



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:

00007FF7ADAB7B39	CC	ret	
00007FF7ADAB7B3E	CC	int3	
00007FF7ADAB7B3F	CC	int3	
00007FF7ADAB7B40	48:83EC 28	sub rsp,28	entry point
00007FF7ADAB7B44	E8 4F030000	call <cryptor.sub_7FF7ADAB7E98>	
00007FF7ADAB7B49	48:83C4 28	add rsp,28	
00007FF7ADAB7B4D	E9 76FEFFFF	jmp cryptor.7FF7ADAB79C8	
00007FF7ADAB7B52	CC	int3	
00007FF7ADAB7B53	CC	int3	

Entry point (00007FF7ADAB7B40 - FIGURA 1)

00007FF7ADAAA277	48:89D3	mov rcx,rcx	
00007FF7ADAAA27A	41:89F8	mov r8d,edi	
00007FF7ADAAA27D	E8 66E20000	call <JMP.&ReadConsoleW>	lee de la consola al apretar enter
00007FF7ADAAA282	85C0	test eax,eax	
00007FF7ADAAA284	0F84 D3FDFFFF	je cryptor.7FF7ADAAA05D	
00007FF7ADAAA28A	8B4424 48	mov eax,dword ptr ss:[rsp+48]	
00007FF7ADAAA28E	48:85C0	test rax,rax	

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

00007FF7ADA96E8E	4C:8B45 C8	mov r8,qword ptr ss:[rbp-38]	
00007FF7ADA96E92	E8 E9FFFFFF	call <cryptor.sub_7FF7ADA95E80>	al grafico 3 ? base64 encode?
00007FF7ADA96E97	EB 00	jmp cryptor.7FF7ADA96E99	
00007FF7ADA96E99	48:8D15 582D0200	lea rdx,qword ptr ds:[7FF7ADA98F8]	00007FF648F19BF8:&"RutPE8RisVNfmXNfbM09X7i7NH9=" rutpe!!!!!!
00007FF7ADA96EA0	48:8D4D 20	lea rcx,qword ptr ss:[rbp+20]	INPUT ingresado en la consola
00007FF7ADA96EA4	E8 A7DCFFFF	call <cryptor.sub_7FF7ADA94850>	fun_140004b50 linea 58810 (alloc string y blabla)
00007FF7ADA96EA9	8B45 BF	mov byte ptr ss:[rbp-41],al	

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!

```
C:\Users\USER\Desktop>cryptor.exe
aBad boy!
C:\Users\USER\Desktop>
```

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)

00007FF7ADA96EAE	> 8A45 BF	mov al,byte ptr ss:[rbp-41]	
00007FF7ADA96EB1	A8 01	test al,1	
00007FF7ADA96EB3	> 75 2F	jne cryptor.7FF7ADA96EE4	CAMBIAR jne a je. Si en rutpe pongo G, voy a la izquierda
00007FF7ADA96EB5	EB 00	jmp cryptor.7FF7ADA96EB7	CAMBIAR ESTE JUMP A 00007FF7D256F24 xxxxxxxxxxxxxxxxxxxxxxxx
00007FF7ADA96EB7	> 48:89E0	mov rax,rsi	
00007FF7ADA96EBA	> 48:C740 20 00000000	mov qword ptr ds:[rax+20],0	
00007FF7ADA96EC2	> 48:8D15 6F2D0200	lea rdx,qword ptr ds:[7FF7ADA89C38]	bad boy!
00007FF7ADA96EC9	> 4C:8D0D 782D0200	lea r9,qword ptr ds:[7FF7ADA89C48]	00007FF7ADA89C48:"called `Result::unwrap()` on an `Err` value"
00007FF7ADA96ED0	> 48:8D8D 88000000	lea rcx,qword ptr ss:[rbp+88]	
00007FF7ADA96ED7	> 41:B8 01000000	mov r8d,1	
00007FF7ADA96EDD	E8 8ED8FFFF	call <cryptor.sub_7FF7ADA94770>	fun_140004770 línea 51815
00007FF7ADA96EE2	EB 74	jmp cryptor.7FF7ADA96F58	ver de no tomar el jump?
00007FF7ADA96EE4	> 48:8D45 E8	lea rax,qword ptr ss:[rbp-18]	le pongo rutpe para probar ss:[rbp+20] (originalmente va a SS:868, el input)
00007FF7ADA96EE8	> 48:8985 80000000	mov qword ptr ss:[rbp+80],rax	
00007FF7ADA96EEF	> 48:8D85 80000000	lea rax,qword ptr ss:[rbp+80]	0x80
00007FF7ADA96EF6	> 48:8945 78	mov qword ptr ss:[rbp+78],rax	
00007FF7ADA96EFA	> 48:884D 78	mov rcx,qword ptr ss:[rbp+78]	
00007FF7ADA96EFE	> 48:8D15 58DAFFFF	lea rdx,qword ptr ds:[<sub_7FF7ADA9496>]	fun_140004960
00007FF7ADA96F05	E8 76D7FFFF	call <cryptor.sub_7FF7ADA94680>	
00007FF7ADA96F0A	> 48:8955 B0	mov qword ptr ss:[rbp-50],rdx	
00007FF7ADA96F0E	> 48:8945 A8	mov qword ptr ss:[rbp-58],rax	[rbp-58]:sub_7FF7ADA94680+329
00007FF7ADA96F12	EB 00	jmp cryptor.7FF7ADA96F14	[rbp-58]:sub_7FF7ADA94680+329
00007FF7ADA96F14	> 48:8845 A8	mov rax,qword ptr ss:[rbp-58]	
00007FF7ADA96F18	> 48:8945 68	mov qword ptr ss:[rbp+68],rax	
00007FF7ADA96F1C	> 48:884D B0	mov rcx,qword ptr ss:[rbp-50]	
00007FF7ADA96F20	> 48:894D 70	mov qword ptr ss:[rbp+70],rcx	
00007FF7ADA96F24	> 48:8D42 20 01000000	mov rdx,rsi	xxxxxxxxxxxxxxxxxxxxxx
00007FF7ADA96F27	> 48:C742 20 01000000	mov qword ptr ds:[rdx+20],1	
00007FF7ADA96F2F	> 48:8D15 EA2C0200	lea rdx,qword ptr ds:[7FF7ADA89C20]	aca hace referencia al dump despues de Your flag is .
00007FF7ADA96F36	> 48:8D4D 38	lea rcx,qword ptr ss:[rbp+38]	
00007FF7ADA96F3A	> 41:B8 01000000	mov r8d,1	
00007FF7ADA96F40	> 4C:8D4D 68	lea r9,qword ptr ss:[rbp+68]	
00007FF7ADA96F48	E8 27D8FFFF	call <cryptor.sub_7FF7ADA94770>	

