
The background of the slide is a dark blue, abstract image. It features a perspective view of a tunnel or a path that recedes into the distance. The walls of the tunnel are composed of glowing binary code (0s and 1s) in various shades of blue and green. Light streaks and bokeh effects are visible, suggesting motion and depth. The overall aesthetic is futuristic and technological.

CWE Challenge - Hash

Michael Mendoza

2023-01-25

Contents

Introduction	2
Initialization	2
For Loop	3
If Else Statement	3
If Even	4
If Odd	5
Return Value	5
Final C Code	5
Flag	6
Conclusion	6
References	6

Introduction

Given the assembly code for this challenge, I was able to make some comparisons to the c code that was also provided. The following shows my methodology into solving this challenge.

Initialization

Since part of the initialization was given, it was easier to learn what some of the equivalent c code amounted to. Looking at the c code given,

```
1 short hash(unsigned char * s){
2     unsigned short h = 0;
3     //complete me
4     return h;
5 }
```

We can see that there was only one argument passed through the function call. Since it's the only argument, we know that only the rdi register will hold the data that needs to be manipulated.

```
endbr64
push    rbp
mov     rbp, rsp
mov     QWORD PTR [rbp-0x28], rdi    # move arg into rbp-0x28 (this is already given QWORD PTR = unsigned char *)
mov     WORD PTR [rbp-0x18], 0x0    # initialize h to 0 (WORD PTR = unsigned short)
mov     DWORD PTR [rbp-0x14], 0x0  # initialize another local variable. i?
mov     rax, QWORD PTR [rbp-0x28]  # move arg into rax
mov     QWORD PTR [rbp-0x10], rax  # move arg to rbp-0x10 (QWORD PTR = original arg from rdi assuming its also
                                   # an unsigned char * since its the same size)
jmp     11cb <hash+0x82>          # This is where the loop starts
```

Figure 1: Initializing the Variables

Since we know the rdi register is being used, we can see that the data from that register is being placed in QWORD PTR at rbp-0x28. So now we can note, QWORD PTR is the same as "unsigned char *". This is also seen right before the loop starts, at the variable located at rbp-0x10, which means another unsigned char pointer is initialized, and with the same data that was passed through the function call.

Furthermore, we can see two variables being initialized to 0; one of them is the "h" variable at rbp-0x18, and since it's placed in a WORD PTR, we can assume future WORD PTR's are the datatype "unsigned short".

The following is the code I came up with:

```
1 short hash(unsigned char * s){
2     unsigned short h = 0;
3     unsigned char *curr = s; //creating rbp-0x10
```

As for the other initialized variable `rbp-0x14`, I noticed that its incrementing after every iteration of the loop so I knew it was the “`int i = 0;`” typically used in a loop. I was able to see this after following the jump to “11cb”.

For Loop

```
add    DWORD PTR [rbp-0x14],0x1    # add 1 to i
add    QWORD PTR [rbp-0x10],0x1    # add 1 to arg original arg that was placed at rbp-0x10
                                         (this shifts the string to the right to get the next letter)
mov     rax,QWORD PTR [rbp-0x10]    # moves the argument back into rax
movzx   eax,BYTE PTR [rax]          # move extend to check the last byte of rax
test    al,al                       # check to see if its 0
jne     116c <hash+0x23>            # if its not 0 it'll continue the loop
```

Figure 2: For Loop

The jump takes the program to right after both the `rbp-0x14` and `rbp-0x10` variables are incremented. After incrementing, `rbp-0x10` was tested to see if it was 0 (or null), if its not, it'll jump to continue the loop. Since I already initialized `rbp-0x10`, the following for loop is what I came up with.

```
1  for (int i = 0; *curr != '\0'; i++, curr++){
2  }
```

If the value of “`curr`” is not 0, then the loop continues. The program jumps to 116c.

Note: As the value of “`curr`” is incremented, the last byte is whats being shifted to the next character. So every iteration is just analyzing the next character in the value past into the function.

If Else Statement

```
mov     rax,QWORD PTR [rbp-0x10]    # places arg into rax
mov     QWORD PTR [rbp-0x8],rax      # places arg into rbp-0x8 (this variable is used to
                                         initialize future argument variables)
mov     eax,DWORD PTR [rbp-0x14]    # moves i value into eax
and     eax,0x1                     # ANDs i with 1, this is equivalent to checking whether
                                         or not i is even or odd
test    eax,eax                     # ANDs the result to see if its 0 (even number) or 1 (odd number)
jne     11a0 <hash+0x57>            # jumps if not equal to 0 (if its odd), else continue (if its even)
```

Figure 3: If Else Statement

This is where the if else statements starts within the for loop. Here we can see that a new variable is created at `rbp-0x8`, where the value of “`curr`” is saved into; because of this, whenever `rbp-0x8` is used, I just use the value of “`curr`”. Then the “`i`” variable (`rbp-0x14`) is ANDed with 1. If the value is not 0, then the program will jump to “11a0”. This can be written in two ways, either using the AND operator, or the modulus operator as such:

<pre> If (i AND 1) { //do something odd } Else { //do something even } </pre>	or	<pre> if (i % 2 ==0) { //do something even } else { //do something odd } </pre>
---	----	---

Figure 4: If Else Example

If we don't jump, we can see what happens in the even statement.

