TiYunZong: An Exploit Chain to Remotely Root Modern Android Devices

——Pwn Android phones from 2015 to 2020

Guang Gong
Alpha Lab, 360 Internet Security Center

Content

ABSTRACT

As more and more mitigations have been introduced into Android, it has become much more difficult to root modern Android devices, in particular, remotely root. This is especially true for Pixel devices as they always have the latest updates and mitigations. In this paper, we will explain why Pixel devices are challenging targets and will give a comprehensive attack surface analysis of remotely compromising Android. Furthermore, we will introduce an exploit chain, code-named TiYunZong, which can be leveraged to remotely root a wide range of Qualcomm-based Android devices including Pixel Devices. The name TiYunZong comes from a very famous Chinese martial arts fiction and this Kungfu allows people to move upward swiftly and effortlessly as if clouds are the stairs. The exploit chain includes three bugs which are just like the cloud stairs of TiYunZong.

The three bugs are found lately and they are numbered CVE-2019-5870, CVE-2019-5877, CVE-2019-10567. We will also present an effective and stable approach to chain these three vulnerabilities for exploitation without any ROP, despite the fact that ROP is the most common technique to exploit complicated vulnerabilities. The exploit chain is the first reported one-click remote root exploit chain on Pixel devices and won the highest reward for a single exploit chain across all Google VRP programs[1].

Keywords

Android, Chrome, Root, Remote Code Execution, Exploitation, V8, Mojo, KGSL

BACKGROUND

Which is the most secure smartphone? It's an open question and I think there is no standard answer. But I will try to convince you that Pixel phone is at least one of the most secure smartphones.

Android and iOS are the two most popular mobile operating systems in the world, and there are too many arguments about which is more secure. There are different answers from different perspectives. If we answer the question from the perspective of the price paid by vulnerability brokers, Android is more secure as higher bounty suggests greater threat the vulnerability may pose and more difficulties to discover it.

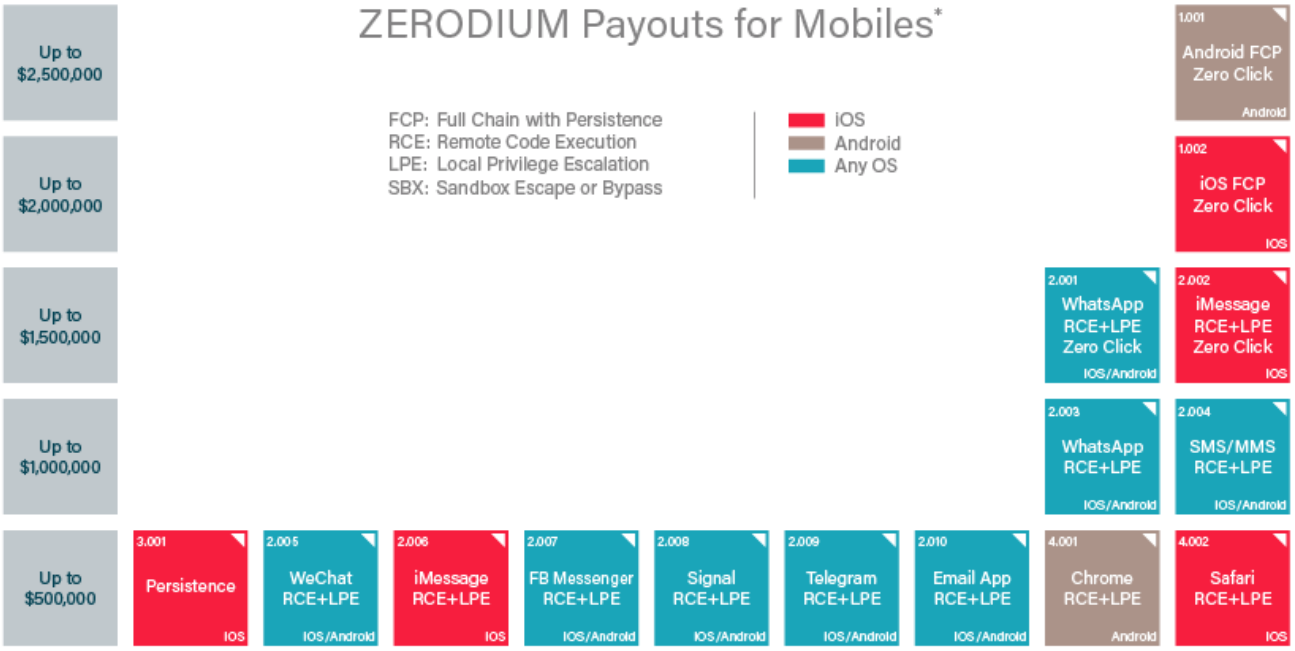

Figure 1. Part of ZERODIM Payouts for Mobiles

As indicated in Figure 1, Zerodium pays up to 2.5 million for Android FCP Zero-click, which is higher than IOS FCP Zero-click. We can say Android is more secure than iOS to some extent.

If we assume Android is more secure than iOS, the remaining question is which is the most secure phones among Android phones. It's easier to answer than the previous one. Google Pixel devices often have more up-to-the-date OS versions, security patches and vulnerability mitigations than other Android phones, which make it more difficult to be attacked.

As shown in Table 1, the Pixel phone is the only device that was not pwned in last three years' Mobile Pwn2Own competitions [2][3][4] while Apple iPhone devices were pwned 7 times in total. Other Android devices including Samsung Galaxy devices, Huawei Mate/P series and Xiaomi Mi Devices were Pwned several times in the contests too. Many exploits demonstrated in the contests target Android devices existed in outdated OS component or unsecured customized code.

Devices / Pwned Times / Year	Apple iPhone	Google Pixel	Samsung Galaxy	Huawei Mate/P	Xiaomi Mi
Mobile Pwn2Own 2017	5(1 partial win)	0	3	2	N/A
Mobile Pwn2Own 2018	2	0	2	0	5
Mobile Pwn2Own 2019	0	0	3(1 partial win)	0	3(1 partial win)
Total	7	0	8	2	8

Table 1. Mobile Pwn2Own results of the latest three years

According to price the vulnerability brokers offer and the results of the Mobile Pwn2Own competitions, Google Pixel Phone is one of the most secure smartphones. It's a tough target, however no device is 100% invulnerable. In this paper, I'll detail how I remotely root the pixel phone.

REMOTE ATTACK SURFACE AND RELEATED WORKS

Remotely compromising a smart phone is very appealing for bug bounty hunters and there have been some interesting researches in recent years. As shown in figure 2, remote attacks of smart phones can be divided into two categories by attack vector: attacks through internet and attacks through adjacent network.

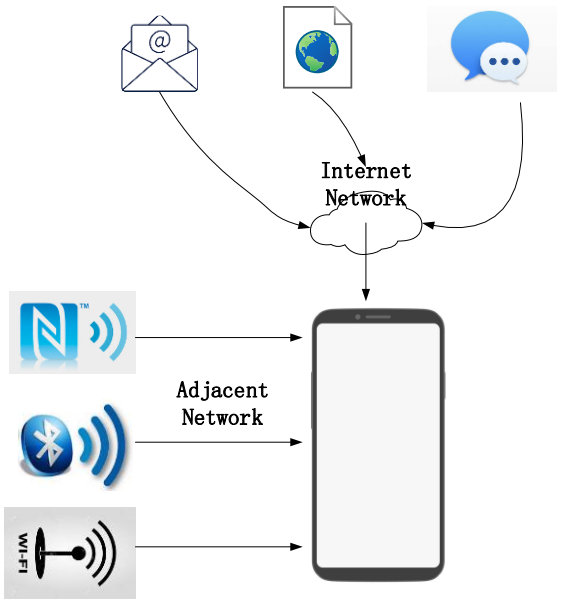

Figure 2. Remote Attack Surface of smart phones

Normally speaking, attacks exploiting vulnerabilities in browsers, IMs and emails can be launched through internet. Attacks which exploits vulnerabilities in NFC, Bluetooth, Wi-Fi and baseband are mostly launched through adjacent network. Attacks from internet is more destructive than attacks from adjacent network. But the former often needs some interaction, such as

clicking a URL, opening an email or instance message. It's relatively easier to launch an attack from adjacent network without interaction than from internet, Broadpwn[5] and BlueBorne[6] are amazing adjacent network attacks. It's hard to say which attack is more harmful between zero-click adjacent network attack and one-click internet deployable attack.

The exploit chain I used to remotely root the pixel phone is a one-click internet deployable attack chain. Table 2 summarizes some remote working exploit chains found by me targeting Android in recent years.

Event	Attack Vector	Target Phone	Obtained Permissions
Pwn2Own 2015	Chrome v8 bug->RCE2UXSS->Google Play Install	Nexus 6	Install any app
Pwn0Rama 2016	Chrome v8 bug ->Chrome IPC weakness->WebView bug	Nexus 6P, Galaxy Note 5, LG G4	Install any app
PwnFest 2016	Chrome v8 bug ->RCE2UXSS->Google Play Install	Pixel XL	Install any app
Pwn2Own 2017	Samsung Internet Browser bug -> exynos gralloc module bug	Galaxy S8	System user permission
ASR 2018	Chrome v8 bug -> libgralloc module bug	Pixel	System user permission
ASR 2019	Chrome v8 bug ->Mojo IPC bug->KGSL bug	Pixel 3	Root user permission

Table 2. Remote working exploit chains targeting Android Found by me in recent years

In Mobile Pwn2Own 2015, I pwned Nexus 6 with a single vulnerability [7], which is an OOB access bug(CVE-2015-6764) in v8 JavaScript Engine, to get RCE in Chrome render process. Then I turned this RCE into an UXSS vulnerability, so any JavaScript code could be injected into Google play website and any app could be installed from it.

In Pwn0Rama 2016, I pwned Nexus 6P, Galaxy Note 5, LG G4 with a single exploit chain [8], the exploit chain also started from a v8 bug, combined with a Chrome IPC weakness: any exported activity including webview activity in system could be launched by JavaScript. At that time, the webview in Android had no sandbox, by exploiting the v8 bug again in a webview, Chrome sandbox could be bypassed easily. This sandbox escaping method doesn't work anymore because webview is also sandboxed now.

