Google Security Blog

The latest news and insights from Google on security and safety on the Internet

PHA Family Highlights: Zen and its cousins

January 11, 2019

Posted by Lukasz Siewierski, Android Security & Privacy Team

Google Play Protect detects Potentially Harmful Applications (PHAs) which Google Play Protect defines as any mobile app that poses a potential security risk to users or to user data—commonly referred to as "malware." in a variety of ways, such as static analysis, dynamic analysis, and machine learning. While our systems are great at automatically detecting and protecting against PHAs, we believe the best security comes from the combination of automated scanning and skilled human review.

With this blog series we will be sharing our research analysis with the research and broader security community, starting with the PHA family, *Zen*. Zen uses root permissions on a device to automatically enable a service that creates fake Google accounts. These accounts are created by abusing accessibility services. Zen apps gain access to root permissions from a rooting trojan in its infection chain. In this blog post, we do not differentiate between the rooting component and the component that abuses root: we refer to them interchangeably as Zen. We also describe apps that we think are coming from the same author or a group of authors. All of the PHAs that are mentioned in this blog post were detected and removed by Google Play Protect.

Background

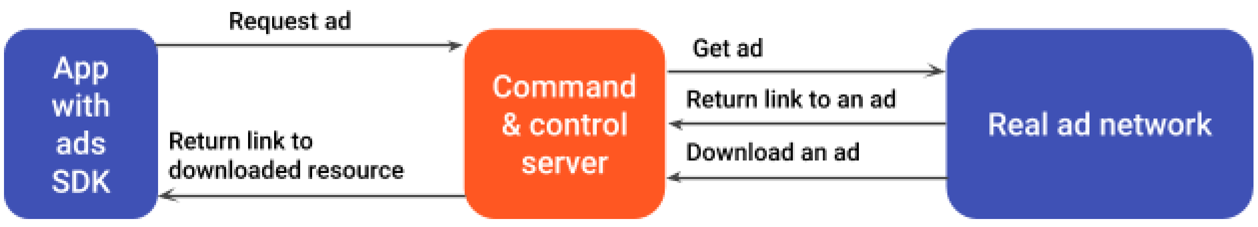
Uncovering PHAs takes a lot of detective work and unraveling the mystery of how they're possibly connected to other apps takes even more. PHA authors usually try to hide their tracks, so attribution is difficult. Sometimes, we can attribute different apps to the same author based on a small, unique pieces of evidence that suggest similarity, such as a repetition of an exceptionally rare code snippet, asset, or a particular string in the debug logs. Every once in a while, authors leave behind a trace that allows us to attribute not only similar apps, but also multiple different PHA families to the same group or person.

However, the actual timeline of the creation of different variants is unclear. In April 2013, we saw the first sample, which made heavy use of dynamic code loading (i.e., fetching executable code from remote sources after the initial app is installed). Dynamic code loading makes it impossible to state what kind of PHA it was. This sample displayed ads from various sources. More recent variants blend rooting capabilities and click fraud. As rooting exploits on Android become less prevalent and lucrative, PHA authors adapt their abuse or monetization strategy to focus on tactics like click fraud.

This post doesn't follow the chronological evolution of Zen, but instead covers relevant samples from least to most complex.

Apps with a custom-made advertisement SDK

The simplest PHA from the author's portfolio used a specially crafted advertisement SDK to create a proxy for all ads-related network traffic. By proxying all requests through a custom server, the real source of ads is opaque. This example shows one possible implementation of this technique.



This approach allows the authors to combine ads from third-party advertising networks with ads they created for their own apps. It may even allow them to sell ad space directly to application developers. The advertisement SDK also collects statistics about clicks and impressions to make it easier to track revenue. Selling the ad traffic directly or displaying ads from other sources in a very large volume can provide direct profit to the app author from the advertisers.

We have seen two types of apps that use this custom-made SDK. The first are games of very low quality that mimic the experience of popular mobile games. While the counterfeit games claim to provide similar functionality to the popular apps, they are simply used to display ads through a custom advertisement SDK.

The second type of apps reveals an evolution in the author's tactics. Instead of implementing very basic gameplay, the authors pirated and repackaged the original game in their app and bundled with it their advertisement SDK. The only noticeable difference is the game has more ads, including ads on the very first screen.

In all cases, the ads are used to convince users to install other apps from different developer accounts, but written by the same group. Those apps use the same techniques to monetize their actions.

Click fraud apps

The authors' tactics evolved from advertisement spam to real PHA (Click Fraud). Click fraud PHAs simulate user clicks on ads instead of simply displaying ads and waiting for users to click them. This allows the PHA authors to monetize their apps more effectively than through regular advertising. This behavior negatively impacts advertisement networks and their clients because advertising budget is spent without acquiring real customers, and impacts user experience by consuming their data plan resources.

