R4ndom's Tutorial #16C: Bruteforcing

by R4ndom on Aug.17, 2012, under Beginner, Reverse Engineering, Tutorials

Introduction

Bruteforcing is a way to extract a serial (or password or whatever…) from a binary when you know the input and output of a encryption/decryption routine, but perhaps do not know how, nor wish to spend the time patching the software. It is the difference between cracked software coming with a patcher (or a copy of the patched executable) and coming with a username/serial that works. If you've ever downloaded cracked software and the person who cracked it includes a username/serial to crack it, they have probably used bruteforcing.

The way it works is, knowing the input and output of the encryption/decryption routine, you try every possibility that turns the input into the output until one matches. For example, if I enter a serial of '12121212′ and the app sends this into the decryption routine, and after the routine the app compares this with "j6^^gD7-L", we have the input as my serial and the output as that strange string. What we want to find out is how the '12121212′ was turned into 'j6^^gD7-L', and how we can enter our a serial that matches what the program expected as output, in other words what serial to put in so the app successfully registers us.

Keep in mind that this only works on binaries that user a username/serial in order to check the legitimacy of registration. If the app queries a database online, this won't work.

All that being said, bruteforcing is not terribly difficult. One requirement is that you know at least one programming language that you can make a bruteforcing program in.In this tutorial I will be discussing mostly assembly language, as that's what we're analyzing the crackme in. I am also including the main brutforcing routine in a couple of other languages, including _____, so you can get a sense of doing it in a higher-level language.

Another requirement is understanding how the username or serial (or both) is converted into the output. The reason for this is that it cuts down on the amount of operation we must try. If I say we must turn the password "SECRET" into the output "MESSAGE", there are an infinite amount of ways. But if I say that the only operations we can user are XORing the username with a certain value, well, that limits it a great deal.

Now we can begin talking specifically about our crackme. As always, you can download the relevant files on the tutorials page. In this tutorial we will be dealing with the same crackme we previously used, as well as our bruteforcing program.