In PwnFest 2016, I pwned a Pixel XL with a single vulnerability once more [9], I exploited a small logical mistake (CVE-2016-9651) in v8, and then I leveraged the same RCE2UXSS method used in Pwn2Own 2015 to install apps from Google play. After PwnFest 2016, Google updated the remote app installation feature to prompt the user to enter their password, which makes this form of attack difficult [10], but it took more than a year for them to mitigate this kind of attack. As of Chrome 77, Site Isolation has been enabled on Android devices with at least 2 GB of RAM [11]. It's nearly impossible to leverage the RCE2UXSS method.

In Mobile Pwn2Own 2017, I pwned a Galaxy S8 with two bugs [12], an OOB access bug in Samsung Internet Browser, and a Use-After-Unmap vulnerability in Samsung exynos gralloc module. Due to the existence of Selinux, most system services in Android are restricted from being accessible from the sandboxed Chrome rendering process, and only a few functions can be called from the isolated_app domain. It's a narrow attack surface from sandbox render processes to system services. But I still found a way that let renderer process reach system_server by binder call with Parcelable object [13]. I can get a reverse shell with system user permission through this exploit chain.

In Android Security Reward Program 2018, I Pwned Pixel phone with two bugs (CVE-2017-5116 and CVE-2017-14904). CVE-2017-5116 is a V8 engine bug that is used to get remote code execution in sandboxed Chrome render process. CVE-2017-14904 is a bug in Android's libgralloc module that is used to escape from Chrome's sandbox. Together, this exploit chain can be used to inject arbitrary code into system_server by accessing a malicious URL in Chrome. This exploit chain won the highest reward in the history of the ASR program [14].

In Android Security Reward Program 2019, I pwned Pixel 3 with three bugs, which are CVE-2019-5870, CVE-2019-5877, CVE-2019-10567. It won the highest reward for a single exploit chain across all Google VRP programs.

THE EXPLOIT CHAIN

The exploit chain is composed of 3 bugs, two Chrome bugs and one QUALCOMM KGSL driver bug. The attack vector is malicious links. Once victims access a web link controlled by attackers, the attacker can get a reverse shell with root privilege of the attacked phone. Figure 3 shows how to remotely get root permission with the exploit chain.

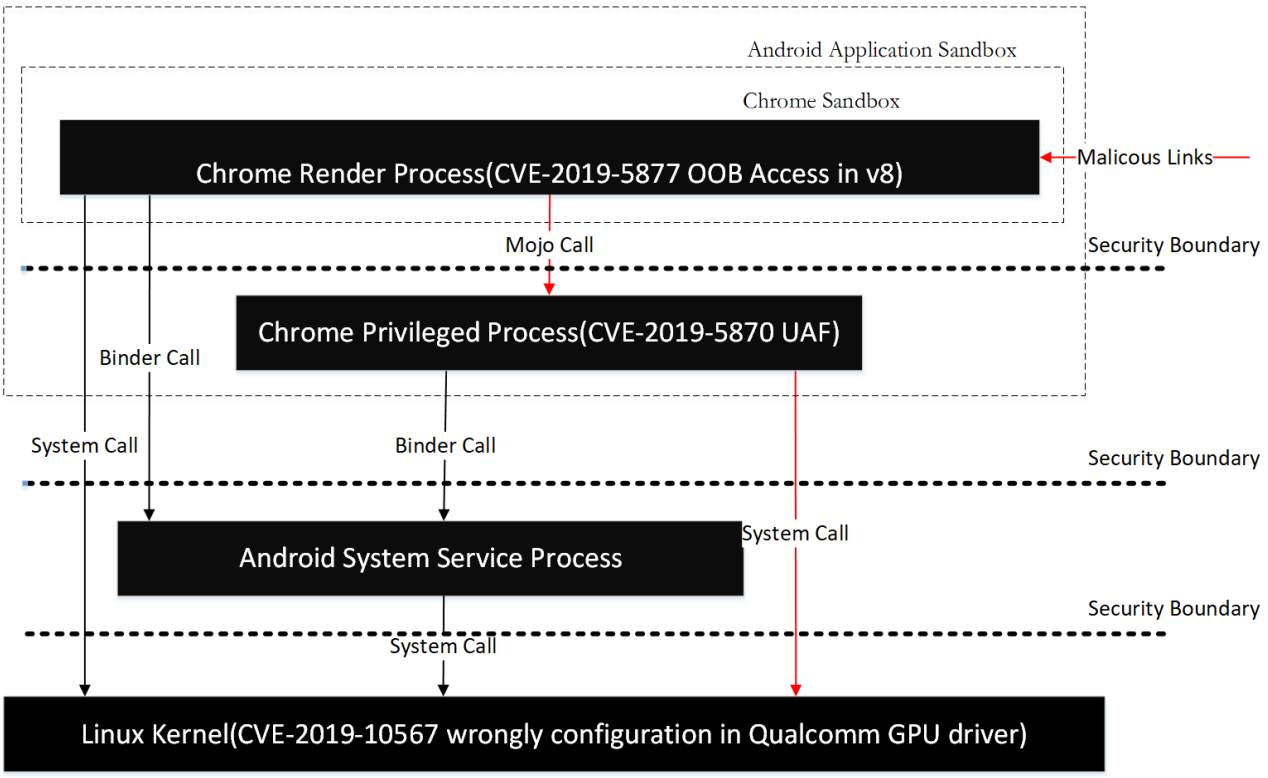

Figure 3. the Exploit Chain

First, an OOB access bug (CVE-2019-5877) in v8 is exploited to achieve remote code execution in a Chrome render process, render process runs in the SELinux domain iosolated_app and is highly sandboxed. According to the following SELinux policies [15], we know render processes can only interact with three Android system services: activity service, display service and webview update service, and it can't access the GPU driver directly.

```
# Isolated apps can only access three services,
# activity_service, display_service, webviewupdate_service.
neverallow isolated_app {
   service_manager_type
   -activity_service
   -display_service
   -webviewupdate_service
}:service_manager find;
# Isolated apps shouldn't be able to access the driver directly.
neverallow isolated_app gpu_device:chr_file { rw_file_perms execute };
```

CVE-2019-10567 is a bug in QUALCOMM GPU driver, but it can't be triggered directly from islolated_app domain. In order to trigger this bug, we need to escape from islolated_app domain with the help of CVE-2019-5870 firstly. CVE-2019-5870 is a use-after-free vulnerability in the media component. A compromised render process can trigger this vulnerability through Mojo call [16]. By exploiting it, arbitrary code can be executed in Chrome privileged process. This process runs in the SELinux domain untrusted_app. As the following SELinux polices [17] shown, process run in this domain can access GPU driver.

```
# Grant GPU access to all processes started by Zygote.
# They need that to render the standard UI.
allow { appdomain -isolated_app } gpu_device:chr_file rw_file_perms;
```

The domain untrusted_app is still restricted by Android application sandbox. We need another bug to break application sandbox.

Generally speaking, normal apps including Chrome can't access most of drivers in Andoid because of the existence of SELinux. To root Android from an app, we often need to find a bug in android system service and then exploit it as a proxy to attack the kernel. Luckily, the KGSL driver bug (CVE-2019-10567) I found can be triggered from Chrome privileged process directly. It's a configuration issue in the QUALCOMM Adreno GPU kernel driver. There is a way to deceive the GPU kernel driver into thinking there is room in the GPU ring buffer and overwriting existing commands could allow unintended GPU opcodes to be executed [18]. the following chapters will introduce the technical details of the three vulnerabilities in the exploit chain.

THE RCE VULNERABILITY (CVE-2019-5877)

Prior Knowledge

V8 is the JavaScript engine that powers Google Chrome. V8's CodeStubAssembler is a custom, platform-agnostic assembler that provides low-level primitives as a thin abstraction over assembly [19]. V8 torque [20] is a V8-specific domain-specific language that is translated to CodeStubAssembler. In V8, many JavaScript built-in functions and objects are implemented by V8 Torque.

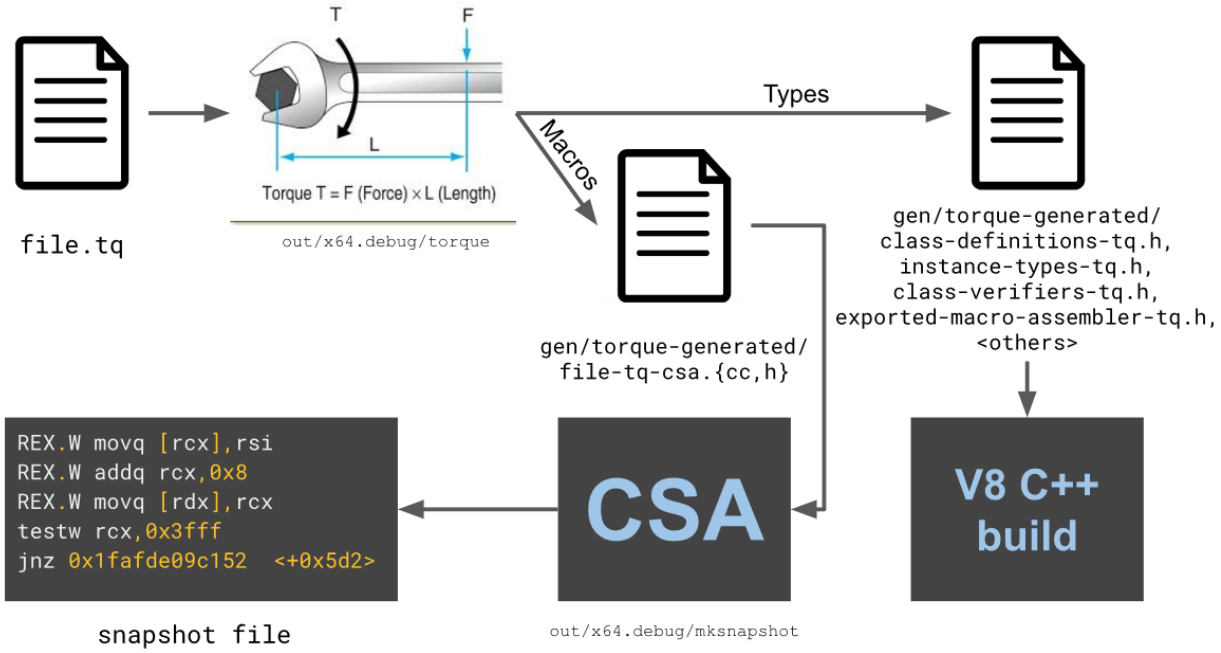

Figure 4. the Build Process of Torque [21]

Figure 4 shows the build process of torque code. If there is any bug in torque code, it'll be propagated to the generated CSA code and then to the generated snapshot file in the end. The snapshot file is embedded into Chrome to speed up the initialization of all built-in functionality in V8's heap.

