R4ndom's Tutorial #16B: Self Modifying Code

by R4ndom on Aug.03, 2012, under Beginner, Reverse Engineering, Tutorials

In part two of this three part series we will go over self-modifying code and will eventually crack this binary. As promised, it will be challenging, but don't worry if you don't get everything- a lot is specific to this binary and you may never see again.

As always, the files you need are included with the download of this tutorial on the tutorials page.

Understanding The App

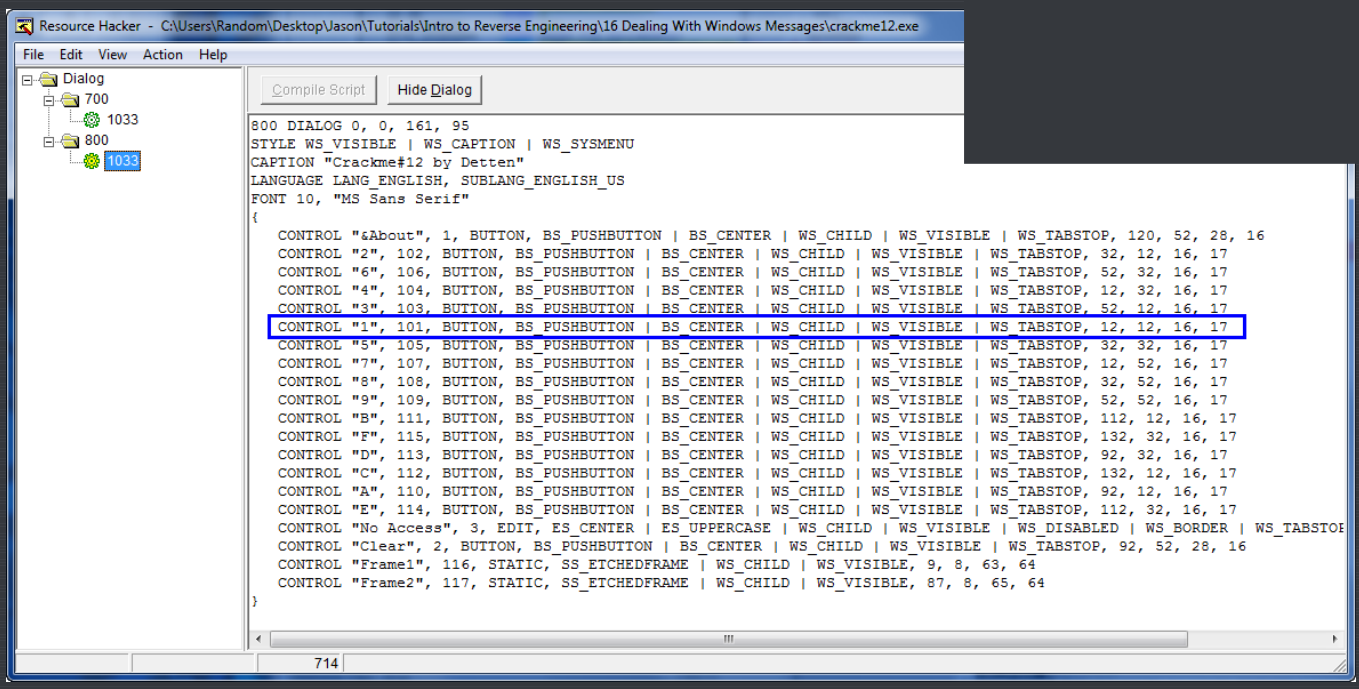
Now that we've seen how the basic message handler callback works, let's see if we can use this to crack this crackme. We can see that there are really only three messages that this app handles; 110 (INITDIALOG), 10 (DESTROY_WINDOW), and 111 (COMMAND). Any other messages are ignored. We've already gone through the init dialog code, and we don't really care about the destroy window code, as that's only called when we close the app. Therefore, anything worth noting happens in the WM_COMMAND section. So let's only pause Olly in that section. Remove any old BPs and set a new one at address 40108e, or after the compare/jump for ID 111:

```
00401072  .v 75 0D        JNZ SHORT crackme1.00401081
00401074  .  FF75 08      PUSH [ARG.1]
00401077  .  E8 32040000  CALL <JMP.&USER32.DestroyWindow>
0040107C  .v E9 CB010000  JMP crackme1.0040124C
00401081  >  817D 0C 11010000  CMP [ARG.2],111
00401088  .v 0F85 B5010000  JNZ crackme1.00401243
0040108E  .  8B45 10      MOV EAX,[ARG.3]
00401091  .  8B55 10      MOV EDX,[ARG.3]
00401094  .  C1EA 10      SHR EDX,10
00401097  .  66:0BD2      OR DX,DX
0040109A  .v 0F85 AC010000  JNZ crackme1.0040124C
004010A0  .  66:83F8 65   CMP AX,65
004010A4  .v 75 0C        JNZ SHORT crackme1.004010B2
004010A6  .  6A 01        PUSH 1
004010A8  .  E8 F8010000  CALL crackme1.004012A5
004010AD  .v E9 40010000  JMP crackme1.004011F2
004010B2  >  66:83F8 66   CMP AX,66
004010B6  .v 75 0C        JNZ SHORT crackme1.004010C4
004010B8  .  6A 02        PUSH 2
004010BA  .  E8 E6010000  CALL crackme1.004012A5
004010BF  .v E9 2E010000  JMP crackme1.004011F2
004010C4  >  66:83F8 67   CMP AX,67
004010C8  .v 75 0C        JNZ SHORT crackme1.004010D6
004010CA  .  6A 03        PUSH 3
004010CC     E8 D4010000  CALL crackme1.004012A5

hWnd = 000B052A ('Crackme#12 by
DestroyWindow
```

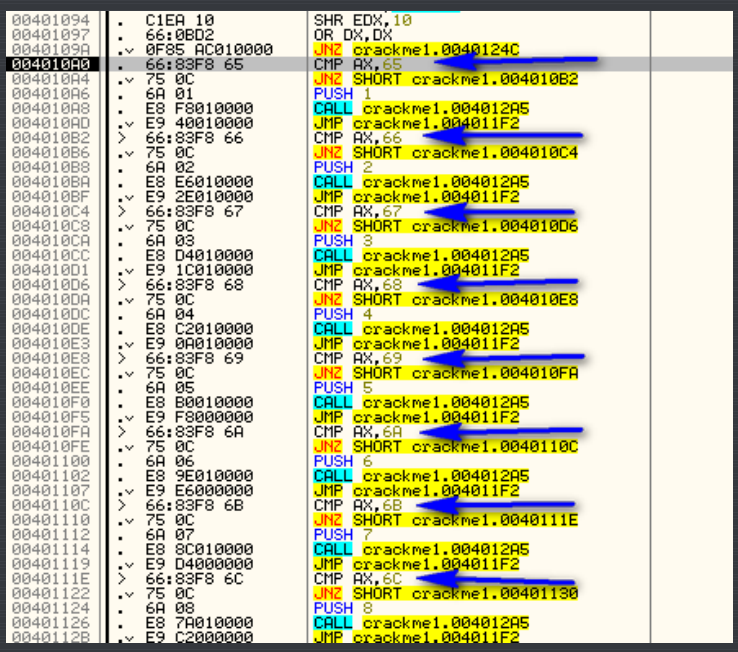
and run the app. You will notice that now if you move the mouse over the window, resize it, move it, or anything that doesn't involve clicking a button, Olly continues to run, as all of these messages are ignored. Now click on the first button, '1'. Olly breaks at our BP. We can also see that the ARG.3 variable contains '65':

```
00401100  .  6A 06        PUSH 6
00401102  .  E8 9E010000  CALL crackme
00401107  .v E9 E6000000  JMP crackme

Stack SS:[0012FAB4]=00000065
EAX=C0000000

Address   Hex dump
00402000  CD 2B D7 76 F3 D8 D7 76 E2 BB D8
00402010  20 20 4B 75 03 3B 4C 75 42 CF 4D
```

If we were to open our crackme in Resource Hacker (from last tutorial) and open up the main dialog, you would see that 65 (or 101 in decimal) is the ID for the number '1' button:



That is the ID that is in ARG.3! It is just the ID of the button. So we step down a couple lines and we see the compares begin, comparing the ID sent in with this message with the ID's hard coded into the app:



So, in the big picture, what this section is doing is checking the ID against all of the possible IDs, and when it finds a match, it calls to a section of code that handles that particular button. Notice also that right before

the call, a value is pushed onto the stack; 1 for 0×65, 2 for 0×66 etc. Since all of the calls are calling the same location, obviously the code at this section will differentiate which button that was clicked by what value is on the stack, again 1 for button 1, 2 for button 2 etc. So let's single step until we perform the call, step in, and see what we have:

```
004012A5  r$  55              PUSH EBP
004012A6  .   8BEC            MOV EBP,ESP
004012A8  .   60              PUSHAD
004012A9  .   8B0D 40304000   MOV ECX,DWORD PTR DS:[403040]
004012AF  .   8B1D 3C304000   MOV EBX,DWORD PTR DS:[40303C]
004012B5  .   A1 38304000     MOV EAX,DWORD PTR DS:[403038]
004012BA  .   807D 08 01      CMP BYTE PTR SS:[EBP+8],1
004012BE  .v  75 10           JNZ SHORT crackme1.004012D0
004012C0  .   81C1 4B050000   ADD ECX,54B
004012C6  .   0FAFD8          IMUL EBX,EAX
004012C9  .   33C1            XOR EAX,ECX
004012CB  .v  E9 17010000     JMP crackme1.004013E7
004012D0  >   807D 08 02      CMP BYTE PTR SS:[EBP+8],2
004012D4  .v  75 12           JNZ SHORT crackme1.004012E8
004012D6  .   81E9 33020000   SUB ECX,233
004012DC  .   6BDB 14         IMUL EBX,EBX,14
004012DF  .   03C8            ADD ECX,EAX
004012E1  .   23D8            AND EBX,EAX
004012E3  .v  E9 FF000000     JMP crackme1.004013E7
004012E8  >   807D 08 03      CMP BYTE PTR SS:[EBP+8],3
004012EC  .v  75 0F           JNZ SHORT crackme1.004012FD
004012EE  .   05 82050000     ADD EAX,582
004012F3  .   6BC9 16         IMUL ECX,ECX,16
004012F6  .   33D8            XOR EBX,EAX
004012F8  .v  E9 EA000000     JMP crackme1.004013E7
004012FD  >   807D 08 04      CMP BYTE PTR SS:[EBP+8],4
00401301  .v  75 0F           JNZ SHORT crackme1.00401312
00401303  .   23C3            AND EAX,EBX
00401305  .   81EB 22121100   SUB EBX,111222
0040130B  .   33C8            XOR ECX,EAX
0040130D  .v  E9 D5000000     JMP crackme1.004013E7
00401312  >   807D 08 05      CMP BYTE PTR SS:[EBP+8],5
00401316  .v  75 0C           JNZ SHORT crackme1.00401324
00401318  .   99              CDQ
00401319  .   F7F9            IDIV ECX
0040131B  .   2BDA            SUB EBX,EDX
0040131D  .   03C1            ADD EAX,ECX
0040131F  .v  E9 C3000000     JMP crackme1.004013E7
00401324  >   807D 08 06      CMP BYTE PTR SS:[EBP+8],6
00401328  .v  75 0F           JNZ SHORT crackme1.00401339
0040132A  .   33C1            XOR EAX,ECX
0040132C  .   23D8            AND EBX,EAX
0040132E  .   81C1 79685400   ADD ECX,546879
00401334  .v  E9 AE000000     JMP crackme1.004013E7
00401339  >   807D 08 07      CMP BYTE PTR SS:[EBP+8],7
0040133D  .v  75 12           JNZ SHORT crackme1.00401351
0040133F  .   81E9 F55F0200   SUB ECX,25FF5
```

