(In)Security of Backend-as-a-Service

Siegfried Rasthofer[1,2], Steven Arzt[1], Robert Hahn[1], Max Kolhagen[1], Eric Bodden[1,2]

[1] Center for Advanced Security Research Darmstadt (CASED)
Technische Universität Darmstadt, Germany
[2] Fraunhofer SIT, Darmstadt, Germany

Abstract. Since recent years, more and more tasks in personal data processing are performed by smartphone applications. Users store and manage an increasing amount of sensitive information inside these apps and expect the data to be available across devices and platforms. Application developers, on the other hand, are pressed to deliver new applications faster and with more features. As a consequence, they outsource tasks such as backend provisioning to specialized service providers.
In this paper, we perform a study on the security of Backend-as-a-Service (BaaS) and its practical use in real-world Android and iOS applications. As we show, many apps embed hard-coded credentials, putting not only the user's data, but the whole platform at risk. We show that with current tools attackers can gain access to huge amounts of sensitive data such as millions of verified e-mail addresses, thousands of health records, complete employee and customer databases, voice records, etc. Often, one can manipulate, and delete records at will. Some BaaS instances even suffer from remote code-execution vulnerabilities.
We provide HAVOC, a fully-automated tool for finding potentially vulnerable applications and a fully-automated exploit generator that extracts the required credentials from the app and checks their validity with the BaaS backend. We analyzed over 2,000,000 applications from the Google Play Store and alternative markets and found over 1,000 backend credentials, many of them re-used in several applications. In total over all apps, we found that more than 18,670,000 records with over 56,000,000 individual data items were freely accessible.

1 Introduction

Smartphone applications are used to manage a broad variety of personal data items. Users store their personal notes, contacts, expenses, and even health records on their phones to have them handy in every situation. Modern apps therefore provide convenient mechanisms for synchronizing their data between devices and platforms. Even if a hardware failure destroys one device, the user still has access to his data from a replacement device or possibly a web interface.

Such features, however, require applications to store their data remotely, even if the application does not offer so-called *social* features like sharing the data with other people on the Internet. But developing and maintaining backend-solutions for data storage is time-consuming and costly. Furthermore, it can

significantly increase app-development costs and time to market. Many application developers do not have a strong background in database development and server-side languages and platforms such as PHP, or ASP.net. This is why a number of companies now provide fully maintained and readily-usable backend solutions that application developers can integrate into their apps with just a few lines of code. This approach promises to leave all the effort of actually running the backend with the respective provider and is called *Backend-as-a-Service* (BaaS). Often, such services also include ready-made solutions for common tasks such as user authentication, storing key-value pairs, social-media integration or push-notifications. BaaS services are provided via specialized software development kits (SDK) and application programming interfaces (APIs) for easy integration.

The simplicity of these services from the application developer's point of view has given rise to a market of multiple competing BaaS providers. The most widely used BaaS providers are Parse.com (recently bought by Facebook)[1], CloudMine[2], and Amazon Web Services[3]. In this paper, we evaluate the security of the most widely used BaaS solutions in real-world Android and iOS applications. We concentrate on the three biggest competitors at the time of writing: Parse, CloudMine and Amazon AWS. We found that while all three BaaS service providers offer security features that would allow for secure data storage, their defaults are mostly alarmingly insecure. Application developers usually accept these defaults for convenience, failing to include appropriate means of protection such as access control or data encryption. By default, most BaaS solutions require an application only to authenticate using an *ID* that uniquely identifies the app, and a so-called "secret" key, used to indicate that the app uses the *ID* legitimately. These credentials, however, neither authenticate a device nor a user. They merely authenticate the app as such and are therefore shared between all installations of this app. As we show, adversaries can extract these two values from apps with ease, allowing them to easily forge a malicious application, which inherits the very same backend-privileges that the original application had. If the original application was able to list all records of a customer database, the impersonator can do so as well.

In theory, all BaaS providers provide documented techniques that layer additional security mechanisms on top or at least minimize the privileges of the application. An app that, e.g., uses a BaaS to store crash reports does not need to operate with privileges to read this data back, delete records, or even manipulate the database schema. These actions only need to be taken by the developer himself through separate channels such as the administrative web-interfaces that many BaaS providers offer - and never through the normal mobile applications distributed to end-users. Developers, however, oftentimes do not use the features for restricting access, but only invest the minimal effort required to build a functionally working application. As a consequence, not only the end users' data is left at risk, but the respective applications and even backend servers are readily

[1] `https://www.parse.com/`
[2] `https://cloudmine.me/`
[3] `http://aws.amazon.com/de/mobile/`

susceptible for data manipulation, exploitation, and misuse. In our study, we find millions of verified e-mail addresses, hundreds of employee- and customer records, thousands of health records, and other highly privacy-sensitive data items. We find servers that allow remote code execution, arbitrary storage misuse, and data manipulation at will.

Previous work [11] has already identified a number of constant Amazon AWS keys in applications downloaded from the Google Play Store. We, however, show that the problem is neither limited to Amazon AWS, nor to constant keys. Keys that are obfuscated or computed at runtime share the same issue. Not only can they as well be easily reconstructed by attackers, but obfuscated keys instead fool the often simpler syntactic security checks that search for such keys when new apps are submitted to an app store—security by obscurity in its perfection.

With this work we expose HAVOC, a fully-automatic exploit generator. HAVOC not only finds simply embedded credentials based on static analysis but also uses a hybrid (static/dynamic) analysis for cases where keys are computed at runtime. This allows HAVOC to find keys which cannot be recognized by simple pattern matching as proposed in [11]. Overall, this paper presents the following original contributions:

- The first comprehensive security evaluation of several popular BaaS providers and APIs as well as their use in real-world Android and iOS applications,
- a fully-automatic and efficient scanning tool that identifies Android applications which use BaaS APIs, even if the app was obfuscated,
- a fully-automatic exploit generator that extracts the necessary credentials from an Android application and verifies if the credentials are still valid with the BaaS provider, and
- a set of proposed mitigation techniques to overcome the vulnerabilities laid out in this paper.

The remainder of this paper is structured as follows: Section 2 gives background information on the most popular BaaS frameworks and providers. In Section 3, we present HAVOC, our fully automated exploit generator. In Section 4, we evaluate our findings and their possible implications on security and privacy. Our responsible disclosure process is discussed in Section 5, while Section 6 presents recommended mitigation strategies against the vulnerabilities we discovered. Section 7 gives an overview of related work, Section 8 outlines our planned future work, and Section 9 concludes the paper.

2 Background

This section gives some background information on the most important BaaS providers on the market: Parse.com, CloudMine, and Amazon AWS. We first explain the basic authentication concept employed by all three providers. Afterwards, we discuss the individual details of the respective services.

2.1 Basic Authentication

All three BaaS providers use the same concept of *application ID* and *secret key* for basic authentication. The application ID uniquely identifies the app and the tenant to which it belongs on the back-end system. The secret key serves as a proof that the application that makes the request is actually authorized to access the database associated with the given application ID. One can think of these two values as user name and password. The key difference, however, is that they do not identify a user or a device, but an app. All copies of the app share the same credentials, regardless of where they are installed and who uses them. The security mechanism solely relies on these credentials only being known to trusted applications which only perform intended operations on the database. If an attacker is able to extract these credentials, he can easily craft an own application, which uses these credentials to authenticate against the backend and impersonates the original application. His own (potentially malicious) application inherits all privileges of the original app.

2.2 Parse.com

The Parse API allows an application developer to store key/value pairs in tables with very little effort. Listing 1.1 shows how to store the value 100 for the key `playerName` in a table called `GameScore`.

Besides storing custom key/value pairs, the Parse.com API offers specialized functions for common backend tasks such as user management. To ease application development, the Parse.com API offers functions to register new users and authenticate existing ones by checking a given combination of user name and password. Internally, ParseUsers are stored in a normal key/value table in the very same backend that also hosts the user tables. The `ParseUser` class represents a single user account, i.e., a single row in the pre-defined user table. By default, accounts have a number of properties such as a user name, an e-mail address, and a password. One can thus use the very same query functions that are available on user-defined tables with the system-provided user table as well. By default, queries on the user table will return all data fields except for the password. This means that, while passwords are protected, all other data items such as the e-mail addresses and further application-specific profile data can be retrieved using simple API calls. As the data is protected only by application authentication, if

```
1 //connection to the backend
2 Parse.initialize(this, APPLICATION_ID, CLIENT_KEY);
3 //access table 'GameScore'
4 ParseObject gameScore = new ParseObject("GameScore");
5 //access record 'playerName' from 'GameScore'
6 String playerName = gameScore.put("playerName", 100);
```

Listing 1.1: Code Snipped necessary for accessing a particular record in database stored in the Parse backend

an attacker can easily extract the application ID and client key from the original application, he can use both to forge an 'authenticated' malicious application that, for instance iterates over all user records and dumps all profile data.

The same issue is shared by the user-defined tables. By default, they can freely be read as long as the application has been 'authenticated'. In contrast to the user table, user-defined tables can by default even be written freely by default, just like in the example above. The Parse.com backend offers a number of configuration options to mitigate the above problems. When saving a new record, for instance, the application can attach an access-control list (ACL) which restricts read and/or write access to that record to the currently logged-in user or specific roles such as administrators. This connects the data tables to Parse.com's built-in user-management system. Once a user has been authenticated using this mechanism, his identity can be matched against the ACL of a table. The developer can configure that certain tables may only be read by authenticated user 'Bob', and not by 'Alice' or an unauthenticated user. ACLs can be defined on the level of tables and records. Records created by Bob can thus be stored in the same table as Alice's records without necessarily becoming visible to Alice. The default settings, however, allow arbitrary accesses to all tables and records except for very few restrictions such as the password field of the user table. A developer who simply uses a code snippet from a tutorial without reading the manual any further is thus likely to inadvertently put his data at risk.

2.3 CloudMine

The CloudMine backend service is very similar to Parse.com in principle. Data storage is done using developer-implemented classes that are derived from a CloudMine base class, as shown in Listing 1.2. In contrast to Parse, objects in CloudMine are divided into *application objects* and *user objects*. Application objects can be read by the application regardless of whether a user is currently authenticated or not. User objects are tied to the currently logged-in user. This means that accessing user objects requires the log-in data of the respective user and is thus not vulnerable to the attacks presented in this paper. While the distinction between application objects and user objects puts a direct emphasis on keeping sensitive data local to its owner's user account, it offers less flexibility than Parse.com's ACL model. It also requires developers to correctly classify objects into one of the two categories.

Just like Parse.com, CloudMine also offers a user-management table with respective API support for signing up new users and authenticating existing ones. Both platforms share the same basic concept of how user accounts are stored. In both systems, the user table is a regular table and can be queried as such. CloudMine also protects the user's password. For an attacker, CloudMine, however, has the advantage that it provides API methods to retrieve all application objects, regardless of their name or type. This API support allows one to obtain data for tables which are not used in the current application under analysis, and for which one thus does not know the table name. Such tables might still contain

```
1 //user class definition
2 public class Order extends CMObject {
3   private String orderNo;
4   public Order() { }  // required by CloudMine
5   public Order(String orderNo) {
6     this.orderNo = orderNo;
7   }
8 }
9
10 //store user elements to the cloud
11 CMApiCredentials.initialize(APPLICATION_ID, CLIENT_KEY,
       getApplicationContext());
12 ClassNameRegistry.register("Order", Order.class);
13 Order order = new Order("123-456");
14 CMStore.getStore().saveObject(order1);
```

Listing 1.2: CloudMine Example for Storing user data

data of interest to an attacker. In Parse, one would need to know the respective table names from some external source to perform the corresponding queries.

2.4 Amazon AWS

The Amazon Web Services (AWS) are a collection of backend services hosted on the Amazon cloud. They are commercially available to developers and consist of various products such as EC2 for running virtual machines, S3 for storing data, and SWS for sending e-mail messages.

Dynamo Amazon Dynamo is part of AWS, the Amazon Web Services. It provides users with a high-availability key/value store similar to the service of Parse.com. The data inside Dynamo is represented and queried as JSON documents, i.e., one table row corresponds to one JSON document, and a field corresponds to a JSON attribute. To simplify the use by Android and Java developers, Amazon provides an object mapper that automatically transfers objects to JSON documents and vice versa.

To use an AWS service such as Dynamo from an application, the application must authenticate against Amazon's servers by providing its *application ID* and *secret key*. This level of authentication is similar to the one required by Parse.com. If a developer, however, specifies his root AWS credentials, which provide full access to all his AWS services, all records inside his Dynamo database and all their attributes are freely readable. Since AWS uses a Single-Sign-On system, such a mistake can not only put the database backend at risk, but also e.g., virtual machine instances hosted on EC2. To obtain access, an attacker only needs to extract the ID and the key from the application.

If an application developer uses Dynamo to store security-sensitive information, he can optionally enable encryption. With this feature, all fields inside the respective table (except for the primary key) are encrypted and exclusions from this rule must be explicitly defined using a `DoNotEncrypt` annotation. However,

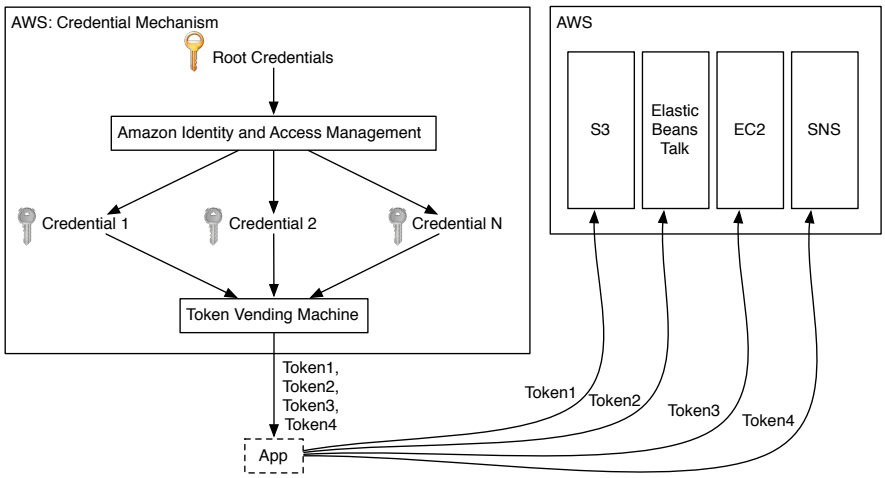

Fig. 1: Amazon AWS Credential Handling

the encryption client for Java does not yet support all data types available in Dynamo and the developer is left with the task of storing and managing his encryption keys. All these factors make the security features less attractive to developers.

Amazon AWS: S3 The Amazon Simple Storage Service (S3) offers easily accessible object storage to application developers. S3 can be thought of as a web-based file system for mass data. For large amounts of unstructured data, the service is assumed to scale better than traditional databases, but on the downside leaves data control and management to the application developer. In particular, it is up to the developer to ensure the integrity of concurrent writes and the correct definition of access-control policies.

To access S3 *buckets* (essentially top-level directories) from within an application, the application needs to authenticate using its *application ID* and *secret key*. Similarly to the other features of AWS, using the developer's default AWS credentials instead of special-purpose credentials with restricted permissions, as would be provided by the Amazon Identity and Access Management (IAM), leads to tremendous security risks. The default credentials have full access to all S3 buckets associated with the ID. This directly allows an attacker who successfully extracted the application ID and the corresponding secret key from a legitimate application to read, write, and delete arbitrary objects on the whole S3 storage associated with that app.

The problem is aggravated by system-level features that rely on the integrity and confidentiality of the S3 buckets. Amazon Elastic Beanstalk, for instance, is a web-application deployment and hosting service that stores its applications as *war* files inside an S3 bucket. If an attacker gains access to that bucket, he can retrieve sensitive configuration information such as credentials for further back-end services or even inject his own code. This code will then be executed

by Amazon's servers just like legitimate user code uploaded by a regular user of Elastic Beanstalk.

Amazon AWS: SNS The Amazon Simple Notification Service (SNS) allows applications to register the phone in a broadcast channel. The developer can then send messages to the enlisted phones. When a message is received, the Android operating system will display it in the notification bar. The developer can freely decide which text to send to the registered phones. When the user clicks on the message, a customizable intent is executed (originally defined during channel subscription). A benign auction application, for instance, could use this feature to notify the user when the auction is going to expire or when a higher bid has been placed on an item.

If an attacker is able to extract the developer's credentials from the app, he can, in the worst case, send push notifications to all registered phones. The users cannot distinguish such rogue messages from benign ones; both are displayed exactly in the same style. To the operating system or the app, there is no difference. Naturally, rogue push notifications are an attractive means for scamming or social engineering.

Securing AWS Access Hard-coded root credentials are a security risk as they allow full access to a developer's AWS instance regardless of the concrete service to be used. Credentials extracted from an app that uses S3 can thus also be used to access virtual machines run in Amazon's Elastic Compute Cloud (EC2). Therefore, these credentials should never be distributed to untrusted clients. Still, clients need to somehow authenticate against their respective AWS services.

This issue can be solved using the Amazon Identity and Access Management (IAM) system (see Figure 1). IAM can be used to derive new keys that are valid for the same application (and thus same backend service), but which have restricted permissions. While the root key can freely perform all supported operations on the backend, an IAM key might, for instance, not be able to delete any data. This system even allows developers to specify fine-grained policies on their AWS objects such as certain records or attributes.

Having said so, the use of IAM requires additional configuration effort. A developer needs to know the IAM concepts, needs to thoroughly read the extensive documentation, or at least the relevant parts of it, and then implement the appropriate policies. On the other hand, if he just integrates his root credentials into the app, using Dynamo is rather easy and the time-to-market is decreased, leaving developers with dangerously wrong incentives.

Even when using IAM, this still requires credentials to be hard-coded into the respective application, which makes it hard to change or rotate them over time. To mitigate this problem, AWS offers temporary credentials. The *Security Token Service* is capable of deriving these token from an IAM credential on a trusted server running a so-called *Token Vending Machine* (TVM). Only this trusted server knows the original IAM credential. Clients first contact the server

to obtain their temporary ID and key and then use these credentials to access to AWS services as normal.

While using IAM with a TVM can be considered a best-practice solution in terms of security, it is not the default implementation variant. The official tutorials on using AWS on Android on Amazon's developer website provide code snippets with placeholders for ID and secret key. A developer pressured on time-to-market might thus simply enter his root credentials there and completely miss the security concept. Furthermore, using a TVM requires the developer to perform a variety of steps on his own: He needs to build the TVM client and server components from samples shipped with the SDK, and then install and run the TVM server on Elastic Beanstalk. In the worst case, this requires the developer to first read about and understand Java Server Pages, and Elastic Beanstalk, frameworks and concepts he would probably never have gotten in touch with if it had not been for the TVM. The usual developer might easily back away from such additional effort that increases security, but does not offer any new features for his app. Furthermore, for using more advanced features such as policy objects (i.e., access control on individual objects on AWS), the TVM samples need to be extended by the app developer. By default, only basic features are available in the TVM samples.

Amazon Cognito provides a more recent alternative to TVMs. Cognito provides user authentication as well as temporary credentials with limited access privileges as a readily-usable solution. This removes the effort to build and set up the TVM. It still requires additional developer actions such as configuring identity pools, but is by far more declarative and simple than TVMs. Since Cognito is rather new and not yet known to the majority of developers, many apps, however, still use hard-coded root credentials or TVMs.

3 Fully-Automatic Exploit Generator

To help assess whether an app that uses BaaS services does so securely, we developed HAVOC, a fully-automated exploit-generation tool consisting of three phases, see Figure 2. First it identifies whether a given application uses one of the supported BaaS libraries. Second it extracts the credentials required for accessing the backend from the app. For all three target services, this is some pair of *ID* and *key*. Lastly, HAVOC collects other useful information such as the names of tables queried inside the app (not required for CloudMine, which allows for full traversal, see Section 2.3) or the names of the accessed S3 buckets.

In HAVOC, we chose to implement support for Parse.com, CloudMine, and Amazon AWS as these services are most widely used. Adding support for other BaaS frameworks is trivial if their API methods used for authentication are known.