Where is The Bug

The JSFunction object in V8 is the internal representation of function in JavaScript. The size of the JSFunction object is flexible. It may contain the field PrototypeOrInitialMap or not. The field PrototypeOrInitialMap is the last field of JSFunction if it has. Listing 1 shows the field information of the Array function. In line 15, has_prototype_slot means the Array function has the field PrototypeOrInitialMap (line 7), so its size is 64 bytes (line 12). Note that Array is a constructor (line 14, line 16). Listing 2 shows the field information of the function parseInt, it has no prototype slot (), so its size is 56 bytes (line 6). Note that parseInt is not a constructor. Mostly, a function has prototype slot when being a constructor otherwise it does not. But there is an exception, as Listing 3 shown, the function Proxy is a constructor (line 13), but it has no prototype slot (line 6).

```
1 d8> %DebugPrint(Array);
2 DebugPrint: 0x2b7bcbd91b79: [Function] in OldSpace
3  - map: 0x2f2f62402ff1 <Map(HOLEY_ELEMENTS)> [FastProperties]
4  - prototype: 0x2b7bcbd82091 <JSFunction (sfi = 0xc83e79480e9)>
5  - elements: 0x3b058dc40bf9 <FixedArray[0]> [HOLEY_ELEMENTS]
6  - function prototype: 0x2b7bcbd91dc9 <JSArray[0]>
7  - initial_map: 0x2f2f62403041 <Map(PACKED_SMI_ELEMENTS)>
8  - shared_info: 0x0c83e79547c9 <SharedFunctionInfo Array>
9
10 0x2f2f62402ff1: [Map]
11  - type: JS_FUNCTION_TYPE
12  - instance size: 64
13   - callable
14  - constructor
15  - has_prototype_slot
16  - constructor: 0x2b7bcbd822e1 <JSFunction Function (sfi = 0xc83e7954449)>
```

Listing 1. The Field Information of Array

```
1 d8> %DebugPrint(parseInt)
2 DebugPrint: 0x2b7bcbd8b999: [Function] in OldSpace
3  - map: 0x2f2f624003e1 <Map(HOLEY_ELEMENTS)> [FastProperties]
4  - prototype: 0x2b7bcbd82091 <JSFunction (sfi = 0xc83e79480e9)>
5  - elements: 0x3b058dc40bf9 <FixedArray[0]> [HOLEY_ELEMENTS]
6  - function prototype: <no-prototype-slot>
7  - shared_info: 0x0c83e79557a1 <SharedFunctionInfo parseInt>
8
9
10 0x2f2f624003e1: [Map]
11  - type: JS_FUNCTION_TYPE
12  - instance size: 56
13  - callable
14  - constructor: 0x3b058dc401b1 <null>
```

Listing 2. The Field Information of parseInt

```
1 d8> %DebugPrint(Proxy)
2 DebugPrint: 0x2b7bcbd8d6d1: [Function] in OldSpace
3  - map: 0x2f2f62401d31 <Map(HOLEY_ELEMENTS)> [FastProperties]
4  - prototype: 0x2b7bcbd82091 <JSFunction (sfi = 0xc83e79480e9)>
5  - elements: 0x3b058dc40bf9 <FixedArray[0]> [HOLEY_ELEMENTS]
6  - function prototype: <no-prototype-slot>
7  - shared_info: 0x0c83e795e749 <SharedFunctionInfo Proxy>
8
9 0x2f2f62401d31: [Map]
10  - type: JS_FUNCTION_TYPE
11  - instance size: 56
12  - callable
13  - constructor
14  - constructor: 0x3b058dc401b1 <null>
```

Listing 3. The Field Information of Proxy

The RCE bug exists in toque code. It's an OOB access vulnerability as shown in Listing 4[22]. In line 6 the Macro GetDerivedMap tries to get the field prototype_or_initial_map of the argument newTarget, but it doesn't check whether

newTarget contains prototype_or_initial_map, and accesses it directly. If newTarget happens to be the Proxy function, it is an out-of-bound access.

```
1 macro GetDerivedMap(implicit context: Context)(
2    target: JSFunction, newTarget: JSReceiver): Map {
3  try {
4    const constructor = Cast<JSFunction>(newTarget) otherwise SlowPath;
5    const map =
6        Cast<Map>(constructor.prototype_or_initial_map) otherwise SlowPath;
7    if (LoadConstructorOrBackPointer(map) != target) {
8      goto SlowPath;
9    }
10    return map;
11  }
12  label SlowPath {
13    return runtime::GetDerivedMap(context, target, newTarget);
14  }
15 }
```

Listing 4. The Torque Code of RCE Bug

How to Exploit it

It's easy to trigger the bug, you need only one line of JavaScript code as follows:

var malformedTypedArray = Reflect.construct(Uint8Array, [4], Proxy)

This line of JavaScript code creates a typed array with Reflect.construct, Uint8Array is target constructor, and Proxy is newTarget. Reflect.construct will call the Torque function CreateTypedArray, which will call the vulnerable macro GetDerivedMap. Because the newTarget argument is set to the Proxy function, the OOB access will be triggered. But if we run the single line of JavaScript code above in Chrome, nothing special will happen although the OOB access has already occurred. Let's review the vulnerable macro in Listing 4. After loading the OOB filed prototype_or_initial_map (line 6), the value will be casted to a Map, then the map's constructor is loaded and compared with target (line 7). Because prototype_or_initial_map is the last filed of JSFucntion object, if it exists, the OOB access gets the first field of the next object. In V8, the first filed of an object happens to be a Map object, so the cast will succeed. But in most situation, the comparison in line 7 will fail because the Map's constructor is not equal to target (Uint8Array). So, execution flow will bail out to slow path, and we lost the exploitation primitive. In order to exploit this bug, we need to replace the object which lies after the Proxy function object to an object whose Map's constructor is Uint8Array. The exploit strategy is as follows:

1. Free the object below the Proxy function.

2. Re-occupy the free space with an object whose Map's (named map x) constructor is Uint8Array, and then drop all reference to Map x so that GC will mark the Map x object as white and will sweep it in scheduled sweep tasks.

3. Trigger the OOB access bug before Max x get swept by GC, so the vulnerable GetDerivedMap macro won't bail out to the slow path. The pointer of Map x will still be used in CreateTypedArray.

4. After GC sweep task finished, re-occupy the freed space of Map x with a map whose constructor is Uint32Array, so we can get a malformed typed array, its map is Uint32Array, but its layout, especially its element kind is Uin8Array type, it's easy to implement arbitrary read and write with this malformed object.

5. With the ability of arbitrary read and write, we can enable MojoJS bindings [23] and exploit the following Mojo vulnerability to escalate from Chrome sandbox.

THE EOP VULNERABILITY (CVE-2019-5870)

Prior Knowledge

Chrome uses a multi-process architecture, which isolates render processes from other privileged processes. As Figure 5[24] shown, the main process that runs the UI and manages tab and plugin processes is called as the browser process. The GPU

process is a process used when Chrome is displaying GPU-accelerated content. Likewise, the tab-specific processes are called render processes. Normally, render process has the lowest privilege, and the GPU process and the browser process have higher privileges. Render processes are highly sandboxed and many interactions between them and the system are proxied by other privileged processes. In Android, render processes run in the domain "isolated_app" and then GPU process and the browser process run in the domain "untrusted_app". The communication between processes is via inter-process communication. The RCE bug occurred in a render process while the EOP Vulnerability occurred in the GPU process.

