Fall 2014

CODEBREAKER CHALLENGE 2.0

Challenge Scenario

"An international terrorist organization has recently revised the operational security (OPSEC) procedures used to communicate with their members in the field. We have recovered a program that we believe is being used to covertly send encrypted messages. On the surface, the program appears to simply check the weather forecast in a few cities, but we believe there is more to the program than meets the eye. Your mission is to figure out how to execute the covert functionality, reverse-engineer the encryption algorithm, create a decryption program, and lastly figure out a way to decipher a message that was captured from a high-value target."

The Challenge

- There are 4 different levels or "tiers" to this challenge problem
 - Tier 1: Determine how to execute the hidden functionality
 - Tier 2: Bypass an authentication check
 - Tier 3: Create a decryption program
 - Tier 4: Decrypt message from a high-value target
- Each tier gets progressively harder and builds off lower tiers

The Challenge (cont.)

- The program will provide you with directions for where to send an encrypted email after you complete Tier 3
 - Please use your *.edu address so we know you are a student
 - You will then be sent the encrypted message for Tier 4
- Solutions are due by the end of 2014

Getting Started

- Review the 'Getting Started' tips in the Codebreaker Challenge document
- Download the IDA Demo from Hex-Rays
 - https://www.hexrays.com/products/ida/support/download_demo.h tml
- Try running the program with different options and observe its behavior
- Disassemble and start analyzing the binary

Reverse Engineering Tips

- Examine strings in the binary using IDA
 - Look for clues that relate to the functionality you are trying to find / reverse
 - Utilize IDA xrefs to find code that references the string(s) of interest
 - Utilize symbols (e.g., function names) to help determine what a section of code does
- Try setting debugger breakpoints to help RE code
 - Single-step after hitting a breakpoint and see how the values in registers/memory change
 - Look for the result of interesting computations. You can sometimes get the data you need from memory
- Leverage online resources, e.g., Intel manuals, RE lectures, etc. for help on reverse-engineering

Reverse Engineering Tips

 Optimizing compilers sometimes generate strange looking code for simple operations

In this challenge, you will encounter one such optimization during Tiers 2 − 4

 To save you some time/frustration, we will walk through an example and explain the math behind the optimization

What does this code do?

```
mov edx, oxAC769185 // edx = oxAC769185
mov eax, ecx // ecx = input value
imul edx
                    // edx:eax = eax * edx
lea eax, [edx + ecx*ox1]// eax = edx + ecx
mov edx, eax // edx = eax
                    // arith right shift; edx = edx >> ox6
sar edx, ox6
                    // eax = ecx
mov eax, ecx
sar eax, ox1f // eax = eax >> ox1f(31)
mov ebx, edx
                    // ebx = edx
sub ebx, eax
                    // ebx = ebx - eax
mov eax, ebx
                   // eax = ebx
imul eax, eax, ox5f
                   // edx:eax = eax * oxsf (95)
mov edx, ecx
                   // edx = ecx
sub edx, eax
                    // edx = edx - eax
                    // edx is the final result
```

- The code computes: edx = ecx % 95
- Why multiply by oxAC769185 and where did that number come from?
 - Division is a time consuming operation
 - When the divisor is a constant, the compiler can optimize the computation
- The basic trick is to multiply by a "magic value" (~ 2³²/d) and extract the leftmost 32 bits of the product
- The following site computes these numbers for you: http://www.hackersdelight.org/magic.htm

■ For wordsize $W \ge 3$ and divisor d, $2 \le d < 2^{W-1}$, we wish to find the least integer m and p such that:

$$\left[\frac{mn}{2^p}\right] = \left[\frac{n}{d}\right] \text{ for o } \le n < 2^{W-1}$$

$$\left[\frac{mn}{2^p}\right] + 1 = \left[\frac{n}{d}\right] \text{ for } -2^{W-1} \le n \le -1 \text{ (1b)}$$

with
$$0 \le m < 2^W$$
 and $p \ge W$

The "magic number" M used in the multiply instruction is given by:

$$M = \begin{cases} m, & \text{if } 0 \le m < 2^{W-1} \\ m - 2^{W}, & \text{if } 2^{W-1} \le m < 2^{W} \end{cases}$$

• Because (1b) must hold for n=-d,

$$\left[\frac{-md}{2^p}\right] + 1 = -1, \text{ which implies}$$

$$\frac{md}{2^p} > 1 \tag{2}$$

• Let n_c be the largest value of n such that $rem(n_c, d) = d-1$. It can be calculated from:

$$n_c = \left| \frac{2^{W-1}}{d} \right| * d - 1$$

$$= 2^{W-1} - rem(2^{W-1}, d) - 1 \quad (3)$$

• Because (1a) must hold for $n=n_c$:

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d} \quad \text{or} \quad \frac{mn_c}{2^p} < \frac{n_c + 1}{d} \quad (4)$$

Combining (4) with (2) gives:

$$\frac{2^p}{d} < m < \frac{2^p}{d} \frac{n_c + 1}{n_c} \tag{5}$$

■ Because m is to be the least integer satisfying (5), it is the next integer greater than $\frac{2^p}{d}$:

$$m = \frac{2^p + d - rem(2^p, d)}{d} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1$$
 (6)

$$2^p > n_c(d - rem(2^p, d)) \tag{7}$$

- Algorithm to find M and shift amount s from d
 - Compute n_c using (3)
 - Solve for p by trying successively larger values, starting at W, until satisfying the inequality in (7)
 - When the smallest $p \ge W$ satisfying (7) is found, m is calculated from (6)
 - ullet The shift amount is computed as: s=p-W
 - M is simply a reinterpretation of m as a signed integer

64-bit Data Types

Consider the following program:

```
int main(){
                                          // sizeof(char) == 1
 char one
                    = 0X11;
 char two
                    = 0X22;
                                          // sizeof(int) == 4
 int three
                    = ox33333333;
 int four
                    = 0 \times 4444444444
                                          // sizeof(long long) == 8
 printf("8b: \%hu 32b: \%u 64b: \%llu\n", one + two, three + four, five + six);
 return o;
```

64-bit Data Types - x86_64

Part 1: Move values onto the stack

```
BYTE PTR [rbp-ox2], ox11
mov
    BYTE PTR [rbp-ox1], ox22
mov
    mov
    DWORD PTR [rbp-ox8], ox44444444
mov
    DWORD PTR [rbp-ox20], ox5555555
mov
    DWORD PTR [rbp-ox1c], ox5555555
mov
    DWORD PTR [rbp-ox18], ox66666666
mov
    DWORD PTR [rbp-ox14], 0x66666666
mov
```

64-bit Data Types - x86_64

Part 2: Load into registers and compute

```
rax, QWORD PTR [rbp-ox18]
                                 mov
      rdx,QWORD PTR [rbp-ox20]
                                 mov
      rcx,[rdx+rax*1]
ea
                                 // rcx = rax + rdx*1
      eax,DWORD PTR [rbp-ox8]
                                 // <mark>0x4444444</mark> in eax
mov
      edx, DWORD PTR [rbp-oxc]
                                 // ox33333333 in edx
mov
add
                                 // edx = edx + eax
      edx,eax
movsx esi,BYTE PTR [rbp-ox2]
                                 // ox11 in esi
movsx eax,BYTE PTR [rbp-ox1]
                                 // ox22 in eax
add
      esi,eax
                                 // esi = esi + eax
```

64-bit Data Types - x86

No 64-bit registers ⊗

Let's make it work with 32-bit ones!