Deciphering the Keys

As homework on the previous tutorial, I asked if you could decipher the various keys, and what modifications the app was performing on each. Here it is filled out completely:

```
004012A9 mov ecx, dword_403040          ; variable 'a'
004012AF mov ebx, dword_40303C          ; variable 'b'
004012B5 mov eax, dword_403038          ; variable 'c'
004012BA cmp [ebp+arg_0], 1             ; ***** Button 1
004012BE jnz short loc_4012D0
004012C0 add ecx, 54Bh                  ; c += 54Bh
004012C6 imul ebx, eax                  ; b *= a
004012C9 xor eax, ecx                   ; a^= c
004012CB jmp loc_4013E7

004012D0 cmp [ebp+arg_0], 2             ; ***** Button 2
004012D4 jnz short loc_4012E8
004012D6 sub ecx, 233h                  ; c -= 233h
004012DC imul ebx, 14h                  ; b *= 14h
004012DF add ecx, eax                   ; c += a
004012E1 and ebx, eax                   ; b &= a
004012E3 jmp loc_4013E7

004012E8 cmp [ebp+arg_0], 3             ; ***** Button 3
004012EC jnz short loc_4012FD
004012EE add eax, 582h                  ; a += 582h
004012F3 imul ecx, 16h                  ; c *= 16h
```

```
004012F6 xor ebx, eax                   ; b ^= a
004012F8 jmp loc_4013E7

004012FD cmp [ebp+arg_0], 4             ; ***** Button 4
00401301 jnz short loc_401312
00401303 and eax, ebx                   ; a &= b
00401305 sub ebx, 111222h               ; b -= 111222h
0040130B xor ecx, eax                   ; c ^= a
0040130D jmp loc_4013E7

00401312 cmp [ebp+arg_0], 5             ; ***** Button 5
00401316 jnz short loc_401324
00401318 cdq
00401319 idiv ecx                       ; a /= c, division rest --> (r)
0040131B sub ebx, edx                   ; b -= r
0040131D add eax, ecx                   ; a += c
0040131F jmp loc_4013E7

00401324 cmp [ebp+arg_0], 6             ; ***** Button 6
00401328 jnz short loc_401339
0040132A xor eax, ecx                   ; a ^= c
0040132C and ebx, eax                   ; b &= a
0040132E add ecx, 546879h               ; c += 546879h
00401334 jmp loc_4013E7

00401339 cmp [ebp+arg_0], 7             ; ***** Button 7
0040133D jnz short loc_401351
0040133F sub ecx, 25FF5h                ; c -= 25FF5h
00401345 xor ebx, ecx                   ; b ^= c
00401347 add eax, 401000h               ; a += 401000h
0040134C jmp loc_4013E7

00401351 cmp [ebp+arg_0], 8             ; ***** Button 8
00401355 jnz short loc_401367
00401357 xor eax, ecx                   ; a ^= c
00401359 imul ebx, 14h                  ; b *= 14h
0040135C add ecx, 12589h                ; c += 12589h
00401362 jmp loc_4013E7

00401367 cmp [ebp+arg_0], 9             ; ***** Button 9
0040136B jnz short loc_401378
0040136D sub eax, 542187h               ; a -= 542187h
00401372 sub ebx, eax                   ; b -= a
00401374 xor ecx, eax                   ; c ^= a
00401376 jmp short loc_4013E7

00401378 cmp [ebp+arg_0], 0Ah           ; ***** Button 10
0040137C jnz short loc_40138A
0040137E cdq
0040137F idiv ebx                       ; a /= b, division rest --> (r)
00401381 add ebx, edx                   ; b += r
00401383 imul eax, edx                  ; a *= r
00401386 xor ecx, edx                   ; c ^= r
00401388 jmp short loc_4013E7

0040138A cmp [ebp+arg_0], 0Bh           ; ***** Button 11
0040138E jnz short loc_4013A3
00401390 add ebx, 1234FEh               ; b += 1234FEh
00401396 add ecx, 2345DEh               ; c += 2345DEh
0040139C add eax, 9CA4439Bh             ; a += 9CA4439Bh
004013A1 jmp short loc_4013E7

004013A3 cmp [ebp+arg_0], 0Ch           ; ***** Button 12
004013A7 jnz short loc_4013B2
004013A9 xor eax, ebx                   ; a ^= b
004013AB sub ebx, ecx                   ; b -= c
004013AD imul ecx, 12h                  ; c *= 12h
004013B0 jmp short loc_4013E7

004013B2 cmp [ebp+arg_0], 0Dh           ; ***** Button 13
004013B6 jnz short loc_4013C8
004013B8 and eax, 12345678h             ; a &= 12345678h
004013BD sub ecx, 65875h                ; c -= 65875h
004013C3 imul ebx, ecx                  ; b *= c
004013C6 jmp short loc_4013E7

004013C8 cmp [ebp+arg_0], 0Eh           ; ***** Button 14
004013CC jnz short loc_4013DB
004013CE xor eax, 55555h                ; a ^= 55555h
004013D3 sub ebx, 587351h               ; b -= 587351h
004013D9 jmp short loc_4013E7

004013DB cmp [ebp+arg_0], 0Fh           ; ***** Button 15
004013DF jnz short loc_4013E7
004013E1 add eax, ebx                   ; a += b
004013E3 add ebx, ecx                   ; b += c
004013E5 add ecx, eax                   ; c += a
```

**** Special thanks go out to figugegl for doing most of this work for me in his tutorial (which I found after I finished two of the three parts of this series 😟). ****

So, now we know what operations are performed by pressing each key. The next thing we need is the inputs and outputs. These we already know- remember, the code in the self-modifying section started as one thing- a group of gobbledygook, and was later XORed into legitimate instructions. They were XORed with the three memory locations ‘a’, ‘b’ and ‘c’. Therefore, the input is the code prior to being XORed, and the outputs are the same locations after being XORed and modified against ‘a’, ‘b’ and ‘c’.

Address 401407 started as EB 3F 90 90 and became B9 B4 C5 9C after XORing with ‘a’
Address 40143B started as 04 66 E7 BB and became FF 75 0C 6A after XORing with ‘b’
Address 40143F started as 4D BD 08 8B and became 03 FF 75 08 after XORing with ‘c’

What we’re eventually going to do is try every combination of these modifications, mimicking trying every possible combination manually by clicking on the buttons. When, after performing 10 operations on a set, we have the correct values in the three variables, we know we have the correct password.

The writer of this crackme also provided the first two characters of the password. They are ’7′ and ’9′. The

reason for this is if you have a slow computer, it may take quite a while to go through all possible combinations. I have a screaming fast computer with 8 cores and it took about an hour to crack the serial starting with no characters known of the password. Starting with the 2 first characters known, it took about a minute. Normally you wouldn't have any characters (obviously) but seeing as this is not a tutorial on patience, I included the first two characters in the source code.