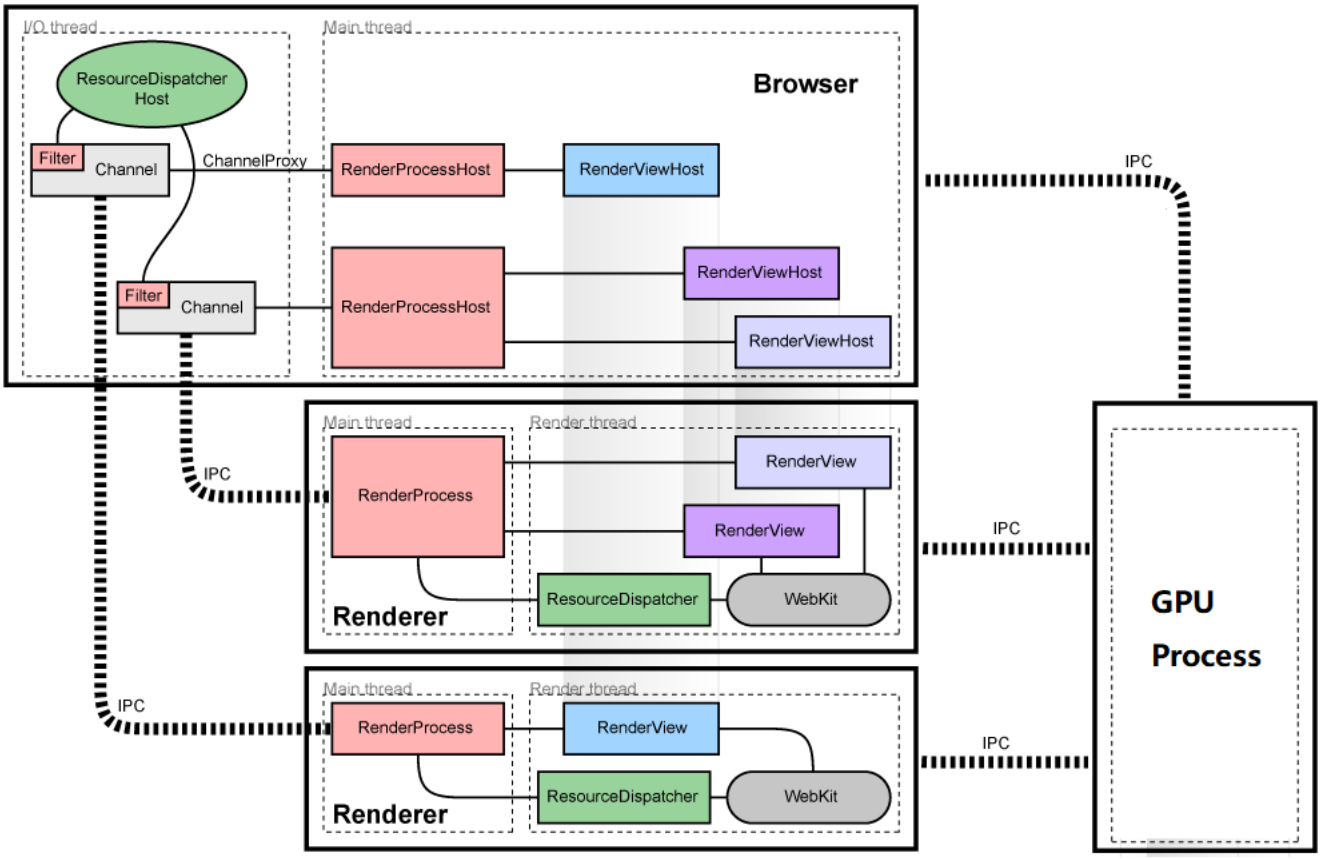

Figure 5. Chrome's Multi-Process Architecture

There are two IPC systems in Chrome, legacy IPC and Mojo IPC [25]. Mojo is Chrome's new IPC system. It is a collection of runtime libraries providing a platform-agnostic abstraction of common IPC primitives, a message IDL format, and a bindings library with code generation for multiple target languages to facilitate convenient message passing across arbitrary inter- and intra-process boundaries. CVE-2019-5870 is a vulnerability that can be triggered by Mojo IPC, or Mojo Call.

Where is The Bug

CVE-2019-5870 is a UAF vulnerability in Chrome media component. It is related with the interface of content decryption module (CDM) which is a module built into Chrome browsers that allow Chrome to play DRM-protected HTML5 video and audio. The IDL format of the CDM is as Listing 5 shown [26]. The function in line 3 is responsible for initializing the CDM. If initialization succeeds, cdm_id will be non-zero and will later be used to locate the CDM at the remote side. The function is implemented in C++ as Listing 6 shown [27]. If initialization succeeds, the function MojoCdmService::OnCdmCreated[28] will be called. The cause of the vulnerability is that there is no restriction to the number of calls of the function MojoCdmService::Initialize. If it was called twice, the same MojoCdmService would be registered twice in the function MojoCdmService: OnCdmCreated in line 12. So in the function MojoCdmServiceContext::RegisterCdm [29], two cdm ids would be mapped to the same MojoCdmService in line 4, Listing 7 when MojoCdmService was destructed. Only one cdm id was unregistered and removed from the map cdm_services_, the other cdm id was mapped to a dangling pointer. Then, if the function MojoCdmServiceContext::GetCdmContextRef[30] was called , the UAF would occur in line 6, Listing 9. MojoCdmService can be configured to be run in different process. It can be configured to run in the browser process, the

GPU process and utility processes. It also can be configured to run in utility processes in desktop Chrome. Utility processes is highly sandboxed just as render processes, so it can't be exploited to escape sandbox in desktop Chrome. But in Android, MojoCdmService is configured to be run in the GPU process by default, so we can exploit it to access the GPU devices directly.

```
1 interface ContentDecryptionModule {
2   SetClient(pending_associated_remote<ContentDecryptionModuleClient> client);
3   Initialize(string key_system,
4              url.mojom.Origin security_origin,
5              CdmConfig cdm_config)
6       => (CdmPromiseResult result, int32 cdm_id,
7           pending_remote<Decryptor>? decryptor);
8   SetServerCertificate(array<uint8> certificate_data)
9       => (CdmPromiseResult result);
10   ......
11 };
```

Listing 5. The Interface Definition of CDM

```
1 void MojoCdmService::Initialize(const std::string& key_system,
2                                 const url::Origin& security_origin,
3                                 const CdmConfig& cdm_config,
4                                 InitializeCallback callback) {
5   DVLOG(1) << __func__ << ": " << key_system;
6   DCHECK(!cdm_);  ------------------>In debug version, this DCHECK will be trigger
7
8   auto weak_this = weak_factory_.GetWeakPtr();
9   cdm_factory_->Create(
10      key_system, security_origin, cdm_config,
11      base::Bind(&MojoCdmService::OnSessionMessage, weak_this),
12      base::Bind(&MojoCdmService::OnSessionClosed, weak_this),
13      base::Bind(&MojoCdmService::OnSessionKeysChange, weak_this),
14      base::Bind(&MojoCdmService::OnSessionExpirationUpdate, weak_this),
15      base::Bind(&MojoCdmService::OnCdmCreated, weak_this,
16                 base::Passed(&callback)));
17 }
```

Listing 6. The Implementation of the Initialized Function of CDM

```
1 void MojoCdmService::OnCdmCreated(
2     InitializeCallback callback,
3     const scoped_refptr<::media::ContentDecryptionModule>& cdm,
4     const std::string& error_message) {
5   mojom::CdmPromiseResultPtr cdm_promise_result(mojom::CdmPromiseResult::New());
6
7   if (!cdm) {
8   ......
9   }
10    cdm_ = cdm;
11    if (context_) {
12      cdm_id_ = context_->RegisterCdm(this); ---------------------------->register twice here
13      DVLOG(1) << __func__ << ": CDM successfully registered with ID " << cdm_id_;
14    }
15    ...
16 }
```

Listing 7. The Callback Function OnCdmCreated

```
1 int MojoCdmServiceContext::RegisterCdm(MojoCdmService* cdm_service) {
2   DCHECK(cdm_service);
3   int cdm_id = GetNextCdmId();
4   cdm_services_[cdm_id] = cdm_service;------------------------------>two cdm ids map to one cdm_service
5   DVLOG(1) << __func__ << ": CdmService registered with CDM ID " << cdm_id;
6   return cdm_id;
7 }
```

Listing 8. The Fucntion RegisterCdm

```
1 std::unique_ptr<CdmContextRef> MojoCdmServiceContext::GetCdmContextRef(
2     int cdm_id) {
3 ......
4   auto cdm_service = cdm_services_.find(cdm_id);
5   if (cdm_service != cdm_services_.end()) {
6     if (!cdm_service->second->GetCdm()->GetCdmContext()) {
7       NOTREACHED() << "All CDMs should support CdmContext.";
8       return nullptr;
9     }
10     return std::make_unique<CdmContextRefImpl>(cdm_service->second->GetCdm());
11   }
12   ......
13   return nullptr;
14 }
```

Listing 9. The Fucntion GetCdmContextRef

How to Exploit it

As Listing 10 shown, the size of the object MojoCdmService is small, it has only 48 bytes (line 2), there is a lot of noise in the heap in this size range (many other objects in the same size range are allocated if we try to occupy the freed MojoCdmServices), so it's hard to reoccupy the freed MojoCdmService object with controlled data, but its member "scoped_refptr<::media::ContentDecryptionModule> cdm_"[31] point to a large object, which is an MediaDrmBridge object with size 168(line 4), we can reoccupy the memory of the freed MediaDrmBridge object with controlled data stably.

```
1 (gdb) p sizeof(media::MojoCdmService)
2 $21 = 48
3 (gdb) p sizeof(media::MediaDrmBridge)
4 $3 = 168 //the size is 160 in release version
5 (gdb) x/10xw 0xb6993300  //the content of MediaDrmBridge
6 0xb6993300:    0xca3f3a0c      0x00000000      0x00000100      0xca3f3a4c
7 0xb6993310:    0xca3f3a6c      0xb6a90750      0xb6a90750      0xb6a90760
8 0xb6993320:    0x00000000      0x00000000
9 (gdb) x/10xw 0xca3f3a0c  //the content of virtual table of MediaDrmBridge
10 0xca3f3a0c <_ZTVN5media14MediaDrmBridgeE+8>:     0xca237e09 0xca207a79 0xca237fad 0xca2382f9
11 0xca3f3a1c <_ZTVN5media14MediaDrmBridgeE+24>:    0xca2384a9 0xca238601 0xca238741 0xca238881
12 (gdb) info symbol 0xca238881
13 media::MediaDrmBridge::GetCdmContext() + 1 in section .text of libmedia.cr.so
```

Listing 10. The Output of GDB

There is an assumption, the content of the MojoCdmService object keep unchanged after free, so its pointer cdm_ still points to the same MediaDrmBridge object. The assumption is easy to be realized if we don't try to reoccupy the freed

MojoCdmService object. The object MojoCdmService has virtual table as shown in line 9, the 8th pointer of the virtual table is the function point to GetCdmContext(line 12, line 13). Note that this function is used in the function GetCdmContextRef(line 6, Listing 9) if we can reoccupy the freed MediaDrmBridge object with controlled data, the virtual table can be controlled.

When the function GetCdmContextRef is called, we can control pc with the modified virtual function pointer GetCdmContext in line 6, Listing 9. The remaining question is which object can be used to reoccupy the freed MediaDrmBridge object. There are many options. I choose the member, namely extra_data_[32] of the VideoDecoderConfig object. extra_data_ is a vector, the size of its backing store and the content are both controllable. Afterwards, we can reoccupy the MediaDrmBridge object with arbitrary data. But there is another problem. We can't allocate memories with known address in GPU process, hence, we don't know where the virtual table pointer of the MediaDrmBridge object is modified to point to. Maybe we can find another information disclosure bug to fake a virtual table with known address, then we modify the virtual table point to point to it to finish the exploit by ROP, however, it's not a good choice. As we know, render processes and the GPU process have the same memory layouts in Android because they are forked from the same process. So we know the base addresses of most of shared libraries in the GPU process. The shared library libllvm-glnext uses the system function. There will be a system function pointer (assume it's stored in address S) in libllvm-glnext after it's loaded. Now, if we modify virtual table pointer to point to S-28 As shown in Figure 6, when the function GetCdmContext is called in line 6, Listing 9, the system function is called, the argument is controllable too. We can use the simple "return to libc" method to execute any shell command in the SELinux domain untrusted_app and escape from the Chrome sandbox.

