VB2020
localhost

30 September - 2 October, 2020 / vblocalhost.com

HUNTING FOR ANDROID 1-DAYS: ANALYSIS OF ROOTING ECOSYSTEM

Eugene Rodionov, Richard Neal & Lin Chen
Google, USA

rodionov@google.com
rmneal@google.com
larchchen@google.com

ABSTRACT

With every new release of *Android* OS it becomes increasingly difficult to gain root privileges on modern devices with locked bootloaders due to improvements and new features in *Android* platform security. However, there still exist a number of applications that offer one-click rooting solutions. Some of the largest rooting providers offer rooting as a service via rooting SDKs too. Usually, such applications exploit unpatched 1-day vulnerabilities present in certain *Android* platforms to gain root privileges. This is a viable strategy since many *Android* devices in the wild are not anywhere close to the most recent security patch level.

During this research the authors took a deep look into the biggest rooting providers targeting modern versions of the *Android* platform (*Android 7.0* and higher) with the aim of better understanding the rooting ecosystem, which vulnerabilities are being used by these applications, and what devices/platforms they are targeting.

In this presentation the authors will share the results of the long-term monitoring of one of the largest rooting providers for *Android* devices: Kingroot. They will provide details of Kingroot's modus operandi, including how to reverse engineer a sophisticated network communication protocol with a C2 server to download the exploits, followed by analysis and deobfuscation of collected payloads. Additionally, the authors will provide analysis of the rooting exploits for various device models and *Android* builds that they managed to obtain in the course of Kingroot monitoring.

To conclude the presentation, the authors will speak about what *Google* is doing to protect *Android* users from unpatched one-days.

INTRODUCTION

Android rooting allows users to gain privileged access to their devices by breaking the *Android* security model. Such demand from users for having complete control over their devices has created an ecosystem of applications that provide rooting services, in particular coming out of China. There is a subset of rooting applications that exploit privilege escalation vulnerabilities to achieve root on the target device, especially for devices with locked bootloaders. Some of the rooting applications integrate third-party SDKs that provide rooting services targeting a wide range of *Android* devices and support exploitation of multiple privilege escalation vulnerabilities.

In this research the authors focused on a rooting provider used in one of the most popular contemporary rooting applications – Kingroot. It claims to support an extensive list of *Android* devices, offering one-click rooting solutions for them. One of the main goals of this research was to gain visibility into which vulnerabilities are exploited by the Kingroot application and to obtain a comprehensive list of the exploits used and the device configurations that they are targeting. The authors hoped to use these insights to improve the exploit detection capabilities of *Google Play Protect* and if any 0-days were found, to get them patched in AOSP. To accomplish these goals the authors reverse engineered Kingroot, and reconstructed its command-and-control (C&C) network communication protocol to be able to download exploits used by the application to root the device.

The rest of the paper is organized as follows. It begins with an overview of the Kingroot application and its modus operandi. Next, the authors provide details on the network communication algorithm Kingroot uses to fetch exploits from the C&C server. The section 'Payload Analysis' provides information on exploits the authors managed to obtain from the C&C server and the device configurations they target. In the 'Remediation' section the authors provide concluding remarks on how information obtained in this research is used to protect *Android* users from unpatched 1-days.

KINGROOT CASE STUDY

Kingroot is one of the most popular rooting applications[1] targeting contemporary *Android* devices and offers two types of rooting solutions: first, an application running on a desktop computer and communicating with the target device over USB, PC root, and second, an *Android* application running on the target device, mobile root. The latter offers one-click rooting services – a user just needs to download the application and it will manage the rooting process from exploiting a privilege escalation vulnerability to gaining root privileges, up to installing a root manager on the device along with other rooting tools. The mobile root version of Kingroot is the main target of this investigation.

From a high-level point of view, the Kingroot application primarily consists of the following components:

- UI code – part of the application that interacts with a user.
- Rooting SDK – a jar file that implements functionality for downloading exploits targeting privilege escalation vulnerabilities from Kingroot's C&C server and running them.
- Rooting tools – a collection of tools for managing the rooted device, such as root manager, su, etc.

The rooting SDK – one of the core components of the rooting application – is stored encrypted in the application's asset files. It is decrypted and dynamically loaded during application execution upon a request from the user to root the device.

---

[1] As of December 2019 Kingroot announced it was shutting down its services. However, the application is still available for download on third-party *Android* markets.

The SDK comes with a licence file which contains information on applications that are authorized to use it. Once loaded, the SDK obtains the package name and hash of the signer certificate of the host application and checks this information against values in the licence. This suggests that this may be a third-party rooting SDK inside the Kingroot application. The rooting SDK checks the authenticity of the licence file by verifying its RSA signature using a hard-coded public key.

To root the target device, Kingroot performs the following steps:

1. Drops rooting tools into its internal directory.
2. Fingerprints the target device and requests a list of rooting solutions (a.k.a. exploits) from the C&C server for the particular device configuration of the target device.
3. Downloads and executes rooting solutions exploiting privilege escalation vulnerability(ies) targeted to the particular target device.
4. Upon successful exploitation of a vulnerability, installs rooting tools onto system and recovery partitions.

In the following section the authors provide information on the network protocol that Kingroot uses to communicate with its C&C servers to download a solution (i.e. exploit) targeting the client's device.

NETWORK COMMUNICATION PROTOCOL

The implementation of the protocol in question is inherently multi-threaded, event-driven, and is specifically designed to provide a robust communication channel in event of a sudden loss of connection due to a number of factors specific to mobile devices, such as: loss of a signal, *Android* task prioritization, low battery, etc. Additionally, the protocol also ensures confidentiality of the data transmitted over the network by using symmetric and asymmetric encryption.