Well now were into the meat of it! After setting up the stack we begin accessing the same memory locations we accessed in the WM_INITDIALOG section, namely starting at address 403038. So let's open that up in the dump so we have a frame of reference:

```
Address   Hex dump                                        ASCII
00403038  AD DE 00 00 AD DE 00 00 42 42 42 42 00 00 00 00  ¡Þ..¡Þ..BBBB....
00403048  00 00 00 00 00 30 40 00 00 00 00 00 00 00 00 00  .....0@.........
00403058  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403068  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403078  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403088  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403098  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030A8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030B8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030C8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030D8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030E8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
004030F8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403108  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403118  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00403128  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

There's our "DEAD" twice along with our 0x42s and the address 403000. Single stepping, we first move the 42s into ECX, and the two 0xDEADs into EBX and EAX:

```
Registers (FPU)
EAX 0000DEAD
ECX 42424242
EDX 00000000
EBX 0000DEAD
ESP 0012FA78
EBP 0012FA98
ESI 00000111
EDI 0012FB20
```

Next, we do a series of compares to find out which button we pushed based on the value that was pushed onto the stack. Here, **SS:[EBP+8]** is directly accessing this pushed value. Since we clicked the first button, we will perform the first set of instructions:

```
004012B5  .   A1 38304000     MOV EAX,DWORD PTR DS:[403038]
004012BA  .   807D 08 01      CMP BYTE PTR SS:[EBP+8],1
004012BE  .v  75 10           JNZ SHORT crackme1.004012D0
004012C0  .   81C1 4B050000   ADD ECX,54B
004012C6  .   0FAFD8          IMUL EBX,EAX
004012C9  .   33C1            XOR EAX,ECX
004012CB  .v  E9 17010000     JMP crackme1.004013E7
004012D0  >   807D 08 02      CMP BYTE PTR SS:[EBP+8],2
```

****One thing you can note: the author actually went through more trouble than he had to. He could have simply pushed the ARG.3 value which is the ID of the button and compared those IDs in this section, as opposed to pushing another value onto the stack and comparing those. Who knows, maybe the author assumed this was harder to read.****

The first thing we will do is add 0x54B to ECX (42424242) which gives us 4242478D. Next we multiply

EAX by EBX (which is 0xDEAD times 0xDEAD) which gives us C1B080E9. Finally, we XOR the ECX register with the EAX register and jump to location 4013E7. Stepping over the jump lands us here:

```
004013C3  .  0FAFD9          IMUL EBX,ECX
004013C6  .v EB 1F           JMP SHORT crackme1.004013E7
004013C8  >  807D 08 0E      CMP BYTE PTR SS:[EBP+8],0E
004013CC  .v 75 0D           JNZ SHORT crackme1.004013DB
004013CE  .  35 55550500     XOR EAX,55555
004013D3  .  81EB 51735800   SUB EBX,587351
004013D9  .v EB 0C           JMP SHORT crackme1.004013E7
004013DB  >  807D 08 0F      CMP BYTE PTR SS:[EBP+8],0F
004013DF  .v 75 06           JNZ SHORT crackme1.004013E7
004013E1  .  03C3            ADD EAX,EBX
004013E3  .  03D9            ADD EBX,ECX
004013E5  .  03C8            ADD ECX,EAX
004013E7  >  FF05 44304000   INC DWORD PTR DS:[403044]
004013ED  .  A3 38304000     MOV DWORD PTR DS:[403038],EAX
004013F2  .  891D 3C304000   MOV DWORD PTR DS:[40303C],EBX
004013F8  .  890D 40304000   MOV DWORD PTR DS:[403040],ECX
004013FE  .  61              POPAD
004013FF  .  C9              LEAVE
00401400  L. C2 0400         RETN 4
00401403  $  55              PUSH EBP
00401404  .  8BEC            MOV EBP,ESP
00401406  .  50              PUSH EAX
```

Which is toward the end of this method. If you scroll back up and take a look, you will see that basically all of the buttons do the same thing; they add a value, XOR a value and jump to the end. They just differ by the values. Then here, at the end, we increment the contents of memory location 403044 (which started as a zero), and we can assume this is some sort of counter. We then store our new values for ECX, EBX and EAX back into the same memory we read them from. After returning, we come back to the main function:

```
00401077  .  E8 32040000     CALL <JMP.&USER32.DestroyWindow>
0040107C  .v E9 CB010000     JMP crackme1.0040124C
00401081  >  817D 0C 11010000 CMP [ARG.2],111
00401088  .v 0F85 B5010000   JNZ crackme1.00401243
0040108E  .  8B45 10         MOV EAX,[ARG.3]
00401091  .  8B55 10         MOV EDX,[ARG.3]
00401094  .  C1EA 10         SHR EDX,10
00401097  .  66:0BD2         OR DX,DX
0040109A  .v 0F85 AC010000   JNZ crackme1.0040124C
004010A0  .  66:83F8 65      CMP AX,65
004010A4  .v 75 0C           JNZ SHORT crackme1.004010B2
004010A6  .  6A 01           PUSH 1
004010A8  .  E8 F8010000     CALL crackme1.004012A5
004010AD  .v E9 40010000     JMP crackme1.004011F2
004010B2  >  66:83F8 66      CMP AX,66
004010B6  .v 75 0C           JNZ SHORT crackme1.004010C4
004010B8  .  6A 02           PUSH 2
004010BA  .  E8 E6010000     CALL crackme1.004012A5
004010BF  .v E9 2E010000     JMP crackme1.004011F2
004010C4  >  66:83F8 67      CMP AX,67
004010C8  .v 75 0C           JNZ SHORT crackme1.004010D6
004010CA  .  6A 03           PUSH 3
004010CC  .  E8 D4010000     CALL crackme1.004012A5
```

and then perform a jump to location 4011F2:

```
004011F2  >  833D 48304000 03 CMP DWORD PTR DS:[403048],3
004011F9  .v 73 2D           JNB SHORT crackme1.00401228
004011FB  .  833D 44304000 0A CMP DWORD PTR DS:[403044],0A
00401202  .v 75 3D           JNZ SHORT crackme1.00401241
00401204  .  E8 43020000     CALL crackme1.0040144C
00401209  .  68 00304000     PUSH crackme1.00403000          ASCII "An error occured"
0040120E  .  FF75 08         PUSH [ARG.1]
00401211  .  E8 ED010000     CALL crackme1.00401403
00401216  .  C705 44304000 00( MOV DWORD PTR DS:[403044],0
00401220  .  FF05 48304000   INC DWORD PTR DS:[403048]
00401226  .v EB 19           JMP SHORT crackme1.00401241
00401228  >  68 11304000     PUSH crackme1.00403011          Text = "Trying to bruteforce?"
0040122D  .  6A 03           PUSH 3                          ControlID = 3
0040122F  .  FF75 08         PUSH [ARG.1]                    hWnd = 000B052A ('Crackme#12 by Detten',cla
00401232  .  E8 89020000     CALL <JMP.&USER32.SetDlgItemTextA> SetDlgItemTextA
00401237  .  C705 44304000 00( MOV DWORD PTR DS:[403044],0
00401241  >v EB 09           JMP SHORT crackme1.0040124C
00401243  >  B8 00000000     MOV EAX,0
00401248  .  C9              LEAVE
00401249  .  C2 1000         RETN 10
0040124C  >  B8 01000000     MOV EAX,1
00401251  .  C9              LEAVE
00401252  L. C2 1000         RETN 10
00401255  .  55              PUSH EBP
00401256  .  8BEC            MOV EBP,ESP
00401258  .  837D 0C 10      CMP [ARG.2],10
0040125C  .v 75 0C           JNZ SHORT crackme1.0040126A
0040125E  .  6A 01           PUSH 1                          Result = 1
00401260  .  FF75 08         PUSH [ARG.1]                    hWnd = 000B052A ('Crackme#12 by Detten',cla
00401263  .  E8 52020000     CALL <JMP.&USER32.EndDialog>    EndDialog
00401268  .v EB 32           JMP SHORT crackme1.0040129C
0040126A  >  817D 0C 11010000 CMP [ARG.2],111
00401271  .v 75 20           JNZ SHORT crackme1.00401293
00401273  .  8B45 10         MOV EAX,[ARG.3]
00401276  .  8B55 10         MOV EDX,[ARG.3]
00401279  .  C1EA 10         SHR EDX,10
```

Next we compare memory location 403048 (which is zero) with 3 (we don't know why yet), then compare our counter at address 403044 with 0x0A. Again, this indicates that 403044 contains a counter that counts to 0x0A. We then jump if it's not equal to 0x0A, telling us that we will run through this loop 10 times before we fall through. You may also have noticed the JNB at address 4011F9 that points to the brute-force message. Obviously, location 403048 will have some sort of counter in it, and if it gets above 3 we will get the brute-force message:

```
004011F9  .v 73 2D          JNB SHORT crackme1.00401228
004011FB  .  833D 44304000 0A CMP DWORD PTR DS:[403044],0A
00401202  .v 75 3D          JNZ SHORT crackme1.00401241
00401204  .  E8 43020000    CALL crackme1.0040144C
00401209  .  68 00304000    PUSH crackme1.00403000          ASCII "An error occured"
0040120E  .  FF75 08        PUSH [ARG.1]
00401211  .  E8 ED010000    CALL crackme1.00401403
00401216  .  C705 44304000 00( MOV DWORD PTR DS:[403044],0
00401220  .  FF05 48304000  INC DWORD PTR DS:[403048]
00401226  .v EB 19          JMP SHORT crackme1.00401241
00401228  > 68 11304000     PUSH crackme1.00403011          Text = "Trying to bruteforce?"
0040122D  .  6A 03          PUSH 3                          ControlID = 3
0040122F  .  FF75 08        PUSH [ARG.1]                    hWnd = 000B052A ('Crackme#12 by Detten
00401232  .  E8 89020000    CALL <JMP.&USER32.SetDlgItemTextA> SetDlgItemTextA
00401237  .  C705 44304000 00( MOV DWORD PTR DS:[403044],0
00401241  >v EB 09          JMP SHORT crackme1.0040124C
00401243  >  B8 00000000    MOV EAX,0
```

Now, let's continue running the program and click on button #2. We break at our BP:

```
0040106E  >  837D 0C 10     CMP [ARG.2],10
00401072  .v 75 0D          JNZ SHORT crackme1.00401081
00401074  .  FF75 08        PUSH [ARG.1]                    hWnd = 00
00401077  .  E8 32040000    CALL <JMP.&USER32.DestroyWindow> DestroyWi
0040107C  .v E9 CB010000    JMP crackme1.0040124C
00401081  >  817D 0C 11010000 CMP [ARG.2],111
00401088  .v 0F85 B5010000  JNZ crackme1.00401243
0040108E  .  8B45 10        MOV EAX,[ARG.3]
00401091  .  8B55 10        MOV EDX,[ARG.3]
00401094  .  C1EA 10        SHR EDX,10
00401097  .  66:0BD2        OR DX,DX
0040109A  .v 0F85 AC010000  JNZ crackme1.0040124C
004010A0  .  66:83F8 65     CMP AX,65
004010A4  .v 75 0C          JNZ SHORT crackme1.004010B2
004010A6  .  6A 01          PUSH 1
004010A8  .  E8 F8010000    CALL crackme1.004012A5
004010AD  .v E9 40010000    JMP crackme1.004011F2
004010B2  >  66:83F8 66     CMP AX,66
```

ARG.3, and in return, EAX and EDX will equal the ID of button number 2, or 0×66:

```
00401134  .v 75 0C          JNZ
Stack SS:[0018F9E8]=00000066
EAX=00000066
Address   Hex dump
00403038  20 99 42 42 E9 80 B0 C1 8D
00403048  00 00 00 00 00 30 40 00 00
```

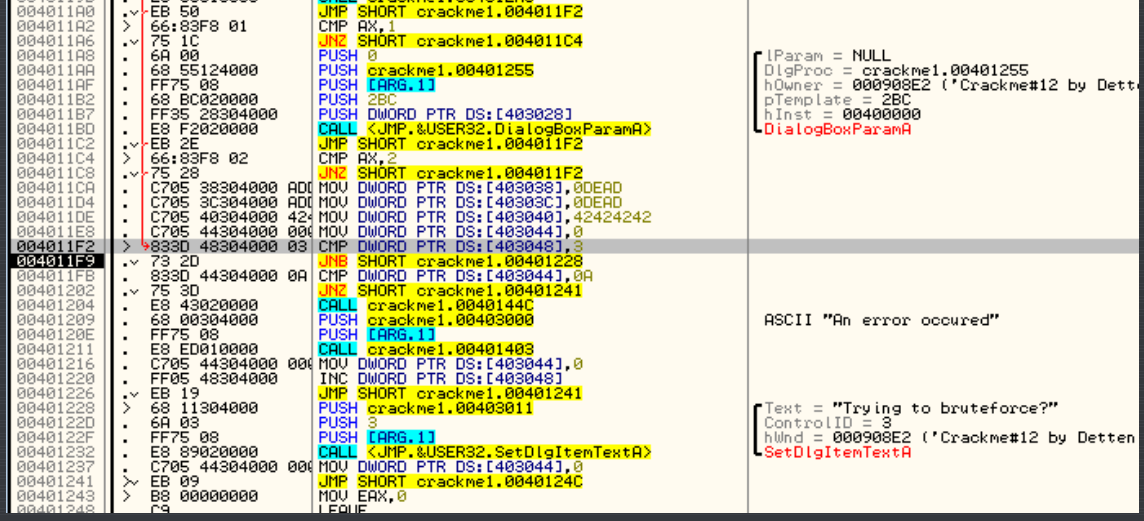
That means we will now run the code associated with button #2:

```
004010A6  .  6A 01          PUSH 1
004010A8  .  E8 F8010000    CALL crackme1.004012A5
004010AD  .v E9 40010000    JMP crackme1.004011F2
004010B2  >  66:83F8 66     CMP AX,66
004010B6  .v 75 0C          JNZ SHORT crackme1.004010C4
004010B8  .  6A 02          PUSH 2
004010BA  .  E8 E6010000    CALL crackme1.004012A5
004010BF  .v E9 2E010000    JMP crackme1.004011F2
004010C4  >  66:83F8 67     CMP AX,67
004010C8  .v 75 0C          JNZ SHORT crackme1.004010D6
004010CA  .  6A 03          PUSH 3
004010CC  .  E8 D4010000    CALL crackme1.004012A5
004010D1  .v E9 1C010000    JMP crackme1.004011F2
004010D6  >  66:83F8 68     CMP AX,68
```

Jumping into the call at 4010BA, we do the same thing we did the first time through, only this time 1) the memory will not contain 0xDEAD and 42424242, but instead will contain adjusted values and 2) since we clicked on the second button, we will perform the code at address 4012D6 which performs a SUB ECX, 233 and IMUL EBX, EBX, 14 etc. We then jump to the end of the routine again:

```
004013D9  .v EB 0C          JMP SHORT crackme1.004013E7
004013DB  >  807D 08 0F     CMP BYTE PTR SS:[EBP+8],0F
004013DF  .v 75 06          JNZ SHORT crackme1.004013E7
004013E1  .  03C3           ADD EAX,EBX
004013E3  .  03D9           ADD EBX,ECX
004013E5  .  03C8           ADD ECX,EAX
004013E7  >  FF05 44304000  INC DWORD PTR DS:[403044]
004013ED  .  A3 38304000    MOV DWORD PTR DS:[403038],EAX
004013F2  .  891D 3C304000  MOV DWORD PTR DS:[40303C],EBX
004013F8  .  890D 40304000  MOV DWORD PTR DS:[403040],ECX
004013FE  .  61             POPAD
004013FF  .  C9             LEAVE
00401400  .  C2 0400        RETN 4
00401403  $  55             PUSH EBP
00401404  .  8BEC           MOV EBP,ESP
00401406  .  50             PUSH EAX
```

Here, we increment the counter at 403044, move the new variables back into their memory locations and return to our main loop. Stepping once jumps to the end of our main loop:

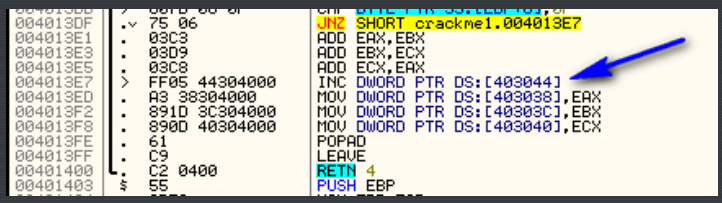


where we compare 403048 (which is still zero) and jump to the brute force message if it's greater than 3. We also compare 403044 with 0A and jump to the error code if our ID is above this (can you figure out why?). We then return from our main loop and return to the Windows loop that wait's for us to do something.

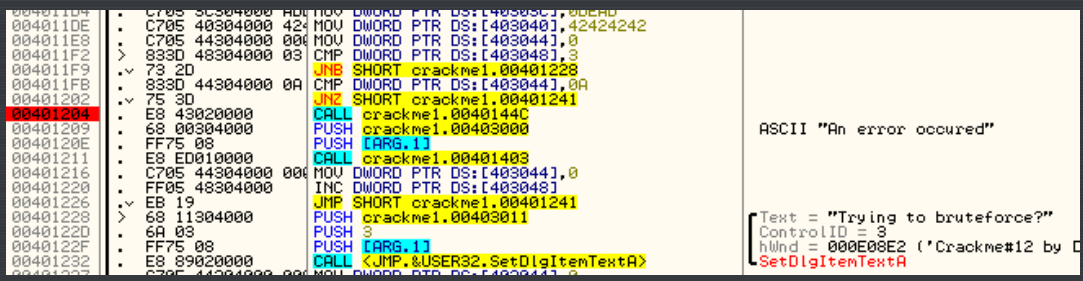
Cracking the App

Now that we understand how the app works, let's patch it. For this app, we need to use a little intuition. By following through the entire flow of this app, we can see that there are not a lot of compare/jumps out of the normal flow. Really, the only ones we see are the jump to the brute force message at address 4011F9, a jump to the 'about' box if we fall all the way through all of the compares from address 4010B2 to 4011A6, a jump to the 'clear' code that resets the memory locations back to 0xDEAD and 42424242 at address 4011CA, and a fall through to the 'error message' at 401204. If you click the 'about' button and trace the code, you will see that it only displays the about box and then returns to our main loop. Doing the same on the 'clear' button does the same. So that leaves either the brute force code or the error code.

Now here's where a little intuition comes in. Every time we checked the address 403048 to see if we should jump to the brute force message, the contents were zero and the jump was never taken. However, the compare at address 4011FB compares the counter at address 403044 and this will jump after reaching 0x0A. We also know that every time through the loop, the contents of 403044 were incremented, so we can assume this counter 'counts' how many buttons we've pressed:



Of course your first thought will be 'yeah, but, that code leads to an error message!". But does it? All the code does is load a pointer to a message that says there was an error, but is this displayed? Not in this code…so maybe not at all. This section of code looks highly suspicious, so let's trace through it. We know that we get to this code by clicking at least 0x0A (10) buttons. So let's place a BP at address 401204, clear our other breakpoints, and re-start the app (so we can count off 10 button presses):



Now, after clicking 10 buttons (I pressed the button number 1 10 times) we should break at our BP. First there is a call to 40144C. Let's step into that and see what it does:

```
0040144C  $  50                PUSH EAX
0040144D  .  68 50304000       PUSH crackme1.00403050                  pOldProtect = crackme1.00403050
00401452  .  6A 04             PUSH 4                                  NewProtect = PAGE_READWRITE
00401454  .  68 F4010000       PUSH 1F4                                Size = 1F4 (500.)
00401459  .  68 07144000       PUSH crackme1.00401407                  Address = crackme1.00401407
0040145E  .  E8 6F000000       CALL <JMP.&KERNEL32.VirtualProtect>     VirtualProtect
00401463  .  A1 38304000       MOV EAX,DWORD PTR DS:[403038]
00401468  .  3105 07144000     XOR DWORD PTR DS:[401407],EAX
0040146E  .  803D 07144000 52  CMP BYTE PTR DS:[401407],52
00401475  .v 75 18             JNZ SHORT crackme1.0040148F
00401477  .  A1 3C304000       MOV EAX,DWORD PTR DS:[40303C]
0040147C  .  3105 3B144000     XOR DWORD PTR DS:[40143B],EAX
```

Hmm, this sets up and then calls VirtualProtect. After reading the info on VirtualProtect, you will see that it is basically used to change the attributes of a section of memory. For example, the section of a binary that contains the executable code has it's attributes generally set to execute, but not to writeable, as there really isn't a lot of point of writing to the code section- that's what the data section is for. If you wanted to change a part of the code section to 'writable' in addition to 'executable', you would use this function. Then you could write to this memory section, in effect changing the code 'on the fly'. This is how self-modifying code works- it calls VirtualProtect on a section of memory in the code section, adds the 'writable' attribute, changes the code (perhaps XORing it with a number) and then calls VirtualProtect again to change the attributes back to executable only. Now, the code has been changed on the fly.

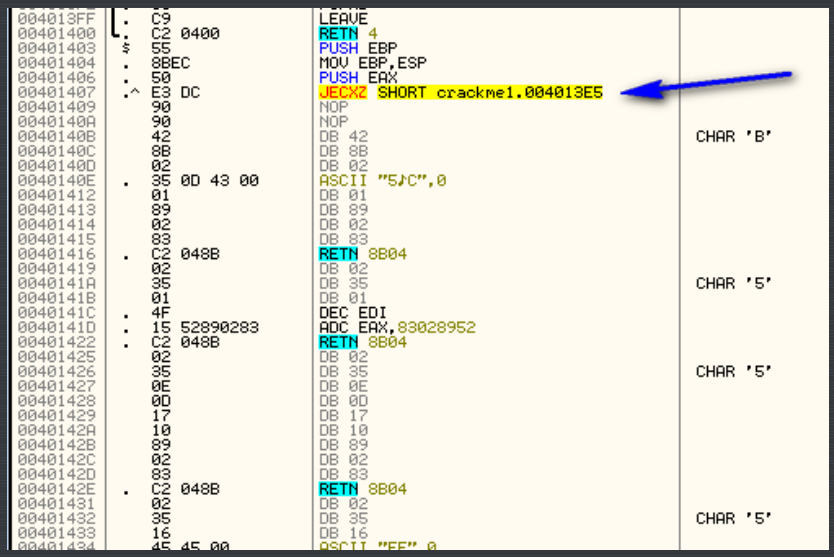
It appears that this app is doing something similar. The last argument to VirtualProtect is the memory location you want to change the attributes for, and the third value is the length in bytes of the section you want to alter. In this case we can see that the starting address is 401407 and the length is 0x1F4 (500). We can also see that the second argument is PAGE_READWRITE, making this section writable as well as readable. Let's look at this section of memory, starting at 401407, and see what is going to change:

```
004013F8  .  890D 40304000     MOV DWORD PTR DS:[403040],ECX
004013FE  .  61                POPAD
004013FF  .  C9                LEAVE
00401400  L. C2 0400           RETN 4
00401403  $  55                PUSH EBP
00401404  .  8BEC              MOV EBP,ESP
00401406  .  50                PUSH EAX                                kernel32.BaseThreadInitThunk
00401407  .v EB 3F             JMP SHORT crackme1.00401448
00401409     90                NOP
0040140A     90                NOP
0040140B     42                DB 42                                   CHAR 'B'
0040140C     8B                DB 8B
0040140D     02                DB 02
0040140E  .  35 0D 43 00       ASCII "5♪C",0
00401412     01                DB 01
00401413     89                DB 89
00401414     02                DB 02
00401415     83                DB 83
00401416  .  C2 048B           RETN 8B04
00401419     02                DB 02
0040141A     35                DB 35                                   CHAR '5'
0040141B     01                DB 01
0040141C  .  4F                DEC EDI
0040141D  .  15 52890283       ADC EAX,83028952
00401422  .  C2 048B           RETN 8B04
00401425     02                DB 02
00401426     35                DB 35                                   CHAR '5'
00401427     0E                DB 0E
00401428     0D                DB 0D
00401429     17                DB 17
0040142A     10                DB 10
0040142B     89                DB 89
0040142C     02                DB 02
0040142D     83                DB 83
0040142E  .  C2 048B           RETN 8B04
00401431     02                DB 02
00401432     35                DB 35                                   CHAR '5'
00401433     16                DB 16
00401434  .  45 45 00          ASCII "EE",0
00401437     89                DB 89
00401438  .  02                DB 02
00401439  .  5A 58             ASCII "ZX"
0040143B  .  0466E7BB          DD BBE76604
0040143F  .  4DBD088B          DD 8B08BD4D
00401443  .  E8 78000000       CALL <JMP.&USER32.SetDlgItemTextA>      SetDlgItemTextA
00401448  >  C9                LEAVE
00401449  .  C2 0800           RETN 8
0040144C  $  50                PUSH EAX                                kernel32.BaseThreadInitThunk
0040144D  .  68 50304000       PUSH crackme1.00403050                  pOldProtect = crackme1.00403050
00401452  .  6A 04             PUSH 4                                  NewProtect = PAGE_READWRITE
00401454  .  68 F4010000       PUSH 1F4                                Size = 1F4 (500.)
00401459  .  68 07144000       PUSH crackme1.00401407                  Address = crackme1.00401407
```

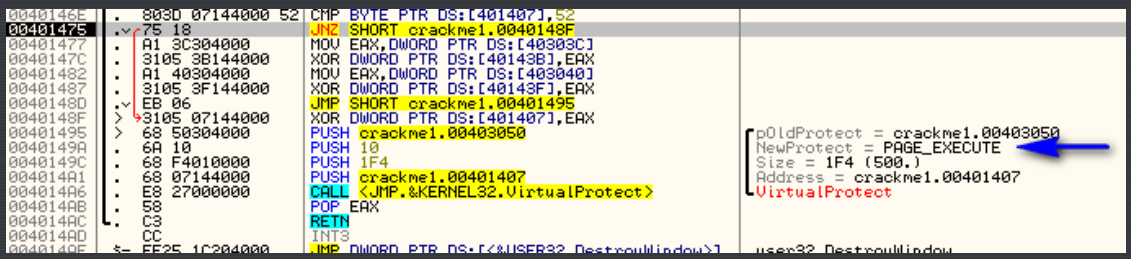
Hmmmmm. That looks really suspicious. It doesn't look like code at all. Let's keep going and see what the app changes in this section of memory. Step just past the call to VirtualProtect:

```
0040144C  $  50                PUSH EAX
0040144D  .  68 50304000       PUSH crackme1.00403050                  pOldProtect = crackme1.00403050
00401452  .  6A 04             PUSH 4                                  NewProtect = PAGE_READWRITE
00401454  .  68 F4010000       PUSH 1F4                                Size = 1F4 (500.)
00401459  .  68 07144000       PUSH crackme1.00401407                  Address = crackme1.00401407
0040145E  .  E8 6F000000       CALL <JMP.&KERNEL32.VirtualProtect>     VirtualProtect
00401463  .  A1 38304000       MOV EAX,DWORD PTR DS:[403038]
00401468  .  3105 07144000     XOR DWORD PTR DS:[401407],EAX
0040146E  .  803D 07144000 52  CMP BYTE PTR DS:[401407],52
00401475  .v 75 18             JNZ SHORT crackme1.0040148F
00401477  .  A1 3C304000       MOV EAX,DWORD PTR DS:[40303C]
0040147C  .  3105 3B144000     XOR DWORD PTR DS:[40143B],EAX
00401482  .  A1 40304000       MOV EAX,DWORD PTR DS:[403040]
00401487  .  3105 3F144000     XOR DWORD PTR DS:[40143F],EAX
0040148D  .v EB 06             JMP SHORT crackme1.00401495
0040148F  >  3105 07144000     XOR DWORD PTR DS:[401407],EAX
00401495  >  68 50304000       PUSH crackme1.00403050                  pOldProtect = crackme1.00403050
0040149A  .  6A 10             PUSH 10                                 NewProtect = PAGE_EXECUTE
0040149C  .  68 F4010000       PUSH 1F4                                Size = 1F4 (500.)
004014A1  .  68 07144000       PUSH crackme1.00401407                  Address = crackme1.00401407
004014A6  .  E8 27000000       CALL <JMP.&KERNEL32.VirtualProtect>     VirtualProtect
004014AB  .  58                POP EAX
004014AC  L. C3                RETN
```

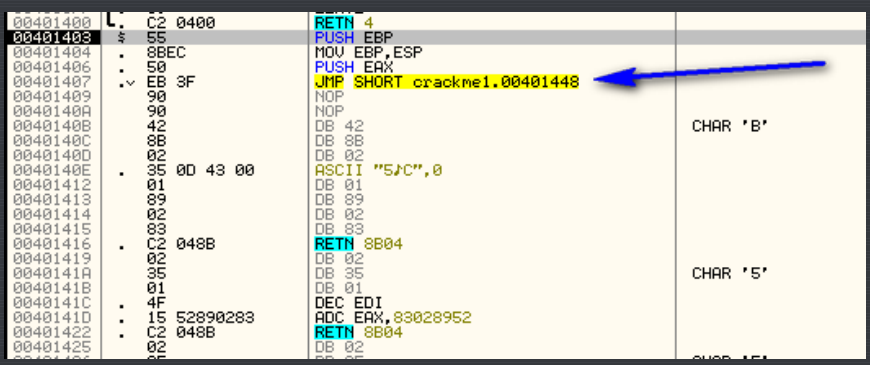
Now, the first thing we do is move the contents of memory location 403038 into EAX and XOR it with memory location 401407, storing the result back into address 401407. Wait a minute! 401407 was the first address of the memory section we changed the attributes for so that we could write to it. And 403038 began as 0xDEAD but was changed depending on which buttons we pressed (and in what order). So this sequence of instructions is changing that memory space based on what buttons and in which order they were pressed. Step over until we get to the JNZ at address 401475 and then let's look at address 401407:



You will notice that address 401407 was changed and now has a valid instruction in it, a **JECXZ SHORT crackme1.004013E5**. The app just added a conditional jump to it's own code! The way it did this was by changing the opcodes, or raw data, at that memory location. Going back to our current instruction, the next thing it does is compare the first byte at 401407 with 0×52 and jumps if it is not equal (to address 40148F). Looking at the above picture, we can see the opcode value at 401407 is "E3" which does not equal 52, so we will jump. The jump is to another setup and call of VirtualProtect, this time locking that section of memory back to executable:



but before this you may have noticed that memory location 401407 was XOR'ed again at address 40148F. Looking again at address 401407 we see that it was changed again:



So now we have a JMP instead of a JECXZ. So in effect, the app just changed it's own memory twice, once to be a JECXZ and the second time to be a JMP. Stepping again we return back to our main loop:

```
004011E8  .  C705 44304000 000 MOV DWORD PTR DS:[403044],0
004011F2  >  833D 48304000 03  CMP DWORD PTR DS:[403048],3
004011F9  .v 73 2D             JNB SHORT crackme1.00401228
004011FB  .  833D 44304000 0A  CMP DWORD PTR DS:[403044],0A
00401202  .v 75 3D             JNZ SHORT crackme1.00401241
00401204  .  E8 43020000       CALL crackme1.0040144C
00401209  .  68 00304000       PUSH crackme1.00403000                   ASCII "An error occured"
0040120E  .  FF75 08           PUSH [ARG.1]
00401211  .  E8 ED010000       CALL crackme1.00401403
00401216  .  C705 44304000 000 MOV DWORD PTR DS:[403044],0
00401220  .  FF05 48304000     INC DWORD PTR DS:[403048]
00401226  .v EB 19             JMP SHORT crackme1.00401241
00401228  >  68 11304000       PUSH crackme1.00403011                   Text = "Trying to bruteforce?"
0040122D  .  6A 03             PUSH 3                                   ControlID = 3
0040122F  .  FF75 08           PUSH [ARG.1]                             hWnd = 000F08E2 ('Crackme#12 by Detten',class='#3
00401232  .  E8 89020000       CALL <JMP.&USER32.SetDlgItemTextA>       SetDlgItemTextA
00401237  .  C705 44304000 000 MOV DWORD PTR DS:[403044],0
00401241  >v EB 09             JMP SHORT crackme1.0040124C
00401243  >  B8 00000000       MOV EAX,0
00401248  .  C9                LEAVE
00401249  .  C2 1000           RETN 10
```

We then push a value (F08E2) into the stack and call another routine at address 401403. stepping in we see that function:

```
00401400  L.  C2 0400          RETN 4
00401403  $   55               PUSH EBP                                 CHAR 'B'
00401404  .   8BEC             MOV EBP,ESP
00401406  .   50               PUSH EAX
00401407  .v  EB 3F            JMP SHORT crackme1.00401448
00401409      90               NOP
0040140A      90               NOP
0040140B      42               DB 42
0040140C      8B               DB 8B
0040140D      02               DB 02
0040140E  .   35 0D 43 00      ASCII "5♪C",0
00401412      01               DB 01
00401413      89               DB 89
```

Well, well, well. we have jumped to the area of memory that the app changed. We recognize the new JMP at address 401407. Let's step onto the jump and see where we go:

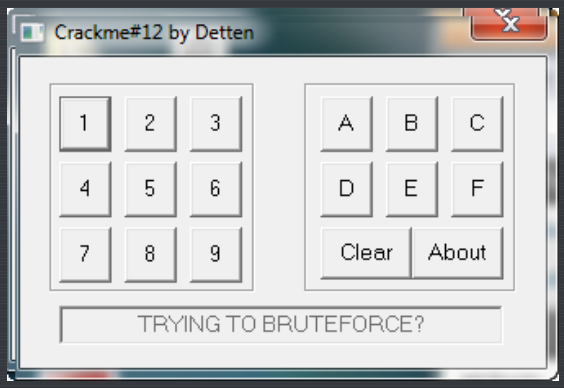
```
00401406  .   50               PUSH EAX
00401407  .v  EB 3F            JMP SHORT crackme1.00401448
00401409      90               NOP
0040140A      90               NOP
0040140B      42               DB 42                                    CHAR 'B'
0040140C      8B               DB 8B
0040140D      02               DB 02
0040140E  .   35 0D 43 00      ASCII "5♪C",0
00401412      01               DB 01
00401413      89               DB 89
00401414      02               DB 02
00401415      83               DB 83
00401416  .   C2 048B          RETN 8B04
00401419      02               DB 02
0040141A      35               DB 35                                    CHAR '5'
0040141B      01               DB 01
0040141C  .   4F               DEC EDI
0040141D  .   15 52890283      ADC EAX,83028952
00401422  .   C2 048B          RETN 8B04
00401425      02               DB 02
00401426      35               DB 35                                    CHAR '5'
00401427      0E               DB 0E
00401428      0D               DB 0D
00401429      17               DB 17
0040142A      10               DB 10
0040142B      89               DB 89
0040142C      02               DB 02
0040142D      83               DB 83
0040142E  .   C2 048B          RETN 8B04
00401431      02               DB 02
00401432      35               DB 35                                    CHAR '5'
00401433      16               DB 16
00401434  .   45 45 00         ASCII "EE",0
00401437      89               DB 89
00401438  .   02               DB 02
00401439  .   5A 58            ASCII "ZX"
0040143B  .   0466E7BB         DD BBE76604
0040143F  .   4DBD088B         DD 8B08BD4D
00401443  .   E8 78000000      CALL <JMP.&USER32.SetDlgItemTextA>       SetDlgItemTextA
00401448  >   C9               LEAVE
00401449  .   C2 0800          RETN 8
0040144C  $   50               PUSH EAX
0040144D  .   68 50304000      PUSH crackme1.00403050                   pOldProtect = crackme1.00403050
00401452  .   6A 04            PUSH 4                                   NewProtect = PAGE_READWRITE
00401454  .   68 F4010000      PUSH 1F4                                 Size = 1F4 (500.)
00401459  .   68 07144000      PUSH crackme1.00401407                   Address = crackme1.00401407
0040145E  .   E8 6F000000      CALL <JMP.&KERNEL32.VirtualProtect>      VirtualProtect
00401463  .   A1 38304000      MOV EAX,DWORD PTR DS:[403038]
00401468  .   3105 07144000    XOR DWORD PTR DS:[401407],EAX
```

Odd, it is jumping to a return. So it appears this didn't really do anything. We are now back at the main program:

```
00401204  .  E8 43020000       CALL crackme1.0040144C
00401209  .  68 00304000       PUSH crackme1.00403000                   ASCII "An error occured"
0040120E  .  FF75 08           PUSH [ARG.1]
00401211  .  E8 ED010000       CALL crackme1.00401403
00401216  .  C705 44304000 000 MOV DWORD PTR DS:[403044],0
00401220  .  FF05 48304000     INC DWORD PTR DS:[403048]
00401226  .v EB 19             JMP SHORT crackme1.00401241
00401228  >  68 11304000       PUSH crackme1.00403011                   Text = "Trying to bruteforce?"
0040122D  .  6A 03             PUSH 3                                   ControlID = 3
0040122F  .  FF75 08           PUSH [ARG.1]                             hWnd = 000F08E2 ('Crackme#12 by Detten',
00401232  .  E8 89020000       CALL <JMP.&USER32.SetDlgItemTextA>       SetDlgItemTextA
00401237  .  C705 44304000 000 MOV DWORD PTR DS:[403044],0
00401241  >v EB 09             JMP SHORT crackme1.0040124C
00401243  >  B8 00000000       MOV EAX,0
00401248  .  C9                LEAVE
00401249  .  C2 1000           RETN 10
0040124C  >  B8 01000000       MOV EAX,1
```

The next thing we are going to do is reset our counter from 0x0A back to zero. We will then increment the counter for the brute force check by one. Now we know how the brute force check works: if you enter a

(wrong) 10 digit code more than 3 times, the contents of location 403048 will be above 3 and we will jump to the brute force message. If you want to try it, go ahead. Just remove the BP at address 401204 and enter in a 10 digit code three times:



And we get the expected message.

Now, make sure our BP is still set at address 401204 (and clear all other BPs) and restart the app. We have to restart the app as once you enter the brute force message it zeroes out the counter every time. Can you see where?

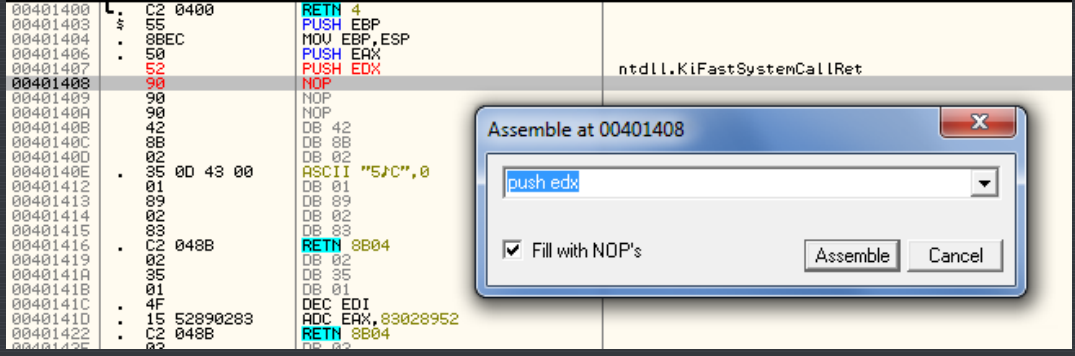
So what we know so far:

1) The password is 10 digits.
2) If you try more than three times with the wrong code, you get a brute force message and have to restart.
3) Every time you hit a button, memory locations 403038, 40303C and 403040 get modified in a different way for each button.
4) After hitting ten buttons, we enter a couple calls that check our code and changes a jump instruction in the code section of memory at address 401407.
5) If the password is not correct, the jump that is created points to a return that just returns us back to the main loop.
6) Therefore, entering the correct password must change this jump to something else, either a jump to a different memory location where our good boy will be, or changing more of the code in this area to create the goodboy at this memory section instead of creating a jump. This sounds more plausible as if it was simply changed into a jump to a new location, what is all this weird looking, non-functioning code for?

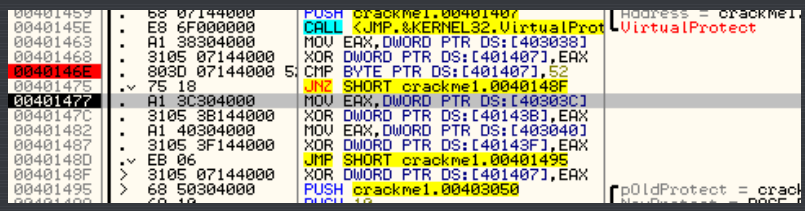
Knowing all this, we know we must zero in on the section of code that does the self modifying changes, namely the code starting at address 40144C. Let's look at that section again:

```
00401438   .  02              DB 02
00401439   .  5A 58           ASCII "ZX"
0040143B   .  0466E7BB        DD BBE76604
0040143F   .  4DBD088B        DD 8B08BD4D
00401443   .  E8 78000000     CALL <JMP.&USER32.SetDlgItemTex  SetDlgItemTextA
00401448   >  C9              LEAVE
00401449   .  C2 0800        RETN 8
0040144C   $  50              PUSH EAX
0040144D   .  68 50304000     PUSH crackme1.00403050           pOldProtect = crackme1.00403050
00401452   .  6A 04           PUSH 4                           NewProtect = PAGE_READWRITE
00401454   .  68 F4010000     PUSH 1F4                         Size = 1F4 (500.)
00401459   .  68 07144000     PUSH crackme1.00401407           Address = crackme1.00401407
0040145E   .  E8 6F000000     CALL <JMP.&KERNEL32.VirtualProt  VirtualProtect
00401463   .  A1 38304000     MOV EAX,DWORD PTR DS:[403038]
00401468   .  3105 07144000   XOR DWORD PTR DS:[401407],EAX
0040146E   .  803D 07144000 5  CMP BYTE PTR DS:[401407],52
00401475   .v 75 18           JNZ SHORT crackme1.0040148F
00401477   .  A1 3C304000     MOV EAX,DWORD PTR DS:[40303C]
0040147C   .  3105 3B144000   XOR DWORD PTR DS:[40143B],EAX
00401482   .  A1 40304000     MOV EAX,DWORD PTR DS:[403040]
00401487   .  3105 3F144000   XOR DWORD PTR DS:[40143F],EAX
0040148D   .v EB 06           JMP SHORT crackme1.00401495
0040148F   >  3105 07144000   XOR DWORD PTR DS:[401407],EAX
00401495   >  68 50304000     PUSH crackme1.00403050           pOldProtect = crackme1.00403050
0040149A   .  6A 10           PUSH 10                          NewProtect = PAGE_EXECUTE
0040149C   .  68 F4010000     PUSH 1F4                         Size = 1F4 (500.)
004014A1   .  68 07144000     PUSH crackme1.00401407           Address = crackme1.00401407
004014A6   .  E8 27000000     CALL <JMP.&KERNEL32.VirtualProt  VirtualProtect
004014AB   .  58              POP EAX                           0012FAD0
004014AC   .  C3              RETN
004014AD      CC              INT3
```

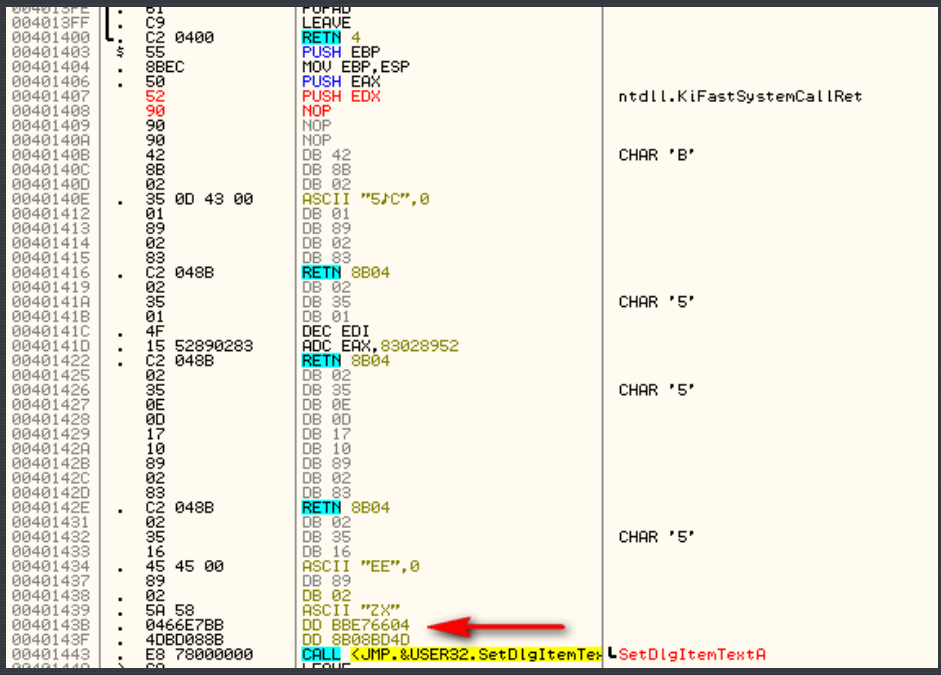
One thing we can gather is that the compare with 0×52 at address 40146e is pretty important. It basically tells the program that the changes to the code that have been made are the correct changes. But what is an opcode of 0×52 mean? Well, after a rather lengthy Google search, I discovered that opcode 0×52 is "PUSH EDX". So therefore, this code checks to see if the first instruction is a "PUSH EDX", and if it isn't, it bugs out. So what happens if we force that instruction to be a push edx? Let's try it. Set a BP at address 40146E where the code checks for the push instruction and run the app. When we break at this address, go to address 401407 and change the value to 0×52:



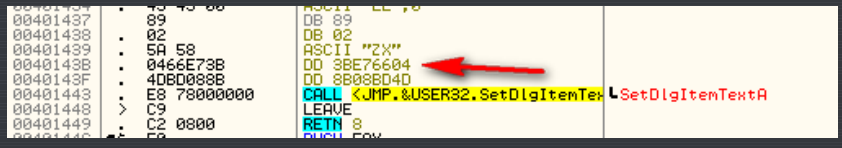
Now, single stepping, we should bypass the JNZ on the 0×52 check:



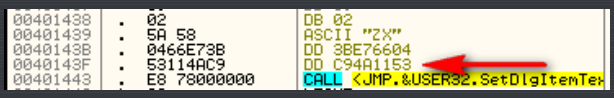
Now, the code moves the value at address 40303C into EAX and XORs it with memory location 40143B. What is that address? Let's look:



As we can see, it is just a memory location toward the end of our self modified code section. After XORing it, we then have:



And we then change the next location at 40143F by XORing that location with the contents of 403040:

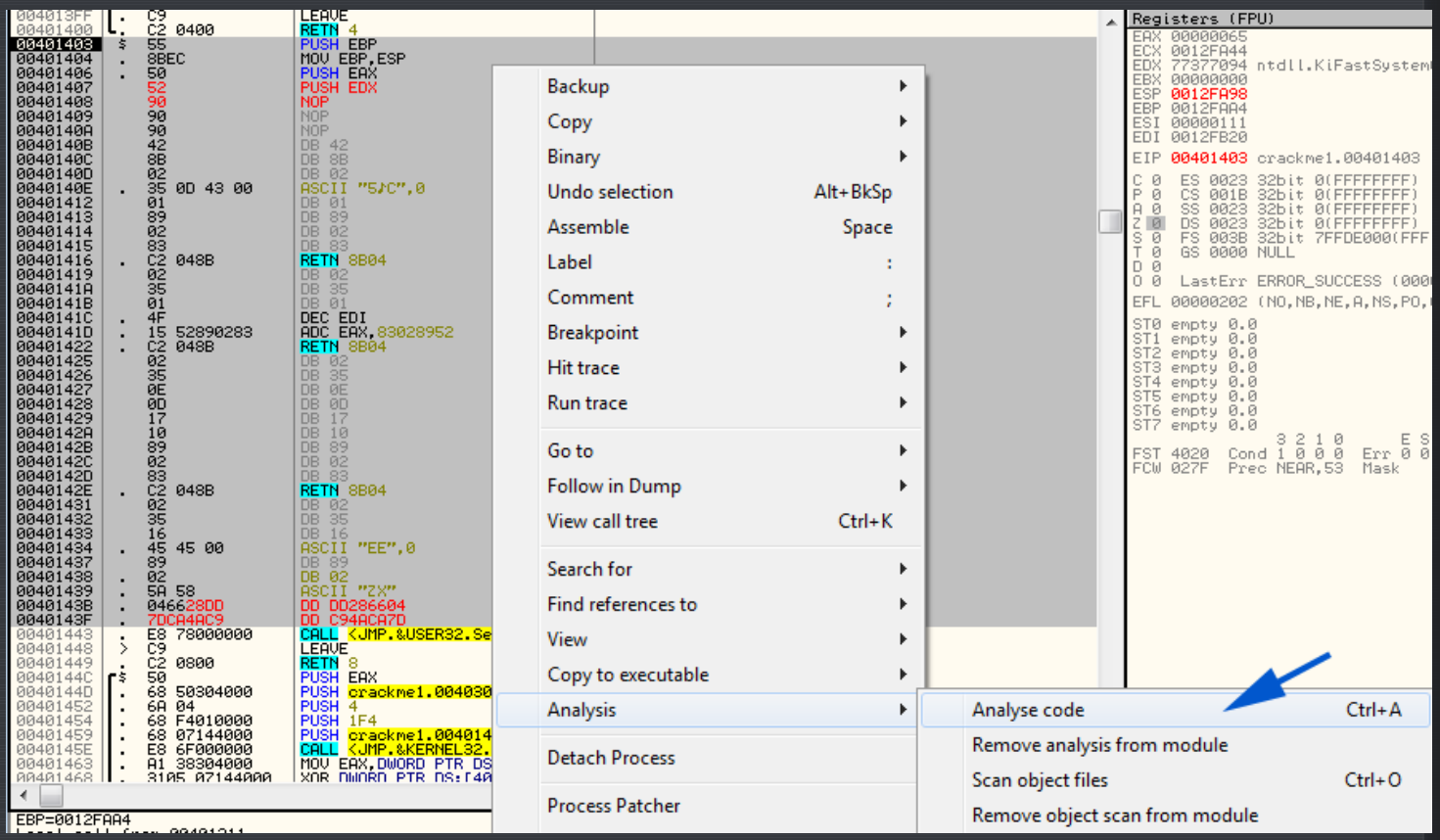


Now we know that these locations are not being changed into the proper code, so it's not really helping us, but seeing as this is the last thing that the app changes, it must be important. Let's keep going, now that we've changed the PUSH EDX and see what this sections does. Step back into the main loop and then

into the call at address 401211:

```
004013FF  .  C9              LEAVE
00401400  .  C2 0400         RETN 4
00401403  $  55              PUSH EBP
00401404  .  8BEC            MOV EBP,ESP
00401406  .  50              PUSH EAX
00401407     52              PUSH EDX                        ntdll.KiFastSystemCallR
00401408     90              NOP
00401409     90              NOP
0040140A     90              NOP
0040140B     42              DB 42                           CHAR 'B'
0040140C     8B              DB 8B
0040140D     02              DB 02
0040140E  .  35 0D 43 00     ASCII "5♪C",0
00401412     01              DB 01
00401413     89              DB 89
00401414     02              DB 02
00401415     83              DB 83
00401416  .  C2 048B         RETN 8B04
00401419     02              DB 02
0040141A     35              DB 35                           CHAR '5'
0040141B     01              DB 01
0040141C  .  4F              DEC EDI
0040141D  .  15 52890283     ADC EAX,83028952
00401422  .  C2 048B         RETN 8B04
00401425     02              DB 02
00401426     35              DB 35                           CHAR '5'
00401427     0E              DB 0E
00401428     0D              DB 0D
00401429     17              DB 17
0040142A     10              DB 10
0040142B     89              DB 89
0040142C     02              DB 02
0040142D     83              DB 83
0040142E     C2 048B         RETN 8B04
```

We are now at the beginning of the self modified section, starting with our added PUSH EDX. Let's tell Olly that things have changed and to re-analyze this section of code:



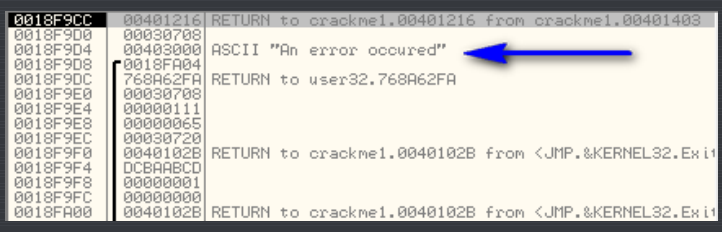
and things start to look a lot better:

It looks like real code now! Well, except for that bit at the end that we created incorrectly.

Now, this is where things get a little challenging and experience will be helpful here. We must look at this code from a 'big-picture' standpoint and think "What is it trying to do?". We have a PUSH EDX at the beginning. This, together with the POP EDX at the end tells us that EDX will be used in this code locally. We then have some empty NOPS which should probably contain code, though we don't know yet which code. We then have a bunch of memory locations being XORed with DWORDs. Experience will tell you that this normally means we are decrypting something, and in this case it's whatever EDX points to. We can deduce that because EDX is PUSHed and then never set, even though it goes on to be changed and referenced. The NOPs are probably the location to set EDX, and EDX will point to something that will be decrypted (or altered) with the XORs.

Lastly, we have several memory locations that were incorrectly decrypted starting at address 40143D. BUT the call to SetDlgItemTextA was not one of them, meaning this instruction was not changed. Generally before a call to SetDlgItemTextA, we have seen that arguments are pushed onto the stack, so we can assume that when we enter the correct password, the instructions from 40143d to 401442 will probably contain several push instructions (probably 3).

Now the big question is what should EDX point to? We have several choices here, and again, this is where experience comes in. An experienced reverse engineer will probably remember that string "An error occurred" and think "we never used that string. We saw that it was just a decoy and was never used. Maybe that is what will be decrypted…". Another hint that tells us that this is a viable solution is that the string is pushed onto the stack but is never used. Why? Here is a picture of the stack when we enter this code:



So assuming that we want to test our hypothesis, we want EDX to point to this string. The easiest way would be lo simply load EDX with the offset in memory that the "An error occurred" string is placed, namely address 403000. The problem is that would take up too many bytes. Looking again at our code, there are only 3 NOPs that we can use to load EDX with a pointer to the error string. Well,putting our assembly hat on, and remembering that the string is currently pushed onto the stack, maybe we can load EDX with the pointer to the string from the stack…

Generally we load a local variable with an instruction like this:

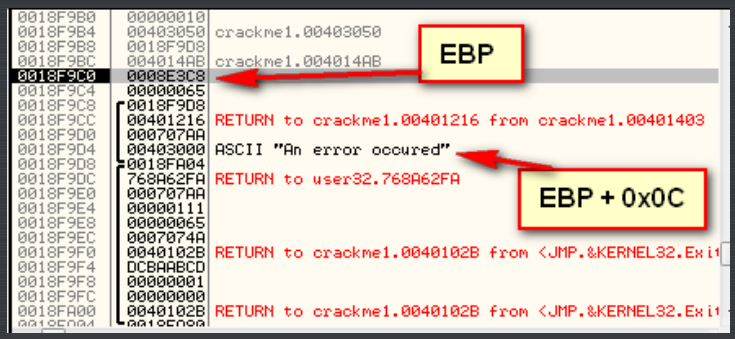
MOV EDX, [EBP + some_#] or MOV EDX, [EBP - some_#]

So the question is what is that number? Step over the first couple of instructions until we get to address

401408:

```
00401403  ┌$  55              PUSH EBP
00401404  .   8BEC            MOV EBP,ESP
00401406  .   50              PUSH EAX
00401407  .   52              PUSH EDX
00401408  .   90              NOP
00401409  .   90              NOP
0040140A  .   90              NOP
0040140B  >   42              INC EDX
0040140C  .   8B02            MOV EAX,DWORD PTR DS:[EDX]
0040140E  .   35 0D430001     XOR EAX,100430D
00401413  .   8902            MOV DWORD PTR DS:[EDX],EAX
00401415  .   83C2 04         ADD EDX,4
00401418  .   8B02            MOV EAX,DWORD PTR DS:[EDX]
0040141A  .   35 014F1552     XOR EAX,52154F01
0040141F  .   8902            MOV DWORD PTR DS:[EDX],EAX
00401421  .   83C2 04         ADD EDX,4
00401424  .   8B02            MOV EAX,DWORD PTR DS:[EDX]
00401426  .   35 0E0D1710     XOR EAX,10170D0E
0040142B  .   8902            MOV DWORD PTR DS:[EDX],EAX
0040142D  .   83C2 04         ADD EDX,4
00401430  .   8B02            MOV EAX,DWORD PTR DS:[EDX]
00401432  .   35 16454500     XOR EAX,454516
00401437  .   8902            MOV DWORD PTR DS:[EDX],EAX
00401439  .   5A              POP EDX                          0008E3C8
0040143A  .   58              POP EAX                          0008E3C8
0040143B  .   04 66           ADD AL,66
0040143D  .   28DD            SUB CH,BL
0040143F  .^  7D CA           JGE SHORT crackme1.0040140B
00401441  .   4A              DEC EDX
00401442      C9              LEAVE
```

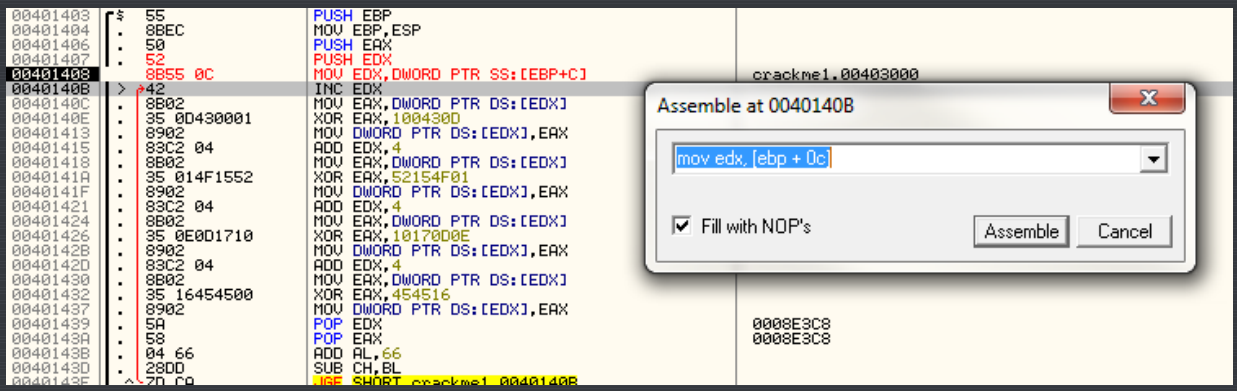
Looking back at the registers we can see that EBP points to 18F9C0 and that the error string is 12 bytes higher than EBP (lower on the stack):



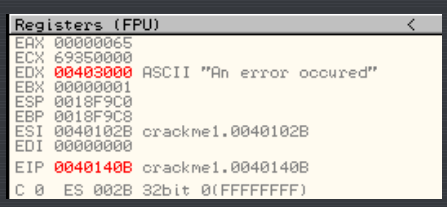
So our instruction that would load a pointer to the error string would be:

MOV EDX, [EBP + 0x0C]

Let's try it and see how many bytes it takes:

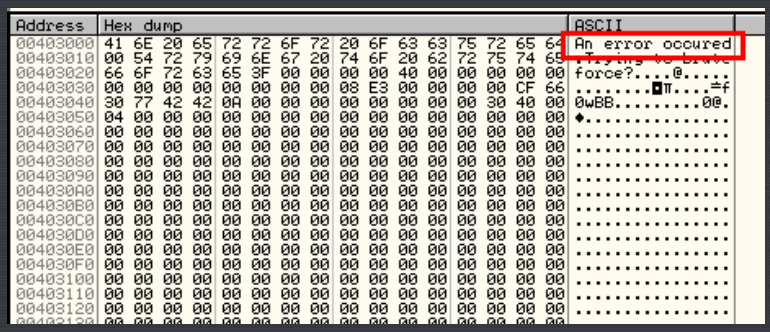


It seemed to fit just right 😀. Now let's single step and see what happens. First, at address 401408, EDX is loaded with a pointer to our text:

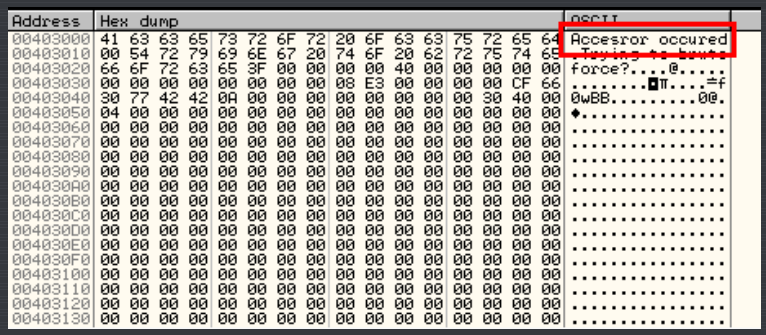


EDX is then incremented, so now points to the second character of our string(the 'n' in 'An error

occurred'). Four bytes (one dword) is loaded into EAX starting at the 'n' in 'An error occurred". EAX is then XORed with 0x100430D, making EAX equal to 0×73656363. This new value is then going to be saved into the address where the error string is located (at 403000). We can see the string before our value is stored:

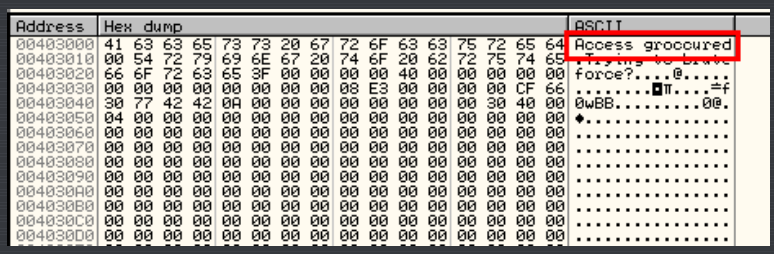


and after it's stored:

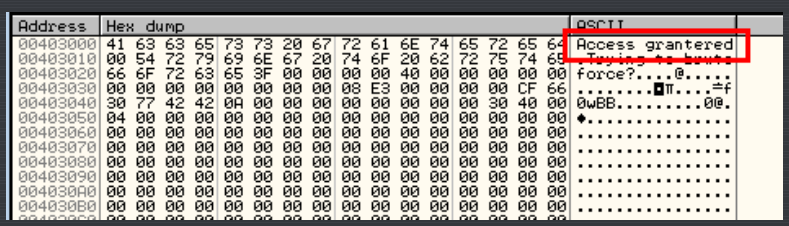


Hmmm. Our string is being modified. Let's keep going.

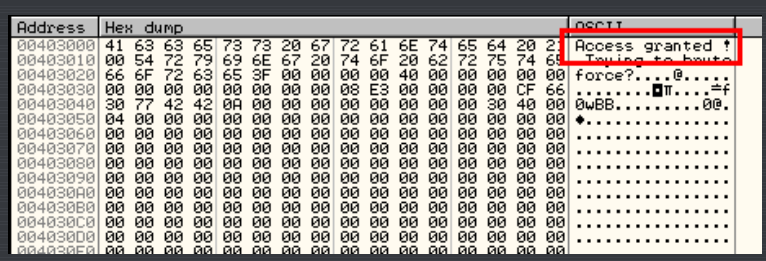
We now load the next four bytes, XOR them with 0x52154F01, and store them back into memory, which makes our string now look like this:



Ahh, now we're getting somewhere. Stepping over the next bit of code gives us the next four bytes:



And now we can probably guess what it's going to say. Stepping over the last modification shows us the entire string:



And we can see that we were right, though we're not out of the woods yet. We now know what the string should say. The problem is, since we entered an incorrect password and the last statements were incorrectly decrypted, our message will never be displayed. What we have to do is rebuild the pushing of the argument onto the stack for the SetDlgItemTextA. Getting help in Olly on SetGlgItemTextA, we can see that there are three arguments that need to be pushed onto the stack (in assembly order):

LPCTSTR lpString // text to set
int nIDDlgItem, // identifier of control
HWND hDlg, // handle of dialog box

The first one is easy:

PUSH [EBP + 0x0c]

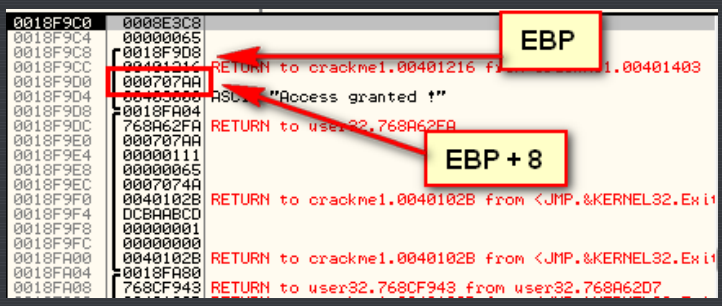
As this is the pointer to our new text string. The second and last options are a little harder, but fortunately we have a reference. There is a SetGlgItemTextA when the bruteforce message is displayed:

```
00401216  .  C705 44304000 000  MOV DWORD PTR DS:[403044],0
00401220  .  FF05 48304000      INC DWORD PTR DS:[403048]
00401226  .v EB 19              JMP SHORT crackme1.00401241
00401228  >  68 11304000        PUSH crackme1.00403011
0040122D  .  6A 03              PUSH 3
0040122F  .  FF75 08            PUSH [ARG.1]
00401232  .  E8 89020000        CALL <JMP.&USER32.SetDlgItemTextA>
00401237  .  C705 44304000 000  MOV DWORD PTR DS:[403044],0
00401241  >v EB 09              JMP SHORT crackme1.0040124C

Text = "Trying to bruteforce?"
ControlID = 3
hWnd = 000707AA ('Crackme#12 by Detten',class='#3277
SetDlgItemTextA
```

We can see that the ControlID equals 3 and the handle to the window is 707AA. The control ID is easy:

PUSH 3

The handle is a little harder, but looking at the stack again, it's not that hard:



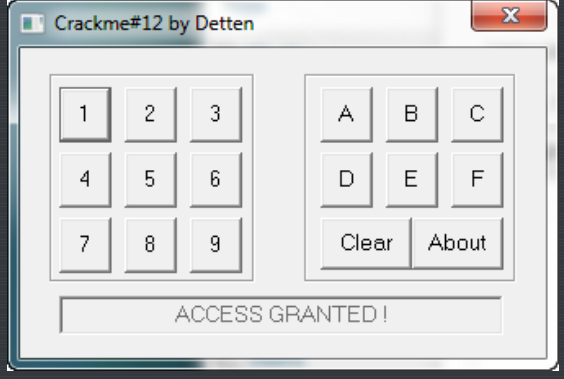
Fortunately, the handle is right on the stack:

PUSH DWORD PTR [EBP + 8]

Inserting our code now makes the disassembly look rather nice:

```
00401400  .  C2 0400            RETN 4
00401403  $  55                 PUSH EBP
00401404  .  8BEC               MOV EBP,ESP
00401406  .  50                 PUSH EAX
00401407  .  52                 PUSH EDX                        crackme1.0040300D
00401408     8B55 0C            MOV EDX,DWORD PTR SS:[EBP+C]    crackme1.00403000
0040140B  >  42                 INC EDX                         crackme1.0040300D
0040140C  .  8B02               MOV EAX,DWORD PTR DS:[EDX]
0040140E  .  35 0D430001        XOR EAX,100430D
00401413  .  8902               MOV DWORD PTR DS:[EDX],EAX
00401415  .  83C2 04            ADD EDX,4
00401418  .  8B02               MOV EAX,DWORD PTR DS:[EDX]
0040141A  .  35 014F1552        XOR EAX,52154F01
0040141F  .  8902               MOV DWORD PTR DS:[EDX],EAX
00401421  .  83C2 04            ADD EDX,4
00401424  .  8B02               MOV EAX,DWORD PTR DS:[EDX]
00401426  .  35 0E0D1710        XOR EAX,10170D0E
0040142B  .  8902               MOV DWORD PTR DS:[EDX],EAX
0040142D  .  83C2 04            ADD EDX,4
00401430  .  8B02               MOV EAX,DWORD PTR DS:[EDX]
00401432  .  35 16454500        XOR EAX,454516
00401437  .  8902               MOV DWORD PTR DS:[EDX],EAX
00401439  .  5A                 POP EDX                         0008E3C8
0040143A  .  58                 POP EAX                         0008E3C8
0040143B     FF75 0C            PUSH DWORD PTR SS:[EBP+C]       crackme1.00403000
0040143E     6A 03              PUSH 3
00401440     FF75 08            PUSH DWORD PTR SS:[EBP+8]
00401443  .  E8 78000000        CALL <JMP.&USER32.SetDlgItemTextA>  SetDlgItemTextA
00401448  .  C9                 LEAVE
00401449  .  C2 0800            RETN 8
0040144C  $  50                 PUSH EAX
```

Running the app finally rewards us with our goodboy:



Saving the binary with our patches now makes it possible to enter any 10 digit password and get the goodboy. We can consider the app cracked.

Of course, it kind of feels like we cheated a little bit (and we did). It seems like it would be much more gratifying to know what the password really is. Well, you're in luck as that's the topic of the next tutorial

Homework

Beginning at location 4012C0, each button dictates various manipulations on the main variables at addresses 402038, 40303C, and 403040. Let's call these variables a, b and c (a = 402038, b = 40303C and c = 403040). Can you figure out what each button does to manipulate these three variables? I'll give you the first one:

```
4012C0   add ecx, 54Bh   ; c += 54Bh
4012C6   imul ebx, eax    ; b *= a
4012C9   xor eax, ecx    ; a^= c
```

Now, can you figure out the remaining 14?

-Till next time.

R4ndom