64-bit Data Types - x86

Part 1: Move values onto the stack (same as x86_64)

```
BYTE PTR [ebp-1], 0X11
mov
     BYTE PTR [ebp-2], 0X22
mov
     DWORD PTR [ebp-8], ox333333333
mov
     DWORD PTR [ebp-12],0X4444444
mov
     DWORD PTR [ebp-24], 0x55555555
mov
     DWORD PTR [ebp-20], 0X55555555
mov
     DWORD PTR [ebp-32], 0x66666666
mov
     DWORD PTR [ebp-28], 0x66666666
mov
```

64-bit Data Types – x86

Part 2: Load into registers and compute

```
eax, DWORD PTR [ebp-32] // 0x66666666 in eax
mov
       edx,DWORD PTR [ebp-28]
                                       mov
       eax, DWORD PTR [ebp-24]
add
                                       // eax = eax + 0×55555555
                                       // \text{ edx} = \text{edx} + \frac{0 \times 55555555}{0 \times 55555555} + \text{CF}
       edx,DWORD PTR [ebp-20]
adc
       eax,DWORD PTR [ebp-12] // ox444444 in eax
mov
       eax, DWORD PTR [ebp-8]
add
                                       // eax = eax + ox33333333
movsx edx,BYTE PTR [ebp-1]
                                       // ox11 in edx
movsx eax, BYTE PTR [ebp-2]
                                       // ox22 in eax
lea eax,[edx+eax]
                                       // eax = edx + eax*
```

Questions



Technical Walkthrough

- Original Codebreaker Challenge
 - Released during Fall 2013
 - Reverse-engineering / crypt challenge
 - Program prompted for a password
 - AES key is derived from SHA256 hash of password
 - AES-encrypted blob w/ instructions for how to submit the solution stored in the executable
 - To verify the correct password was entered, the derived key is used to compute an HMAC of the encrypted blob
 - If HMAC is correct, the blob is decrypted, revealing instructions.
 - Weakness of the design is in the key-derivation function
 - Only first 2-bytes of SHA256 hash matter
 - Easy to brute force

Running the program

```
Command Prompt - CodeBreaker.exe
C:∖challenge>
C:\challenge>
C:\challenge>CodeBreaker.exe
Welcome to the NSA Codebreaker Challenge!
Loading....
To win the challenge, you must be the first to decrypt the code
protected by the password. You are free to use any means at your
disposal to reverse-engineer and/or modify this binary in order to discover
the encryption key. Instructions for submitting your solution will be 'revealed'
when you are successful...good luck!
Enter Password:
```

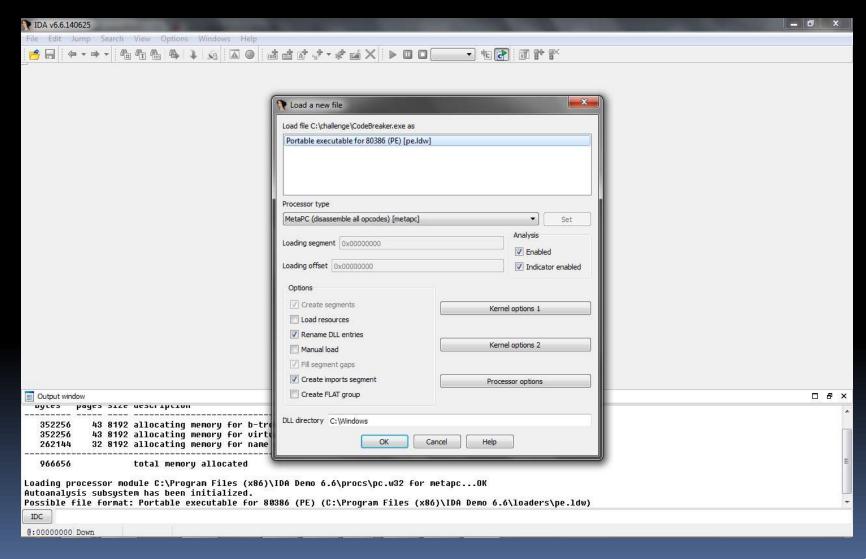
Running the program (2)

```
Command Prompt
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>
C:\challenge>CodeBreaker.exe
Welcome to the NSA Codebreaker Challenge!
Loading.....
To win the challenge, you must be the first to decrypt the code
protected by the password. You are free to use any means at your
disposal to reverse-engineer and/or modify this binary in order to discover
the encryption key. Instructions for submitting your solution will be 'revealed'
when you are successful...good luck!
Enter Password: password
ACCESS DENIED
C:\challenge>
```

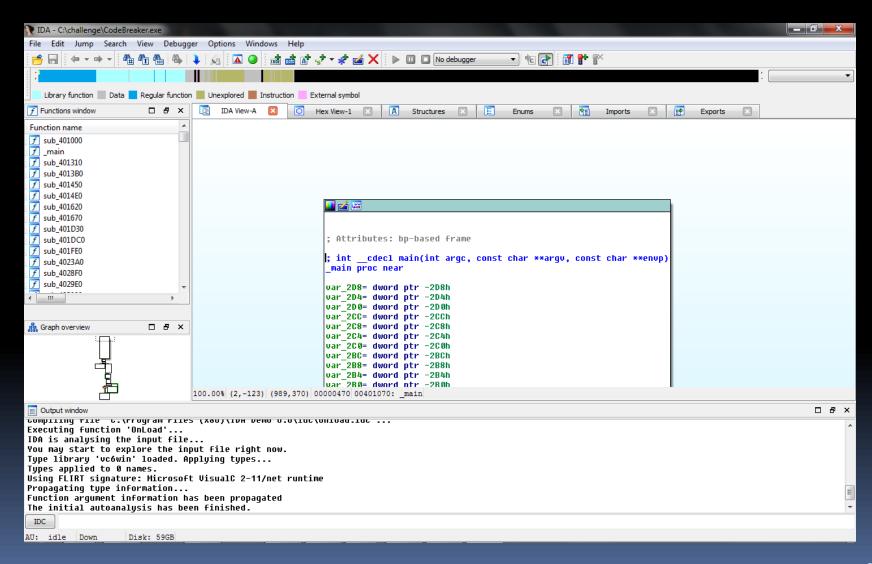
Disassemble

- Disassemble the Codebreaker binary
 - If asked whether you want to use Proximity View
 - Click no
 - Use graph view

Disassemble (2)



Disassemble (3)



Observe Strings

- Observe the strings that show up in IDA
 - Click Views->Open Subviews->Strings
 - You should see the strings that are displayed when you run the program

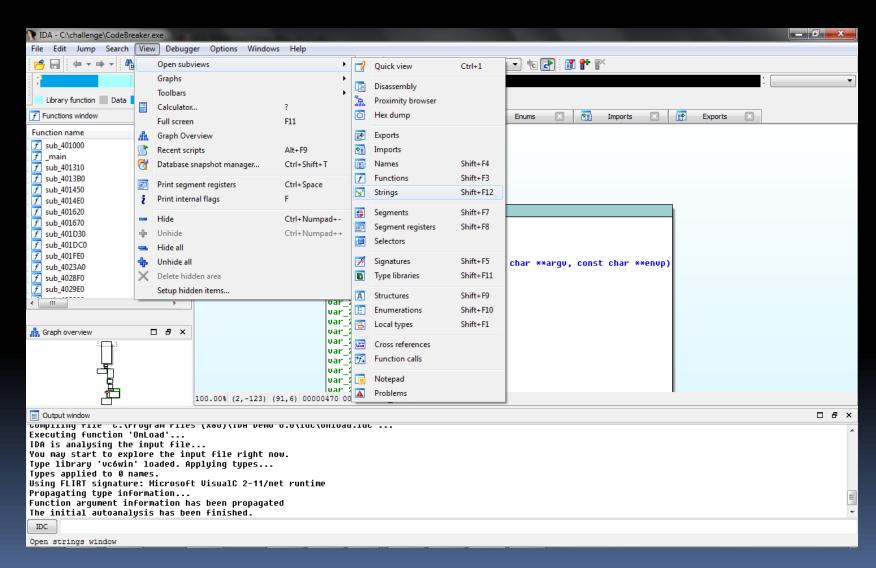
Welcome to the NSA Codebreaker Challenge!

Loading.....

