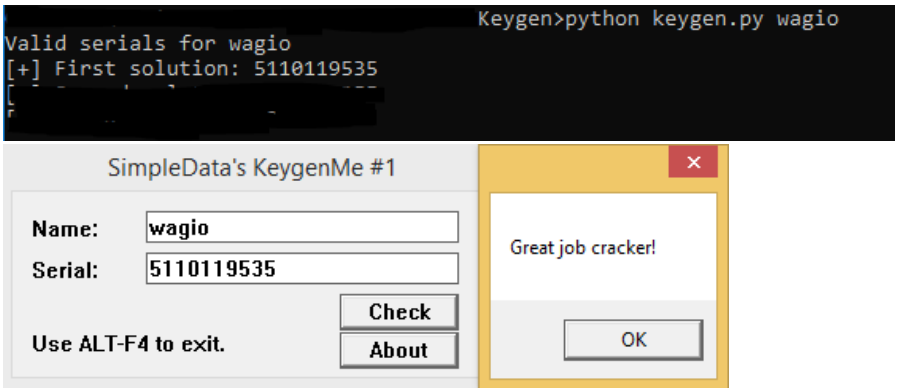
Both loops are valid.

With the information we have got from our analysis we are now able to get the first valid serial. Here is the code snippet in python that would do the job for us:

```
# Loop
var13C = 0
var29C = 0
i = 0
while var29C < len(args.name):
    var13C = var13C + ord(args.name[i:i+1])
    var29C = var29C + 1
    i = i + 1

# First serial = NAME.length + 0x6E + NAME.substr(0,1) + var13C
print('[+] First solution: ' + str(len(args.name)) + str(int(0x6E)) + str(ord(args.name[:1])) + str(var13C))
```

Lets check it is correct.



Nice!

Other solutions

The remaining valid solutions are modified on the order of the parameters and the hardcoded part, so, given that you have understood the first solution, it is just a matter of reordering the values to get the other correct serials.

Keygen

I have made a keygen in python to get the three valid serials for any given name. The keygen is stored at my github page (https://github.com/JavierYuste/write-ups/tree/master/Crackmes_one/Level%201/simpledatas_keygenme_1)

References

- https://en.wikipedia.org/wiki/DOS_MZ_executable
- https://en.wikipedia.org/wiki/Portable_Executable
- http://www.cplusplus.com/reference/cstdio/sprintf/
- https://msdn.microsoft.com/es-es/library/windows/desktop/ms644950(v=vs.85).aspx
- Intel SW Dev's Manual (pdf)
  https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf