WriteUp reto Cryptor (Reversing) - Ekoparty 2020 Pre-ctf
Autor: twitter.com/_BrOoDkIlLeR_

Real crypto always win.

Attachment

Cryptor

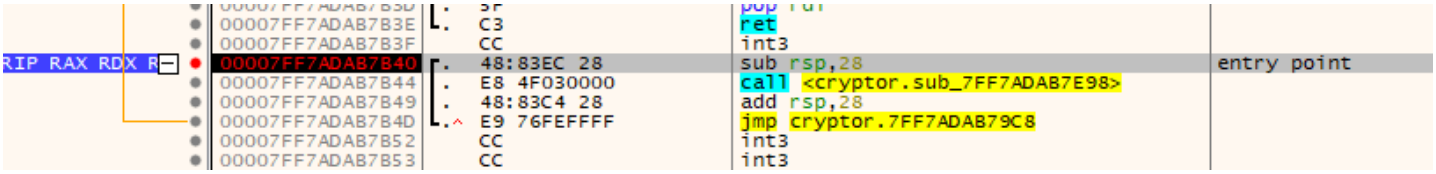
Nos descargamos el archivo, es un ejecutable 64bits de Windows. Ejecuto “strings cryptor.exe”

Muestra strings comunes salvo una q llama la atención: RutPE8RisVNfmXNfbM09X7i7NH9= (lo llamaremos “rutpe”). Parece un base64 pero al decodificar el resultado es ilegible (volvemos más tarde con esto)

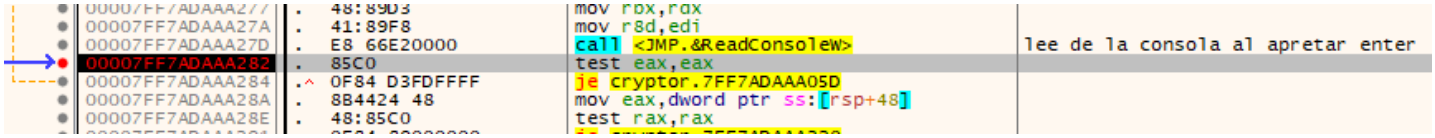
Utilizo x64dbg para analizar el ejecutable. **Utilizo el carácter ‘a’ como input (b64(a)=YQ)**

Revolviendo el proceso me doy cuenta que hay muchas funciones (alrededor de 1075) y pareciera que se encuentra desarrollado en RUST

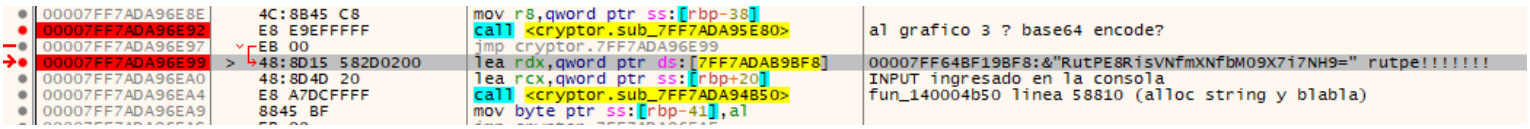
Establezco breakpoints en:



Entry point **(00007FF7ADAB7B40 - FIGURA 1)**

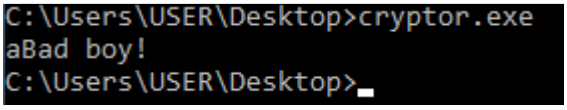


Luego de la llamada a call <JMP.&ReadConsoleW> (**00007FF7ADAAA282 - FIGURA 2**)



Busco strings references del modulo actual y establezco breakpoint (**00007FF7ADA96E99 - FIGURA 3**)

Si ingreso un caracter y presiono enter, pareciera que no hace nada. Al presionar ctrl+z dos veces el proceso se cierra y muestra el cartel Bad boy!



El string bad boy no se encuentra en el código binario, pero sí utilizando la herramienta strings2 http://split-code.com/strings2.html

Avanzando con F7 me encuentro con una función gigante (ver **00007FF7ADA96E92 - FIGURA 3**). Esta función realiza el base 64 de un string (y quizá más cosas)

Al retornar de esta función, en la **FIGURA 3** se realiza un lea rdx y lea rcx del string “rutpe” y del input ingresado en la consola. Estos dos valores son parámetros de la función que les sigue.

Al retornar de esta función se puede ver lo siguiente **(punto crítico del reto)**

FIGURA 4

Se realiza un test al,1 y continua la ejecución en 00007FF7ADA96EE4 (true) o 00007FF7ADA96EB7 (false, este es el valor por defecto)

Realizando un paralelo entre las funciones exportadas del ghidra (**provisto por @DBorgogno**) tenemos lo siguiente (en el mismo estado del proceso): los 4 últimos caracteres de los nombres de las funciones del ghidra corresponden con los últimos 4 caracteres del x64dbg, por ejemplo

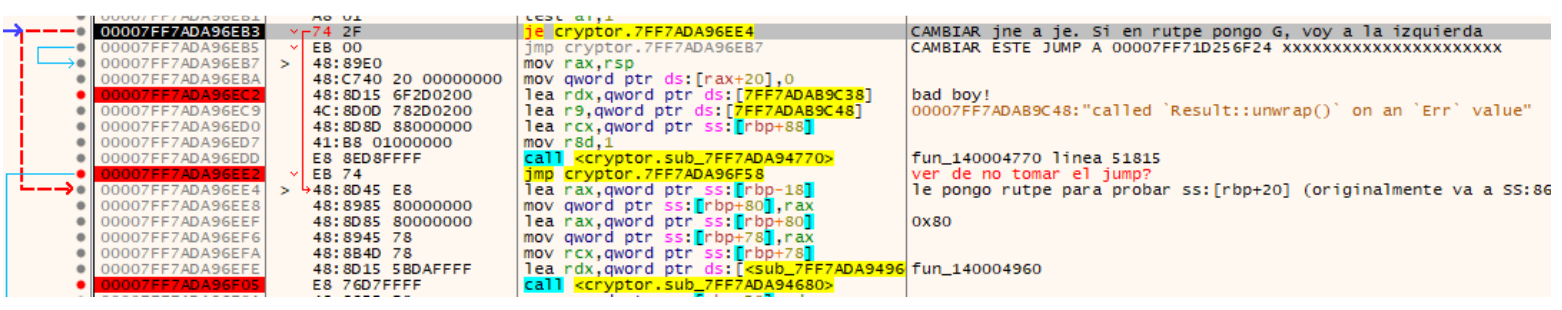
- Ghidra: fun_14000**4770**
- X64dbg: call <cryptor.sub_7FF7ADA9**4770**>

```
rbp2 = reinterpret_cast<struct s216*>(reinterpret_cast<int64_t>(__zero_stack_offset()) - 8 - 0x140 + 0x80);
rcx3 = reinterpret_cast<void**>(reinterpret_cast<int64_t>(rbp2) + 0xfffffffffffffff8);
fun_140004980(rcx3);
fun_14000f300(rcx3);
r8_4 = reinterpret_cast<void**>(reinterpret_cast<int64_t>(rbp2) + 0xfffffffffffffff8);
fun_14000f570(rbp2, reinterpret_cast<int64_t>(rbp2) + 24, r8_4, r9_5);
fun_140007d30(rbp2, 0x140029bc0, r8_4);// 0x140029bc0 es el INPUT que ingresamos en la consola
fun_1400050a0(reinterpret_cast<int64_t>(rbp2) + 24);
rax6 = fun_140004b10(reinterpret_cast<int64_t>(rbp2) + 0xfffffffffffffff8, 0x140029bc0, r8_4);
fun_140005e80(reinterpret_cast<int64_t>(rbp2) + 32, rax6, 0x140029bc0);
al7 = fun_140004b50(reinterpret_cast<int64_t>(rbp2) + 32, 0x140029bf8, 0x140029bc0);//0x140029bf8 =>rutpe. 0x140029bc0 esta
if (al7 & 1) {// great job
    fun_140004680(reinterpret_cast<int64_t>(rbp2) + 0x80, fun_140004960, 0x140029bc0);//cambio 0x140029bc0 x 0x140029bf8
    rdx8 = reinterpret_cast<void**>(0x140029c20);//ss:[rbp+38]
    r9_9 = reinterpret_cast<void**>(reinterpret_cast<int64_t>(rbp2) + 0x68);
    fun_140004770(reinterpret_cast<int64_t>(rbp2) + 56, 0x140029c20, 1, r9_9);
    fun_1400106d0(reinterpret_cast<int64_t>(rbp2) + 56, 0x140029c20, 1, r9_9);
} else {// bad boy
    rdx8 = reinterpret_cast<void**>(0x140029c38);
    r9_9 = reinterpret_cast<void**>("called `Result::unwrap()` on an `Err` value");
    fun_140004770(rbp2 + 8, 0x140029c38, 1, "called `Result::unwrap()` on an `Err` value");
    fun_1400106d0(rbp2 + 8, 0x140029c38, 1, "called `Result::unwrap()` on an `Err` value");
}
*rcx = 3;
fun_140005100(reinterpret_cast<int64_t>(rbp2) + 32, rdx8, 1, r9_9);
fun_140005100(reinterpret_cast<int64_t>(rbp2) + 0xfffffffffffffff8, rdx8, 1, r9_9);
return rcx;
```

Por lo tanto, la instrucción jne cryptor.7FF7ADA96EE4 corresponde con “if (al7 & 1)” (recuadro rojo).

Por defecto el “if” continuará por el else (y mostrará el cartel Bad Boy!), por lo tanto:

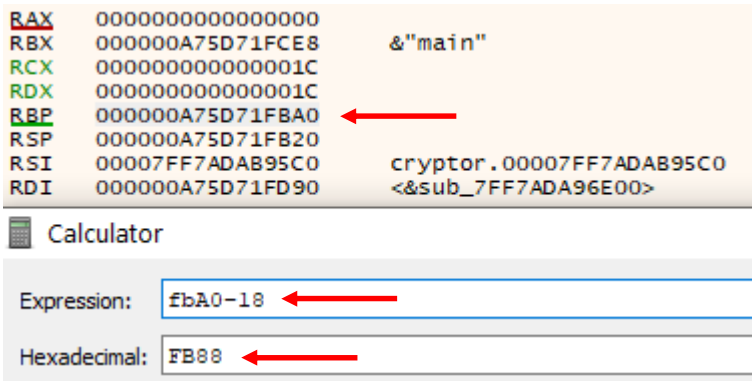
parcheo instrucción: `jne cryptor.7FF7ADA96EE4` a `je cryptor.7FF7ADA96EE4`

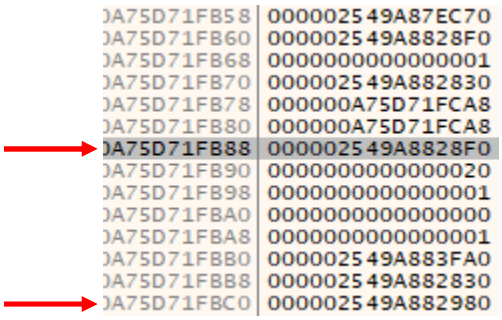


Se puede ver que el hilo de ejecución continuará ahora en 00007FF7ADA96EE4

Previo a la llamada de la función call <cryptor.sub_7FF7ADA94680> en 00007FF7ADA96F05, se ejecuta la instrucción **lea rax,qword ptr ss:[rbp-18]** **(punto crítico del reto)**

Leerá en rax el valor de **ss:[rbp-18]**. RBP tiene el valor 000000A75D71FBA0, si le resto 18 da como resultado 000000A75D71FB88, por lo tanto en rax se almacenará ss[000000A75D71FB88]





Registros del CPU **Stack**

El offset 000002549A8828F0 en el ds[000002549A8828F0] se muestra en la siguiente figura (click derecho en ss[000000A75D71FB88] -> Follow QWORD in Dump). Es el carácter 'a' que ingresamos

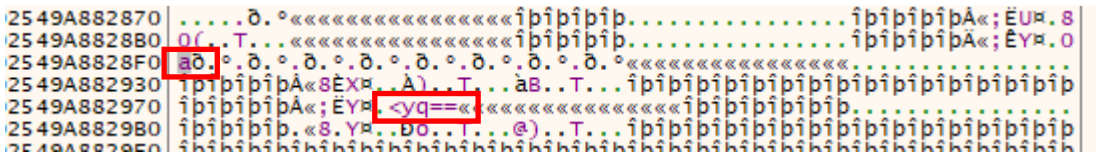
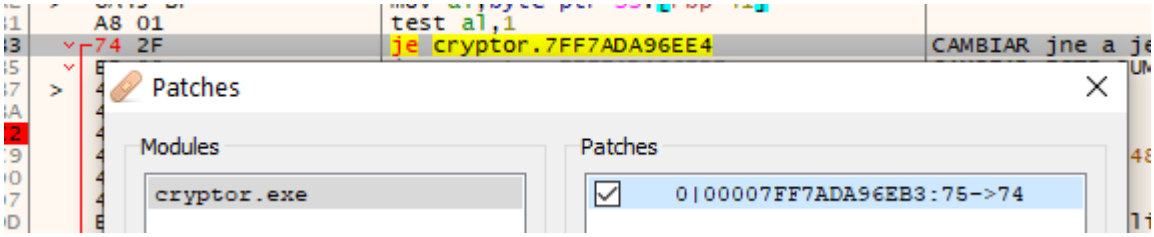


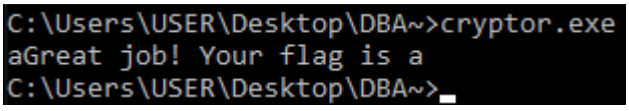
FIGURA 5

Ahora volvamos al stack y visualicemos el offset **ss:[rbp+20]** (000002549A8828F0) que está en ss[000000A75D71FBC0] (click derecho en ss[000000A75D71FBC0] -> Follow QWORD in Dump). Es el comienzo de yq== (el base 64 resultado de nuestro input).

Si parcheamos el ejecutable con el único cambio que realizamos hasta el momento, utilizando la "curita" del x64dbg

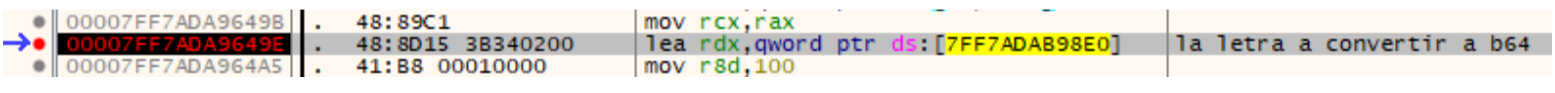


Ejecutamos cryptor.exe con 'a' como input y obtenemos lo siguiente:



Logramos modificar el flujo del proceso y por lo menos nos devuelve un cartel más esperanzador que Bad boy! y nuestro flag es el propio input. Obviamente, no es la respuesta final.

Ahora, volviendo a la **FIGURA 3**, se llama a la función cryptor.sub_7FF7ADA95E80 (base 64). En estos gráficos siguientes se puede ver que se convierte la letra ingresada 'a' por su equivalente en base64 (el primer carácter debería ser 'Y', pero retorna 'y' minúscula.