Message layout

The messages Kingroot exchanges with its C&C server are called ClientShark[2] (transmitted from the application to the server) and ServerShark (received by the application from the server). Figure 1 demonstrates the layout of a ClientShark message.

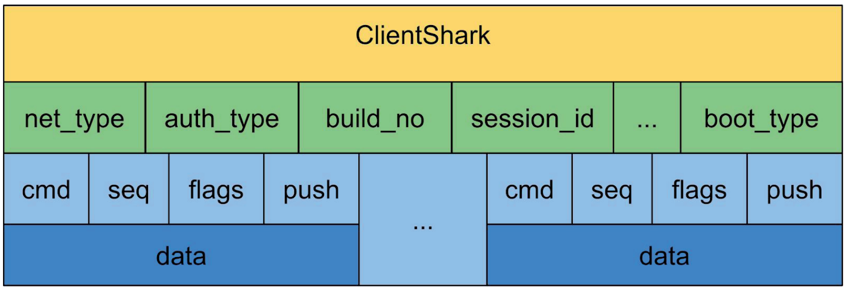


Figure 1: Layout of a ClientShark message.

The fields of a ClientShark message have the following purpose:

- **net_type**, **auth_type** – integer, the type of the network connection: Wi-Fi, mobile network.
- **build_no** – integer, build number that identifies the implementation of the network protocol.
- **session_id** – unique randomly generated 16-byte identifier of the session between the Kingroot application and the C&C server.
- **cmd** – integer, an identifier of the command/message type.
- **seq** – integer, a sequence number of the message in the communication protocol (i.e. identifier of the protocol transaction). This field is used to match requests and responses from Kingroot and the C&C server respectively.
- **flags** – integer, determines if the payload transmitted in the message is encrypted and/or compressed.
- **push** –integer, used by the C&C server to send out-of-band commands/messages to the Kingroot application.
- **data** – byte array, the actual payload transmitted from/received by the Kingroot application.

Kingroot uses a custom TLV (tag-length-value) scheme to encode the fields referenced above in a ClientShark message. For instance, it has separate tag-length values for the following types: Byte, Short, Int, Long, Float, Double, String, Map, Array, List and Byte array. Complex data types such as class objects are deserialized into ClientShark messages using Map type (i.e. it maps the names of object fields to their corresponding values). For integers, the scheme attempts to encode the value using the least amount of bytes. As a result, the actual size of the encoded integer field within the message depends on the actual value of the field.

---

[2] The actual names used in the paper correspond to the names of Java classes in the application that the authors analysed in the course of this research.

Protocol setup stage

Communication between Kingroot and the C&C server can be split into two stages: setup and payload fetching. During the setup stage Kingroot establishes a session with the C&C server and negotiates certain parameters like the session identifier and encryption key to protect confidentiality of the data. Figure 2 demonstrates a sequence of messages that are exchanged between Kingroot (C, on the left-hand side of the figure) and the server (S, on the right-hand side of the figure) during that stage.

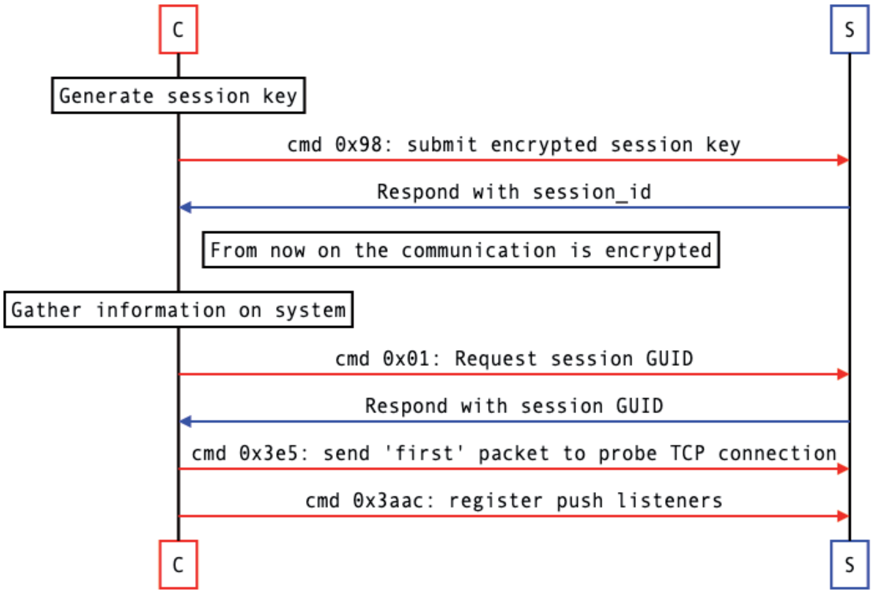


Figure 2: A sequence of protocol transaction during setup stage.

Before sending out the very first ClientShark message, Kingroot generates a 16-byte symmetric key to protect the confidentiality of the data transmitted/received to/from the C&C server using the XTEA cipher. If we look at the implementation of the key generation routine (as shown in the listing below) one can see that it uses an object of java.util.Random[3] type to provide entropy for the new key. This effectively reduces the entropy of the generated key from 128 bits to 48 bits.

```
private String a(int p6) {
   java.util.Random v2_1 = new java.util.Random();
   StringBuffer v3_1 = new StringBuffer();
   while (v0_0 < p6) {
     v3_1.append("abcdefghijklmnopqrstuvwxyzABCDEF" +
                 "GHIJKLMNOPQRSTUVWXYZ0123456789".charAt(v2_1.nextInt(62)));
     v0_0++;
   }
   return v3_1.toString();
}
```

Once the session key is generated Kingroot encrypts it using a 1024-bit RSA algorithm with the C&C public key hard coded in the application dex code and sends the result to the server (cmd=0x98). To confirm, the C&C server replies with a message containing the session ID. From this point on, the communication between Kingroot and the C&C server is encrypted using the XTEA cipher and the generated key.

Fingerprinting Android devices

As the next step in the setup stage Kingroot requests a session GUID from the C&C server, which identifies a type of the communication session. To request the GUID Kingroot gathers extensive information about the hardware, firmware and software configuration of the device and sends it to the server. The list below contains some of the information collected by Kingroot:

- Unique device ID – IMEI for GSM and MEID or ESN for CDMA
- Wi-Fi MAC address
- Android ID

---

[3] Class java.security.SecureRandom provides a cryptographically strong random number generator.

- CPU information -- cat /proc/cpuinfo, number of cores, maximum CPU frequency
- Screen size of the device
- Amount of available memory --/proc/meminfo
- Total size and amount of available space on system and data partitions
- Total size and amount of available space on the external storage
- Device build information
- Version of baseband firmware
- Device brand, manufacturer, product name and release version number
- etc.

Exploit fetching stage

Once Kingroot has obtained a session ID and session GUID during the setup stage it proceeds with obtaining information on exploits available for the configuration of the user's device. As shown in Figure 3, the exploit fetching happens in two steps:

- Request for a list of exploits that target the configuration – ClientShark message with cmd=0x14f3
- Request for statistics (number of successful exploitations/rootings) on exploits provided in the previous step; this information is used to prioritize exploits received at the previous step – ClientShark message cmd=0x14f6

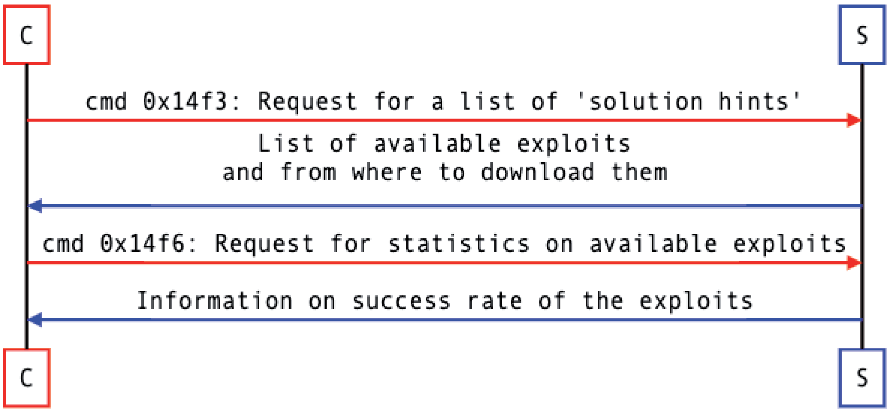


Figure 3: A sequence of protocol transactions during exploit fetching stage.

Along with ClientShark message 0x14f3 Kingroot submits information about the target device. This device fingerprinting information is different from what was previously sent to the C&C server during the setup stage. This time Kingroot sends the contents of the /proc/version file and the value of the android.os.Build.FINGERPRINT field to uniquely identify the build of the system.

The response from the C&C server contains an array of XML data structures describing exploits to download. Each element in the array describes a single exploit and contains a URL from where the rooting SDK downloads the actual binary.

PAYLOAD ANALYSIS

Understanding of the C&C network communication protocol enabled the authors to reimplement it in a laboratory environment and allowed them to obtain multiple exploits targeting various configurations of *Android* devices from the Kingroot C&C server. The rest of the paper is devoted to analysis of the payloads – downloaded exploits.

Payload containers

The content of Kingroot payloads has varied over time, though newer payloads tend to conform to the scheme described here. The downloaded payload is a JAR file, which contains an ELF executable called *krmain*. *Krmain* is a 32-bit executable. It performs some environmental checks, and if these pass, it unpacks further files from its *.data* section. The files inside the *.data* section can be stored in raw byte format or gzip-compressed TAR files (named *mypack.tar*). Files are often stored as a byte array, followed by an integer containing the byte length. This allows a form of automated brute-force scanning to be used to identify likely files, which can then be extracted and examined. Older payloads have different numbers and types of embedded files and some do not have the content/size variables nicely ordered to support automated extraction. Manual analysis has to be used in some cases to extract the payloads.

Looking at the additional files, there is usually another ELF named for the exploit, and a configuration file with parameter information to allow the exploit to work on different devices. A TAR file usually contains post-rooting-related utilities.

The exploit ELF can itself contain further payloads stored in the same manner, but these do not seem to play any role in exploiting the device. These are completely scrambled using what appear to be pseudo-randomly generated perturbation tables.

Simple scrambling of the first four bytes of a file's data can occur:

```
def _unscramble_data(data):
   """Unscramble the given data.

   Args:
      data: The data to unscramble.  This is modified in-place.

   Returns:
      Unscrambled version of the data.
   """
   raw_data = bytearray(data)
   # None of these bytes can be zero, as the % operator won't work. So if they
   # are, just return the data.
   for index in range(0, 4):
      if raw_data[index + 4] == 0:
         return raw_data
   # Only the first four bytes are scrambled.
   for index in range(0, 4):
      scramble_key = len(data) % raw_data[index + 4]
      current_value = raw_data[index]
      raw_data[index] = ((current_value & ~scramble_key) |
                         (scramble_key & ~current_value)) & 0xff
   return raw_data
```

This scrambling is enough to change any identifying magic numbers, for example for GZip or ELF headers.

Most, but not all, of the ELF binaries are compiled with obfuscator-LLVM. There seems to be a lot of common code between instances of *krmain* and also pre- and post-rooting activities in the actual exploit binaries, so this helps understand the behaviour of the obfuscated files.

Payload configuration file

The payload configuration files are usually called *katana*, though a small number of other names have been observed. They follow the same general schema, though the exact record format is different for each exploit. A file consists of a series of identically sized records, each of which contains a device identifier and configuration data for that device. Two examples of configuration records for different exploits are as follows:

```
00000000  41 4b f2 a1 03 de 15 ed 88 44 b4 50 59 ad 78 32  |AK      D.PY.x2|
00000010  21 c4 d6 77 6c 25 92 0d d9 cf b7 5a 4f 6f e9 d2  |!..wl% ZOo..|
00000020  cc d2 13 00 c0 ff ff ff a0 5d 08 00 c0 ff ff ff  |.........].......|
00000030  0c ba 17 00 c0 ff ff ff d0 7e b6 00 c0 ff ff ff  |.........~.......|
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000090  70 83 6c 01 c0 ff ff ff 02 00 00 00 00 00 00 00  |p.l    |
000000a0  e9 2f 4d f2 a8 6c 42 9a 92 3e 9e d3 d7 93 77 22  |./M..lB..>    w"|
000000b0  d6 b7 4a a9 d4 85 a0 ad 79 bc 63 4c 47 6b 92 75  |..J    y.cLGk.u|
000000c0  cc d2 13 00 c0 ff ff ff a0 5d 08 00 c0 ff ff ff  |.........].......|
000000d0  0c ba 17 00 c0 ff ff ff 5c 6c b6 00 c0 ff ff ff  |........\l     |
000000e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
000000f0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000130  68 83 6c 01 c0 ff ff ff 02 00 00 00 00 00 00 00  |h.l    |
```

In this first example, two records are shown. The records are 0xa0 bytes in size. Each record begins with 32 bytes of device identification data, and is followed by exploit-specific data. In this case, the data appears to be four kernel addresses, ten empty values, another kernel address, and a small integer.

```
00000000  a4  34  fd  45  15  58  f3  67  69  35  0c  dd  88  c8  f1  ff  |.4.E.X.gi5      |
00000010  24  2d  65  c4  85  bd  4c  58  51  7d  95  3c  13  fc  a3  d0  |$-e...LXQ}.<    |
00000020  88  b4  1a  01  c0  ff  ff  ff  a0  b3  1a  01  c0  ff  ff  ff  |................|
00000030  44  b7  37  00  c0  ff  ff  ff  3c  9e  93  00  c0  ff  ff  ff  |D.7.....<       |
00000040  78  9e  93  00  c0  ff  ff  ff  d0  3f  1b  01  c0  ff  ff  ff  |x.......?       |
00000050  0c  f1  19  01  c0  ff  ff  ff  90  bf  fb  00  c0  ff  ff  ff  |................|
00000060  a6  bf  4f  f5  06  b5  02  1e  8a  e7  c6  ec  0a  5b  aa  52  |..O     [.R|
00000070  26  25  e3  25  5e  46  85  5e  be  3d  ae  c3  a4  31  7a  0c  |&%.%^F.^.=...1z.|
00000080  88  84  19  01  c0  ff  ff  ff  a0  83  19  01  c0  ff  ff  ff  |................|
00000090  44  b7  37  00  c0  ff  ff  ff  68  53  93  00  c0  ff  ff  ff  |D.7.....hS      |
000000a0  a4  53  93  00  c0  ff  ff  ff  d0  0f  1a  01  c0  ff  ff  ff  |.S      |
000000b0  0c  c1  18  01  c0  ff  ff  ff  80  7f  fb  00  c0  ff  ff  ff  |................|
```

The second example also shows two records, which in this case are 0x60 bytes in size. They again start with 32 bytes of device identification data, but the exploit data is eight kernel addresses.

One exploit with a different format file, called *lollipop*, has been seen. This has the potential for records of different fixed sizes; as one is twice the size of the other, we conjecture that this is to support 32- and 64-bit configuration in the same file. The file we have only contains records of the larger size, however.

The device identification data is generated from information about the device and the kernel it is running. This allows Kingroot to parameterize exploits to support a range of devices without recompiling the main payload. The device identification is worked out as follows:

```
SHA256(
        FORMAT("%s|%s|%.1023s",
                device_brand,
                device_model,
                kernel_version))
```

Device brand and model are taken from device properties, the kernel version string from */proc/version*. The exploit binary generates the required hash, and then reads the configuration file linearly either until it finds a match or until there are no more entries.

Given sufficient examples of the information triple required to calculate the device identification hash (ideally collected over time in order to see the different kernel version strings), it is possible to create a lookup table of hashes and therefore see which devices/kernels are supported by particular exploits from Kingroot. The support period for a particular device, as indicated by the kernel versions, can indicate when the exploit was patched (assuming that device vendors apply patches promptly, and that Kingroot would maintain support for popular devices as far as possible).

Interestingly, we can usually only identify around 50% of the devices in any given configuration file. Obviously this shows some form of shortcoming in our lookup table, but currently we do not have a definitive reason for why we do not have the brand/model/version for so many devices. One guess is that Kingroot is targeting the Chinese device ecosystem and that many devices which are not *Android*-certified and which do not have *Google Play Protect* installed exist there.