3.1 Library Identification

When given a set of Android applications, HAVOC first needs to identify whether one of the supported BaaS frameworks is used inside an app, and if so, which one.

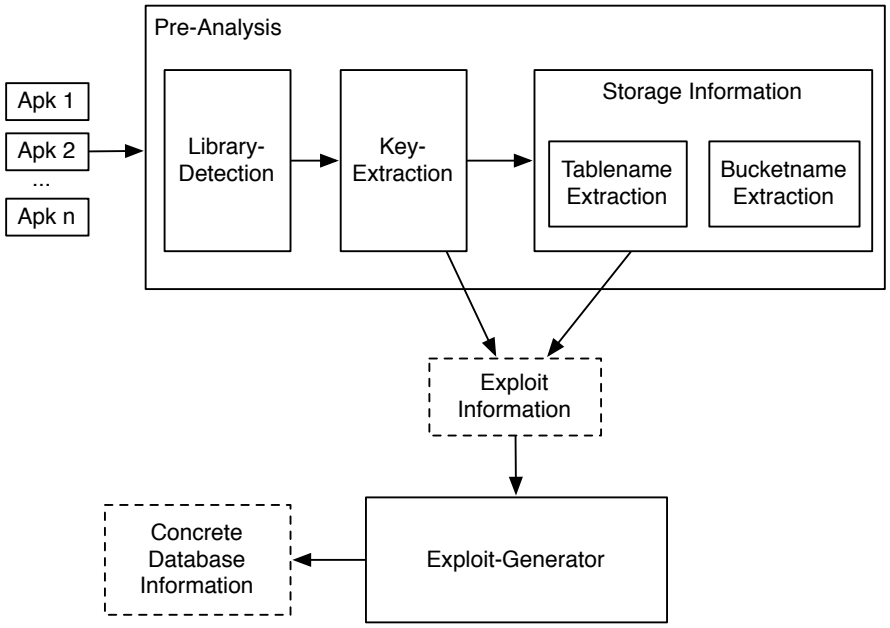

Fig. 2: Overview of the Exploit Generation Process

This detection is not trivial, as applications can be highly obfuscated and have randomly generated strings instead of the original package, class, and method names.

To overcome this challenge, we created signatures for the supported BaaS libraries. A signature consists of the numbers and types of certain selected API calls as well as some characteristics of the containing class. Even if the method `Parse.initialize()`, for instance, is obfuscated, it will usually still receive the same number of arguments with the same types. Furthermore, we exploit the observation that an obfuscator needs to retain the original behavior of the program. All calls that were made from within the original `initialize()` must still be made from within the obfuscated version. If the sequence of these calls (order and target) still match, the method is very likely to be the `initialize()` method we are looking for, regardless of its name or the name of its containing class.

All these hints contribute to a score. A method that only has the correct parameter types will receive a lower score than a method that has the correct call sequence (or a subsequence of it) inside as well. If the score exceeds a configurable threshold, we assume the library to be present. If the calls inside the Parse.com library are obfuscated as well, the call sequence might not be fully detected. In that case, the score is lower, i.e., the uncertainty is higher.

3.2 Pre-Filtering

Though HAVOC is able to identify apps that use BaaS services efficiently, scanning whole app store with millions of applications still takes considerable time. For getting a first overview over the BaaS security in today's popular app stores, we therefore propose a fast pre-filter that is able to identify likely candidate applications. The pre-filter is based on the observation that even if an application is obfuscated, the libraries are oftentimes not. They still bear their original package names. Though this filter will miss those apps that are fully obfuscated, we argue that the pre-filtered set still gives a useful bottom line on BaaS security in real-world app stores. Nevertheless, it is not a replacement for HAVOC's more thorough scanner explained in Section 3.1. A store provider might for instance want to screen new applications when they are uploaded. He would then operate on fewer apps at a time and would therefore rather use the more expensive, but also more through library identification without applying the pre-filter.

3.3 Simple Key Extraction

HAVOC is built on top of the Soot [7] program analysis framework. Soot can directly process Android APK files and convert the Dalvik bytecode into the Jimple intermediate language, a typed three-operand IL. HAVOC scans through all method bodies to find the calls to the authentication methods for the supported BaaS frameworks. For Amazon AWS, these are, for instance, calls to the constructor of the `BasicAWSCredentials` class. For Parse.com, the calls to the static `initialize` method of the `com.parse.Parse` object are of interest. The APIs for Amazon AWS, Parse.com, and CloudMine have in common that the application credentials are always passed as parameters to such methods before any other operation on the BaaS service is performed.

In the easiest (and most insecure) case, the values are constant strings and can directly be retrieved by HAVOC. This is, however, not always the case. Even if developers do not attempt to obfuscate their applications, normal software engineering practices can lead to them being places elsewhere. In many cases the credentials are stored in static final fields, either inside the class that initializes the BaaS client framework, or inside a special configuration class. These strings are nevertheless constant as shown in the example in Listing 1.3.

To efficiently cope with such cases, HAVOC applies FlowDroid's interprocedural constant-value propagator [1] before inspecting the app. The propagator looks for occurrences of static strings in static initializers or assignments. A value is considered static if the respective field or local variable is only ever assigned the same constant value, which is commonly the case for configuration classes as they comprise static final fields. All references to these fields or local variables are then replaced by the constant value found for the respective field or variable. If successful, the constant will then appear in the call to the authentication API method for one of the supported BaaS services. The code from the example in Listing 1.3 will be transformed to the code shown in Listing 1.4.

```
1 public class MyActivity extends
      Activity{
2   private String APP_ID;
3   private String CLIENT_KEY;
4
5   private void prePhase() {
6      APP_ID = "12345678";
7      CLIENT_KEY = "87654321";
8   }
9
10  @Override
11  public void onCreate(Bundle b) {
12    prePhase();
13    Cloud.initialize(APP_ID,
       CLIENT_KEY);
14  }
15 }
```

Listing 1.3: Example That Requires Inter-procedural Constant Propagation

```
1 public class MyActivity extends
      Activity{
2   private String APP_ID;
3   private String CLIENT_KEY;
4
5   private void prePhase() {
6      APP_ID = "12345678";
7      CLIENT_KEY = "87654321";
8   }
9
10  @Override
11  public void onCreate(Bundle b) {
12    prePhase();
13    Cloud.initialize("12345678",
       "87654321");
14  }
15 }
```

Listing 1.4: Example That Requires Inter-procedural Constant Propagation

The inter-procedural constant propagator can only deal with constants, but runs in $O(n)$ where n is the number of Jimple statements in the program. Only in cases where this does not yield a result, i.e., there is still no constant in the call to the authentication method, a more sophisticated (and thus more costly) analysis needs to be applied, as we detail next.

3.4 Harvesting Credentials from Applications

Some applications also store the credentials in configuration files included in the respective app's `assets` folder. Other applications have been obfuscated using string encryption. Such an app only contains an encrypted string or byte array, and then apply either some simple transformations or operations of the Java Crypto API to reconstruct the original credentials at runtime. Yet other applications hide the credentials in native code, e.g., by having simple native getters return the respective values or have a native method perform some string transformations such as in the example in Listing 1.5.

However, since the application needs the credentials during normal execution, all data and code required to reconstruct them needs to be contained inside the app in some way. In the example in Listing 1.5, one can retrieve the credentials by performing the same de-obfuscation steps that the native method in the original app would perform. To achieve this, one could place the native library into a new, artificial app, and call it with the obfuscated credentials just as the original app would do. Consequently, the method would then return the same de-obfuscated result that the original app computed. This, however, requires more complex processing than the content value propagator.

```
1 public class MyActivity extends Activity{
2
```

```
3   private native String deobfuscate(String data);
4
5   private final String APP_ID = "dsgfdsf";
6   private final String CLIENT_KEY = "lkhkjkgfhdhsfdsb";
7
8   @Override
9   public void onCreate(Bundle b) {
10    // other code
11    String appID = deobfuscate(APP_ID);
12    // other code
13    String clientKey = deobfuscate(CLIENT_KEY);
14    // other code
15    Cloud.initialize(appID, clientKey);   // s0
16    // other code
17  }
18 }
```

Listing 1.5: Example That Requires Harvester

Fortunately, encoded credentials like the one in the figure can be easily extracted with Harvester, a fully automatic tool developed by Rasthofer et al. [10]. Harvester uses a combination of static slicing and concrete dynamic execution to extract values such as the above. The tool is configured with so-called *logging points*. A logging point is a pair $\langle v, s \rangle$ where v is a variable or field access and s is a statement such that v is in scope at s. For the purpose of HAVOC, s is chosen from the set of all BaaS-service authentication methods. v are the arguments that get passed to the authentication methods and that contain the *ID* and *key*. In the example, this would be $\langle appID, s0 \rangle$ and $\langle clientKey, s0 \rangle$.

Harvester then performs a static backward slicing starting from each logging point. Every logging point (i.e., every authentication statement) yields a new slice. Any slice comprises exactly only those statements that could contribute to the computation of the v variables (i.e., the arguments containing the ID and the key). All other statements are excluded from the slice. All slices are combined to build a new, artificial executor application. This application executes every slice one after the other and reports the runtime values of the logging points to a database, as shown in Listing 1.6. The executor application can be run either on an emulator or a real phone. After this process is complete, HAVOC can directly access the runtime values of the ids and keys from the Harvester database.

```
1 public class ExecutorMainActivity extends Activity {
2
3   private native String deobfuscate(String data);
4
5   private final String APP_ID = "dsgfdsf";
6   private final String CLIENT_KEY = "lkhkjkgfhdhsfdsb";
7
8   @Override
9   public void onCreate(Bundle b) {
10    runSlice1();
11    runSlice2();
12  }
13
14  private void runSlice1() {
15    String appID = deobfuscate(APP_ID);
16    // Cloud.initialize(appID, clientKey);
17    Reporter.report("appID", "s0", appID);
18  }
```

```
19
20   private void runSlice2() {
21     String clientKey = deobfuscate(CLIENT_KEY);
22     // Cloud.initialize(appID, clientKey);
23     Reporter.report("clientKey", "s0", clientKey);
24   }
25 }
```

Listing 1.6: Slices and Executor Application Computed by Harvester

Harvester can even recover such values that are hidden inside native code (as in the example) or which are obfuscated using other concepts that a purely static analysis cannot handle.

3.5 Extracting Additional Data

In addition to the credentials as such, HAVOC also extracts other information. For Parse.com, these are the names of the tables accessed inside the app. For Amazon AWS, these are the features that are used inside the app (Dynamo, S3, etc.) as well as the S3 bucket names if S3 is used. These additional pieces of information are required for the exploit generator in case one cannot list all tables / buckets of the respective BaaS instance due to ACLs other restrictions. Even then, in many cases the tables or buckets themselves are not secured properly and can, for instance, be read or even manipulated.

To extract this additional information, HAVOC uses the same techniques as for the credentials. For table and bucket names, the constant-value propagator is sufficient in most cases and Harvester does not need to be employed. Note that we do not automatically extract the table names for Amazon Dynamo. While Dynamo is an AWS service and suffers from the same configuration issues as S3 in many applications, Dynamo is much less widespread. Additionally, the effort for building a Dynamo exploit is much higher. For querying Dynamo, one would need to reconstruct the schema of the key entries in the database. For the few applications that we found to use Dynamo, we therefore crafted exploits by hand.

3.6 Verifying the Credentials

To test whether the credentials extracted from the app are still valid, we ran carefully-chosen read-only commands on the backend BaaS infrastructure. Our main goal was to never adversely affect the productive operations of the apps from which we extracted the credentials or of the BaaS infrastructure.

For applications that used Amazon AWS, we tried to run an `ls` command to the S3 API. If successful, this command proves that the credentials are still valid and also give a list of the S3 buckets in the developer's S3 instance. The latter can give a first impression of the privacy or security implications of the vulnerability. If this command fails, it returns the reason for the failure. This can either be a non-existing application ID or missing privileges. In the first case, we assumed that the credentials were no longer valid. In the second case, we re-tried an `ls` on the concrete bucket used inside the application.

We also ran an `info` command against S3. This command returns, among other information, the access privileges of the current application ID and key. This shows whether a malicious attacker is able to not only read but also manipulate files on the S3 storage.

With a `describe-instances` command against EC2, we checked whether the credentials used inside the application also had access to EC2 which it should normally never have. For an Android app with fixed credentials, there is usually no reason to list, or even start or stop, virtual machines inside the Amazon cloud. In either case, if this check succeed, this is an indicator that the tested credentials might be full-access root credentials for AWS.

For Parse.com, we obtain the schema of the tables references inside the applications. The Parse.com API does not provide a means to list all tables in the backend, so our exploit cannot detect if additional data is freely accessible if an attacker is able to guess the correct table name. Note that system tables like `user` can directly be tested even if they do not occur in the app's bytecode as they always have the same pre-defined name.

4 Evaluation

In this section, we evaluate our findings in real-world applications. Note that we only tested accesses that did not put the regular operations of the application or the BaaS provider's infrastructure at risk. However, from our results, we are sufficiently sure that even such requests would be possible.

We do not provide the concrete names of the individual applications in which we found the security vulnerabilities. Due to the magnitude of the problem there are still quite a number of apps for which the problem has not yet been addressed. Section 5 will detail our responsible-disclosure process.

4.1 Exploit Generator

We evaluated our exploit generator over 2,000,000 real-world Android applications obtained from the Google Play Store and various third-party app stores. We also ran HAVOC on 350,000 malware applications. Many of these malware applications were re-packaged benign applications which had a malicious part (such as sending premium-rate SMS messages) attached on top of the original functionality. It is therefore not surprising that these malware applications share many security vulnerabilities with the benign ones.