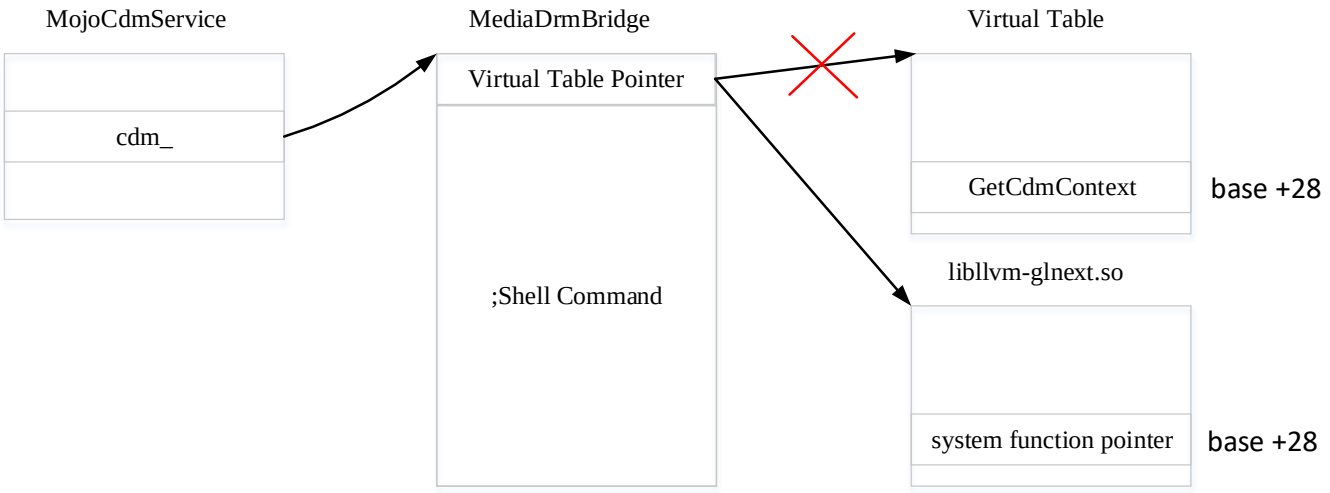

Figure 6. Exploit the EOP bug

THE ROOT VULNERABILITY (CVE-2019-10567)

Prior Knowledge

The Google Pixel phone has an Adreno [33] GPU which uses the KGSL driver [34] developed by Qualcomm. KGSL means kernel graphics support layer. It communicates with the user land Apps and system services to render graphics.

Figure 7 shows the architecture of the KGSL driver, Apps can create Adreno contexts with different priorities. Normally, there are four context priorities (0,1,2,3). 0 denotes the highest priority, and 3 denotes the lowest priority. Adreno GPUs use IOMMUs, each context has its own GPU page tables. To implement per-context GPU page tables, basically all the driver needs to do is to bang a few IOMMU registers to change the page table base address and invalidate the TLB [35]. Similar as CPU has different privilege level, GPU can run in different mode. Adreno GPU can run in two modes, privileged mode and unprivileged mode. Pages and registers can be configured to different attributes in different mode. Some pages and registers can only be accessed or written in privileged mode. Although every context uses different page tables, some pages are mapped globally to all contexts.

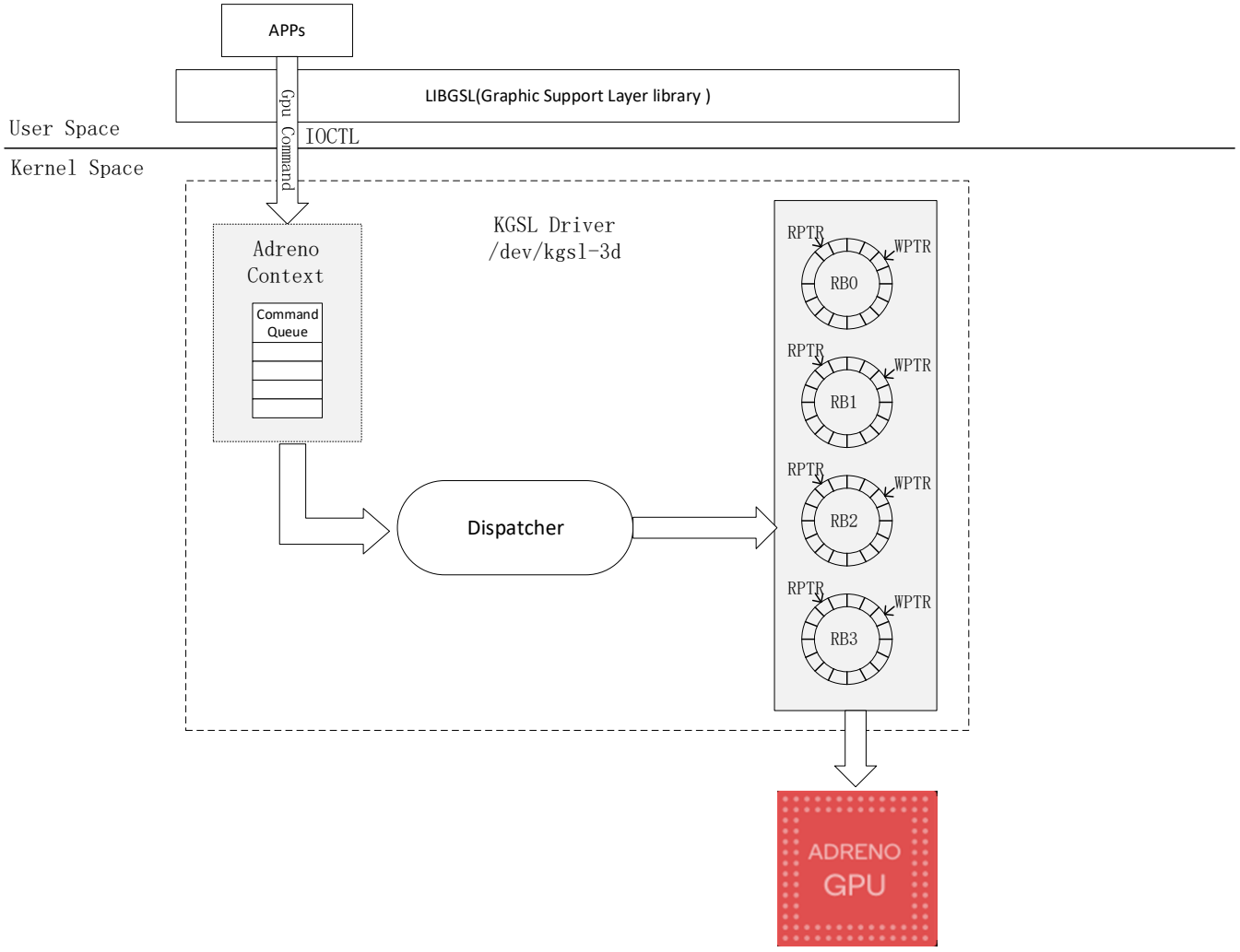

Figure 7. The KGSL Driver Architecture of Adreno GPU

Listing 11 shows all globally mapped pages in Pixel crosshatch. Note that, “scratch” in line 5 is a globally mapped page and we will use it latter. Some pages are not only mapped in GPU space, but also mapped in CPU space. Each Adreno context is bound to a ring buffer depending on its priority. Because Adreno contexts have four priorities, there are four ring buffers relatively (RB0, RB1, RB2, RB3). After an App creates an Adreno context, the App can send GPU commands to the context by IOCTL. Each context has a command queue. The received GPU commands are queued. The dispatcher is the core module in KGSL. It runs in a separate kernel thread and keeps reading commands from the queue and submitting them to ring buffers. After commands are submitted to a ring buffer, the write pointer (WPTR) of the ring buffer is updated. After the Adreno GPU executes some commands, the read pointer (RPTR) is updated by the GPU.

Offset	Length(Bytes)	Content
0	4*4	RB0 RPTR,RB1 RPTR, RB2 RPTR, RB3 RPTR
0x10	8*4	RB0 Context Restore Address, RB1 Context Restore Address RB2 Context Restore Address, RB3 Context Restore Address

