Solution to Keygenme 8 by qpt^J

March 6, 2011

Ohi. It’s my first attempt @ crackmes.de

Since I work, It took me few days two finish it, I’ve used mostly IDA Pro.

VM loop starts at 0x4012E9, VM code starts at 0x401749 (we’ll get back to this one), handlers are in an array at 0x401231.

I’ve analyzed almost all handlers and gave them some more sensible names. Here’s small example:

```
.text:004012A5 dd offset subValFromCell_B_D
.text:004012A9 dd offset andCellsByVal_Bd_Ds
.text:004012AD dd offset pushOnStack_Dp
.text:004012B1 dd offset shlCellPtrValByCell_B_B
.text:004012B5 dd offset orCellPtrValByCell_B_B
.text:004012B9 dd offset subValFromCellPtr_Bd_Ds
.text:004012BD dd offset cmpValWithCellPtr_Bd_Ds
```

The machine uses 8 32-bits registers, which I will call CELLS later, stack with 32-bit values, Z-flag register, and some memory.

The '_B' and '_D' suffixes in handler names means type of argument, which can be BYTE or DWORD. If it’s a BYTE in most cases it’s a number of a CELL.

I’ve defined enum with mnemonics based on handler names and some helper structs.

Thanks to this I got all the VM code which you can see in appendix A

I was actully to lazy to write decompiler, mainly cause VM has less than 0x100 instructions. So the stuff in appendix A is mada mainlt by pressing ’Alt+Q’ in IDA (that is ”struct var”).

So basically what the VM does at the beginning is:

- take the input from a dialog box,
calculate MD2 from the username,

convert the password into two DWORDS, these are passDw1 and passDw2 on calls at 0x401808 and 0x40181D

when it has that it makes to calls to subProc at 0x401832 and 0x401854

bad guy/good guy depends on the result from the subproc (conditional jumps at 0x401840 and 0x401862)

So now let’s look at the subProc. As params it takes some 256-byte magicTable (we’ll get back to that) and a dwords from the ”password” field.

I’ve rewritten piece of the VM code mnemonic (from 0x4018EB) to more understandable pseudo-code (at least imo):

```
VM_SETCELLFROMCELLPTR BD BS 1 5
VM_SUBVALFROMCELL B D 5 dd 0xFFFFFFFF8
VM_SETCELLFROMCELLPTR BD BS 0 6
VM_ADDVALTO_CELL B D 6 dd 4
VM_XORCELLBYCELL 3 3
VM_ANDCELLS BD BS 1 0
VM_JMPIFZ D dd offset byte 401925
VM_SETCELLFROMCELL_BS BD BS 2 1
VM_SUBVALFROMCELL B D 2 dd 1
VM_ADDVALTO_CELL B D 3 dd 1
VM_ANDCELLS BD BS 1 2
VM_JMPNOZ_D dd offset byte 40190E

401925:
VM_ADDVALTO_CELL B D 5 dd 0xFFFFFFFF4h
VM_SETCELLFROMCELLPTR BD BS 1 5
VM_SUBVALFROMCELL B D 5 dd 0xFFFFFFFF4h
VM_SETCELLFROMCELLPTR BD BS 0 6
VM_ADDVALTO_CELL B D 6 dd 4
VM_XORCELLBYCELL 7 7
VM_ANDCELLS BD BS 1 0
VM_JMPIFZ D dd offset byte 40195F
401948:
VM_SETCELLFROMCELL_BS BD BS 2 1
VM_SUBVALFROMCELL B D 2 dd 1
VM_ADDVALTO_CELL B D 7 dd 1
VM_ANDCELLS BD BS 1 2
```

cell[1] = passDw2

cell[0] = tbl[tblPtr]
tblPtr+ = sizeof(DWORD)
cell[3] = 0
cell[1] &= cell[0]
cell[2]– = 1
cell[3]++ = 1
cell[1] = passDw2
cell[0] = tbl[tblPtr]
tblPtr+ = sizeof(DWORD)
cell[7] = 0
cell[1] &= cell[0]
cell[2]– = 1
cell[7]++ = 1

So first after performing an AND on a DWORD from magicTable and on passDw1 it counts set bits using Kernighan’s method. Then it does the same on another DWORD from magicTable and passDw2.

And another fragment:
So it xors the $bitCount(passDw1 & magicTable[i])$ with $bitCount(passDw2 & magicTable[i+1])$ and then it checks the parity and it stores the result os one of the bits in some dword, decrements counter and goes again.

There are 256 bytes in magicTable, in each loop, 2 DWORD are taken, there are 0x20 loops, $0 \times 20 \times 2 \times sizeof(DWORD) = 256$.

After it calculates all the 0x20 bits it compares them with first of the MD2 calculated DWORDS.

Before we can continue, we need some math.

I suppose you know what matrix is and how matrix multiplication works.

Now imagine matrix in $\mathbb{Z}_2$. Elements of a matrix are 1’s and 0’s. Row multiplication becomes AND on bits, Addition is simply a xor.

Now what the VM code above does is basically calculating$^1$:

$$A \ast passDw1 + B \ast passDw2$$

where both $A$ and $B$ are 32 $\times$ 32 matrices, and both DWORD are column vectors.