For efficiency reasons, we first applied a pre-filtering to only analyze those applications that were likely to contain one of the supported BaaS libraries. According to AppBrain,[4] Parse is used in 1.35% of all apps in the Google Play Store and in 4.23% of all apps that are part of the Google Play Store's Top 500 list. While this number might appear low, our evaluation shows that nevertheless, a large number of data records is affected. Furthermore, many of these records

[4] `http://www.appbrain.com/stats/libraries/details/parse/parse`

pose severe threats to system security (e.g., through remote-code execution) or user privacy. Our pre-filtered set of apps likely to contain the Parse library consisted of 902 apps.

We then applied to these apps the analysis based on the inter-procedural constant-value propagator (see Section 3.4). In 268 apps we found a total of 308 ID/key pairs for Parse.com and 440 names of accessed Parse tables. We automatically generated exploits to verify which of the tables are freely accessible to all adversaries who know the respective application ID and key. In total, all tables contained more than 18,670,000 records. The largest number of records found through one single app was 2,611,760. On average, every app gave access to about 70,000 records. In the notion of Parse, one record is an object. Just like a table can have multiple columns, serialized Parse objects can have fields. We found that, on average, a Parse object has 6.66 fields, making the number of individual data items being exposed even larger. Counted individually, all tables contained more than 56,000,000 data items.

For Amazon AWS, our pre-filter returned 1,922 apps. These apps were different ones than those that we found for Parse.com as developers usually do not combine frameworks from different BaaS providers inside one app. In 763 of these apps, we found 772 ID/key pairs. Only 106 of these keys, however, were unique, which means that application developers who create multiple applications often share one ID/key pair between multiple apps. The worst case was one key which was used in 195 apps. Two other keys were used in 124 and 122 apps respectively. This is critical, because it hinders a conceptual isolation in the backend. A security vulnerability with one of the apps leads to all apps with that key being affected as well. In only 72 of the apps that were returned by the prefilter and that did indeed contain the AWS library, was the constant propagation-based exploit generator too weak to extract the keys, which is why for these cases we had to extract the keys with Harvester (Section 3.4). This was successful in every instance.

We also verified whether iOS apps from the official App Store were vulnerable to the same security issue. We found 82 unique Amazon AWS keys in 128 apps from a set of 11,000 apps crawled from the App Store. This shows that iOS apps share the same problem as Android apps: Developers on both platforms embed their keys into the apps rather than using more secure authentication methods. iOS developers also share keys between applications. One key, for instance, was used in 9 different apps of the test set.

4.2 Privacy-Sensitive User Data

The keys we were able to extract from various applications give an attacker access to a broad variety of privacy-sensitive information. A single app's backend store, for instance, contained more than one million e-mail addresses. This list would be especially helpful for a spammer or scammer as the addresses were part of user profiles managed with Parse.com. This means that all addresses were verified by the Parse.com framework during user registration and are thus guaranteed to be valid (or at least have been at the time of registration).

Another app stored sensitive health information in an insecure way. The AWS key contained in the application (in plain text) gave access to more than 1,400 records of baby-growth and feeding data as well as photos of the respective children. The e-mail addresses of the parents which were also accessible were even a minor concern in this case. An attacker did not need to know any credentials from the parents or the children to get access; the application ID and key extracted from the app's code were sufficient.

A chat application gave access to pictures uploaded by the respective users and the contents of websites that were hosted via the same S3 storage instance. In an online dating application, tables containing voice-chat messages were freely accessible. The app vendor claims that more than 90 million people have signed up for their dating website and app. The Google Play Store reports that the app has between 10 and 50 million installations.

One application allowed users to record car-crash data. When a crash happened, the user can take photos of the scene, record voice comments on the situation, and make additional notes. However, since the application ID and key can easily be extracted from the app and the backend database is not properly secured, an unauthenticated user who only knows this ID and key can extract all this data from all users.

Applications that use Parse.com benefit from a convenient integration into the Facebook API. This is not surprising, as Parse.com is now owned by Facebook Inc. In some apps, however, this integration impairs the privacy mechanisms of Facebook. One app, for instance, downloads a user's friend list and stores it in a custom Parse table. Since there is no access restriction on this table, it is sufficient to extract the vulnerable app's ID and key to gain access to all friend records (including non-public friends) of all users of this app. The app's users generally must allow the app to access their Facebook data. But unsuspecting users will not assume that this makes their friend data permanently available to arbitrary remote adversaries. We have furthermore identified a similar issue in an app that integrates with Twitter.

Some applications use AWS credentials that have access to buckets that are not even required for the application. With these credentials, we would have been able to retrieve Excel spreadsheets containing customer data (full names and customer IDs), server logs, and database backups. Especially the customer records could be abused by scammers who can try to impersonate a customer using social engineering against a human company representative.

4.3 Data-Manipulation Vulnerability

With the default settings of user-defined tables in Parse.com, an attacker who only knows the application ID and key can manipulate each records inside every table of the whole database. In the car-accident documentation app discussed earlier, this means that an attacker can manipulate the records taken by his opponent before charges are filed in court. Even if these records are not allowed as evidence in court, manipulating or deleting them makes it harder for the opponent to prepare his defense.

Parse.com write-protects the pre-defined `user` table by default. Write operations to this table are only permitted after a user authentication has been performed and only to the record of the authenticated user, but not to any other records inside the table. Still, Parse allows a user to freely modify his own record. As a consequence, if an application stores additional profile data along with the user object, this data can be tampered with. One app, for instance, added a flag `isPaid`. Knowledge of the application ID and the key is sufficient to set this flag to `true` for the own user account even if no payment was ever made.

4.4 Financial Harm to BaaS Clients

The various commercial BaaS providers have different revenue models. Some of them charge for the data transferred between their datacenter and the customer (or, more precisely, his applications), others bill their clients based on the average number of requests per minute. Attackers can exploit this model by downloading large datasets (or, in case of frequency, requesting a large number of records) from the backend. Since applications based on BaaS usually perform their business logic on the client, i.e., inside the app, there is no trusted entity between the attacker-controlled client and the backend. Therefore, the validity of requests can only be checked with very general anomaly-based fraud-detection schemes on the side of the BaaS provider. Such fraud-detection schemes, if in place at all, however, fundamentally suffer from the problem of having to cater to a broad variety of unknown client applications that all have their individual usage and data transfer patterns.