Table 3. The Format of the Scratch Memory

The scratch memory is one-page data that is mapped into the GPU. This allows for some 'shared' data between the GPU and CPU. For example, it will be used by the GPU to write updated RPTR for each ring buffer. The format of the scratch is as Table 3. The first 16 bytes are the read pointers of the ring buffers.

```
 1 crosshatch:/ # cat /sys/kernel/debug/kgsl/globals
 2 0x00000000fc000000-0x00000000fc000fff             4096 setstate
 3 0x00000000fc001000-0x00000000fc040fff           262144 gpu-qdss
 4 0x00000000fc041000-0x00000000fc048fff            32768 memstore
 5 0x00000000fc049000-0x00000000fc049fff             4096 scratch
 6 0x00000000fc04a000-0x00000000fc04afff             4096 pagetable_desc
 7 0x00000000fc04b000-0x00000000fc052fff            32768 ringbuffer
 8 0x00000000fc053000-0x00000000fc053fff             4096 pagetable_desc
 9 0x00000000fc054000-0x00000000fc05bfff            32768 ringbuffer
10 0x00000000fc05c000-0x00000000fc05cfff             4096 pagetable_desc
11 0x00000000fc05d000-0x00000000fc064fff            32768 ringbuffer
12 0x00000000fc065000-0x00000000fc065fff             4096 pagetable_desc
13 0x00000000fc066000-0x00000000fc06dfff            32768 ringbuffer
14 0x00000000fc06e000-0x00000000fc09dfff           196608 profile
15 0x00000000fc09e000-0x00000000fc0a5fff            32768 ucode
16 0x00000000fc0a6000-0x00000000fc0a8fff            12288 capturescript
17 0x00000000fc0a9000-0x00000000fc113fff           438272 capturescript_regs
18 0x00000000fc114000-0x00000000fc114fff             4096 powerup_register_list
19 0x00000000fc115000-0x00000000fc115fff             4096 alwayson
20 0x00000000fc116000-0x00000000fc116fff             4096 preemption_counters
21 0x00000000fc117000-0x00000000fc326fff          2162688 preemption_desc
22 0x00000000fc327000-0x00000000fc327fff             4096 perfcounter_save_restore_desc
23 0x00000000fc328000-0x00000000fc537fff          2162688 preemption_desc
24 0x00000000fc538000-0x00000000fc538fff             4096 perfcounter_save_restore_desc
25 0x00000000fc539000-0x00000000fc748fff          2162688 preemption_desc
26 0x00000000fc749000-0x00000000fc749fff             4096 perfcounter_save_restore_desc
27 0x00000000fc74a000-0x00000000fc959fff          2162688 preemption_desc
28 0x00000000fc95a000-0x00000000fc95afff             4096 perfcounter_save_restore_desc
29 0x00000000fc95b000-0x00000000fc95bfff             4096 smmu_info
```

Listing 11. Globally Mapped Pages

Where is The Bug

As we know, the RPTRs of ring buffers are stored in the "scratch" memory. The "scratch" memory is mapped in GPU space and CPU space. RPTRs are sensitive data for KGSL driver, the driver reads RPTRs when allocates buffer from ring buffers. But the "scratch" memory is wrongly set to writable by normal GPU Command Processor instructions in the function adreno_ringbuffer_probe as shown in Listing 12. In line 9, kgsl_allocate_global allocates a global memory without KGSL_MEMFLAGS_GPUREADONLY and KGSL_MEMDESC_PRIVILEGED, so RPTRs can be modified by GPU Command Processor instructions in unprivileged mode.

In the function adreno_ringbuffer_allocspace as shown in Listing 13, the variable rptr in line 5 is read from the scratch memory. As rptr can be modified to any value, the function can be controlled to return a wrong ring buffer pointer. As a result, we can overwrite the exist Command Processor instructions and inject malicious Command Processor instructions into ring buffer to execute.

```
1 int adreno_ringbuffer_probe(struct adreno_device *adreno_dev, bool nopreempt)
2 {
3    struct kgsl_device *device = KGSL_DEVICE(adreno_dev);
4    struct adreno_gpudev *gpudev = ADRENO_GPU_DEVICE(adreno_dev);
5    int i;
6    int status = -ENOMEM;
7
8    if (!adreno_is_a3xx(adreno_dev)) {
9         status = kgsl_allocate_global(device, &device->scratch,  ---------->scratch is allocated as writable by normal
Command Processor instructions
10              PAGE_SIZE, 0, KGSL_MEMDESC_CONTIG, "scratch");
11        if (status != 0)
12            return status;
13    }
14 ...
15 }
```

Listing 12. The Code of adreno_ringbuffer_probe

```
1 unsigned int *adreno_ringbuffer_allocspace(struct adreno_ringbuffer *rb,
2       unsigned int dwords)
3 {
4    struct adreno_device *adreno_dev = ADRENO_RB_DEVICE(rb);
5    unsigned int rptr = adreno_get_rptr(rb);   ---------------------------->read rptr from scratch memory
6    unsigned int ret;
7
8    if (rptr <= rb->_wptr) {
9       unsigned int *cmds;
10
11       if (rb->_wptr + dwords <= (KGSL_RB_DWORDS - 2)) {
12          ret = rb->_wptr;
13          rb->_wptr = (rb->_wptr + dwords) % KGSL_RB_DWORDS;
14          return RB_HOSTPTR(rb, ret);
15       }
16
17       /*
18          * There isn't enough space toward the end of ringbuffer. So
19          * look for space from the beginning of ringbuffer up to the
20          * read pointer.
21          */
22       if (dwords < rptr) {
23          cmds = RB_HOSTPTR(rb, rb->_wptr);
24          *cmds = cp_packet(adreno_dev, CP_NOP,
25             KGSL_RB_DWORDS - rb->_wptr - 1);
26          rb->_wptr = dwords;
27          return RB_HOSTPTR(rb, 0);
28       }
29    }
30
31    if (rb->_wptr + dwords < rptr) {
32       ret = rb->_wptr;
33       rb->_wptr = (rb->_wptr + dwords) % KGSL_RB_DWORDS;
34       return RB_HOSTPTR(rb, ret);
35    }
36    return ERR_PTR(-ENOSPC);
37}
```

Listing 13. The Code of Allocating Space from a Ring Buffer

How to Exploit it

There are few public documents about the internal architecture of the Adreno GPU. We can only speculate some internal behaviors of the Adreno GPU. RPTRs in the “scratch” memory are updated by the GPU, but it seems that they are just shadows of the real read pointers used by the GPU to fetch instructions. Modifying the RPTRs in the “scratch” memory doesn’t affect the executing flow of the GPU. Luckily, it affects the kernel allocating space from ring buffers.

The function adreno_ringbuffer_allocspace in Listing 13 shows the code of allocating space from a ring buffer in KGSL. The variable rptr points to the next instruction which will be executed by the GPU and _wptr points to the start of free space of a ring buffer. As shown in figure 8, there are two scenarios of the positions of rptr and _wptr, rptr<=_wptr or rptr>_wptr.

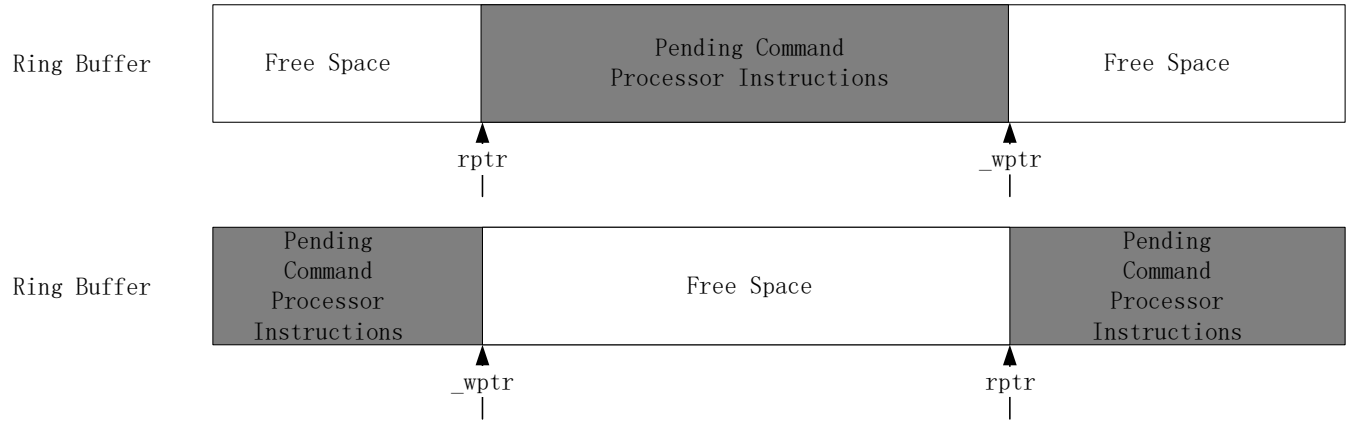

Figure 8. The Positions of rptr and _wptr of a Ring Buffer

As show in figure 9, when rptr is less than or equal to _wptr, if there is enough space from _wptr to the end of the ring buffer, allocating is simple, we just need to advance _wptr and return the original _wptr. If there isn't enough space toward the end of the ring buffer, we have to look for space from the beginning of the ring buffer up to the read pointer. If there is enough space, advance _wptr from the begin of the ring buffer. Otherwise, the allocation fails. When rptr is larger than _wptr, if there is enough space from _wprt to rptr, we just need to advance _wptr and return the original _wptr, otherwise the allocation fails.