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_140004770
- X64dbg: call <cryptor.sub_7FF7ADA94770>

```
rbp2 = reinterpret_cast<struct s216*>(reinterpret_cast<int64_t>(_zero_stack_offset()) - 8 - 0x140 + 0x80);
rcx3 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(rbp2) + 0xffffffffffffffe8);
fun_140004980(rcx3);
fun_14000f300(rcx3);
r8_4 = reinterpret_cast<void*>(reinterpret_cast<int64_t>(rbp2) + 0xffffffffffffffe8);
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) + 0xffffffffffffffe8, 0x140029bc0, r8_4);
fun_140005e80(reinterpret_cast<int64_t>(rbp2) + 32, rax6, 0x140029bc0);
a17 = fun_140004b50(reinterpret_cast<int64_t>(rbp2) + 32, 0x140029bf8, 0x140029bc0); // 0x140029bf8 => rutpe. 0x140029bc0 esta
if (a17 & 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) + 0xffffffffffffffe8, rdx8, 1, r9_9);
return rcx;
```

Por lo tanto, la instrucción jne cryptor.7FF7ADA96EE4 corresponde con “if (a17 & 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

00007FF7ADA96EB3	> 75 2F	je cryptor.7FF7ADA96EE4	CAMBIAR jne a je. Si en rutpe pongo G, voy a la izquierda
00007FF7ADA96EB5	EB 00	jmp cryptor.7FF7ADA96EB7	CAMBIAR ESTE JUMP A 00007FF7D256F24 xxxxxxxxxxxxxxxxxxxxxxxx
00007FF7ADA96EB7	> 48:89E0	mov rax,rsi	
00007FF7ADA96EBA	> 48:C740 20 00000000	mov qword ptr ds:[rax+20],0	
00007FF7ADA96EC2	> 48:8D15 6F2D0200	lea rdx,qword ptr ds:[7FF7ADA89C38]	bad boy!
00007FF7ADA96EC9	> 4C:8D0D 782D0200	lea r9,qword ptr ds:[7FF7ADA89C48]	00007FF7ADA89C48:"called `Result::unwrap()` on an `Err` value"
00007FF7ADA96ED0	> 48:8D8D 88000000	lea rcx,qword ptr ss:[rbp+88]	
00007FF7ADA96ED7	> 41:B8 01000000	mov r8d,1	
00007FF7ADA96EDD	E8 8ED8FFFF	call <cryptor.sub_7FF7ADA94770>	fun_140004770 línea 51815
00007FF7ADA96EE2	EB 74	jmp cryptor.7FF7ADA96F58	ver de no tomar el jump?
00007FF7ADA96EE4	> 48:8D45 E8	lea rax,qword ptr ss:[rbp-18]	le pongo rutpe para probar ss:[rbp+20] (originalmente va a SS:868, el input)
00007FF7ADA96EE8	> 48:8985 80000000	mov qword ptr ss:[rbp+80],rax	
00007FF7ADA96EEF	> 48:8D85 80000000	lea rax,qword ptr ss:[rbp+80]	0x80
00007FF7ADA96EF6	> 48:8945 78	mov qword ptr ss:[rbp+78],rax	
00007FF7ADA96EFA	> 48:884D 78	mov rcx,qword ptr ss:[rbp+78]	
00007FF7ADA96EFE	> 48:8D15 58DAFFFF	lea rdx,qword ptr ds:[<sub_7FF7ADA9496>]	fun_140004960
00007FF7ADA96F05	E8 76D7FFFF	call <cryptor.sub_7FF7ADA94680>	