The click fraud PHA requests a URL to the advertising network directly instead of proxying it through an additional SDK. The command & control server (C&C server) returns the URL to click along with a very long list of additional parameters in JSON format. After rendering the ad on the screen, the app tries to identify the part of the advertisement website to click. If that part is found, the app loads Javascript snippets from the JSON parameters to click a button or other HTML element, simulating a real user click. Because a user interacting with an ad often leads to a higher chance of the user purchasing something, ad networks often "pay per click" to developers who host their ads. Therefore, by simulating fraudulent clicks, these developers are making money without requiring a user to click on an advertisement.

This example code shows a JSON reply returned by the C&C server. It has been shortened for brevity.

```
{
 "data": [{
  "id": "107",
  "url": "<ayud_url>",
  "click_type": "2",
  "keywords_js": [{
   "keyword": "<a class=\"show_hide btnnext\"",
   "js": "javascript:window:document.getElementsByClassName(\"show_hide btnnext\
  {
   "keyword": "value=\"Subscribe\" id=\"sub-click\"",
   "js": "javascript:window:document.getElementById(\"sub-click\").click();"
```

Based on this JSON reply, the app looks for an HTML snippet that corresponds to the active element (`show_hide btnnext`) and, if found, the Javascript snippet tries to perform a `click()` method on it.

Rooting trojans

The Zen authors have also created a rooting trojan. Using a publicly available rooting framework, the PHA attempts to root devices and gain persistence on them by reinstalling itself on the system partition of rooted device. Installing apps on the system partition makes it harder for the user to remove the app.

This technique only works for unpatched devices running Android 4.3 or lower. Devices running Android 4.4 and higher are protected by Verified Boot.

Zen's rooting trojan apps target a specific device model with a very specific system image. After achieving root access the app tries to replace the framework.jar file on the system partition. Replicating framework.jar allows the app to intercept and modify the behavior of the Android standard API. In particular, these apps try to add an additional method called `statistics()` into the Activity class. When inserted, this method runs every time any Activity object in any Android app is created. This happens all the time in regular Android apps, as Activity is one of the fundamental

Android UI elements. The only purpose of this method is to connect to the C&C server.

The Zen trojan

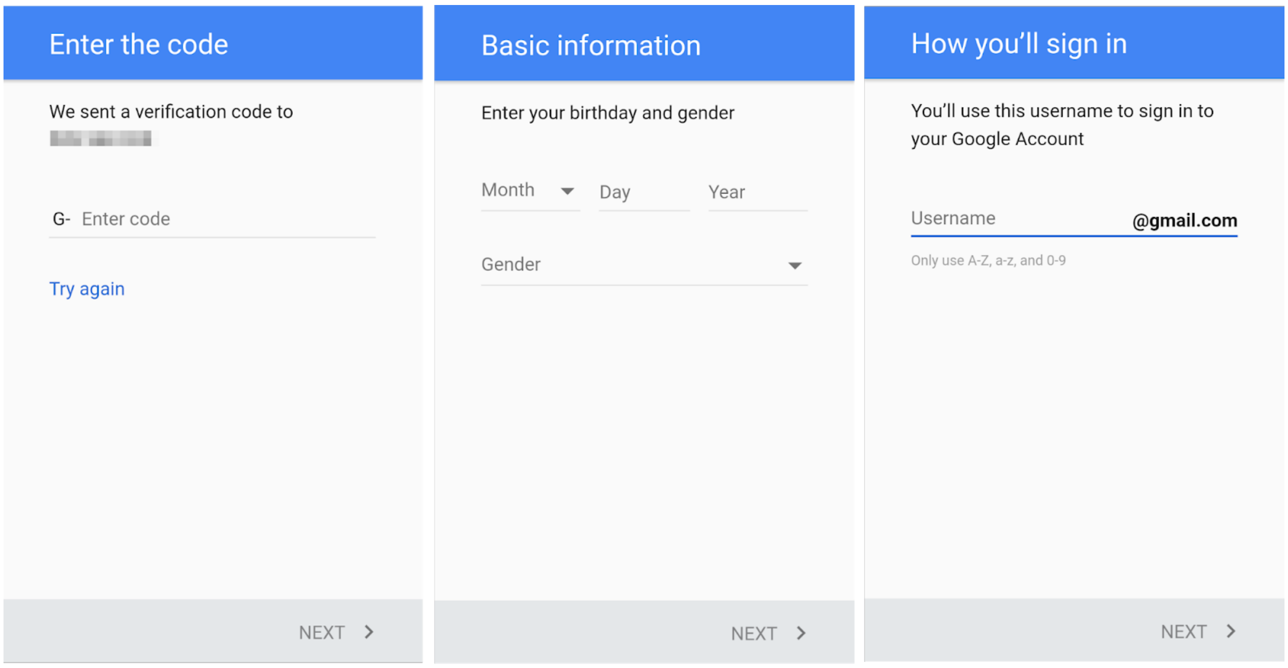
After achieving persistence, the trojan downloads additional payloads, including another trojan called Zen. Zen requires root to work correctly on the Android operating system.

The Zen trojan uses its root privileges to turn on accessibility service (a service used to allow Android users with disabilities to use their devices) for itself by writing to a system-wide setting value `enabled_accessibility_services`. Zen doesn't even check for the root privilege: it just assumes it has it. This leads us to believe that Zen is just part of a larger infection chain. The trojan implements three accessibility services directed at different Android API levels and uses these accessibility services, chosen by checking the operating system version, to create new Google accounts. This is done by opening the Google account creation process and parsing the current view. The app then clicks the appropriate buttons, scrollbars, and other UI elements to go through account sign-up without user intervention.

During the account sign-up process, Google may flag the account creation attempt as suspicious and prompt the app to solve a CAPTCHA. To get around this, the app then uses its root privilege to inject code into the Setup Wizard, extract the CAPTCHA image, and sends it to a remote server to try to solve the CAPTCHA. It is unclear if the remote server is capable of solving the CAPTCHA image automatically or if this is done manually by a human in the background. After the server returns the solution, the app enters it into the appropriate text field to complete the CAPTCHA challenge.

The Zen trojan does not implement any kind of obfuscation except for one string that is encoded using Base64 encoding. It's one of the strings - "How you'll sign in" - that it looks for during the account creation process. The code snippet below shows part of the screen parsing process.

```
if (!title.containsKey("Enter the code")) {
  if (!title.containsKey("Basic information")) {
    if (!title.containsKey(new String(android.util.Base64.decode("SG93IHlvdeKAmW
      if (!title.containsKey("Create password")) {
        if (!title.containsKey("Add phone number")) {
```



Enter the code

We sent a verification code to
[redacted]

G- Enter code

Try again

NEXT >

Basic information

Enter your birthday and gender

Month ▾ Day Year

Gender ▾

NEXT >

How you'll sign in

You'll use this username to sign in to your Google Account

Username @gmail.com

Only use A-Z, a-z, and 0-9

NEXT >

Apart from injecting code to read the CAPTCHA, the app also injects its own code into the `system_server` process, which requires root privileges. This indicates that the app tries to hide itself from any anti-PHA systems that look for a specific app process name or does not have the ability to scan the memory of the `system_server` process.

The app also creates hooks to prevent the phone from rebooting, going to sleep or allowing the user from pressing hardware buttons during the account creation process. These hooks are created using the root access and a custom native code called `Lmt_INJECT`, although the algorithm for this is well known.

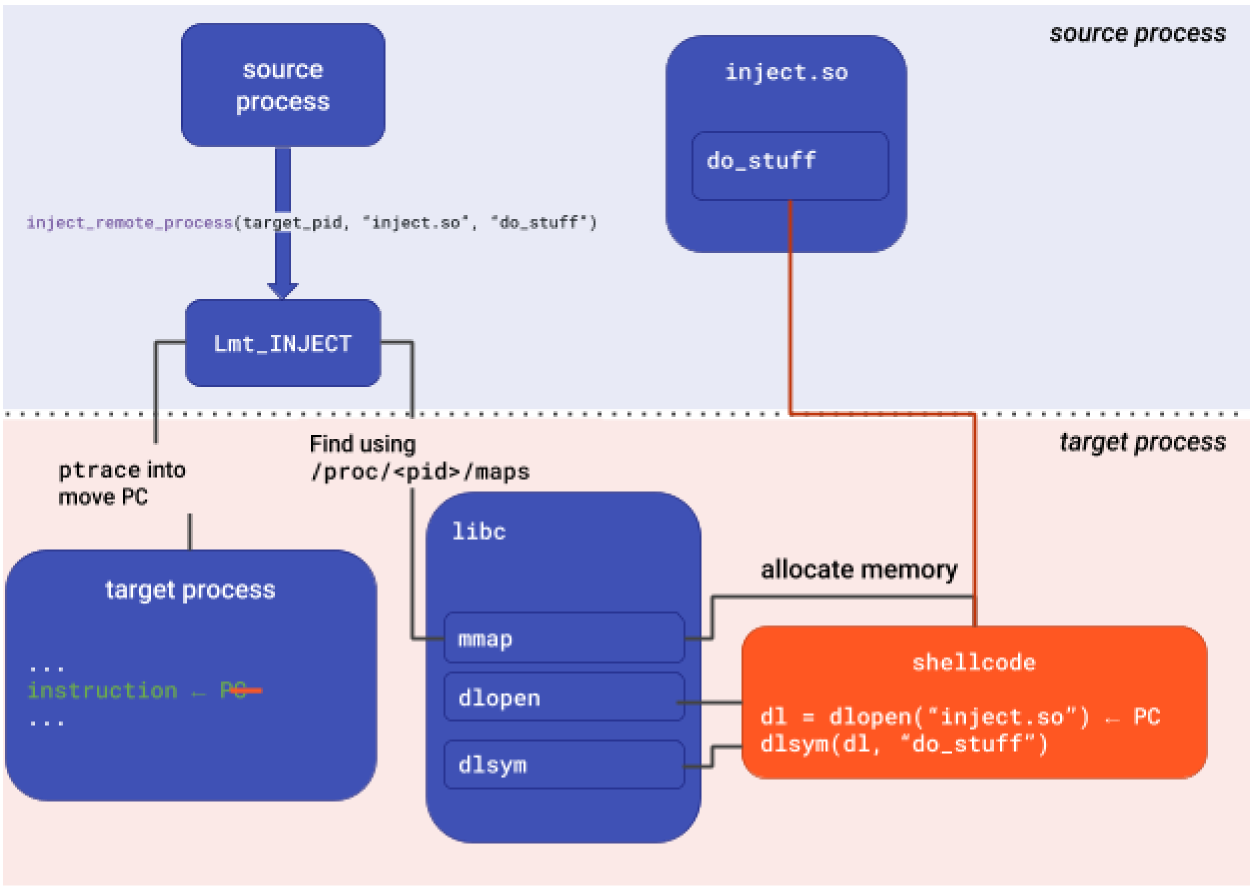
First, the app has to turn off SELinux protection. Then the app finds a process id value for the process it wants to inject with code. This is done using a series of syscalls as outlined below. The "source process" refers to the Zen trojan running as root, while the "target process" refers to the process to which the code is injected and [pid] refers to the target process pid value.

1. The source process checks the mapping between a process id and a process name. This is done by reading the `/proc/[pid]/cmdline` file.

   This very first step fails in Android 7.0 and higher, even with a root permission. The `/proc` filesystem is now mounted with a `hidepid=2` parameter, which means that the process cannot access other process `/proc/[pid] directory`.

2. A `ptrace_attach` syscall is called. This allows the source process to trace the target.
3. The source process looks at its own memory to calculate the offset between the beginning of the `libc` library and the `mmap` address.
4. The source process reads `/proc/[pid]/maps` to find where `libc` is located in the target process memory. By adding the previously calculated offset, it can get the address of the `mmap` function in the target process memory.
5. The source process tries to determine the location of `dlopen`, `dlsym`, and `dlclose` functions in the target process. It uses the same technique as it used to determine the offset to the `mmap` function.
6. The source process writes the native shellcode into the memory region allocated by `mmap`. Additionally, it also writes addresses of `dlopen`, `dlsym`, and `dlclose` into the same region, so that they can be used by the shellcode. Shellcode simply uses `dlopen` to open a .so file within the target process and then `dlsym` to find a symbol in that file and run it.
7. The source process changes the registers in the target process so that PC register points directly to the shellcode. This is done using the `ptrace` syscall.

This diagram illustrates the whole process.



Summary

PHA authors go to great lengths to come up with increasingly clever ways to monetize their apps.