Examples of device information for *Google* devices that were supported by a Kingroot exploit are:

- 0525a720c6afbc972d4bd24176a93d418d086cf24c402ba291b317020630877d
    - google
    - Pixel XL
    - Linux version 3.18.52-g0b28c9afaba8 (android-build@wphn10.hot.corp.google.com) (gcc version 4.9.x 20150123 (prerelease) (GCC)) #1 SMP PREEMPT Wed Jul 26 21:51:18 UTC 2017
- c12c1c2296df9c5130709b69919839f6839c99a4602051382171c2b9c4708d95
    - google
    - Pixel XL
    - Linux version 3.18.52-g99dda0323132 (android-build@wprf7.hot.corp.google.com) (gcc version 4.9.x 20150123 (prerelease) (GCC)) #1 SMP PREEMPT Fri Aug 18 00:56:04 UTC 2017

In total we were able to create lookup entries for around 2.7 million sets of brand/model/kernel information. This data allows us to identify 598 unique devices (a combination of device brand and device model) across 42 device brands targeted by 12 Kingroot exploits over a time range from 2013-07 through 2018-10.

Different Kingroot exploits support very different numbers of devices/kernels. The exploit supporting the fewest devices was *m4u*, which contained 73 device configurations, of which 52 came from just four different hardware brands. The highest observed number of supported device configurations in a single exploit was 5,482 for *tga*, of which 3,610 device types were identified. The information on the exploits is provided in the Exploit Analysis section of the report.

One notable exception to the presence of the device configuration file are the exploits for CVE-2016-5195, a.k.a. DirtyCow. This vulnerability is a race condition resulting in incorrect permissions being applied to memory pages, so kernel addresses are not required to exploit this.

Visualizing vulnerability

Given the existence of the device information lookup table, which contains kernel compilation date/time information, an attempt can be made to visualize available vulnerabilities for a particular device or the identified devices as a whole.

The compilation time/date of the last supported kernel for a device is interesting, as this potentially shows the last kernel that was vulnerable to a particular exploit, i.e. the next build had a patch applied and the exploit could not be made to work. If Kingroot's device support was driven by user requests for capability against particular devices/kernels, this could also indicate a drop in demand leading to a reallocation of resources. It seems unrealistic that user demand for rooting support on multiple devices from a given brand would drop simultaneously though.

The date/time of the earliest vulnerable kernel does not indicate when the vulnerability was discovered, as support for older vulnerable kernels could be added at any time after the discovery depending on user demand. If devices are not receiving updates, or users are not updating their devices, then it may be worthwhile to backport an exploit to older devices/kernels.

Looking specifically at the *tga* exploit in Figure 4, the last supported kernels on *Google* devices for this exploit (and hence potentially the last unpatched kernel) were compiled in August 2017 (the first five rows in the figure). Devices from other brands were apparently still vulnerable into 2018.

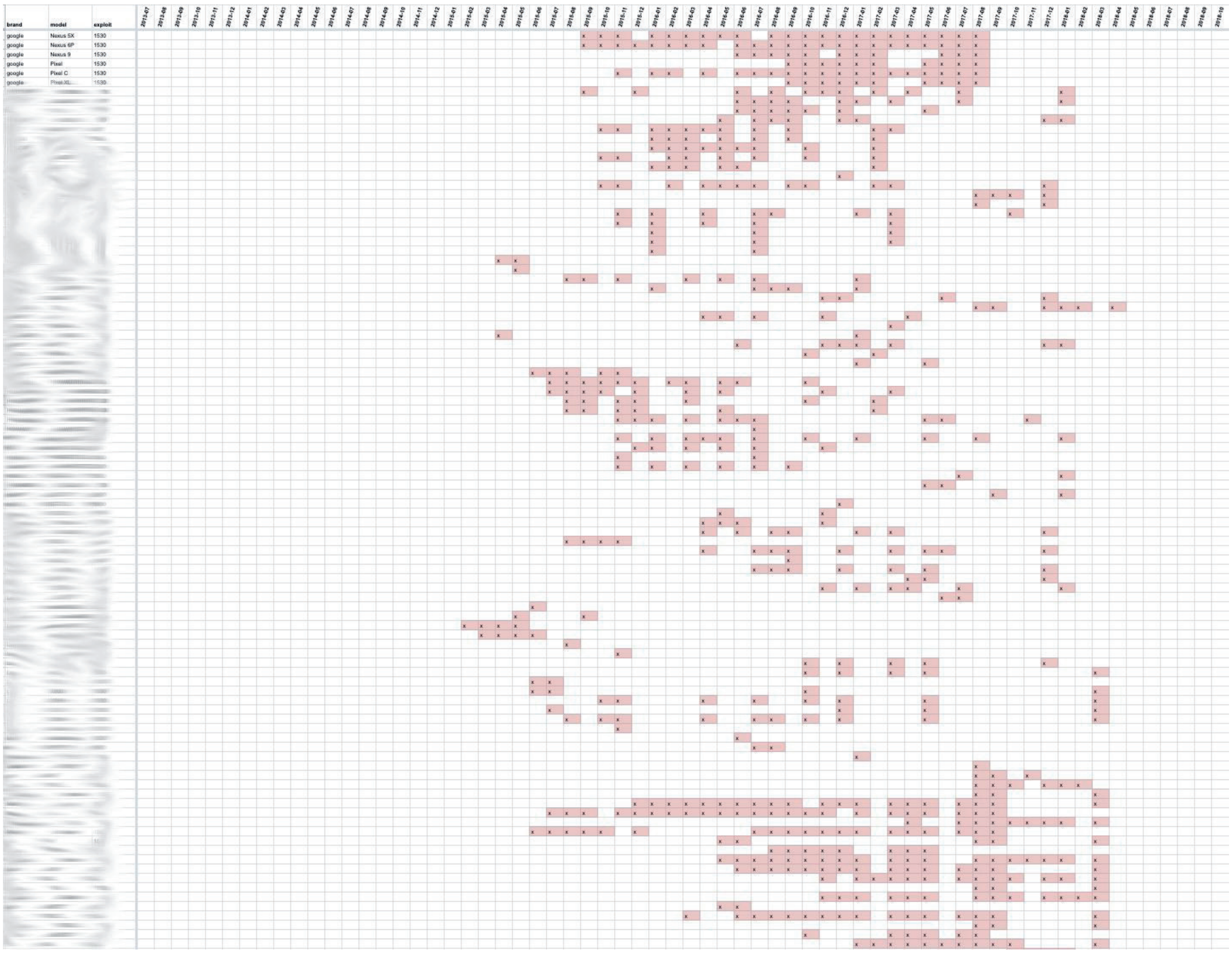


Figure 4: Comparison of Kingroot 1530/tga support for Google and some other OEM devices (2013-07 through 2018-10).

The situation for patching 1545/hecate was worse; some devices were vulnerable up to October 2018. This implies that many device types sold in 2013 did not receive security updates after 2017 when hecate was fixed. We also believe that these devices did not get updates in 2018 but most likely use of these devices dropped off rapidly after five years so the Kingroot developers stopped adding support for them at this point.

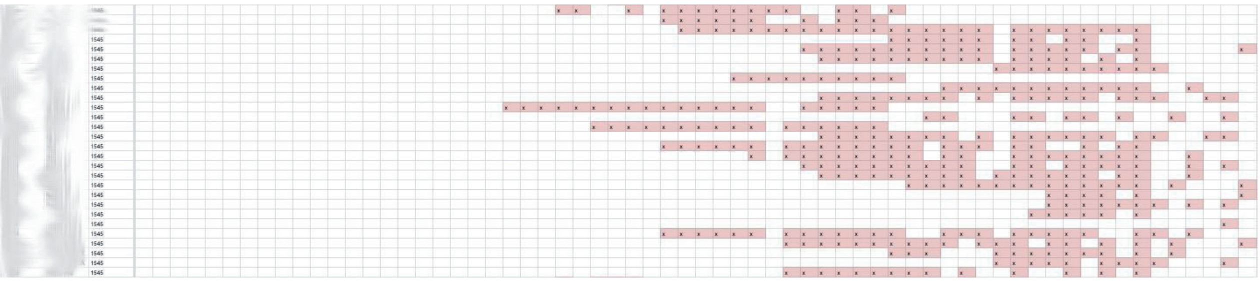


Figure 5: Kingroot 1545/hecate support for OEM A devices (2013-07 through 2018-10).

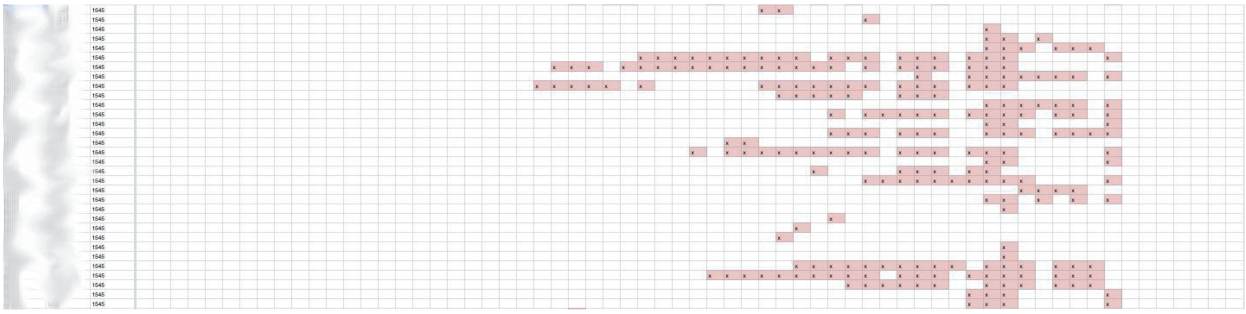


Figure 6: Kingroot 1545/hecate support for OEM B devices (2013-07 through 2018-10).