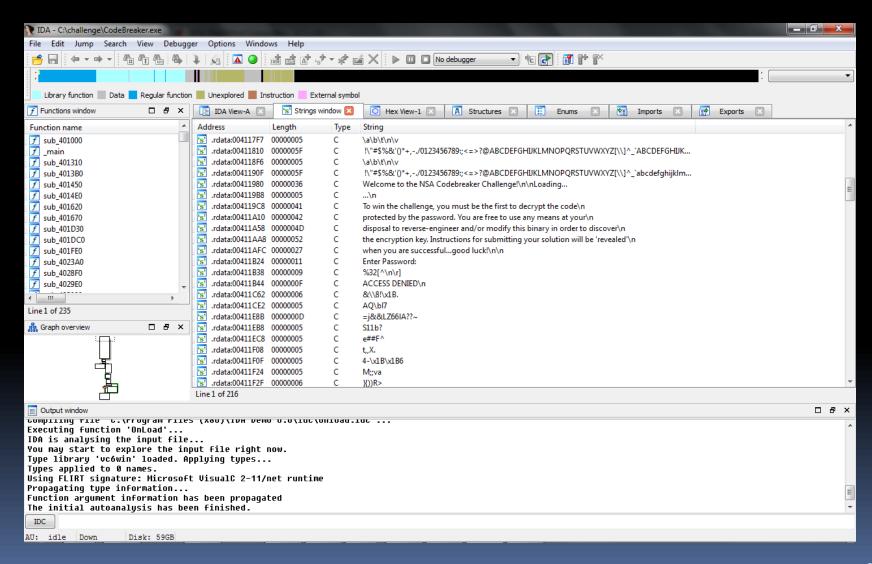
To win the challenge, you must be the first to decrypt the code protected by the password. You are free to use any means at your disposal to reverse-engineer and/or modify this binary in order to discover the encryption key. Instructions for submitting you solution will be 'revealed' when you are successful...good luck!

Enter Password:

Observe Strings (2)



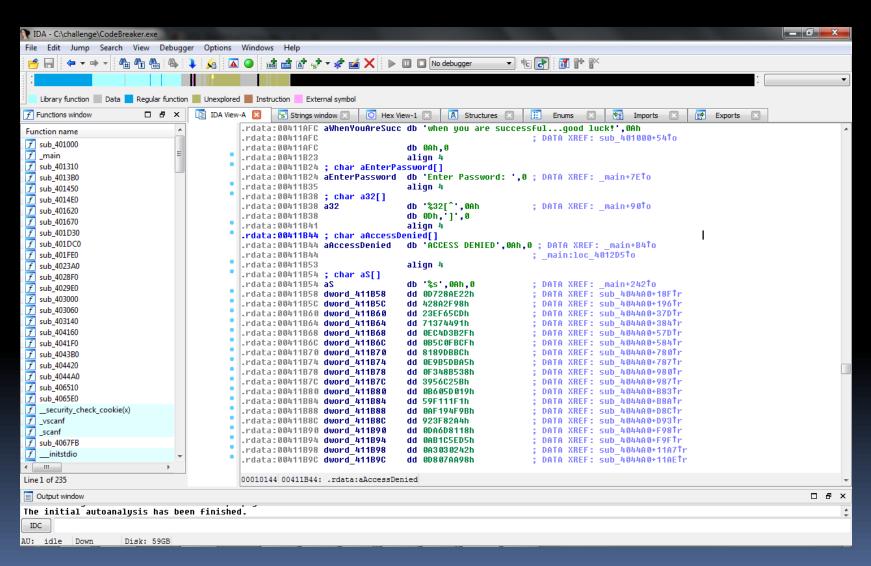
Observe Strings (3)



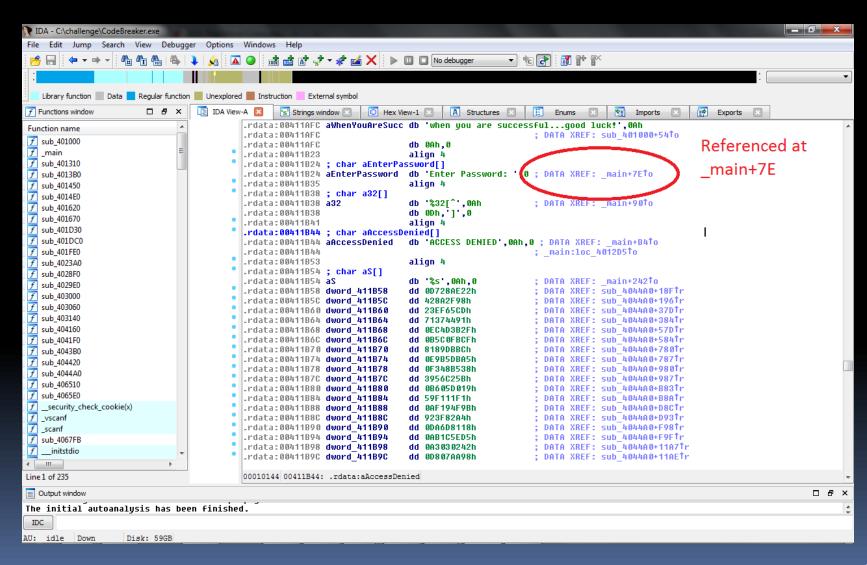
ACCESS DENIED

- Double click on the "ACCESS DENIED\n" string
 - This takes you to the data section of the binary where the string is stored
- To the right of the string are cross references to this address (show up as DATA XREF in IDA)
- Press ctrl-x to pull up a cross-references window; you will see two different references

ACCESS DENIED (2)



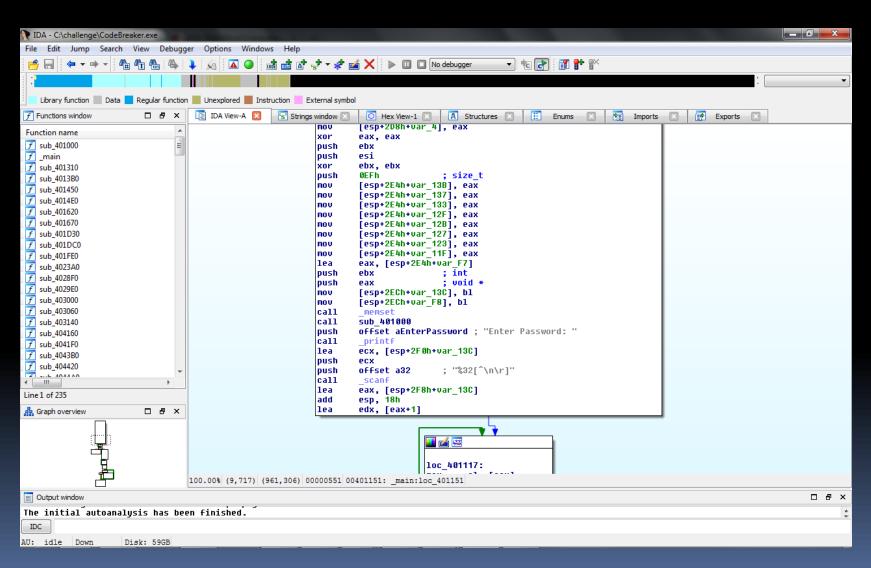
ACCESS DENIED (3)



Double-click Reference

- You should now be looking at disassembled x86 code
 - We just leveraged the fact that in order to print the ACCESS DENIED message to the screen, the code had to reference the address in the data section of the program where the string was stored.
- Using xrefs in IDA is a quick and easy way to find interesting code sections

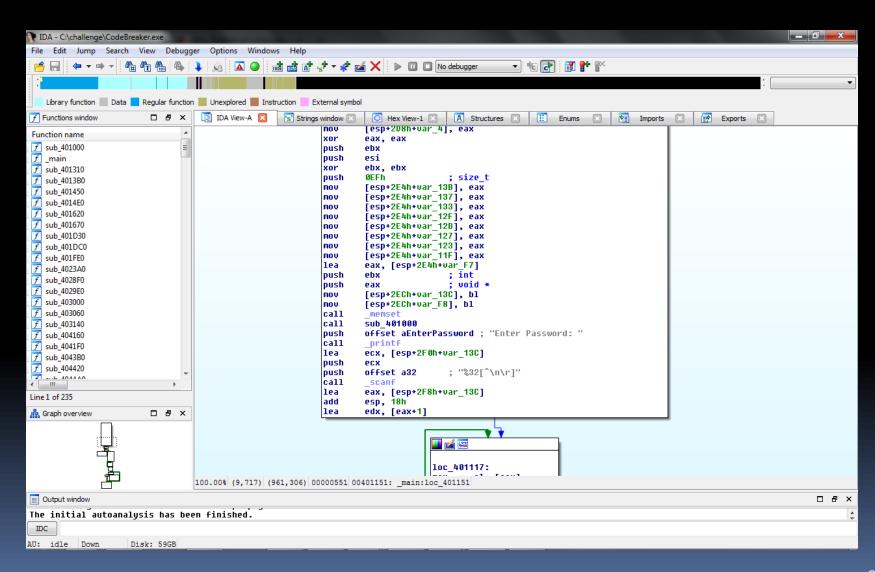
Double-click Reference (2)



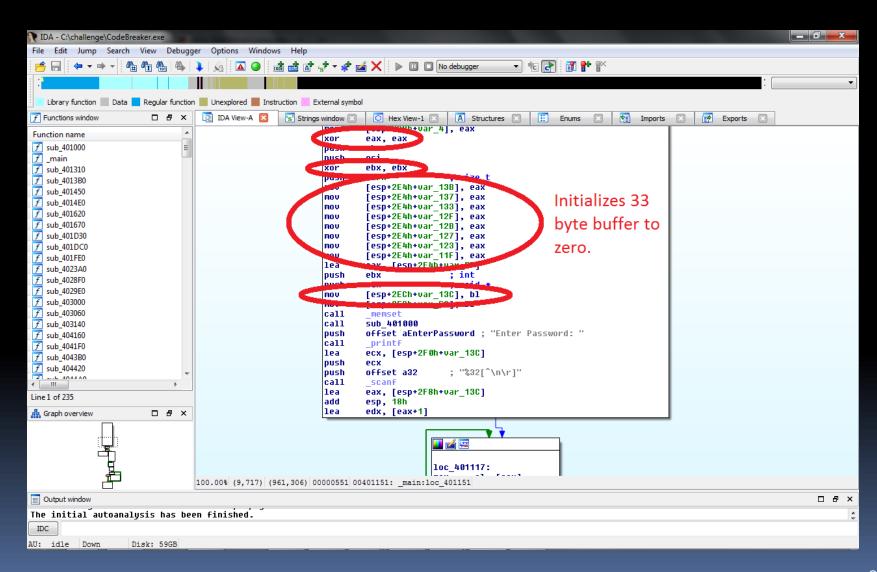
Explore Code Block

- Explore the code block preceding the access denied message (note: you can double-click call statements to visit the function body)
- You will see routines to:
 - print the initial welcome screen,
 - prompt the user for the password and
 - compute the password length
- The branch for "ACCESS DENIED" is taken when the strlen of the password is o.

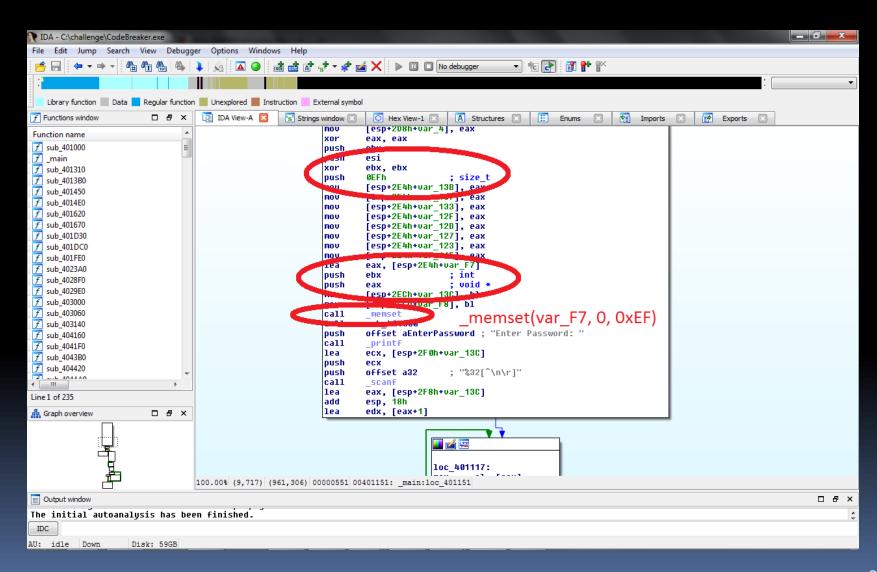
Explore Code Block (2)



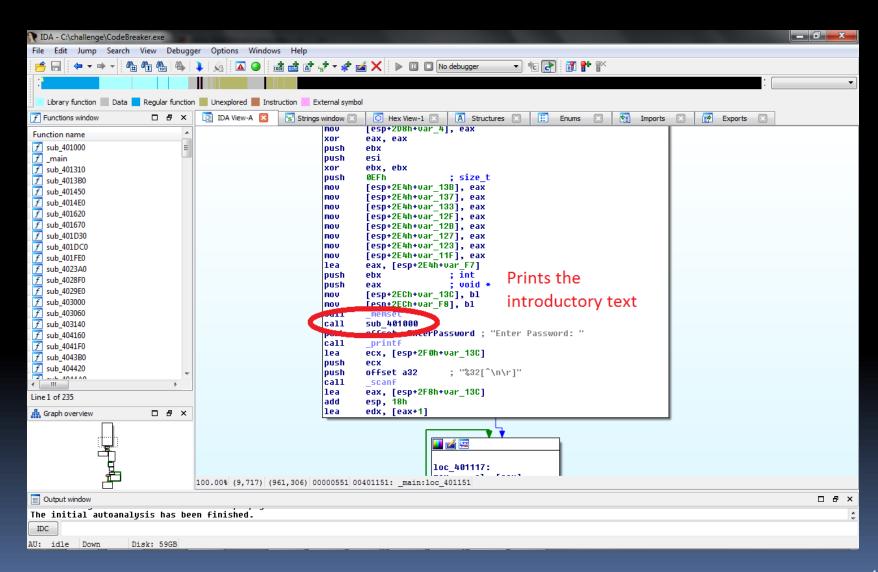
Explore Code Block (3)



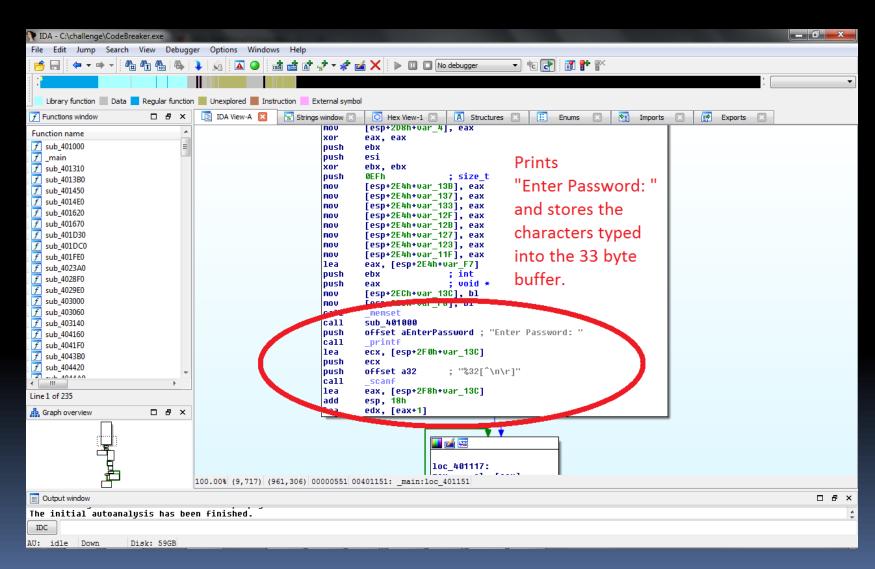
Explore Code Block (4)