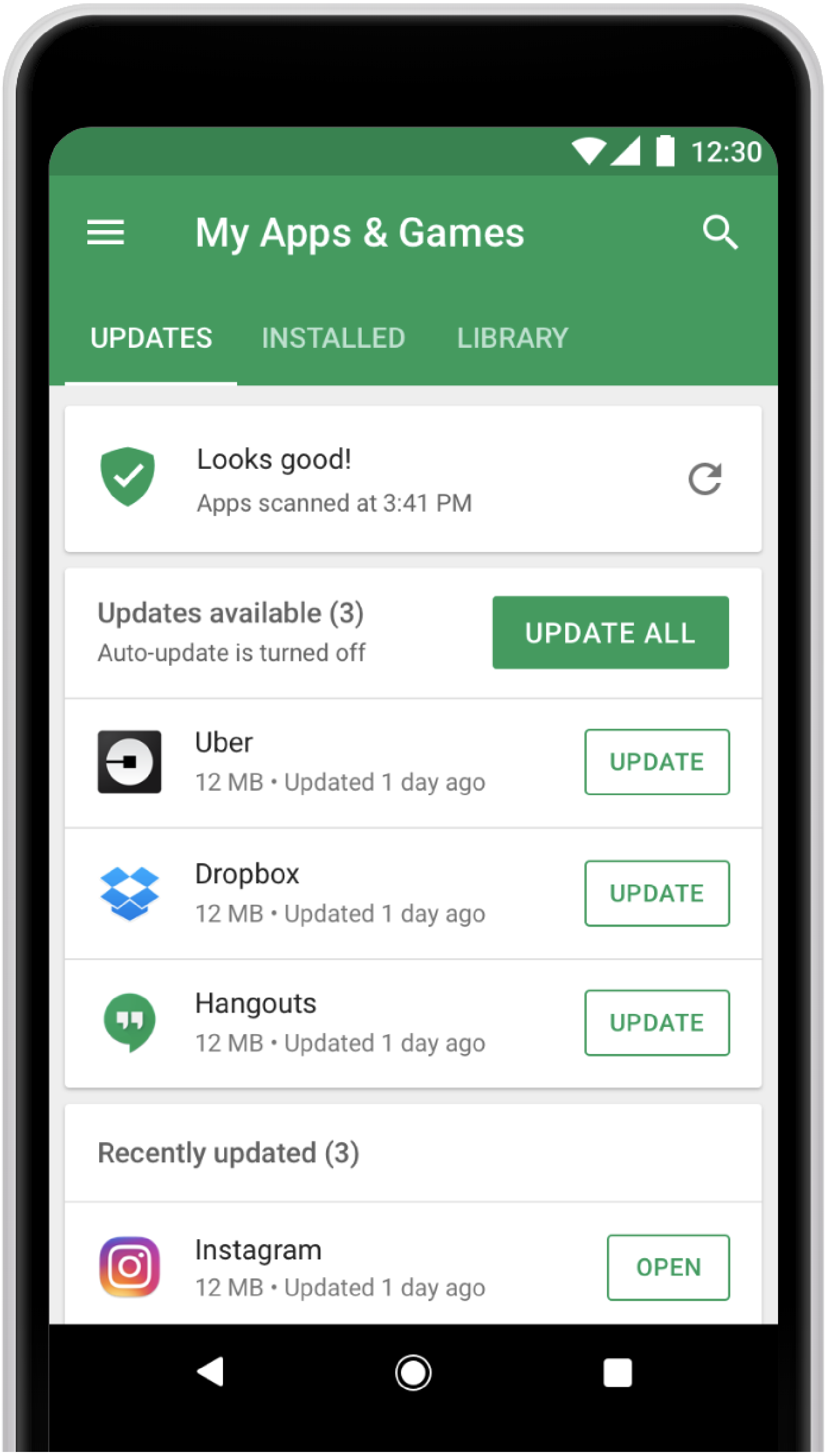
Zen family PHA authors exhibit a wide range of techniques, from simply inserting an advertising SDK to a sophisticated trojan. The app that resulted in the largest number of affected users was the click fraud version, which was installed over 170,000 times at its peak in February 2018. The most affected countries were India, Brazil, and Indonesia. In most cases, these click fraud apps were uninstalled by the users, probably due to the low quality of the apps.

If Google Play Protect detects one of these apps, Google Play Protect will show a warning to users.

We are constantly on the lookout for new threats and we are expanding our protections. Every device with Google Play includes Google Play Protect and all apps on Google Play are automatically and periodically scanned by our solutions.

You can check the status of Google Play Protect on your device:

1. Open your Android device's Google Play Store app.
2. Tap Menu>Play Protect.
3. Look for information about the status of your device.



Hashes of samples

Type	Package name	SHA256 digest
Custom ads	com.targetshoot.zombieapocalypse.sniper.zombieshootinggame	5d98d8a7a012a858f0fa4cf8d2ed3d5a82937b1a98ea2703d440307c63c6c928
Click fraud	com.counterterrorist.cs.elite.combat.shootinggame	84672fb2f228ec749d3c3c1cb168a1c31f544970fd29136bea2a5b2cefac6d04
Rooting trojan	com.android.world.news	bd233c1f5c477b0cc15d7f84392dab3a7a598243efa3154304327ff4580ae213
Zen trojan	com.lmt.register	eb12cd65589cbc6f9d3563576c304273cb6a78072b0c20a155a0951370476d8d

Labels: android security

No comments :

Post a Comment

Links to this post

Create a Link