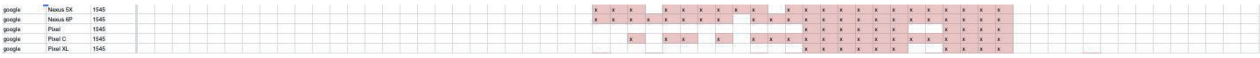
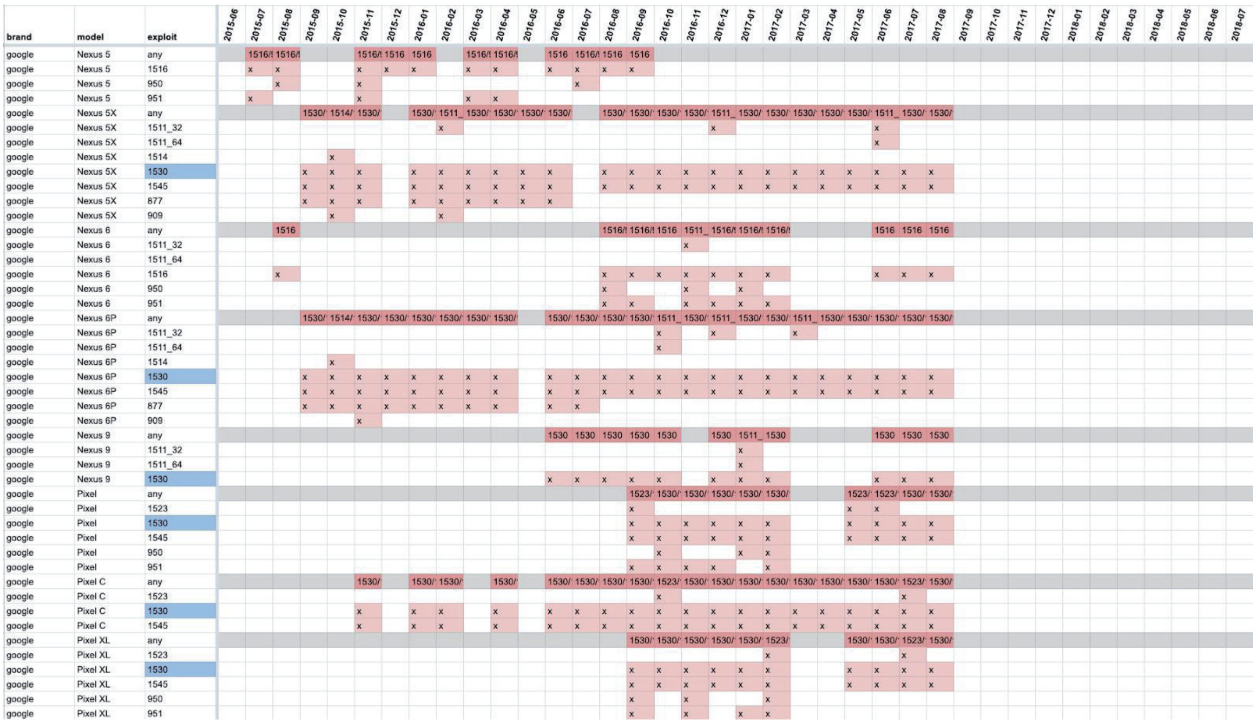


Figure 7: Kingroot 1545/hecate support for Google devices (2013-07 through 2018-10).

Google Nexus and *Pixel* devices were fairly well supported by Kingroot early on, however there was then no support after the third quarter of 2017, suggesting major improvements to the resilience of *Google* devices to privilege escalation exploits around that time.



brand	model	exploit	2015-06	2015-07	2015-08	2015-09	2015-10	2015-11	2015-12	2016-01	2016-02	2016-03	2016-04	2016-05	2016-06	2016-07	2016-08	2016-09	2016-10	2016-11	2016-12	2017-01	2017-02	2017-03	2017-04	2017-05	2017-06	2017-07	2017-08	2017-09	2017-10	2017-11	2017-12	2018-01	2018-02	2018-03	2018-04	2018-05	2018-06	2018-07
google	Nexus 5	any		1516/	1516/			1516/	1516	1516		1516/	1516/		1516	1516/	1516	1516																						
google	Nexus 5	1516		x	x			x	x	x		x	x		x	x	x	x																						
google	Nexus 5	950			x			x								x																								
google	Nexus 5	951		x				x				x	x																											
google	Nexus 5X	any				1530/	1514/	1530/		1530/	1511_	1530/	1530/	1530/	1530/		1530/	1530/	1530/	1530/	1511_	1530/	1530/	1530/	1530/	1530/	1511_	1530/	1530/											
google	Nexus 5X	1511_32									x										x						x													
google	Nexus 5X	1511_64																									x													
google	Nexus 5X	1514					x																																	
google	Nexus 5X	1530				x	x	x		x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Nexus 5X	1545				x	x	x		x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Nexus 5X	877				x	x	x		x	x	x	x	x	x																									
google	Nexus 5X	909					x				x																													
google	Nexus 6	any			1516												1516/	1516/	1516	1511_	1516/	1516/	1516/				1516	1516	1516											
google	Nexus 6	1511_32																		x																				
google	Nexus 6	1511_64																																						
google	Nexus 6	1516			x												x	x	x	x	x	x	x				x	x	x											
google	Nexus 6	950															x			x		x																		
google	Nexus 6	951															x	x		x	x	x	x																	
google	Nexus 6P	any				1530/	1514/	1530/	1530/	1530/	1530/	1530/	1530/		1530/	1530/	1530/	1530/	1511_	1530/	1511_	1530/	1530/	1511_	1530/	1530/	1530/	1530/	1530/											
google	Nexus 6P	1511_32																	x		x			x																
google	Nexus 6P	1511_64																	x																					
google	Nexus 6P	1514					x																																	
google	Nexus 6P	1530				x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Nexus 6P	1545				x	x	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Nexus 6P	877				x	x	x	x	x	x	x	x		x	x																								
google	Nexus 6P	909						x																																
google	Nexus 9	any													1530	1530	1530	1530	1530		1530	1511_	1530				1530	1530	1530											
google	Nexus 9	1511_32																				x																		
google	Nexus 9	1511_64																				x																		
google	Nexus 9	1530													x	x	x	x	x		x	x	x				x	x	x											
google	Pixel	any																1523/	1530/	1530/	1530/	1530/	1530/			1523/	1523/	1530/	1530/											
google	Pixel	1523																x								x	x													
google	Pixel	1530																x	x	x	x	x	x			x	x	x	x											
google	Pixel	1545																x	x	x	x	x	x			x	x	x	x											
google	Pixel	950																	x			x	x																	
google	Pixel	951																x	x	x	x		x																	
google	Pixel C	any						1530/		1530/	1530/		1530/		1530/	1530/	1530/	1530/	1523/	1530/	1530/	1530/	1530/	1530/	1530/	1530/	1530/	1523/	1530/											
google	Pixel C	1523																	x									x												
google	Pixel C	1530						x		x	x		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Pixel C	1545						x		x	x		x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	x											
google	Pixel XL	any																1530/	1530/	1530/	1530/	1530/	1523/			1530/	1530/	1523/	1530/											
google	Pixel XL	1523																					x					x												
google	Pixel XL	1530																x	x	x	x	x	x			x	x	x	x											
google	Pixel XL	1545																x	x	x	x	x	x			x	x	x	x											
google	Pixel XL	950																x		x			x																	
google	Pixel XL	951																x		x		x	x																	

Figure 8: Overall Kingroot exploit support for Google devices (2015-06 through 2018-07).