Here is the C source code for the bruteforcer (the entire project for Visual Studio is in the download):

```
#include <iostream>
using namespace std;

void brute( void )
{
    char finalAsciiSerial[11] = "";
    int    i, varA, varB, varC, tempVar, tempSerial[10];

    // we know the first number is '7'
    for (tempSerial[0] = 7; tempSerial[0] <= 7; tempSerial[0]++)
    {
     // and we know the second number is '9'
     for (tempSerial[1] = 9; tempSerial[1] <= 9; tempSerial[1]++)
     {
      for (tempSerial[2] = 1; tempSerial[2] <= 15; tempSerial[2]++)
      {
       for (tempSerial[3] = 1; tempSerial[3] <= 15; tempSerial[3]++)
       {
        for (tempSerial[4] = 1; tempSerial[4] <= 15; tempSerial[4]++)
        {
         cout << ".";    // Update display
         for (tempSerial[5] = 1; tempSerial[5] <= 15; tempSerial[5]++)
         {
          for (tempSerial[6] = 1; tempSerial[6] <= 15; tempSerial[6]++)
          {
           for (tempSerial[7] = 1; tempSerial[7] <= 15; tempSerial[7]++)
           {
            for (tempSerial[8] = 1; tempSerial[8] <= 15; tempSerial[8]++)
            {
             for (tempSerial[9] = 1; tempSerial[9] <= 15; tempSerial[9]++)
             {
                // Reset variables
                varA = 0xDEAD;
                varB = 0xDEAD;
                varC = 0x42424242;

                // Apply each digit
                for (i = 0; i < 10; i++)
                {
                    switch (tempSerial[i])
                    {
                    case 1:
                        varC += 0x54B;
                        varB *= varA;
                        varA ^= varC;
                        break;

                    case 2:
                        varC = varC - 0x233 + varA;
                        varB = (varB * 0x14) & varA;
                        break;

                    case 3:
                        varA += 0x582;
                        varC *= 0x16;
                        varB ^= varA;
                        break;

                    case 4:
                        varA &= varB;
                        varB -= 0x111222;
                        varC ^= varA;
                        break;

                    case 5:
                        if (varC != 0)         // Watch divide by zero!
                        {
                            varB -= (varA % varC);
                            varA /= varC;
                            varA += varC;
                        }
                        break;

                    case 6:
                        varA ^= varC;
                        varB &= varA;
                        varC += 0x546879;
                        break;

                    case 7:
                        varC -= 0x25FF5;
```