00007FF7ADA9649B	48:89C1	mov rcx,rcx	
00007FF7ADA9649E	48:8D15 3B340200	lea rdx,qword ptr ds:[7FF7ADAB98E0]	1a letra a convertir a b64
00007FF7ADA964A5	41:88 00010000	mov r8d,100	

00007FF7ADAB98605.....«.÷.....*.....«.÷.....N.....«.÷.....
00007FF7ADAB98A0C.....«.÷.....2.....«.÷.....
00007FF7ADAB98E0	ä.....B.....c.....D.....e.....F.....g.....H.....i.....J.....k.....L.....m.....N.....o.....P.....
00007FF7ADAB9920	q.....R.....s.....T.....u.....v.....w.....X.....y.....z.....A.....b.....c.....d.....E.....f.....
00007FF7ADAB9960	G.....h.....I.....j.....K.....l.....M.....n.....O.....p.....Q.....r.....S.....t.....U.....v.....
00007FF7ADAB99A0	W.....x.....Y.....z.....9.....8.....7.....6.....5.....4.....3.....2.....1.....0.....+.....-.....
00007FF7ADAB99E0«.÷.....;..........«.÷.....;..........«.÷.....;.....

```

-> 00007FF7ADA964D2 |> 48:8845 20 | mov rax,qword ptr ss:[rbp+20]
-> 00007FF7ADA964D6 | 8B8C85 A0050000 | mov ecx,dword ptr ss:[rbp+rax*4+5A0] | esta convirtiendo a b64, a=YQ (selecciona y)
-> 00007FF7ADA964DD | 48:8D95 A0060000 | lea rdx,qword ptr ss:[rbp+6A0]

```

00A75D71F530
00A75D71F570).....T.....a.....B.....c.....D.....e.....F.....g.....H.....
00A75D71F5B0	i.....J.....k.....L.....m.....N.....o.....P.....q.....R.....s.....T.....u.....v.....w.....X.....
00A75D71F5F0	y.....z.....A.....b.....c.....d.....E.....f.....g.....h.....I.....j.....k.....l.....M.....n.....
00A75D71F630	O.....p.....Q.....r.....S.....t.....U.....v.....w.....x.....Y.....z.....9.....8.....7.....6.....
00A75D71F670	5.....4.....3.....2.....1.....0.....+.....-.....
00A75D71F6B0

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.

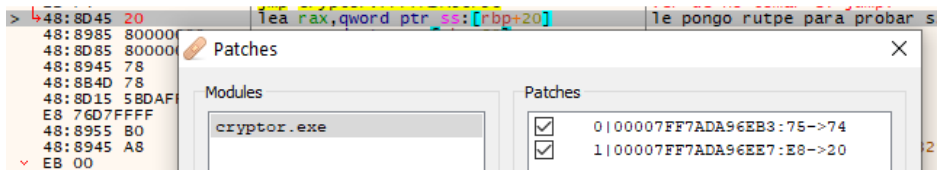
Conocemos que la flag tiene el formato EKO{}, y que EKO{ en base64 es RUtPew (parecido al comienzo del string que encontramos en el ejecutable, no?).

(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 (123456789012345678901234567890)

```

C:\Users\USER\Desktop\DBA~>cryptor.exe
123456789012345678901234567890Great job! Your flag is mTizNDu7NzG4mDeYmq8NjC5oTaxmjm9NTy6oDKW
C:\Users\USER\Desktop\DBA~>

```

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

```

REAL -> MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM0NTY3ODkw
TRUCHO -> mTizNDu7NzG4mDeYmq8NjC5oTaxmjm9NTy6oDKW

```

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. Habría 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 -> RuTpElRISVNfMxNfbm90X2i2NH0
```

EKO{THIS_1s_not_b64}

(Of course, that was not fucking b64)