If Even

<pre> mov rax,QWORD PTR [rbp-0x8] movzx eax,WORD PTR [rax] mov WORD PTR [rbp-0x16],ax movzx eax,WORD PTR [rbp-0x16] imul ax,ax,0x1906 mov WORD PTR [rbp-0x16],ax movzx eax,WORD PTR [rbp-0x16] xor WORD PTR [rbp-0x18],ax jmp 11c2 <hash+0x79> </pre>	<pre> # moves arg into rax # move extend the arg to get the last byte and places it into eax # moves that last byte into rbp-0x16 as a WORD which is the same as h variable (unsigned short) # move extend the byte and place it into eax # multiply 0x1906 to that last byte and place result into ax # move the result into rbp-0x16 # move extend the result into eax to get the last byte # xor the last byte with h and place result in h # jmp back to beginning of for loop to check if end is reached </pre>
---	--

Figure 5: Even Statement

Now we see rbp-0x8 being used to create another variable, rbp-0x16. Since we know rbp-0x8 is "curr", we can just cast "curr" to the datatype needed for rbp-0x16. Since a WORD PTR is used to create the variable, we know that it will be an "unsigned short" datatype because the "h" variable was also a WORD PTR. The imul operator is used which just multiplies the 2 source operands and places the product in the destination operand; that value is then placed into rbp-0x16. Lastly, this value is xor'd with the "h" variable before jumping back to the for loop.

```

1  if (i % 2 == 0)
2  {
3      unsigned short val = *(unsigned short *) curr; //creating rbp-0x16
4      val *= 0x1906;
5      h ^= val;
6  }

```

Next we can see the odd statement from the following:

If Odd

```

mov     rax,QWORD PTR [rbp-0x8]      # start of odd i variable, move the arg in rap-0x8 into rax
movzx   eax,BYTE PTR [rax]          # move extend to get the last byte in rax and place it in eax;
                                      # this time its a byte PTR which I assume its unsigned char *

mov     BYTE PTR [rbp-0x19],al       # move that last byte into rbp-0x19
movzx   edx,BYTE PTR [rbp-0x19]      # place that last byte into edx
mov     eax,edx                     # moves that byte into eax
shl     eax,0x3                     # shift the byte value left (value << 3)
add     eax,edx                     # add the shifted value to the value it was before it was shifted
                                      # and place in eax
add     eax,eax                     # add the value to itself and place back into eax
mov     BYTE PTR [rbp-0x19],al       # move the new value of al into rbp-0x19
movzx   eax,BYTE PTR [rbp-0x19]      # move the new value into eax
xor     WORD PTR [rbp-0x18],ax        # xor the the ax register with h and place the value in h

```

Figure 6: Odd Statement

The variable `rbp-0x8` is used to create `rbp-0x19`, which means again we are going to use the value of “curr” to create the new variable. Since `rbp-0x19` is a `BYTE PTR`, we are going to use an “unsigned char” datatype to create this variable. The new variable seems to be placed in two registers, `eax` and `edx`. The value in `eax` is shifted to the left by 3; this can be done in c using “<<”. Now the original value is still in `edx`, and that is added to the new value that was shifted in `eax`. Which can look like this “(val << 3) + val”. This value is then added to itself before being xor’d with the “h” variable and returned to the loop.

```

1  else
2  {
3      unsigned char val = *curr; //creating rbp-0x19
4      val = ((val << 3) + val) *2;
5      h ^= val;
6  }

```

Return Value

Lastly, the function call ends when the for loop hits null and breaks.

```

mov     rax,QWORD PTR [rbp-0x10]      # moves the argument back into rax
movzx   eax,BYTE PTR [rax]           # move extend to check the last byte of rax
test    al,al                        # check to see if its 0
jne     116c <hash+0x23>              # if its not 0 it'll continue the loop
movzx   eax,WORD PTR [rbp-0x18]      # move extend h and then return to main with this value in eax
pop     rbp
ret

```

Figure 7: Return Value

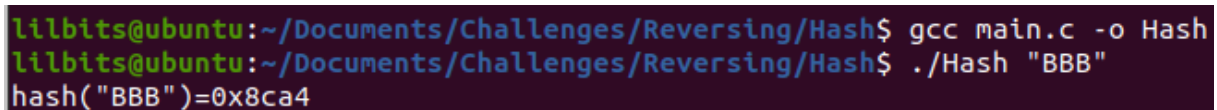
Final C Code

The final c code:

```
1 short hash(unsigned char * s)
2 {
3     unsigned short h = 0;
4     unsigned char *curr = s; //creating rbp-0x10
5     for (int i = 0; *curr != '\0'; i++, curr++)
6     {
7         if (i % 2 == 0)
8         {
9             unsigned short val = *(unsigned short *) curr; //rbp-0x16
10            val *= 0x1906;
11            h ^= val;
12        }
13        else
14        {
15            unsigned char val = *curr; //creating rbp-0x19
16            val = ((val << 3) + val) *2;
17            h ^= val;
18        }
19    }
20    return h;
21 }
```

After compiling the main.c file we created, we can get the flag.

Flag



```
lilbits@ubuntu:~/Documents/Challenges/Reversing/Hash$ gcc main.c -o Hash
lilbits@ubuntu:~/Documents/Challenges/Reversing/Hash$ ./Hash "BBB"
hash("BBB")=0x8ca4
```

Figure 8: Flag

Conclusion

This challenge was crazy. I learned so much about disassembly and how to do it manually. I know there was probably an easier option of turning the c code provided as well as the hash function into object files and linking them together to create the binary needed to get the flag, but learning how to reverse the assembly seemed more fun learn, especially since I wasn't on a time crunch to get this done.

References

1. <https://www.felixcloutier.com/x86/imul>

2. <https://www.felixcloutier.com/x86/movzx>
3. https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf
4. <https://stackoverflow.com/questions/23367624/intel-64-rsi-and-rdi-registers>