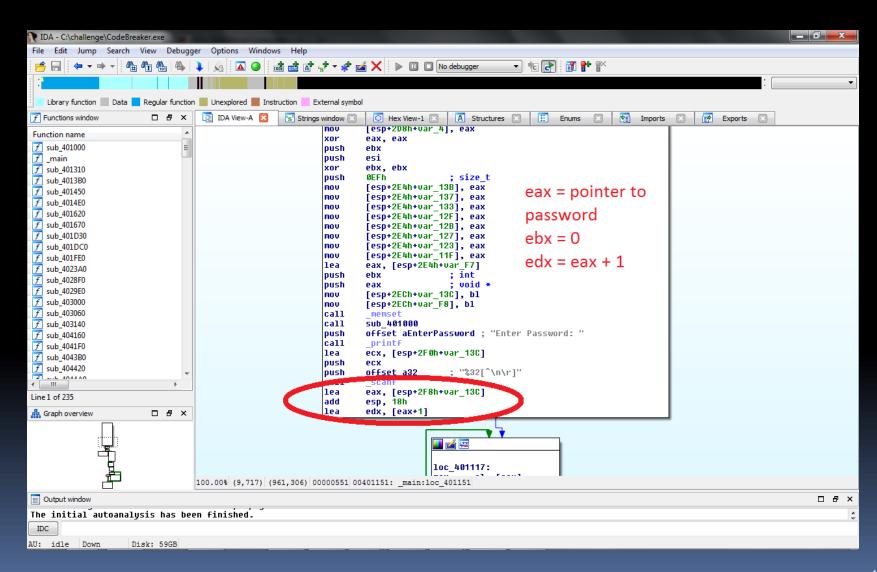
Explore Code Block (5)



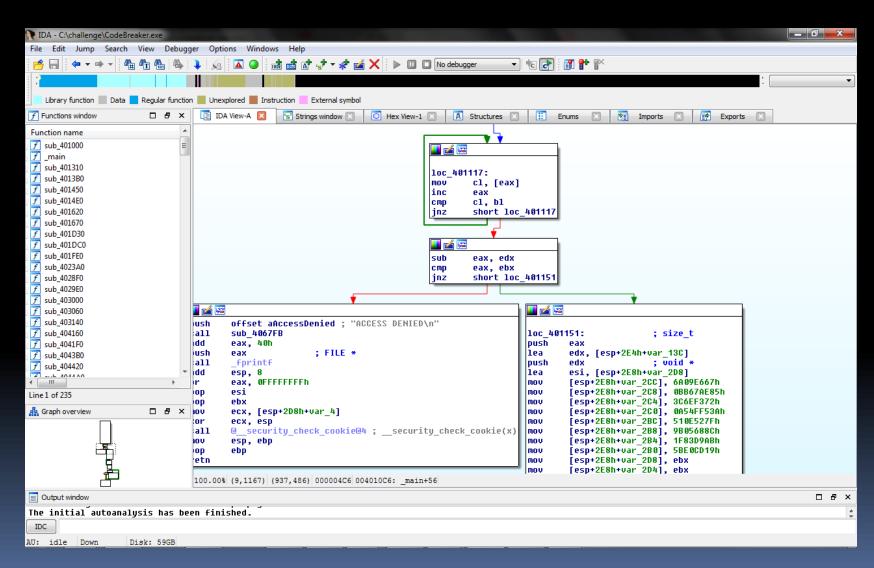
Explore Code Block (6)



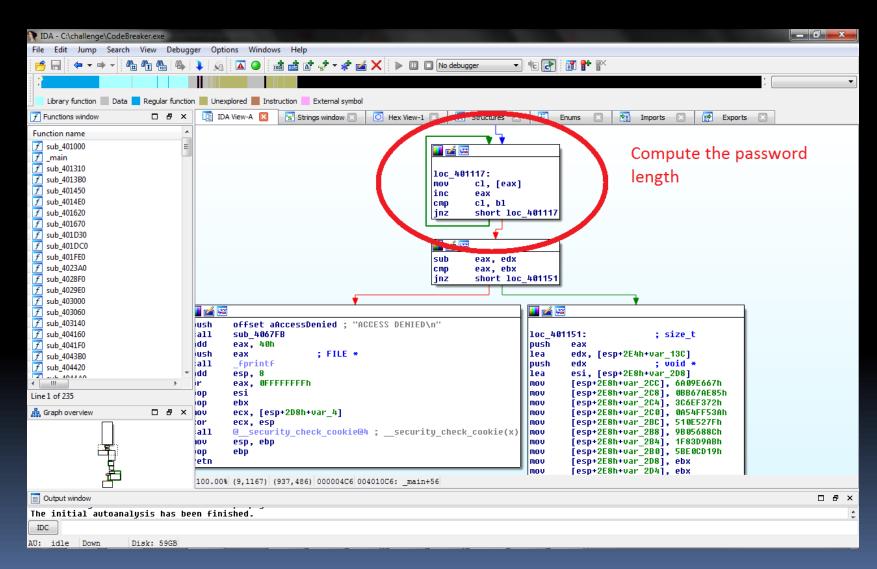
Explore Code Block (7)



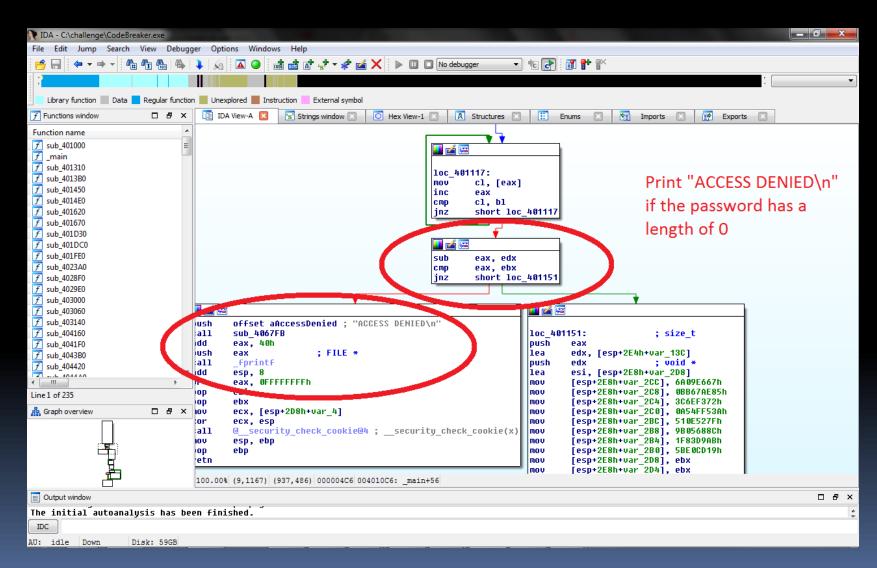
Explore Code Block (8)



Explore Code Block (9)



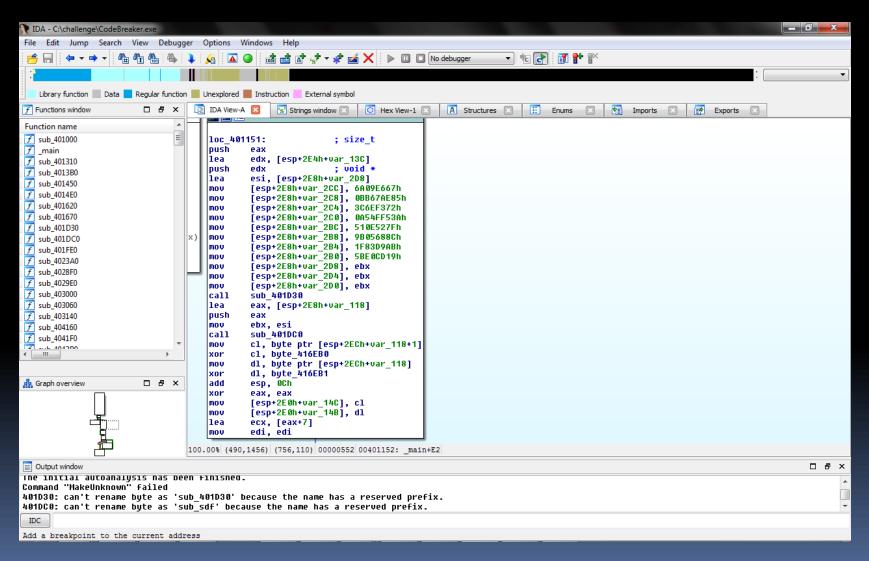
Explore Code Block (10)