Each of the 0x20 loops described above calculates ("row from $A^\prime$ $\times$ passDw1") + ("row from $B^\prime$ $\times$ passDw2"). Multiplication itself is hidden behind that bitcounting + XOR (at 0x40195F) + AND (at 0x401962)

Then it compares the result with the md2Dword1.

Because there are two calls to the subProc we get pair of equations:

$^1$But it took me some time to realize that, I’ve just left it went sleeping, and the next day when I was taking a shower enlightenment has come :D
\[
\begin{align*}
A \cdot \text{passDw}1 &+ B \cdot \text{passDw}2 = \text{md2Dword}1 \\
C \cdot \text{passDw}1 &+ D \cdot \text{passDw}2 = \text{md2Dword}2
\end{align*}
\]

Where \(A, B, C\) and \(D\) are all matrices of size \(32 \times 32\).

Due to linearity, we can rewrite the above
\[
\begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} \text{passDw}1 \\ \text{passDw}2 \end{bmatrix} = \begin{bmatrix} \text{md2Dword}1 \\ \text{md2Dword}2 \end{bmatrix}
\]

Where \(ABCD\) is a \(64 \times 64\) matrix, and the two vectors are of sizes \(64 \times 1\).

Now the only thing that’s left is calculate the inverse of the matrix, so the solution is simply
\[
\begin{bmatrix} \text{passDw}1 \\ \text{passDw}2 \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} \begin{bmatrix} \text{md2Dword}1 \\ \text{md2Dword}2 \end{bmatrix}
\]

I have precalculated the inverse matrix, using Gauss-Jordan elimination (it’s in tblz.py).

So basically what the keygen does is it multiplies inverted matrix with a vector made from calculating \(MD2\).

Because this is done only on bits, the multiplication isn’t straightforward in the keygen :)

Cheers and thanks to qpt\(^{\text{\textasciitilde}}\)J for this keygenme :)
A Appendix

.text:00401749  stru_401749
.text:0040174E  V_D (VM_PUSHONSTACK_D, offset aUsername)
.text:00401753  V_D (VM_PUSHONSTACK_D, 3EH)
.text:00401758  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:0040175E  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:00401760  V_B (VM_DOCALL_RESCELLO_B, offset GetDlgItemTextA, 10h)
.text:00401766  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:00401769  nameFilled  V_D (VM_PUSHONSTACK_D, 100h); DATA XREF: .text:00401769  o
.text:0040176A  V_D (VM_PUSHONSTACK_D, offset passw)
.text:0040176B  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:0040176D  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:00401772  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:00401778  V_D (VM_PUSHONSTACK_D, offset aEnterYourName); "Enter your name!"
.text:0040177E  V_D (VM_PUSHONSTACK_D, 0)
.text:00401781  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:00401785  passFilled  V_D (VM_PUSHONSTACK_D, 100h); DATA XREF: .text:00401769  o
.text:00401786  V_D (VM_PUSHONSTACK_D, offset passw)
.text:0040178B  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:0040178D  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:0040179A  V_D (VM_PUSHONSTACK_D, 3EH)
.text:0040179F  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:0040179E  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:0040179F  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:004017A5  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017A7  V_B (VM_MOVEFROMMENTOCALL_B_DP, 0, 403BA0h)
.text:004017AD  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017A7  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017B0  V_D (VM_PUSHONSTACK_D, offset passFilled)
.text:004017B5  V_D (VM_PUSHONSTACK_D, 0)
.text:004017B5  V_D (VM_PUSHONSTACK_D, 0)
.text:004017B8  V_D (VM_PUSHONSTACK_D, offset aEnterYourSerial); "Enter your serial!"
.text:004017C4  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017CA  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017C9  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017C9  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017D2  V_D (VM_DOCALL_RESCELLO_B, offset j_user32_MessageBoxA, 10h)
.text:004017D7  passFilled  V_D (VM_CALLFUNC_D, offset zero16b)
.text:004017D7  ; DATA XREF: .text:004017B0  o
.text:004017DD  V_D (VM_PUSHONSTACK_D, offset aUsername)
.text:004017E1  V_D (VM_PUSHONSTACK_D, offset aUsername)
.text:004017E7  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017E7  V_B (VM_PUSHFROMCELLOSTACK_B, 0)
.text:004017E9  V_D (VM_PUSHONSTACK_D, offset aUsername)
.text:004017EE  V_D (VM_PUSHONSTACK_D, offset bufAppend16, 8)
.text:004017F4  V_D (VM_CALLFUNC_D, offset calcMd2)
.text:004017F9  V_D (VM_PUSHONSTACK_D, 8)
.text:004017FE  V_D (VM_PUSHONSTACK_D, offset passDw1)
.text:00401803  V_D (VM_PUSHONSTACK_D, offset passw)
.text:00401808  V_D (VM_DOCALL_RESCELLO_B, offset strToInt, 0Ch)
.text:0040180E  V_D (VM_PUSHONSTACK_D, 8)
.text:00401813  V_D (VM_PUSHONSTACK_D, offset passDw2)
.text:00401818  V_D (VM_PUSHONSTACK_D, offset passDw2); "09876543"
.text:0040181D  V_D (VM_PUSHONSTACK_D, offset striToHex, 0Ch)
.text:00401823  V_D (VM_PUSHONSTACK_D, offset magicTable)
.text:00401828  V_D (VM_PUSHONSTACK_D, offset passDw2)
.text:0040182D  V_D (VM_PUSHONSTACK_D, offset passDw1)