Poster: *DyPolDroid*: User-Centered Counter-Policies Against Android Permission-Abuse Attacks

Matthew Hill and Carlos E. Rubio-Medrano
Texas A&M University - Corpus Christi
Corpus Christi, Texas, USA
mhill10@islander.tamucc.edu,
carlos.rubiomedrano@tamucc.edu

Luis M. Claramunt, Jaejong Baek and Gail-Joon Ahn
Arizona State University
Tempe, Arizona, USA
{*lclaramu, jbaek7, gahn*}*@asu.edu*

Abstract—**Android applications are extremely popular, as they are used for banking, social media, e-commerce, etc. However, several *malicious* applications have recently carried out data leaks and spurious credit card charges by *abusing* the Android Permissions granted initially to them by unaware users in good faith. To alleviate this pressing concern, we present *DyPolDroid*, a dynamic, semi-automated security framework that builds upon Android Enterprise, a device-management framework for organizations, allowing for users to design and enforce custom *Counter-Policies*, effectively protecting against such malicious applications without requiring advanced security and/or technical expertise.**

1. Introduction

In recent years there has been an increase in the number of malicious applications in the Android Ecosystem [1], targeting users with a large variety of attacks, e.g., harvesting private data, making unwanted credit card charges, retrieving the location of users, etc. Whereas the root causes for such attacks have been largely explored in the literature [2], an increasing number of applications look to use and abuse the permissions granted legitimately by users to carry out attacks. These so-called *Permission-Abusing Applications* (P-A Apps) initially pose as *benign* and request users to grant a seemingly normal set of permissions to deliver some *harmless* functionality, e.g. sorting out contact information. However, they later *abuse* the granted permissions to facilitate attacks, e.g., leaking the user's contacts to a remote server via the Internet.

To effectively defeat these P-A Apps, we present *DyPolDroid* (Dynamic Policies in Android), a dynamic, semi-automated security framework, which allows for users and enterprise administrators to easily write *Counter-Policies* restricting a series of *Attack Patterns*. These are sets of Permissions that, if used in combination, may allow for carrying out an attack, e.g., combining the `Internet` and `Read-Contacts` permissions to perform a data leak. Later, Counter-Policies are evaluated and translated into lists of permissions that are allowed or denied for each potential P-A App, and are sent for enforcement on the user's device by means of the remote configuration features offered by the Android Enterprise, providing a convenient solution that offers an advanced degree of automation and requires no advanced security expertise.

Dr. Gail-Joon Ahn is also affiliated with Samsung Research.

2. Background and Problem Statement

Android Permission Model. Prior to Android 6.0, all permissions requested by an app needed to be granted by users at installation time; users were presented with a list of permissions to accept or deny once the app have been downloaded but before installation could begin. If users would choose to deny the requested permissions, the installation of the app would fail. With the release of Android 6.0, the permission model was modified such that apps needed to request access to a permission the first time that they wanted to use it [3], which allowed for a more fine-grained approach in which users would accept or reject each permission individually.

Android Enterprise. Android Enterprise is a device management framework that allows for organizations to remotely configure a series of Android-run devices, e.g., installing and uninstalling apps on devices without extensive user intervention [4]. In addition, Android Enterprise leverages the permission model, as described before, to dynamically update, e.g., grant or deny, the permissions requested by individual apps, thus allowing for Enterprise administrators to remotely restrict the functionality of all the apps installed on a managed device at will.

Permission-Abusing Applications. A *Permission-Abusing Application* (P-A App) is a seemingly *benign* app that is secretly *malicious*. Its formal or informal usage documentation states that it uses permissions in an expected, harm-free way, e.g., for sending messages to contacts via the Internet, but it may also use them in a malicious, unwanted, and potentially user-harming way as well, e.g., for installing tracking software [5] or collecting extraneous user data [6].

Problem Statement. For the purposes of this paper, we assert that apps that request access to permissions and knowingly misuse them are potentially malicious, i.e., they are P-A Apps, as such permissions may allow for them to successfully carry out their attack(s). Therefore, we aim to detect all potential apps installed on devices that may be P-A Apps, and aim to prevent them from successfully using, a.k.a., exploiting, any granted permissions at run-time. In such a scenario, there may be an overlap between the permissions that allow for benign functionality and the ones used for carrying out malicious functionality, e.g., the `Internet` permission being simultaneously used for sending messages (benign) and leaking private data (malicious).

DOI 10.1109/EuroSP51992.2021.00053

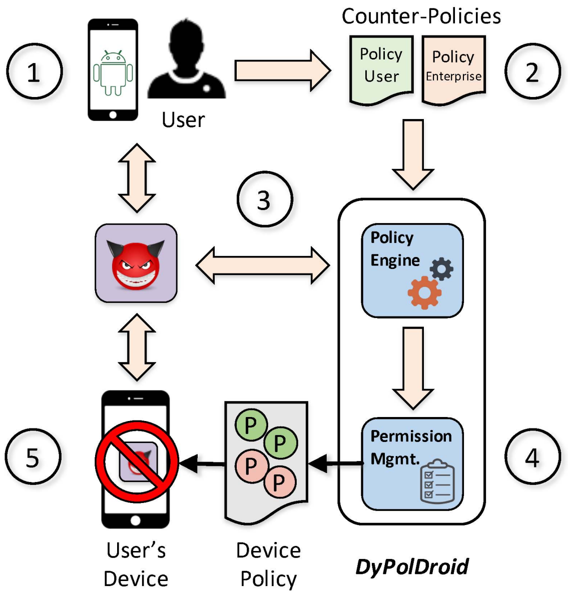

Figure 1. The Workflow of *DyPolDroid*: a User signs up for an Android Enterprise (1) and moves on to write Counter-Policies (2), which are later evaluated against the Attack Patterns obtained from any installed P-A Apps (3), producing a Device Policy that is then sent to the Device (4). As a result, P-A Apps have their permissions blocked (5).

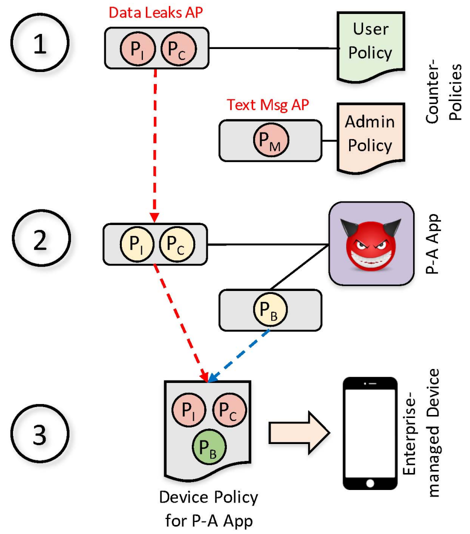

Figure 2. Creating Device Policies in *DyPolDroid*. The set of *authorized* permissions for each P-A App are obtained by evaluating Counter- Policies(1), whereas the set of *requested* Permissions are obtained via Data Flow and Taint Tracking analysis (2). The set of *resulting* permissions is calculated by *subtracting* the authorized permissions from the requested ones, and it is later encoded and sent out as a Device Policy (3).

3. Our Approach: Dynamic Enforcement of Counter-Policies via Android Enterprise

To address the previously stated problem, we have envisioned an approach in which both Users and Android Enterprise Administrators can actively restrict the functionality of potential P-A Apps by leveraging the dynamic permission updates provided by Android Enterprise. Our approach, called *DyPolDroid*, allows for the specification of so-called *Counter-Policies* restricting the Attack Patterns described in Sec. 2. Such patterns are in turn discovered utilizing contextual information obtained by analyzing the flow of data inside the P-A Apps installed on a user's device. Following Fig. 1, our approach can be further described as:

(1) Android Enterprise Sign Up. Initially, users are allowed to sign up for the Android Enterprise on their mobile device. One key functionality of *DyPolDroid* is that users do not need to make any modification to their operating system, e.g., requiring the device to be *rooted*, which allows for them to simply register and get protected in an easy and straightforward way.

(2) Writing Counter-Policies. Counter-Policies are written using a series of *templates* depicting a subset of XACML, the *de facto* language for authorization and access control. Users are then able to specify a policy to help protect their device by specifying a variety of rules including features like: which applications can be installed, the default permission policy of any newly installed application, and what potential attacks the user would like to defend against. In addition, Counter-Policies leverage the conflict resolution features provided by XACML for the case when multiple policies are applied to the same device, allowing for *DyPolDroid* to resolve conflicts before any resulting policies are sent to the user's device.

(3) Discovering Attack Patterns. Our proposed Attack Patterns are inspired by a set of predetermined attack vectors that were found to be common place across a number of known malicious applications [7]. These map data from a *source* to a *sink* inside code. For example, the Attack Pattern: {Contacts, Internet} extracts a user's contact information and sends them to a remote server via the Internet.

DyPolDroid leverages taint tracking from FlowDroid [8] to gain insight into the data usage within a P-A App. The resulting data flow is cross-referenced against known Android class functions that are used for interacting with permissions. To ensure only matching applications are updated, *DyPolDroid* uses the SHA 256 hash in conjunction with the application package to ensure that only matching applications have the appropriate actions taken against them. This is important when there are multiple versions of the same application installed on devices for different users within the Android Enterprise, e.g. v1.1.33 and v1.1.34. If a vulnerability that the user wants to protect against is found and the application matches, the policy is automatically updated to block the permission(s) required to carry out the attack.

Fig. 2 gives an overview of how the Device Policies are created and checked before being send to the user's device. First, the permissions requested by the P-A App is obtained. Second, the results of the application analysis are used to determine what permissions, if any, are being abused by the application to leak user data. These offending permissions are then updated within the Device Policy to block their usage by the application.

(4) Device Policies and Enforcement. Once the Android Enterprise has received the Device Policy from *DyPolDroid*, it sends it to the device. Once received, the policy will immediately begin to apply. If there are any conflicts between the user's device and the new-applied policy, e.g., an installed application is not allowed by the

```
<Rule RuleId="Laverna_attacks" Effect="Deny">
 <Target>
  <AnyOf> <AllOf> <Match Id="boolean-equal">
   <AttributeValue>true</AttributeValue>
    <AttributeDesignator
    AttributeId="Laverna"/>
  </Match> </AllOf> </AnyOf>
  <AnyOf> <AllOf> <Match Id="boolean-equal">
   <AttributeValue>true</AttributeValue>
   <AttributeDesignator
   AttributeId="\textSteal_Contacts"/>
  </Match> </AllOf>
   <AllOf><Match Id="boolean-equal">
    <AttributeValue>true</AttributeValue>
    <AttributeDesignator Category="action"
     AttributeId="Steal_Messages"/>
  </Match></AllOf> </AnyOf>
 </Target>
</Rule>
```

Listing 1. A Counter-Policy for the Laverna P-A App.

policy, the device manager will freeze the profile until the device is compliant with the policy, e.g., forcing the user to manually uninstall the offending application. With the policy now applied to the mobile device and in full effect, the user can begin to get protected.

4. Preliminary Evaluation

For the purpose of evaluating our approach, we have developed *Laverna*: a *proof-of-concept* P-A App that requests several permissions for benign functioning, getting full access to the user's contacts, real time location, and SMS so it can serve as a messaging application. However, it also silently exploits the granted permissions to collect and leak data to a remote server when the user is messaging another user. The leaked data includes the contact's full name and phone number and the messages sent, including who the sender and receiver are. The Counter-Policy shown in Listing 1 gives the response to the different types of attacks a users wants to defend against. In this case the two attacks are: Steal Contacts, and Steal Messages. Should any of the attacks be found when analyzing the application, the action taken against the used permissions will be to deny them. This change in allowed permissions is reflected in the JSON-based Device Policy shown in Listing 2.

In our experiments, Laverna was downloaded on an experimental device, and a user was allowed to select what permissions can be granted before installed such P-A App. Our tests show that *DyPolDroid* was able to block this application from collecting the user's data and sending it off the device. Since a subset of the permissions requested by Laverna were found to be malicious, the default policy was overridden to block them on the device. While this approach does not preemptively block the leaking of user data, once *DyPolDroid* has been performed its analysis future cases will mitigate such attacks.

5. Conclusions and Future Work

P-A Apps are still an ongoing problem for Android Ecosystems. In such regard, *DyPolDroid* offers an effective and convenient solution that requires no root access to user's devices nor any modifications to the code of P-A Apps: two constraints that have limited the deployment in practice of previous related approaches. As a matter of ongoing and future work, we are currently analyzing several P-A Apps to identify Attack Patterns and potential templates for Counter-Policies that can effectively defeat them. We plan to use this insight later on to conduct a comprehensive study in which users sign up for an experimental Android Enterprise. Then, we aim to collect data on how the devices are used, and verify whether *DyPolDroid* was able to accurately detect when permissions were improperly used. Also, we will collect data regarding the level of user satisfaction with respect to the restrictions observed in the functionality of potential P-A Apps as a result of using *DyPolDroid*.

```
{ "defaultPermissionPolicy": "PROMPT",
  "applications": [{
    "packageName": "com.example.laverna",
    "installType": "REQUIRED_FOR_SETUP",
    "permissionGrants": [
      { "permission": "android.permission.
          READ_CONTACTS",
        "policy": "BLOCK"},
      { "permission": "android.permission.
          READ_SMS",
        "policy": "BLOCK"}
    ]}
```

Listing 2. A Device Policy for the Laverna P-A App.

Acknowledgments

This work is partially supported by a grant from the National Science Foundation (NSF-SFS-1129561), a grant from the Center for Cybersecurity and Digital Forensics at Arizona State University, and by a startup funds grant from Texas A&M University – Corpus Christi.

References

[1] ZDNet. (2020) Play store identified as main distribution vector for most android malware. [Online]. Available: https://www.zdnet.com/article/play-store-\identified-as-main-distribution-vector-for-most-android-malware/

[2] Y. Shao, J. Ott, Q. A. Chen, Z. Qian, and Z. Mao, "Kratos: Discovering inconsistent security policy enforcement in the android framework," in *Proc. of the Network and Distributed System Security Symposium (NDSS) 2016*, January 2016.

[3] Google. (2021) Permissions on android. [Online]. Available: https://developer.android.com/guide/topics/permissions/overview

[4] Google. (2021) Android Enterprise. [Online]. Available: https://www.android.com/enterprise/

[5] Android Authority. (2020) Report: Hundreds of apps have hidden tracking software used by the government. [Online]. Available: https://www.androidauthority.com/government-tracking-apps-1145989/

[6] The New York Times. (2020) The Lesson We're Learning From TikTok? It's All About Our Data. [Online]. Available: https://www.nytimes.com/2020/08/26/technology/personaltech/tiktok-data-apps.html

[7] A. Arora, S. K. Peddoju, and M. Conti, "Permpair: Android malware detection using permission pairs," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1968–1982, 2020.

[8] S. Arzt, "Static data flow analysis for android applications," Ph.D. dissertation, Technische Universität, Darmstadt, 2017. [Online]. Available: http://tuprints.ulb.tu-darmstadt.de/5937/