En el siguiente ciclo retorna ‘q’ también minúscula y concatena ‘==’.

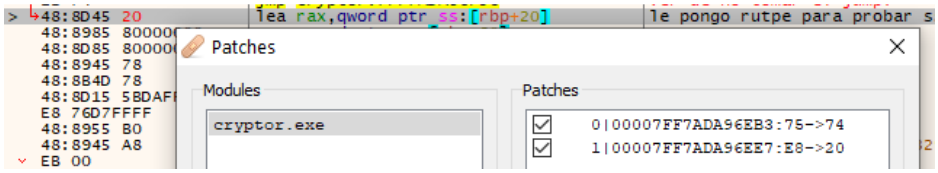
El motivo por el cual nuestro proceso realiza el base 64 (parecido al real) no lo sabemos todavía.

Conocemos que la flag tiene el formato EKO{}, y que EKO{ en base64 es RUtPew (parecido al comienzo del string que encontramos en el ejecutable, no?).

(punto crítico del reto)
Que pasa si el proceso, a propósito, modifica uno o más caracteres de nuestro base 64 y por eso el string “rutpe” al decodificarlo se rompe?

Si modificamos la instrucción en 00007FF7ADA96EE4 como sigue:

parcheo instrucción: `lea rax,qword ptr ss:[rbp-18]` a `lea rax,qword ptr ss:[rbp+20]`



Parcheamos el ejecutable, y corremos el proceso con input ‘a’, obtenemos:

```
C:\Users\USER\Desktop\DBA~>cryptor.exe
aGreat job! Your flag is yq==
C:\Users\USER\Desktop\DBA~>
```

Por lo tanto, nuestro nuevo ejecutable nos devuelve el base 64 “trucho” del input que le ingresemos.

Probamos ingresar una cadena más larga (12345678901234567890123456789O)

```
C:\Users\USER\Desktop\DBA~>cryptor.exe
12345678901234567890123456789OGreat job! Your flag is mTizNDu7NzG4mDeYmzq8NjC5oTaxmjm9NTy6oDKW
C:\Users\USER\Desktop\DBA~>
```

El base64 real de esta cadena es el siguiente (lo comparo con el “trucho”)

REAL -> MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkw
TRUCHO ->mTizNDu7NzG4mDeYmzq8NjC5oTaxmjm9NTy6oDKW

Los valores en rojo son los valores distintos al b64 real, y analizándolos, nos podemos dar cuenta que las letras son las mismas (solo que algunas se encuentran inverted case) y los números cambian, pero siempre cambian por los mismos (probamos otros strings hasta que nos devuelva todas las combinaciones)

REAL -> 1234567890
TRUCHO -> 8765432109

```
REAL ->   abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
TRUCHO -> AbCdEfGhIjKlMnOpQrStUvWxYzaBcDeFgHiJkLmNoPqRsTuVwXyZ
```

SOLUCION DEL RETO

En este momento se me ocurren TRES soluciones:

1. Buscar la forma de ingresar como input a "rutpe" y que nuestro proceso decodifique la misma. Habria que buscar la función que realice base64 decode y editarla para que funcione con los caracteres invertidos/cambiados (UN BARDO).
2. Parchear (en tiempo de ejecución?) las tablas del data segment de la función cryptor.sub_7FF7ADA95E80 para que los valores devueltos sean los reales y no los invertidos, y luego hacer base64 decode (tarea para el hogar)
3. Tomar "rutpe" y modificar manualmente los valores del string por los correspondientes invertidos y decodificar

Por supuesto que la tercer opción es la más directa.

```
TRUCHO -> RutPE8RisVNfmXNfbM09X7i7NH9=
 REAL ->  RUtPe1RISVNfMXNfbm90X2I2NH0
```

EKO{THIS_1s_not_b64}

(Of course, that was not fucking b64)