A similar issue arises with malicious uploads, since all providers analyzed in this paper charge their clients based on the size of the data stored on their services. With Parse.com, 20 GB cost $200.[5] This is a severe issue given that the default settings of Parse.com allow clients to not only arbitrarily create new records, but also completely new tables. Even worse, this also allows for file uploads. Such malicious uploads might not even be detected by the application developer who owns the storage, due to a design issue with Parse.com: One first uploads a file and receives an identifier. Only in a second step is this identifier then associated with a record in a table. If one uploads a file and discards the identifier, the file becomes orphaned and can never be retrieved again. Parse.com staff even acknowledges that this might be a problem in case of a buggy application [6]. Their platform offers a *cleanup* function to remove those orphaned files in bulk, but this is neither a real solution to the original problem of orphaned files, nor does it alert the developer that a cleanup is necessary in case of storage filling up due to abuse.

[5] `https://www.parse.com/questions/how-to-get-a-list-of-uploaded-files-that-are-not-associated-with-objects`

[6] `https://www.parse.com/questions/how-to-get-a-list-of-uploaded-files-that-are-not-associated-with-objectsandoffers`

4.5 Remote Code Execution

For some applications it is insufficient to simply store data in the cloud. Amazon therefore offers a service called *Elastic Beanstalk* to host web applications inside the Amazon cloud. These applications can be written in a variety of languages such as Java, PHP, and .NET. A Java application for Elastic Beanstalk, for instance, will be compiled into a `war` file and uploaded to an S3 bucket for storage. An *EC2* (Elastic Compute Cloud) instance then executes this file.

If a user gains access to the S3 bucket in which the code is stored, he can download, potentially decompile, and then inspect the code. If the code accesses further backend services like databases, the credentials for these services become available to the attacker as well. He can simply extract them from the server-side code or from configuration data stored inside the S3 bucket alongside the code.

From various applications, we were able to extract credentials that could not only read arbitrary S3 buckets (including ones hosting web pages and server-side code), but which also had write access. Such write access allows an attacker to run arbitrary server-side code inside the Amazon cloud. Furthermore, the attacker could modify and misuse S3-hosted websites to distribute malware or perform phishing scams against unsuspecting users with ease.

Even worse, we were able to extract keys that gave us administrative access to EC2, i.e., that would have allowed us to view the developer's instances running on EC2. A malicious user could also have created new server VM instances in the developer's account for hosting activities such as Bitcoin mining, the sending of spam, or for hosting malware. It would also have been possible to shut down or even delete existing VMs.

4.6 Sending Spam Messages

Amazon offers a service called *Simple Email Service* (SES), which can be used to send purely outgoing (mass) e-mail messages. To use the service, one only needs a valid application ID and key. We have found apps, which use this service. However, if the app is able to send e-mail messages using these credentials, everyone with the same credentials can. Attackers thus get an easy mechanism to send spam and phishing e-mails using Amazon's infrastructure at the cost of the app developer who gets charged based on the number of messages sent. If the e-mail service is used from within EC2 or Elastic Beanstalk, Amazon even allows 62,000 free messages. An attacker who exploits a foreign key can thus send a large number of messages even without risking exposure through unexpected invoices.

4.7 Malware

Malware applications also use BaaS frameworks. Some of them store malware-specific configuration data in a Parse.com table. One application, for instance, used Parse to manage the list of country codes for which no malicious behavior should be performed. Security experts and law-enforcement officials could use

this knowledge to first revoke write access to this table and then add all existing country codes to this table to disable the malware. The user-registration information with which the Parse.com instance was created could also give hints to the identity of the malware author.

4.8 Push Notifications

We were able to extract credentials from Android applications that allowed us to view a list of existing Amazon Simple Notification Service (SNS) channels. For some applications and their channels, we were even able to list the devices registered with these channels. The device information contained the location, the manufacturer and model, the Android OS version, and other information with which the respective device can be fingerprinted. Additionally, such information also allows to automatically scan for outdated devices and deliver appropriate exploits. The actual delivery can, for instance, happen by sending a push notification pointing the user to a malicious download or URL. The user is then likely to believe that he acts upon a legitimate notification from the respective app, giving the message increased credibility. Several automated trading platforms digest news messages to initiate automated stock trades. In the past, misinformation in news channels has caused havoc at stock exchanges several times. One might envision situations in which the vulnerabilities here could cause such an effect if the respective channel is subscribed by the respective trading companies.

4.9 Key-Hiding Attempts

Some applications try to mitigate the security problem by obfuscating the code that uses the BaaS API, especially those functions that provide the credentials. While such attempts can increase the time required for an attacker to recover the credentials, it does not resolve the fundamental issue of all required authentication data being available in the app. Attackers can use debuggers, symbolic execution, or hybrid data extraction [10] to obtain the data.

Some apps tried to hide the credentials in native code, but used functions that were mere "getters". Harvester can easily extract such keys. More thorough approaches encrypted the credentials using AES and obfuscated the AES encryption key using various string-manipulation operations. There was, however, even a large set of apps from one developer that simply flipped the key string (from back to front) as an "obfuscation".

4.10 Legacy Application Data

Some of the applications that we analyzed have been removed from the Google Play Store in the meantime. However, even if an application is no longer publicly available, this does not directly resolve a backend-related security vulnerability. In many cases, the keys that we extracted from locally cached copies of the applications which we crawled months ago were still valid. We were still able to

access the respective BaaS instances and extract the data that the app used to store there.

This poses the question of what happens to the data of an application that is no longer maintained and who is responsible for maintaining the security of this data. While users who still have such applications installed on their devices might appreciate still being able to use the respective service, their data is also greatly at risk due to vulnerabilities no longer being fixed. The situation gets aggravated with apps that do not provide their users a means of deleting their data once they no longer trust the application. Worse, if an application is no longer maintained, there is oftentimes no notice to the user that would make them aware of the potential security problems of continued use of that app.

5 Responsible Disclosure

Since the problems we describe in this paper directly concern the security and privacy of many smartphone users, we follows a responsible disclosure process prior to submitting this publication. We informed the security teams of Amazon and Facebook (who now owns Parse.com) as these providers were mainly affected by our findings.

Due to the high number of vulnerable applications, we were unable to contact all corresponding application developers directly. Because all applications were (or used to be) available through the Google Play Store or the Apple App Store, we worked together with our government's CERT,[7] Google and Apple to help spread the word through appropriate channels. With this approach, we leveraged on Google's and Apple's mass capabilities for reaching out to the developers in their respective stores. The goal of this effort was to get as many applications as possible secured before publishing our results to the general public. Nevertheless, for vulnerabilities that are such widespread, it is almost impossible to avoid information leaks to the public, as several hundred people will need to be involved in the process.

Legacy applications that have been removed from the respective application store but for which data is still hosted with a provider (see Section 4.10) are especially hard to handle. Not all of these developers are still interested in maintaining these apps or securing the data. In this case, we relied on the BaaS provider to handle the case. We provided the respective official security contacts with all identifiers extracted from the apps. All communications were encrypted to make sure that none of our extracted IDs became available to unauthorized people.