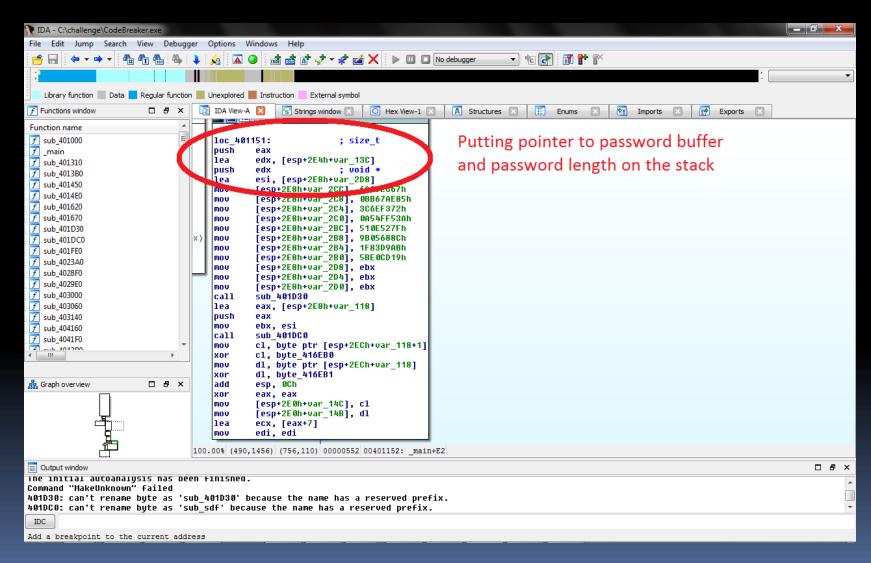
Secure Hash Algorithm

- Hash values can be used to verify the integrity of the data without providing any means to derive the original data. The algorithm is irreversible (SHA-2 is implemented in some widely used security applications and protocols: TLS, SSL, PGP, SSH, S/MIME, Bitcoin and IPsec)
- The first call computes the SHA256 hash of the password
 - You can figure out that it is a SHA256 by Googling for some of the large constants that you will find in the code (0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a, 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5beocd19)

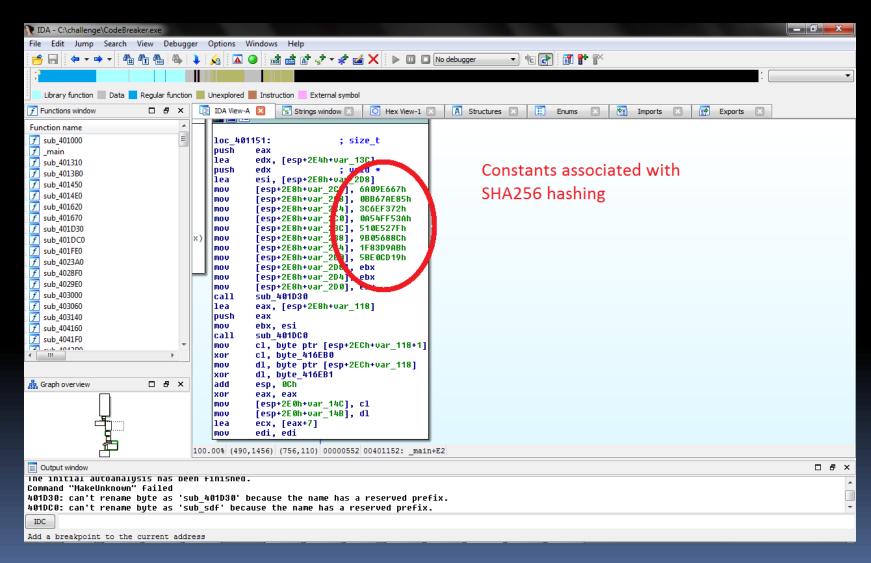
Secure Hash Algorithm (2)



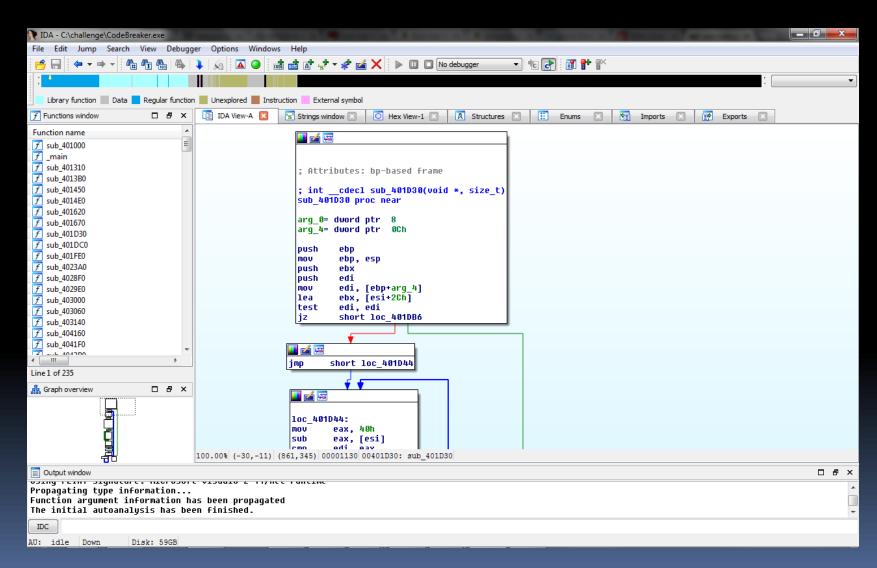
Secure Hash Algorithm (3)



Secure Hash Algorithm (4)



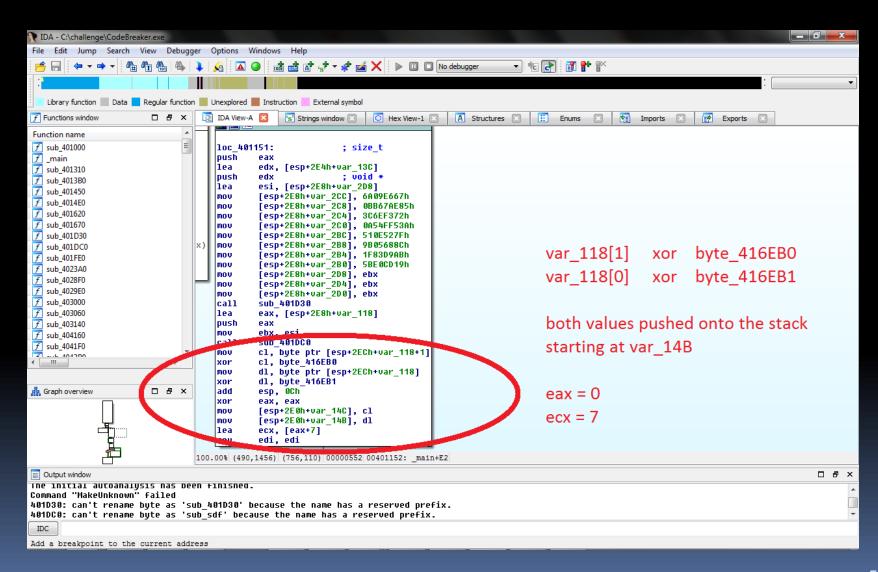
Secure Hash Algorithm (5)