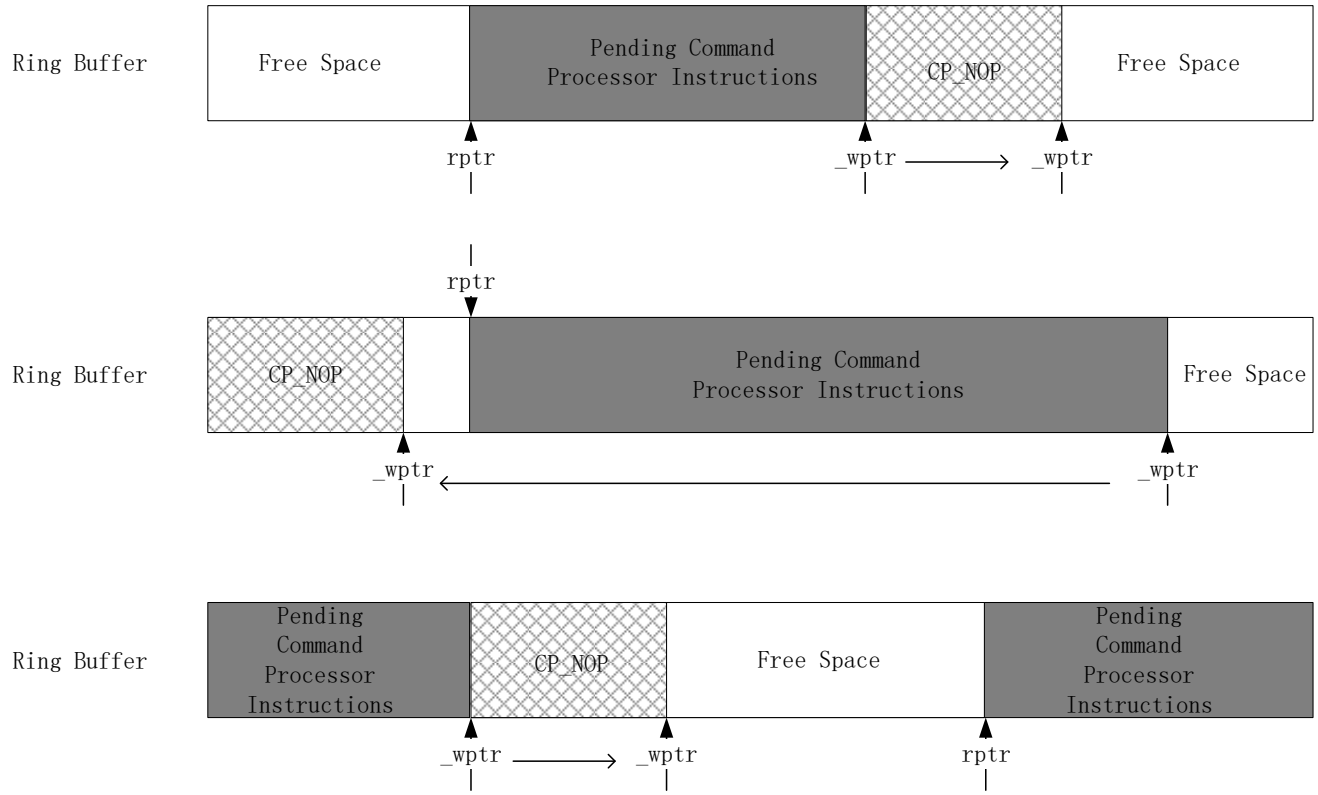

Figure 8. Allocating Space from Ring Buffer

As mentioned before, RPTR in “scratch buffer” can be set to any value by normal GPU instruction. So we can fool the function adreno_ringbuffer_allocspace. As Figure 9 shows, assume that after GPU executing some instructions, RPTR is less than _wptr. A request of allocating a large space from ring buffer is issued. Because the space at the begin and the end is

small, the request will fail. If we modify RPTR in "scratch" memory, for example advance it near _wptr, then some space will wrongly marked as free. At this moment, if another request of allocating a large space from ring buffer is issued, it'll succeed, but some existing Command Processor instructions will be overwritten. The Adreno GPU will execute instruction still from the original RPTR, resulting in executing unaligned instructions, which means executing from the middle of an instruction, treating some data of the instructions as opcode. So it's possible to inject arbitrary command processor instructions into ring buffer if we can write arbitrary data to ring buffer.

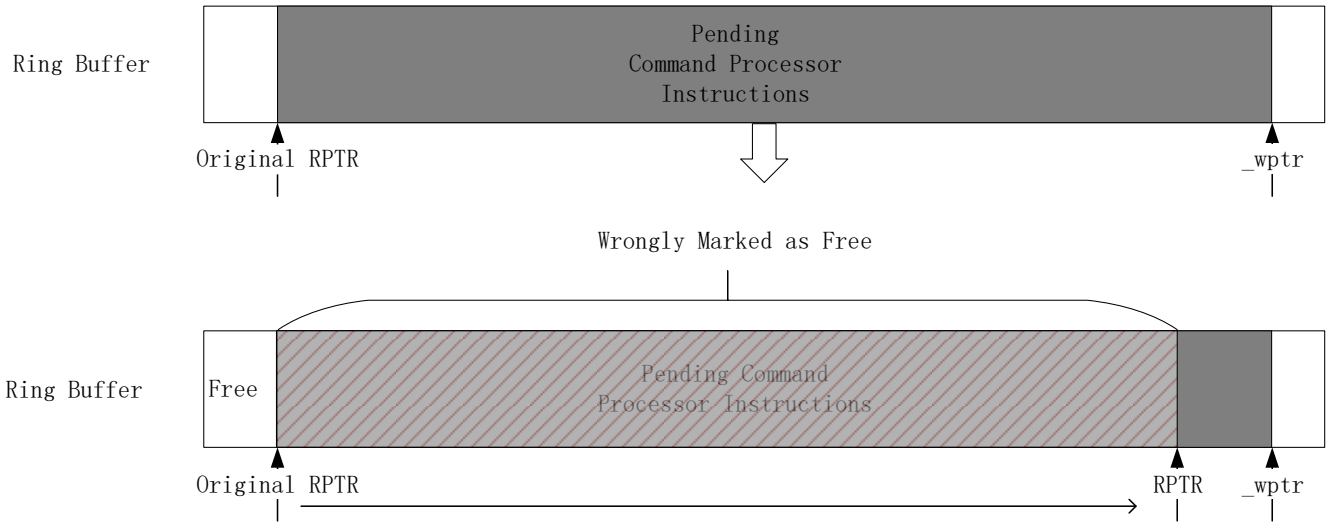

Figure 9. Overwrite Exist Instructions

When GPU instructions are dispatched to ring buffers, CP_INDIRECT_BUFFER_PFE instructions are inserted into ring buffers by KGSL driver. GPU instructions provided in the user mode are in indirect buffers. They are not copied to ring buffer directly. The execution flow will jump to indirect buffer when CP_INDIRECT_BUFFER_PFE instruction is executed. It's difficult to write arbitrary data into ring buffer. Fortunately, we found another bug in user profiling command. User profiling command is used to read the GPU ticks at the start and the end of GPU command and write them into the appropriate profiling buffers. The GPU address of the profiling buffer is controllable by user mode and can be set to any value [36]. GPU address is 8 bytes. It's enough to write an CP_NOP or CP_SET_PROTECTED_MODE instruction into it.

As shown in Figure 10, it's CP instruction sequence of executing IOCTL_KGSL_GPU_COMMAND one time. CP_INDIRECT_BUFFER_PFE instructions are wrapped by other instructions, such as instructions to enable and disable protected mode, instructions to start and end user profiling. If protected mode is disabled, GPU runs in privileged mode, many privileged registers can be modified. Protected mode can only be disabled by CP instructions which executes in ring buffer. CP instructions executed in indirect buffer have no permission to disable protected mode. There is a CP_SET_PROTECTED_MODE instruction enabling protected mode before jump to indirect buffer, so, the CP instructions in indirect buffer are executed in unprivileged mode normally.

As mentioned before, we can execute a CP instruction from the middle. We can exploit the bug as shown in figure 11. Before we trigger the bug, the layout of the ring buffer 3 is shown on the left. The first CP_INDIRECT_BUFFER_PFE will jump to an indirect buffer, in which there is a CP_WAIT_REG_MEM instruction. The CP_WAIT_REG_MEM instruction will wait until the value in a specific GPU address to become a specific value. At this moment, the condition is not satisfied. So RPTR will point to the second CP_INDIRECT_BUFFER_PFE instruction. Then we trigger the bug in the highest priority (ring buffer 0) Adreno context, so we can preempt the wait instruction to modify RPTR near wptr, which deceives the GPU kernel driver into thinking the space before RPTR is all free. Then we execute IOCTL_KGSL_GPU_COMMAND two times to overwrite the ring buffer as shown in the middle. The GPU address in the second and the fourth user profiling command are carefully designed as a CP_NOP instruction and a CP_SET_PROTECTED_MODE instruction. If the wait condition is satisfied at this moment, after executing some prefetched instructions, the CP_NOP will be executed as shown on the right of figure 11. The CP_NOP will nop all the instructions until the CP_SET_PROTECTED_MODE instruction. the CP_SET_PROTECTED_MODE instruction will disable protected mode and the following CP_INDIRECT_BUFFER_PFE instruction will jump to indirect buffer with protected mode off. Now we can change TTBR to any value by instructions in

indirect buffer. It's a powerful primitive. We can exploit it to read and write arbitrary physical memory including the code segment in kernel. It's easy to exploit it to get arbitrary kernel code execution.