Exploit analysis

Table 1 lists a number of exploit payloads that were obtained during the investigation of Kingroot, mapping the payload files to actual CVEs that they exploit. The blank fields in the table indicate missing values for the exploit file name and configuration file name in the payload container. The blank field for CVE indicates the actual vulnerability exploited by the payload hasn't yet been identified.

ID1[4]	ID2[5]	Identifier string	Exploit file name	Configuration file name	CVE
356	1000	PU			
363	1000	MS3			
437	1001	NFT			
671[6]					
812	1002	CVR			CVE-2013-6282
813	1002	MS2			
814	1002	FT	libfutex		CVE-2014-3153
840	1003	SDC	sdc32	N/A	CVE-2016-5195[7]
877	1002	VKE	valkyrie	445c0900	CVE-2016-6787
909	1043	YME	yumie64	katana	CVE-2015-1805
910	1043	YME	yumie		CVE-2015-1805
919	1002	ODC	[8]	N/A	Unsure
947	1003	M4U	m4u64	katana	Unsure[9]
950	1009	IOV	iov32	katana	CVE-2015-1805[10]
951	1010	IOV	iov32	katana	CVE-2015-1805
1512	1016	DTC	dirtyc0w64	N/A	CVE-2016-5195
1545	1001	HCT	hct64	katana	Unsure[11]
1511	1028	IZA	izanami	katana	CVE-2017-0403
1511	1028	IZA	izanami64	katana	CVE-2017-0403
1523	1003	MBS	mebius64	katana	CVE-2017-7533
1512	1005	MCW[12]	sdc32-mtk	N/A	CVE-2016-5195
1516	1004	ONE	one32	katana	CVE-2017-8890[13]
1513	1143	SDR	schrodinger	?[14]	CVE-2015-3636
1513	1143	SDR	schrodinger64	lollipop	CVE-2015-3636
1530	1023	TGA	tga64	katana	Unsure[15]
1514	1051	WKL	winkle	?[16]	CVE-2015-0569
1514	1051	WKL	winkle64	flintlock	CVE-2015-0569

Table 1: List of exploits obtained from Kingroot C&C server.

---

[4] This ID number comes from the download information.

[5] This ID number is hard coded as a string in the *krmain* dropper binary, as is the identifier string.

[6] We were not able to obtain this payload.

[7] VDSO-patching variant, persists by patching libc.

[8] This payload contains a shared object exporting JNI_OnLoad rather than an executable, together with a very small DEX file that loads the SO using a passed-in string for the name, and passes a string to a function in it.

[9] There have been a number of CVEs in the driver concerned (e.g. CVE-2017-0500 to CVE-2017-0506) though this exploit uses a different IOCTL to the known vulnerabilities – possibly patched as part of other fixes and never reported individually. All known vulnerable kernels were compiled before mid-2017.

[10] A modification of iovyroot [1].

[11] Unsure; probably patched in November 2017.

[12] Also referred to as MTKCOW.

[13] Exploit code is very similar to [2].

[14] The *krmain* file extracts as expected, however there is not a configuration file present.

[15] Unsure; probably patched in November 2017. Has strong similarities to CVE-2017-8890.

[16] The same situation as for *schrodinger.*

REMEDIATION

There are many CVEs reported and patched, but not all of these will be turned into exploits. Behavioural detection attempts to look at what a piece of code does (or in the case of static analysis, what it might be capable of doing), in order to make a decision about whether something is dangerous or not. Rooting exploits often need to interact with the device kernel in some way in order to affect the device, which requires them to exhibit specific behaviours.

Understanding exploits and exploitation techniques allows us to develop behavioural signatures that can be applied to unknown code to look for evidence of attempts to exploit vulnerabilities.

CONCLUSION

The number of devices supported by some exploits implies either significant manual effort in obtaining the configuration values for each device, or reliable automation to obtain them. That said, we were not able to obtain a large number of exploits due to the high number of exploits for device configurations unknown to us. Generally, each device either received most of the overall set obtained, or a small subset of it. This suggests the pool of exploits available to Kingroot was limited.

Some of the exploits Kingroot has used are very similar to proofs-of-concept available on *GitHub*. Others are using vulnerabilities that it is much harder to find information about, suggesting either internal research and development effort or non-public sources.

The patching strategy of *Android* devices from different OEMs clearly differs. Exploits that seem to be patched in *Google*-manufactured devices continue to work on other devices sometimes for months and years afterwards. This places some users more at risk than others, depending on their choice of device.

Coincidentally, since we started this analysis work, a number of popular *Android* rooting applications have announced they are closing their services down. This includes Kingroot. We don't know what caused this nearly simultaneous decision among large rooting app developers.

REFERENCES

[1] https://github.com/dosomder/iovyroot.

[2] https://github.com/thinkycx/CVE-2017-8890/blob/ec16acd01a6c0e9edc017cf5f66918ccf79a4b4b/nexus6p%40kernel-3.10/jni/exp.c.