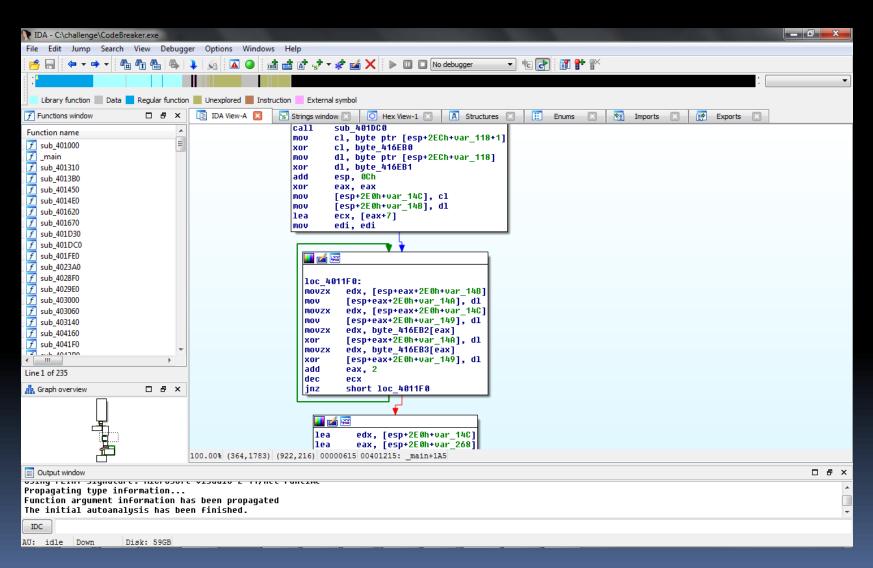
AES Key Derivation Routine

- AES is a symmetric-key algorithm; same key is used to encrypt and decrypt the data.
- The AES key is being derived from the hash of the password
- Lets take a closer look at what is being done to the password hash

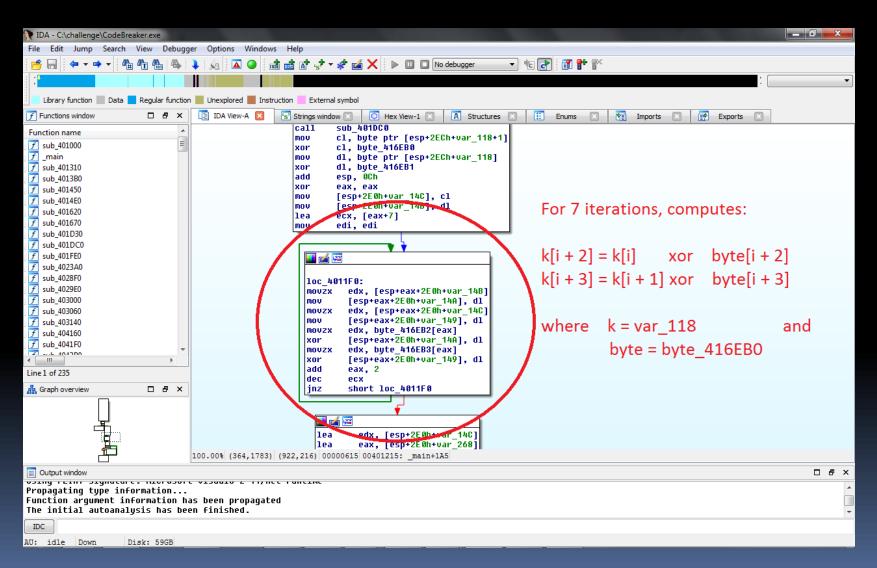
AES Key Derivation Routine (2)



AES Key Derivation Routine (3)



AES Key Derivation Routine (4)



AES Key Derivation

- The 16-byte key is derived as follows:
- Ho, H1, ..., H31 represents the 32-bytes of the SHA256 hash
- Ko, K1, ..., K15 are the 16-bytes of key
- Ro, R1, ..., R15 are the 16-bytes of fixed random data that is in the program

(0x23,0x8e,0x60,0xe1,0xd2,0x96,0x38,0xc7,0xa5,0xc0,0x22,0x74,0x4f,0x31,0x5b,0xcd)

AES Key Derivation

- Ko = H1 xor Ro
- K1 = Ho xor R1
- K2 = K1 xor R2
- $K_3 = K_0 \times R_3$
- $K_4 = K_3 \times R_4$
- $K_5 = K_2 \times R_5$
- •
- K₁₅ = K₁₂ xor R₁₅
- Looking at the above equations, do you notice any problems?
 - How many bytes of the password hash are ultimately used in the derivation of the key?
 - Given this, how many possible AES keys could be derived from this function?
 - Is that something that you can efficiently compute and brute-force?

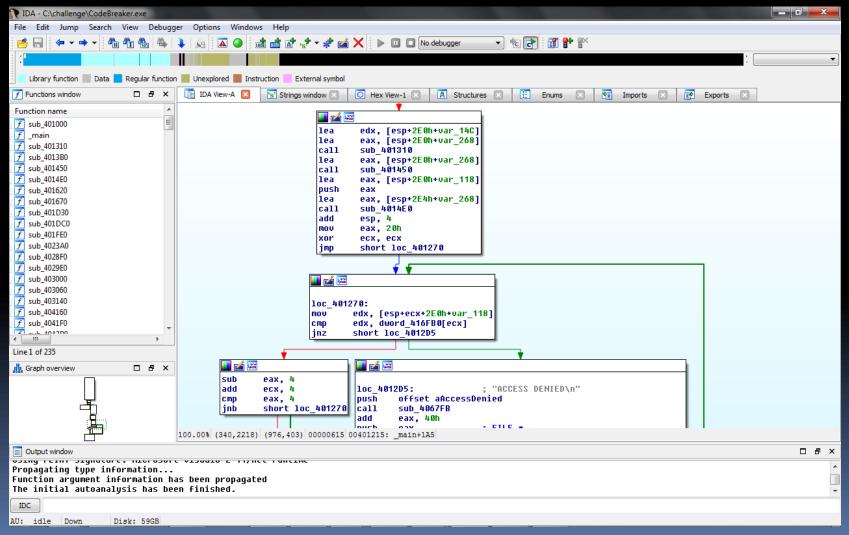
Keyed-Hash Message Authentication Code

- Following the code you will see that the AES key is passed to a function that computes the Keyed-Hash Message Authentication Code (HMAC) of the cipher text stored in the program.
- If the computed HMAC-SHA256 matches the expected HMAC

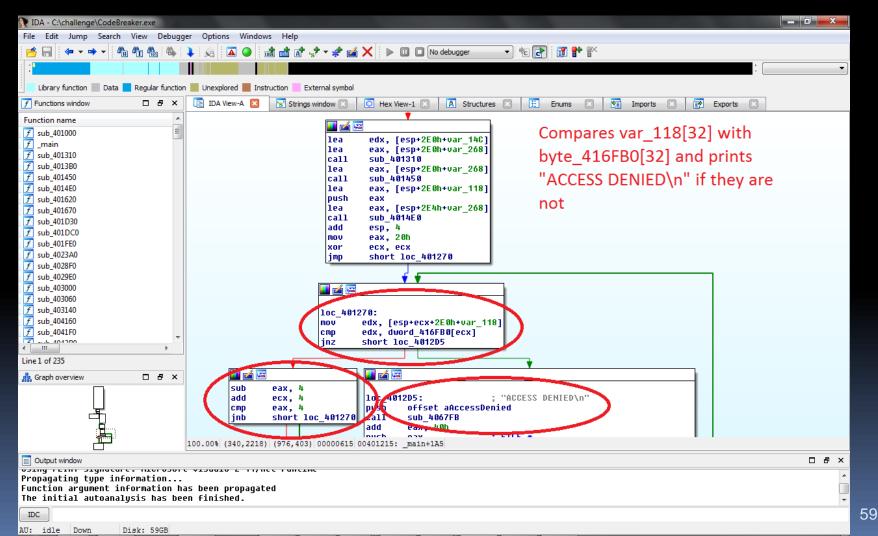
(oxbc,oxbo,ox78,ox41,oxaa,oxdo,oxo6,ox68,oxdd,oxa4,ox74,oxo6,oxd7,ox1a,ox2b,oxd2,ox9b,oxoe,oxdb,oxod,oxf9,oxce,oxf4,oxa7,oxf9,ox51,ox6c,ox99,oxfc,oxb6,ox95,oxf8)

- Then, the AES key is used to decrypt the cipher text
- Else, "ACCESS DENIED" message is displayed

