UAM Futurama 3 parte 2 partial writeup: ROP contra servidor web sin LEAKS

Introducción

Primero de todo, este no es un write-up completo. Describiré mi exploit para lograr la flag en la segunda parte del reto de Futurama 3 de este mes de la UAM. Se considera el reversing del binario “carl” ya completado y las respectivas vulnerabilidades encontradas para provocar el **Buffer Overflow**.

El servidor web nos muestra una caja de texto donde podemos introducir el parámetro que recogerá el binario vulnerable carl, y su salida estándar se nos mostrará en el textarea de debajo. Siendo un reto llamado “SSRF”, parecería lógico llegar a imprimir la flag en el textarea tras explotar correctamente el BOF encontrado durante la fase de reversing.

No leaks

Este reto se podía resolver de una manera relativamente habitual en los CTFs de exploiting de BOF: primero, leak. Luego, llamada a system. Sin embargo, y parcialmente inspirado por el reto de c0rn0n4con “Twicat”, me puse como objetivo lograr imprimir la flag con una sola ejecución del payload, sin leaks. Reverseando el binario nos encontrábamos claramente con las llamadas dlsym, dlopen y la cadena “system”. Así pues, llegar a ejecutar system(); sin ningún leak era totalmente factible. Además, el binario estaba repleto de llamadas a funciones que son ideales como primitivas write-what-where (sprintf, strcpy, strncpy, strcat…), por lo que todo apuntaba a que debía ser factible lograr mi objetivo. Como principal problema a superar debía considerar 2 cosas:

Dirección donde escribir el comando a ejecutar

Siendo un binario no PIE, las direcciones siempre son las mismas. A simple vista, la dirección **0x604150** parecía una muy buena candidata. De hecho, en el offset **0x00400f6c** de carl veíamos el siguiente código:

```
[0x00400f6c]> pd 3
            0x00400f6c      bf50416000      mov edi, 0x604150          ; 'PA`'
            0x00400f71      ffe0            jmp rax
            0x00400f73      0f1f440000      nop dword [rax + rax]
```

Snippet de codigo que mueve a RDI el valor 0x604150 y luego hace un jump a rax.

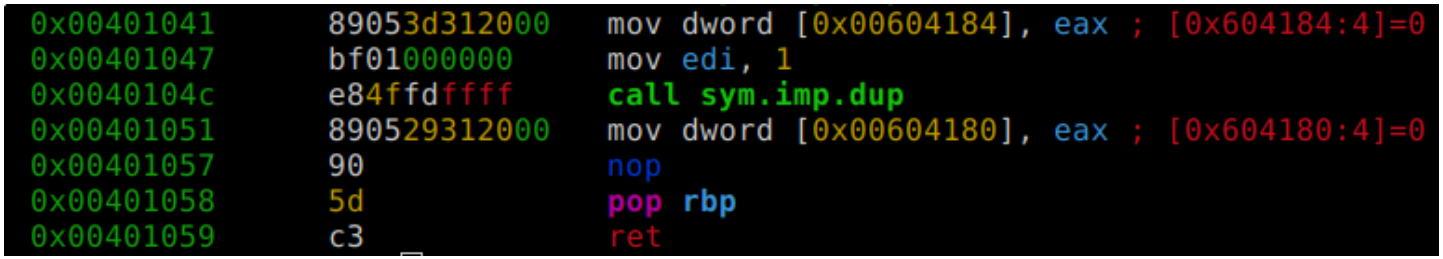
Este código era ideal para cargar la dirección 0x604150 sobre RDI (único argumento de system) y, teniendo en RAX la dirección de system (gracias a dlsym), ejecutar system con el comando deseado. Sin embargo, tras el salto a RAX (ejecución de system), carl ejecutaría correctamente el comando pero acabaría con un segfault. El servidor web nos devolvería un "Internal Server Error" sin mostrarnos la salida estándar de carl en el textarea, y como mi intención era mostrar la flag directamente, esta vía estaba descartada.

Dirección del comando a ejecutar y primitiva write-what-where

Para lograr escribir el comando deseado sobre una dirección de mi elección dentro de una sección rw- del binario, busqué primero ROP gadgets del estilo mov [registro], registro. Este tipo de gadget me permitiría hacer cosas del estilo:

```
# mov [ r14 ], r15
payload += p64(_pop_r14_r15_ret)
payload += p64(0x604150)
payload += "ls l/; \x00"
payload += p64(_write_mem_ret)
```

Sin embargo, no encontré ninguno que me sirviera. Así que descartada esta opción, seguí buscando, y encontré esto en la función fcn.00400ff7:



Una primitiva write-what-where sobre 0x604180 (nibble) + 0x60184 (otro nibble).

Esto me permitía escribir hasta 8 bytes en dos pasadas, aunque debía luego compensar RBP en el stack. Llegó a funcionar aunque entonces me quedé sin espacio para mis otros ROP gadgets. En aquel momento no me percaté de que dlopen no era necesaria en absoluto, y que llamando a dlsym directamente pasandole un NULL en rax como handle (gracias @GNLZ) también retornaba correctamente la dirección de system sobre RAX. Con esto, me habría ahorrado los 3 bytes de la llamada a dlopen y hubiera tenido suficientes gadgets para lograr el objetivo sobreescribiendo mi comando en 0x604180. Y digo 3 y no 4 bytes porqué yo sí me percaté que dlopen sigue funcionando a pesar de tener un valor indeterminado en RSI (flags) ;-).

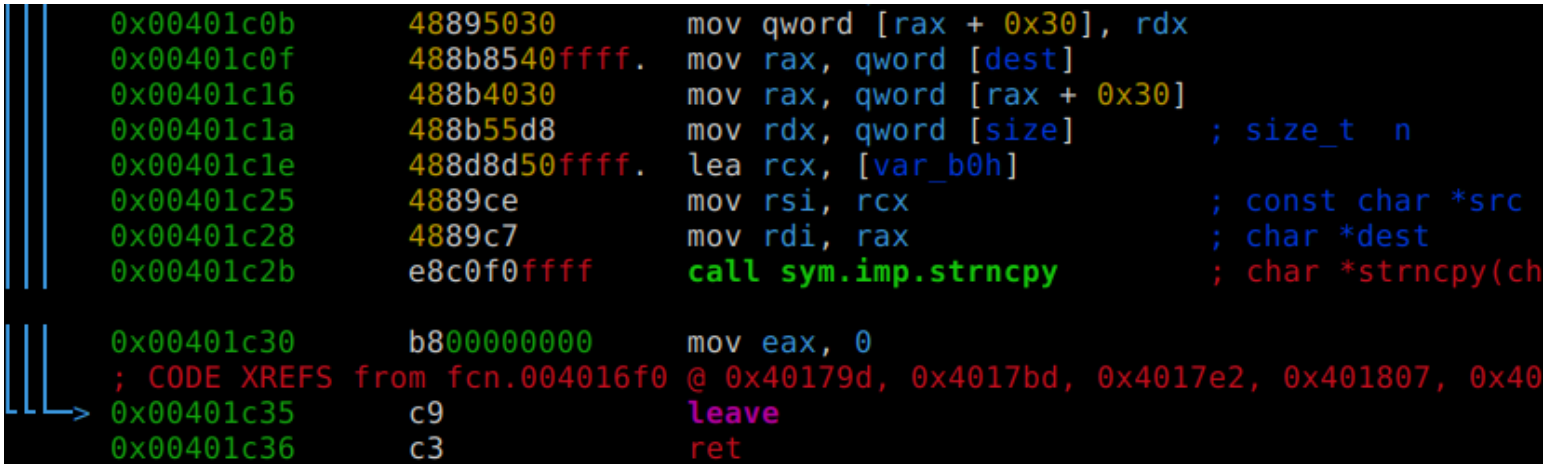
Así que sólo me quedaba introducir el comando a ejecutar en la pila del programa y buscar la manera de referenciar dicha dirección dinámica (recordemos, ASLR) usando ROP. Lo primero era fácil, aprovechar el “padding”. Rellené mi padding con algunos patterns básicos “abcdef…ABCDEF…0123..” hasta EIP, y lancé mi exploit dentro de gdb, parando la ejecución de mi payload en el primer gadget y observé los offsets respecto de mi padding y el valor de RSP. Como es lógico, los OFFSET no varían. Por supuesto, por la manera en que se apilan los datos en el stack, era evidente que mi padding estaría cerca de RSP:

Offset respecto del valor de RSP del padding.