6 Mitigation

An important step in mitigating the security vulnerabilities described in this paper is for application developers to properly apply the security features already

[7] Name omitted due to double-blind review

offered by the various BaaS frameworks. Only blaming the developers would be too simplistic, though. Their main goal usually is to get the app into the market as soon as possible without spending too much extra effort on nun-functional aspects. BaaS providers such as Parse explicitly benefit from this requirement. They abstract away from backend handling and reduce it to a handful of lines of code that every developer can just copy&paste into his app without further knowledge or consideration. Every additional mandatory step would contradict their own business model of abstraction and simplicity.

Still, given the severity of the problem, we argue that the default security settings need to be more restrictive. We argue that forcing developers to explicitly allow access to tables instead of granting full access to everything by default is a reasonable cost to pay. The security configuration must be sufficiently simple, however. While Amazon's IAM is much more flexible than Parse's ACLs, they are also too complex for the average developer to handle. Providing pre-configured templates (e.g., "read-only bucket", "records only visible to the user who created them") could help solve this problem.

In general, developers need to be better educated in the security and privacy implications of software development. We hope that examples of vulnerabilities such as the ones in this paper help raise developer awareness. The exploit generator presented in this paper could also be used by app-store providers to check new uploads and counter the distribution of vulnerable apps. Apps from which keys can automatically be extracted could then be rejected from stores such as Google Play, effectively forcing the respective developer to fix the issue. Amazon's app store already applied such checks that warn the developer if she uploads a potential insecure app containing Amazon credentials. Furthermore, BaaS features that are not necessary for the majority of productive applications such as creating new tables on the fly should be disabled by default.

Still, all these measures require application developers and users to trust the BaaS provider. To mitigate this risk, the providers should include easily usable end-to-end encryption and authentication methods into their open-source client SDKs.

7 Related Work

Static and dynamic data flow tracking tools such as FlowDroid [1] and TaintDroid [3] can detect whether private information is leaking from an app to a remote service. Sending the e-mail address of a customer to a back-end service is however expected behavior and the more existence of the flows therefore does not indicate a security issue. Assessing the security of the used back-end service is out of scope for these approaches.

To find apps that use a specific library such as *Parse* to interact with their respective backends, one can use approaches like DroidSearch [9]. DroidSearch crawls the Google Play Store and various other stores. It conducts lightweight pre-analyses on all the apps it downloads such as extracting all calls to library

methods or finding all requested permissions. A security analyst can then query these results to find interesting apps for further investigation.

In previous work, Fahl et. al [4] have shown that Android application developers make security-critical mistakes in other areas as well, in particular when accessing remote web sites using SSL. Viennot et. al. [11] already found and verified a number of AWS credentials contained in apps as constant strings. They, however, did not investigate further the privacy and security implications of the access gained through these credentials, nor did they evaluate non-constant or obfuscated credentials in applications. We were furthermore able to prove that the issue is not limited to Amazon AWS, but is shared between backend-as-a-service providers in general. Bugiel et al. [2] were able to extract sensitive information including credentials from publicly available pre-configured images for Amazon EC2 virtual machine instances. With these credentials, an attacker could gain full control over the respective virtual machines.

Mansfeld-Devine [8] points out that the Mercury tool from MWR Labs can find hard-coded online service passwords in the manifest. In our investigation, the credentials were however contained in the code or other custom configuration files, not in the manifest. An article by Steve Gold [5] quotes a bi-annual investigation on Android security by Veracode. They report that 40% of all Android apps contained at least one hard-coded cryptographic key. While our focus is on credentials, cryptographic keys are an equally important topic and share many of the security implications discussed in this paper.

Kalutarage et. al. [6] propose a certification process for Android apps which checks these apps according to various *Common Vulnerability Enumeration* (CWE) entries, including CWE 259 ("Hard-Coded Password").

8 Future Work

As future work, we plan to extend HAVOC to automatically generate exploits for iOS apps as well. Since many BaaS frameworks are available for both Android and iOS, the structure and the security implications are often very similar. Furthermore, many frameworks also have client-side interfaces that allow them to be used from normal desktop applications written in Java or C/C++. Web-based interfaces that allow the frameworks to be integrated into web applications are also common. We plan to investigate whether these are vulnerable as well, and, especially, whether data sanitization is applied correctly. By storing data on one platform and retrieving it on another, additional vulnerabilities might be exploited.

9 Conclusion

In this paper, we presented HAVOC, a fully automated exploit generator for Backend-as-a-Service (BaaS) frameworks used in Android applications. All frameworks we investigated require the applications to authenticate using an application ID and a key. While this is not a proper security feature, application developers

tend to solely rely on it and refrain from implementing further security precautions. Measures like ACLs or restricted credentials as offered by Parse.com or Amazon AWS are only used in the minority of cases, leaving many apps unprotected against data extraction and manipulation.

Unlike previous work, this paper shows the severity of the issue and proves that it is not limited to a single provider, but affects all commonly used BaaS providers. We show that iOS applications suffer from the same issue. Our exploit generator allows application developers and store operators to check apps for this kind of vulnerability.

We have suggested several mitigations to these problems, from better defaults for BaaS platforms, to better developer education and automatic vulnerability checks on applications uploaded to app stores. In general, app developers need to better understand that every app has security implications, which must be taken into consideration as part of the basic design of the app.

Acknowledgements This work was supported by the Deutsche Forschungsgemeinschaft within the project RUNSECURE, the BMBF within EC SPRIDE and ZertApps, by the Hessian LOEWE excellence initiative within CASED, and by the DFG Collaborative Research Center CROSSING.

References

1. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 29. ACM, 2014.
2. S. Bugiel, S. Nürnberger, T. Pöppelmann, A.-R. Sadeghi, and T. Schneider. Amazonia: When elasticity snaps back. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 389–400, New York, NY, USA, 2011. ACM.
3. W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.
4. S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.
5. S. Gold. Android: A secure future at last? *Engineering & Technology*, 7:50–54(4), March 2012.
6. H. Kalutarage, P. Krishnan, and S. Shaikh. A certification process for android applications. In A. Cerone, D. Persico, S. Fernandes, A. Garcia-Perez, P. Katsaros, S. A. Shaikh, and I. Stamelos, editors, *Information Technology and Open Source: Applications for Education, Innovation, and Sustainability*, volume 7991 of *Lecture Notes in Computer Science*, pages 288–303. Springer Berlin Heidelberg, 2014.
7. P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, Oktober 2011.

8. S. Mansfield-Devine. Android architecture: attacking the weak points. *Network Security*, 2012(10):5 – 12, 2012.
9. S. Rasthofer, S. Arzt, S. Huber, M. Kohlhagen, B. Pfretschner, E. Bodden, and P. Richter. Droidsearch: A tool for scaling android app triage to real-world app stores. *Proceedings of the IEEE Technically Co-Sponsored Science and Information Conference*, 2015.
10. S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. 2015.
11. N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '14, pages 221–233, New York, NY, USA, 2014. ACM.