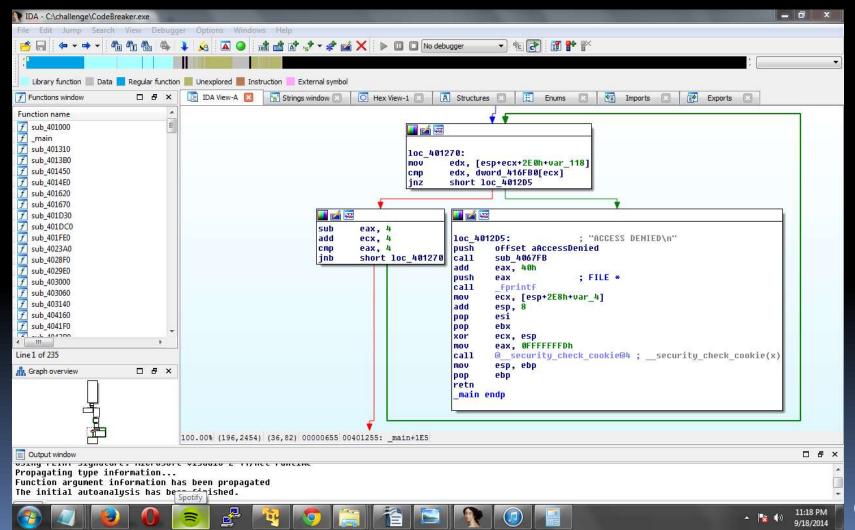
Keyed-Hash Message Authentication Code (2)



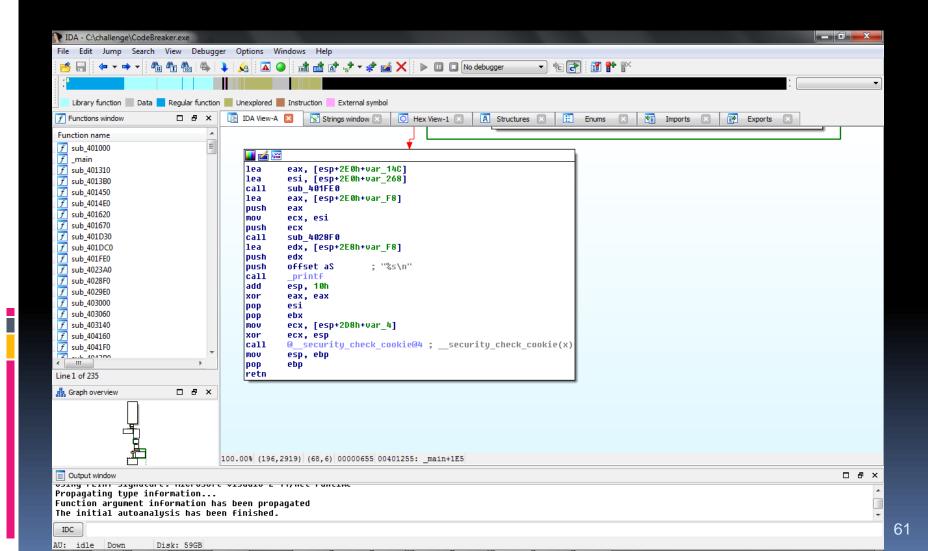
Keyed-Hash Message Authentication Code (3)



Keyed-Hash Message Authentication Code (4)



AES Decryption



Brute Force Attack

- Leverage the fact that you know the expected HMAC and you know how to derive putative key
 - (oxbc,oxbo,ox78,ox41,oxaa,oxdo,oxo6,ox68,oxdd,oxa4,ox74,oxo6,oxd7,ox1a,ox2b,oxd2,ox9b,oxoe,oxdb,oxod,oxf9,oxce,oxf4,oxa7,oxf9,ox51,ox6c,ox99,oxfc,oxb6,ox95,oxf8)
- Write a simple program to brute force the key and decrypt the ciphertext
 - The CyaSSL library is easy to use and contains all the functions that you need (it was used to write the challenge problem)

Brute Force Code

```
printf("Searching for a solution...\n");
for (i=0; i < (1 << 16); i++)
      memcpy(hashGuess, (uint8_t*)&i, 2);
      // Derive the AES key to decrypt secret code
      retCode = deriveAESKey(hashGuess, (uint8_t*)AESKey);
      // Compute HMAC of ciphertext
      computeHMAC(ciphertext, sizeof(ciphertext), AESKey, AES_KEY_SIZE, HMACdigest);
      // Compare computed HMAC with expected value
      if (memcmp(HMACdigest, HMAC_cipher, SHA256_DIGEST_SIZE) == 0)
               fprintf(stderr, "FOUND SOLUTION!\n");
               fprintf(stderr, "SHA256 Hash of Password -> Byte 1: ox%x\tByte 2: ox%x\n", hashGuess[0], hashGuess[1]);
               fprintf(stderr, "AES Key: ");
               printHexString(AESKey, AES_KEY_SIZE);
               printf("\n");
               // Decrypt the challenge ciphertext
               decrypt(AESKey, plain, sizeof(plain));
               printf("%s\n", plain);
```

Ciphertext

• uint8_t ciphertext[] = {ox34,ox3b,ox71,ox62,oxba,ox9f,ox55,oxb8,oxd9,ox88,ox71,ox98,ox24,ox41,ox3e,ox4d,oxa4,oxbd,ox1a,oxe1,ox2c,oxae,ox2c,oxff,ox1d,ox4e,ox1a,ox9e,ox94,ox8a,ox4d,ox07,ox71,ox5d,ox66,ox1f,oxef,ox91,ox09,ox67,oxda,oxfa,ox37,ox96,ox11,oxe1,ox67,oxd6,ox3e,ox31,ox5e,ox58,ox0b,ox81,oxdd,oxb2,oxaf,ox5d,oxde,oxde,ox9d,ox82,oxb3,ox72,ox36,ox86,oxa6,ox72,oxea,ox3e,ox5a,ox00,ox21,ox4e,ox94,oxbf,ox51,ox12,oxbe,oxfc,oxb6,ox07,ox3d,ox51,ox36,oxcf,ox76,ox93,oxab,oxc6,ox6c,ox7b,ox5f,oxc8,ox16,oxa2,ox11,oxc0,oxe6,ox87,ox9e,oxab,ox40,ox56,oxa4,oxb7,oxa5,ox20,ox44,oxbf,oxb0,oxb7,ox5b,ox43,ox4a,ox02,ox19,ox09,ox1d,oxb2,ox30,oxbb,ox15,oxce,ox1c,ox97,oxd8,ox77,oxbc,ox42,ox87,ox14,ox93,ox85,oxd2,ox0a,ox7d,oxc4,ox44,ox0e,ox82,ox35,ox3b,oxc4,ox40,ox78,ox7a,ox59,ox10,ox59,ox10,ox90,ox22,ox17,ox68,oxc0,oxfc,ox7a,ox5f,ox67,ox5b,ox2a,oxb3,oxfc,ox53,oxbc,oxe0,ox92,oxff,ox0d,ox84,ox74,ox31,ox1f,oxf5,ox16,ox4f,ox17,ox50,ox8d,ox95,ox51,ox66,oxf7,oxbc,oxda,ox15,ox05,ox76,oxb5,ox10,ox78,ox34,ox31,ox1f,ox45,oxf1,ox6e,ox78,ox2c,ox3a,ox01,ox4e,ox82,ox68,ox4f,oxe8,ox12,ox69,oxdd,ox00,ox77,ox17,oxec,ox95,ox76,oxbc,ox8c,ox43,oxc5,ox99,ox53,oxba,ox86,ox4c,ox6b,ox46,ox4a,ox35,ox82,oxe1,ox10,oxee,ox2a,ox73,ox4d,ox80,ox55,oxdc,oxbc};

Brute Force Solution

```
Searching for a solution...
FOUND SOLUTION!
SHA256 Hash of Password -> Byte 1: 0x12 Byte 2: 0x2
AES Key: 0x21 0x9c 0xfc 0xc0 0x12 0x6a 0x52 0xd5 0x70 0x92 0xb0 0x4 0x4b 0x81 0x
da 0x86

Congratulations! You have successfully decrypted the challenge cipher.

Please send an email (use your *.edu address) with a writeup of your solution an d the string bd461a21f932a4130a6f670d to the following address: Senior_Project@n sa.gov
```

Plaintext

- Congratulations! You have successfully decrypted the challenge cipher.
- Please send an email (use your *.edu address) with a writeup of your solution and the string bd461a21f932a413oa6f67od to the following address:

Questions