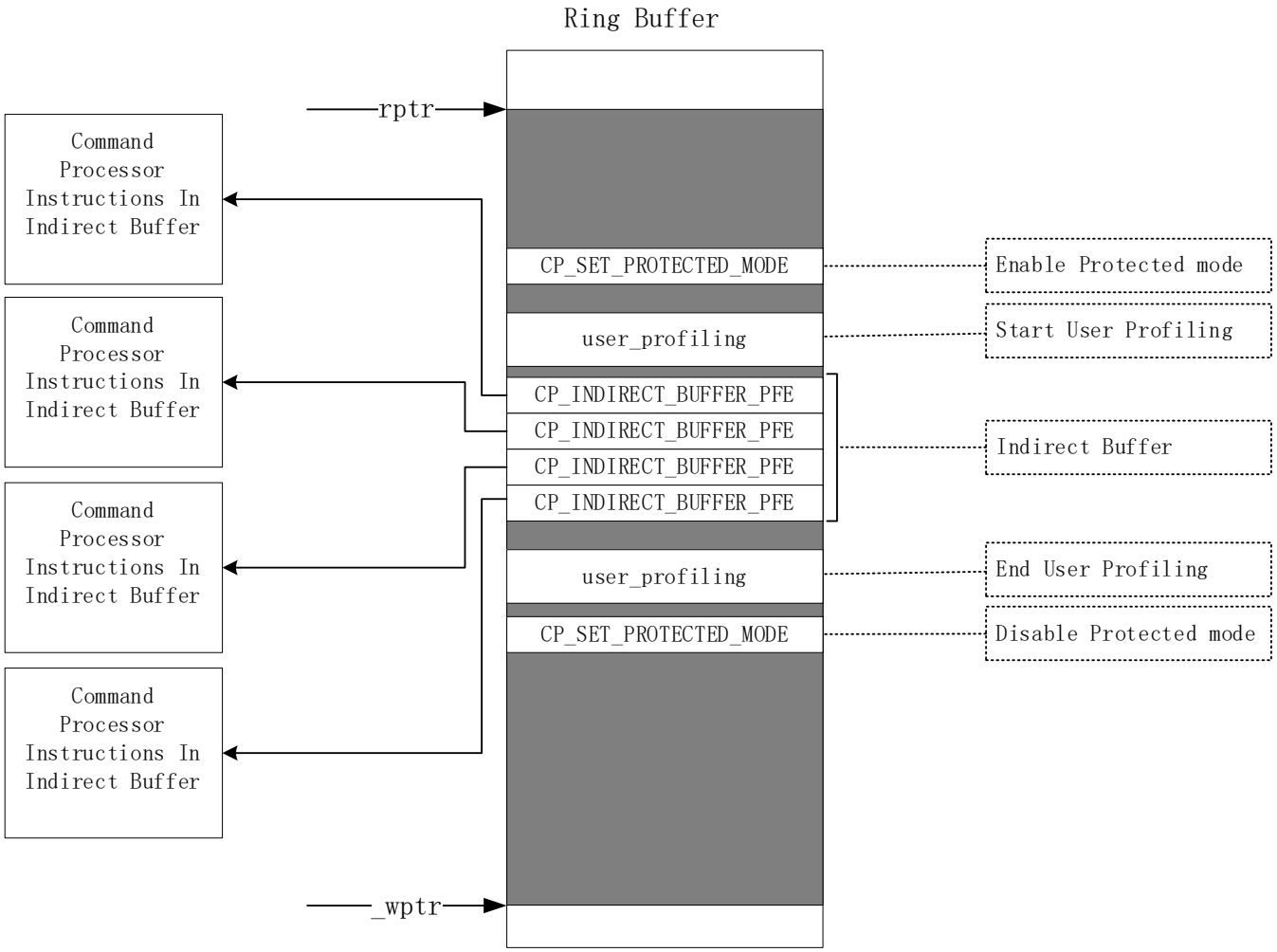

Figure 9. CP Instruction Sequence of Executing IOCTL_KGSL_GPU_COMMAND One Time

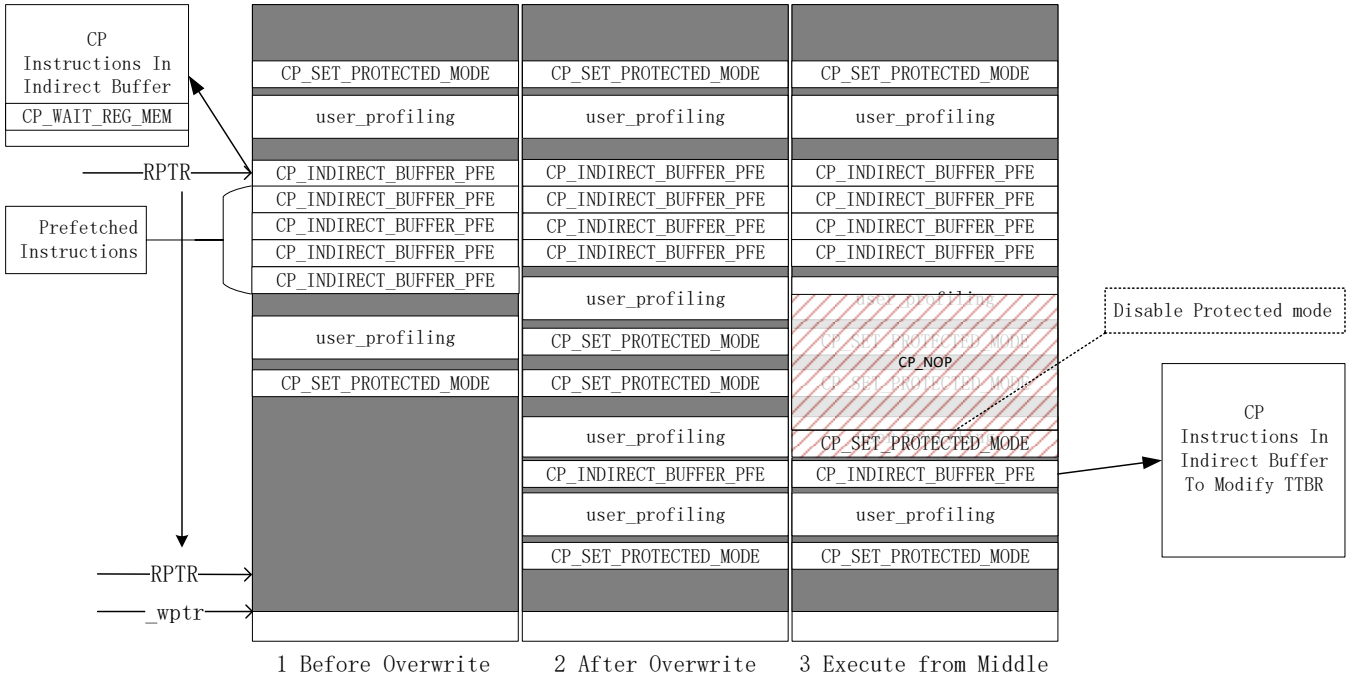

Figure 10. The process of exploiting CVE-2019-10567

CONCLUSION

In this paper, I explained why Google pixel phone is one of the most secure smartphones. Then I introduced the remote attack surface of Android phones and 6 Android exploit chains found by me over recent years. After that, we overviewed the exploit chain which can remotely root the modern Android devices including Google pixel phones. At last, I detailed three vulnerabilities in the chain and how I exploit them.

The exploit chain is the first reported one-click remote root exploit chain on Pixel devices that Google received from security researchers. In the exploitation process of the exploit chain, despite the fact that ROP are the most common techniques to exploit complicated vulnerabilities, we found an effective and stable approach to chain these three vulnerabilities for exploitation without any ROP. The specific approach is that when exploiting CVE-2019-5877, we used an attack on data, and exported the JavaScript interface of Mojo Call instead of hijacking the control flow. For CVE-2019-5870, we used a simpler "return to libc" technology other than ROP, which greatly reduces the complexity of the exploitation. When attacking with CVE-2019-10567, we used the "Command Processor Instruction Injection" technology as a novel approach. We successfully modified the code pages without writable attribute by modifying the physical memory with the use of GPU instructions to hijack control flow. Some features of ROP, such as the frequency and intensity of returned instruction calls, are often used for 0day detection. Therefore, it will make our attack more covert to be detected without using ROP technology.

REFERENCES

1. https://security.googleblog.com/2019/11/expanding-android-security-rewards.html
2. https://www.thezdi.com/blog/2017/11/1/the-results-mobile-pwn2own-day-one
3. https://www.thezdi.com/blog/2018/11/13/pwn2own-tokyo-2018-day-one-results
4. https://www.thezdi.com/blog/2019/11/6/pwn2own-tokyo-2019-day-one-results
5. https://www.blackhat.com/docs/us-17/thursday/us-17-Artenstein-Broadpwn-Remotely-Compromising-Android-And-iOS-Via-A-Bug-In-Broadcoms-Wifi-Chipsets.pdf
6. https://www.blackhat.com/docs/eu-17/materials/eu-17-Seri-BlueBorne-A-New-Class-Of-Airborne-Attacks-Compromising-Any-Bluetooth-Enabled-Linux-IoT-Device-wp.pdf
7. https://cansecwest.com/slides/2016/CSW2016_Gong_Pwn_a_Nexus_device_with_a_single_vulnerability.pdf
8. https://twitter.com/Pwn0R/status/712537388849963009
9. https://github.com/secmob/pwnfest2016
10. https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf,29-29.
11. https://www.chromium.org/Home/chromium-security/site-isolation
12. https://cansecwest.com/slides/2018/Attacks%20and%20Analysis%20of%20the%20Samsung%20S8%20from%20Mobile%20PWN2OWN%20-%20Guang%20Gong%20and%20Jianjun%20Dai,%20Qihoo%20360.pdf
13. https://github.com/secmob/mosec2016
14. https://android-developers.googleblog.com/2018/01/android-security-ecosystem-investments.html
15. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/private/isolated_app.te
16. https://chromium.googlesource.com/chromium/src.git/+/master/docs/mojo_and_services.md
17. https://android.googlesource.com/platform/system/sepolicy/+/refs/heads/master/public/app.te
18. https://www.qualcomm.com/company/product-security/bulletins/february-2020-bulletin
19. https://v8.dev/docs/csa-builtins
20. https://v8.dev/docs/torque-builtins
21. https://v8.dev/docs/torque
22. https://cs.chromium.org/chromium/src/v8/src/builtins/base.tq?rcl=568f3984d3ead0863deb3e84eec4c0ccd33a4936&l=372
23. https://googleprojectzero.blogspot.com/2019/04/virtually-unlimited-memory-escaping.html
24. https://www.chromium.org/developers/design-documents/multi-process-architecture
25. https://chromium.googlesource.com/chromium/src.git/+/master/docs/mojo_ipc_conversion.md
26. https://source.chromium.org/chromium/chromium/src/+/master:media/mojo/mojom/content_decryption_module.mojom
27. https://cs.chromium.org/chromium/src/media/mojo/services/mojo_cdm_service.cc?rcl=a64ec63d6caf3838818b97a49dd95950f29ef6ad&l=58
28. https://cs.chromium.org/chromium/src/media/mojo/services/mojo_cdm_service.cc?rcl=a64ec63d6caf3838818b97a49dd95950f29ef6ad&l=140
29. https://source.chromium.org/chromium/chromium/src/+/master:media/mojo/services/mojo_cdm_service_context.cc;l=72;drc=3bcb70cef58efe3a14d211aff71e72e2d402c894
30. https://source.chromium.org/chromium/chromium/src/+/master:media/mojo/services/mojo_cdm_service_context.cc;l=103;drc=3bcb70cef58efe3a14d211aff71e72e2d402c894
31. https://source.chromium.org/chromium/chromium/src/+/master:media/mojo/services/mojo_cdm_service.h;l=100;drc=ee4cff87f02e46e1fbbdaef0aa123e05761b35e8

32. https://source.chromium.org/chromium/chromium/src/+/master:media/base/video_decoder_config.h;drc=becc5bbb0aa6233f60a2daf65d5e1704b2f63d46;l=188
33. https://developer.qualcomm.com/software/adreno-gpu-sdk/gpu
34. https://lwn.net/Articles/394665/
35. http://bloggingthemonkey.blogspot.com/2014/06/fire-in-root-hole.html
36. https://source.codeaurora.org/quic/la/kernel/msm-4.14/commit/?id=fb37ff663a3d28e3a07549b074c54feb3e4376b5