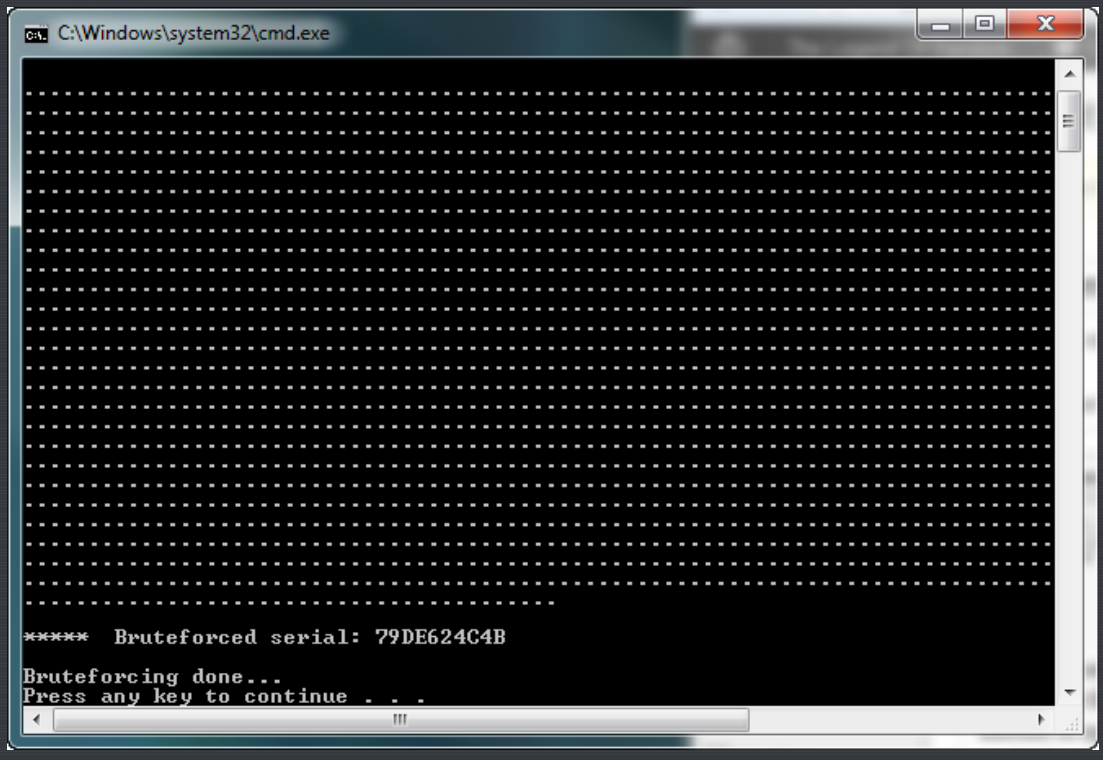
```
                        varB ^= varC;
                        varA += 0x401000;
                        break;

                    case 8:
                        varA ^= varC;
                        varB *= 0x14;
                        varC += 0x12589;
                        break;

                    case 9:
                        varA -= 0x542187;
                        varB -= varA;
                        varC ^= varA;
                        break;

                    case 10:
                        if (varB != 0)         // Watch divide by zero!
                        {
                            tempVar = varA % varB;
                            varA /= varB;
                            varB += tempVar;
                            varA *= tempVar;
                            varC ^= tempVar;
                        }
                        break;

                    case 11:
                        varB += 0x1234FE;
                        varC += 0x2345DE;
                        varA += 0x9CA4439B;
                        break;

                    case 12:
                        varA ^= varB;
                        varB -= varC;
                        varC *= 0x12;
                        break;

                    case 13:
                        varA &= 0x12345678;
                        varC -= 0x65875;
                        varB *= varC;
                        break;

                    case 14:
                        varA ^= 0x55555;
                        varB -= 0x587351;
                        break;

                    case 15:
                        varA += varB;
                        varB += varC;
                        varC += varA;
                        break;
                    }
                }

                // stop if serial equals proper values
                if ((varA == 0x9CC5B4B9)
                                && (varB == 0xD1EB13FB)
                                && (varC == 0x837D424E))
                {
                    // Convert to ASCII
                    for (i = 0; i < 10; i++)
                    {
                        if (tempSerial[i] <= 9)
                        {
                            finalAsciiSerial[i] = tempSerial[i] + 0x30;
                        }
                        else
                        {
                            finalAsciiSerial[i] = tempSerial[i] + 0x37;
                        }
                    }
                    cout << "\n\n*****  Bruteforced serial: ";
                                        cout << finalAsciiSerial << "\n";
                    return;
    }}}}}}}}}}
}

int main()
{
    cout << "Bruteforcer by R4ndom\n\n";

    brute();

    cout << "\nBruteforcing done...\n";

    return 0;
}
```

First, after setting up our variables, we cycle through each character of the password. The first and second characters we know are '7' and '9', but the others can be anywhere between 1 and 15 (1-F in hex). I inserted a cout instruction that prints a dot after a certain amount of operations because I hate programs that have no feedback. This gives us a growing string of dots so we know it has not crashed.

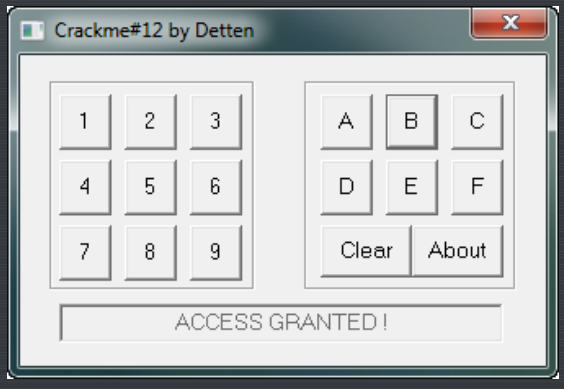
Next we perform the modifications on the variables depending on which key was pressed, just like the modifications we showed above.

After each set of 10 modifications (as the password is 10 characters long) we check the three variables to see if they are the same as the outputs we are looking for. If they are, we stop, convert the password string to ASCII, and show it. If they don't match, we continue on to the next permutation.

Here is the output from running the bruteforcer in a command window:



and here we can see our password Entering in this password, we see we have it right:



We have now bruteforced our crackme…

-Till next time

R4ndom