Descubrí que podía referenciar mi padding con algunas instrucciones presentes en el propio binario del estilo **[rbp – OFFSET].** Pero para ello, claro, debía inicializar RBP al valor de RSP. Encontré un gadget ideal para esto:

mov rbp, rsp; mov rdi, rsi; ret;

Lo siguiente era encontrar o bien un ROP, o bien código legítimo del binario, que me permitiera ejecutar un strcpy, strncpy o similar, rellenando los valores adecuados sobre cada uno de los registros mediante referencias del estilo: mov registro [rbp – OFFSET], teniendo en cuenta que OFFSET debía cumplir el requisito de “caer” dentro de mi padding. Lancé la búsqueda de gadgets (sin éxito), así que busqué en el propio código del binario. Necesitaba que, tras la llamada legítima a la función de escritura de memoria, la función regresara (con LEAVE o RET) para no romper mi ROP chain que venía luego. Me encontré con esta joya:

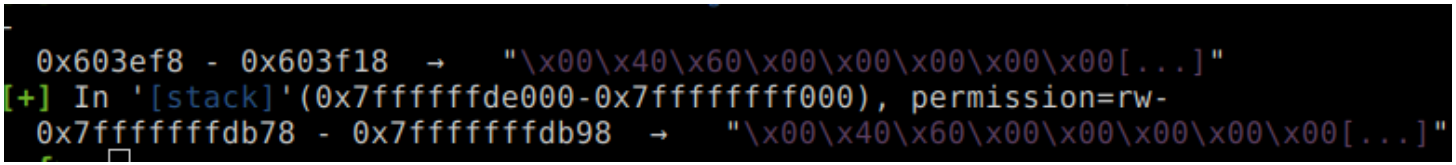


Primitiva write-what-where con referencias al stack y OFFSETS dentro de mi padding.

Todos los OFFSET presentes caían dentro de mi padding. Podía simplemente escribir valores en mi padding y estos de cargarían sobre los registros **RAX, RDX y RCX**. RDX tendría la longitud del comando. Bastaría con escribir en el OFFSET correcto de mi padding la longitud del comando como p64(len(cmd)). El comando a ejecutar lo escribiría tal cual directamente sobre el padding, en el OFFSET correcto para que RCX cargase bien dicha dirección **[rbp – 0xb0]**. Finalmente, RAX debía acabar computando una dirección dentro del binario que tuviera el valor QWORD adecuado (una dirección dentro de una sección rw- donde poder escribir). El valor 0x604150 que he comentado al principio era ideal, pero el código mov rax, qword [rax+0x30] lee un QWORD. Así que busqué el patrón 0x0000000000604150 sin encontrarlo:

gef➤ search-pattern 0x0000000000604150
[+] Searching '\x50\x41\x60\x00\x00\x00\x00\x00' in memory

Seguí buscando, empezando con la .got.plt:



En la dirección 0x603ef8 tenemos el valor QWORD de la .got.plt

Bien, siempre que mi comando sólo fuera como máximo 8 bytes de largo y no sobreescribiera la siguiente dirección, en un principio podía cargar sobre RAX el valor 0x603ef8-0x30 y así acabaría teniendo la dirección 0x603ef8 sobre RAX. Este valor, 0x00603ec8, debía colocarlo en mi padding en el OFFSET adecuado.

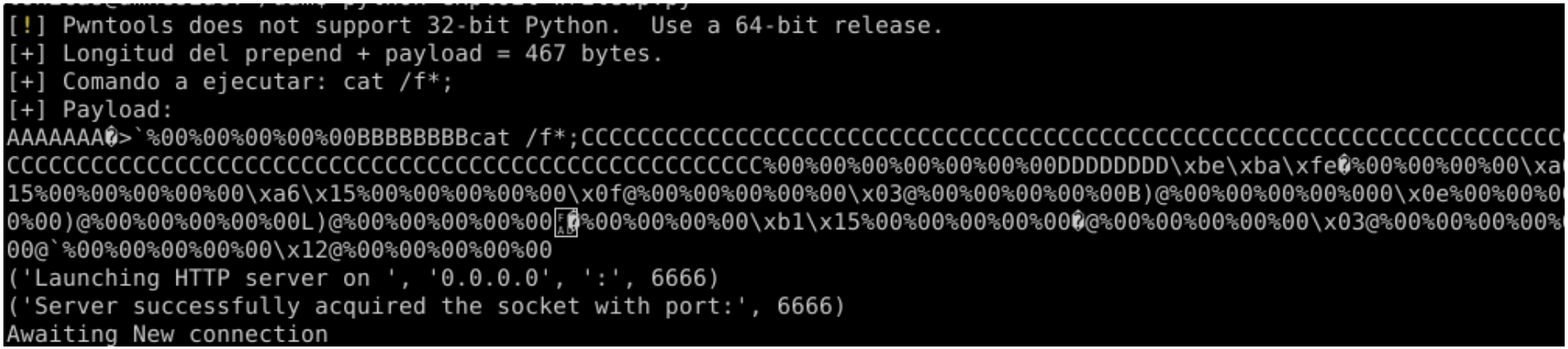
Así, con un primer ROP mov rbp, rsp; mov rdi, rsi; ret; tenía acceso a mi padding pero junto con el slash "/". No podía controlar el "/", que se convertía en 0x2f como valor numérico sobre RAX, estropeándome el ataque por completo. Así que ejecuté un segundo ROP idéntico para desplazarme dentro de mi padding y poder controlar la totalidad del valor a escribir sobre RAX.

El comando

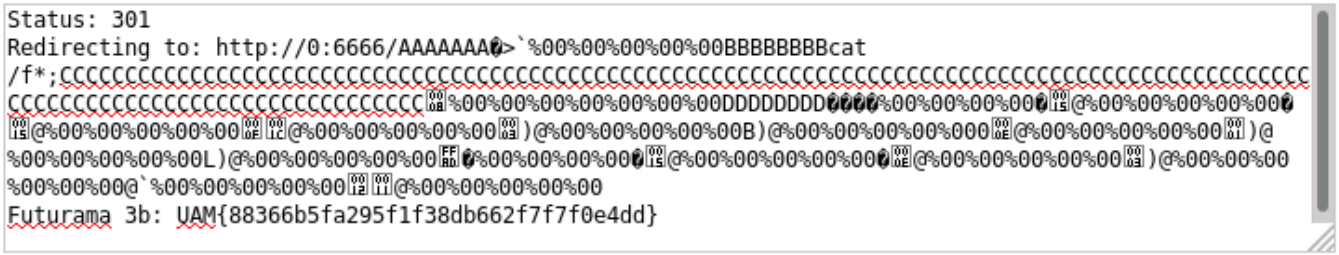
Sin leaks, ya podía escribir cualquier cosa de hasta 8 bytes de longitud en mi padding y obtener RCE en el servidor. Pero tras mi comando, había parte de morralla de la siguiente dirección concatenada con mi string dentro de la .got.plt y system se quejaba. Así que la única solución era reducir la longitud del comando a 7 bytes y añadir **";"** al final. Probando contra el servidor de la UAM, se ejecutaba el primer comando, system se quejaba del segundo pero saliendo con un ROP exit(0) el servidor devolvía la salida de carl en el textarea. Para obtener la flag directamente por pantalla no tenía suficientes caracteres, pero gracias a la shell eso da igual. Algo tan simple como **cat /fl*;** era más que suficiente para lograr mi objetivo!

La flag

Tras pulir un poco mi script y lanzarlo, sin leaks y con una sola ejecución, la tan ansiada flag apareció por pantalla.



Lanzando el exploit…



… y consiguiendo la FLAG sin LEAKS y en SSRF!

Podéis bajar el exploit AQUÍ.

@socialkas