Security Smells Pervade Mobile App Servers

Pascal Gadient
Marc-Andrea Tarnutzer
Oscar Nierstrasz
Software Composition Group, University of Bern
Bern, Switzerland
scg.unibe.ch/staff

Mohammad Ghafari
University of Auckland
Auckland, New Zealand
m.ghafari@auckland.ac.nz

ABSTRACT

[Background] Web communication is universal in cyberspace, and security risks in this domain are devastating. [Aims] We analyzed the prevalence of six security smells in mobile app servers, and we investigated the consequence of these smells from a security perspective. [Method] We used an existing dataset that includes 9 714 distinct URLs used in 3 376 Android mobile apps. We exercised these URLs twice within 14 months and investigated the HTTP headers and bodies. [Results] We found that more than 69% of tested apps suffer from three kinds of security smells, and that unprotected communication and misconfigurations are very common in servers. Moreover, source-code and version leaks, or the lack of update policies expose app servers to security risks. [Conclusions] Poor app server maintenance greatly hampers security.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

security smells, web communication, mobile apps

ACM Reference Format:
Pascal Gadient, Marc-Andrea Tarnutzer, Oscar Nierstrasz, and Mohammad Ghafari. 2021. Security Smells Pervade Mobile App Servers. In *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21), October 11–15, 2021, Bari, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3475716.3475780

1 INTRODUCTION

Globally accessible, reliable, and scalable web services are on the rise, with more than 24 000 currently known public web APIs.[1] Likewise, native apps are starting to decline while web apps arise that depend on application servers.[2] Additionally during the past years, the complexity of developing a web-enabled app has massively increased due to the growing number of involved application frameworks, programming languages, and supported device categories, *e.g.*, desktops, notebooks, tablets, smartphones, and wearables.

Web communication has already received much attention in the security community, leading to improved tool support. For example, programs exist that can continuously monitor web APIs to ensure that an app remains compliant with the API specification [14], and tool support to mitigate insecure communication channels has been built into recent releases of the Android ecosystem.[3] Generally speaking, the existing literature has covered the transmitted payload by using client-side static [5] and dynamic analysis techniques [12], server side analyses of web service source code [9] as well as connection properties, *e.g.*, the URL [16] or two-factor authentication [13]. Another topic that has received extensive attention is that of hard-coded credentials in apps that may allow adversaries unrestricted access to the relevant infrastructure [11, 15].

Unfortunately, the server configurations of off-the-shelf application servers have received much less attention. A recent work has identified eight web API security smells, but did not assess their prevalence [3]. These smells are *symptoms in the code that signal the prospect of a security vulnerability* [4]. In this work we assess app servers that are used for communicating with mobile apps. We investigated the presence of six app server security smells, and the corresponding server maintenance activity based on the dataset that contains 9 714 distinct URLs that were used in 3 376 apps. We address the following research questions:

RQ$_1$: *What is the prevalence of the server side security smells in the web communication of mobile apps?* We found 231 URLs from 44 apps that leak the source code of the web service implementation if processing errors occur. We can further confirm that most app servers communicate with apps over insecure HTTP connections [10], and fail to enforce use of the HTTP strict transport security policy. Finally, we found that on average almost every second app server suffers from version information leaks.

RQ$_2$: *What is the relationship between security smells and app server maintenance?* In particular, we are interested in configuration changes, because they provide insights into established maintenance processes of mobile app servers. Based on the collected HTTP header information from two measurements over 14 months, we evaluated what software changes are introduced by system administrators. We observed that servers are usually set up once and never touched again, yielding severe security risks. For instance, criminals can attack outdated app servers by exploiting vulnerabilities listed in public databases or illicit websites. On the positive

[1] https://www.programmableweb.com/apis/directory
[2] https://www.forbes.com/sites/victoriacollins/2019/04/05/why-you-dont-need-to-make-an-app-a-guide-for-startups-who-want-to-make-an-app/
[3] https://developer.android.com/training/articles/security-config

ESEM '21, October 11–15, 2021, Bari, Italy

ACM ISBN 978-1-4503-8665-4/21/10…$15.00
https://doi.org/10.1145/3475716.3475780

side, we noted that version upgrades are much more common than version downgrades, and that developers occasionally use Cloudflare to protect their infrastructure against adversaries, especially for non-JSON-based app servers.

In summary, this work reveals the prevalence of insecure app server configurations accessed by Android mobile apps, and their maintenance protocol. The list of apps that we analyzed in this study is available online,[4] and we share the aggregated data for research purposes on request due to the contained sensitive information such as credentials, API keys, and email addresses.

The remainder of this paper is organized as follows. We report on app server security smells relevant for this work in Section 2. We describe the dataset used for our app server investigations in Subsection 3.1. We provide the prevalence of app server security smells in Subsection 3.2, and we shed light on server infrastructure maintenance in Subsection 3.3. Finally, we recap the threats to validity in Section 4, and we summarize related work in Section 5. We conclude this paper in Section 6.

2 SECURITY SMELLS IN APP SERVERS

In this section, we briefly explain six of the eight security smells that we identified in previous work [3]. The two remaining security smells, *i.e.*, *Credential leak* and *Embedded languages* are not within the scope of this study, because they require a deep understanding of the app and the context where they occur.

2.1 Insecure transport channel

Web communication relies on HTTP or HTTPS; both variants exist in app server configurations. *Issue:* HTTP does not provide any security; neither the address, nor the header information or the payload are encrypted. *Symptom:* The URL begins with `http://`.

2.2 Disclosure of source code

Error messages leak valuable information regarding the implementation of a running system. *Issue:* Error messages that include the relevant stack trace are transmitted as plain text in the server's message response body. Such a message reveals information like the used method names, line numbers, and file paths disclosing the internal file system structure and configuration of the server. *Symptom:* The returned HTTP body contains a stack trace or a code snippet that shows the problematic code. The structure of such data depends on the used framework, however terms related to application crashes are common, *e.g.*, "stack," "trace," and "error."

2.3 Disclosure of version information

Besides useful connection parameters, HTTP headers leak information regarding the software architecture and configuration of a running system. *Issue:* Outdated software suffers from severe security vulnerabilities. For instance, a server that returns `X-PoweredBy: PHP/5.5.23` in the response header uses a PHP version that is at the time of writing more than 6 years old, and a quick search in the Common Vulnerabilities and Exposures (CVE) database shows that this framework suffers from 65 known security vulnerabilities, six of which received the most severe impact score of 10.[5] *Symptom:* One of the following keys exists in the response header: `engine`, `server`, `x-aspnet-version`, or `x-powered-by`.

2.4 Lack of access control

Authentication by a user name and a password provides tailored experiences to end users, *e.g.*, individual chat logs or friend lists, and at the same time enables access control to separate and protect sensitive user data. *Issue:* The access to sensitive data or actions is not restricted by a sane authentication mechanism such as a user name and password pair, instead, easy-to-forge identifiers or no identification data at all are used to secure the access. *Symptom:* A server does not respond with the status code *401 Unauthorized* or *403 Forbidden*. In other words, the server successfully responds without asking for any credentials.

2.5 Missing HTTPS redirects

We found servers that do not redirect clients to encrypted connections although they would have been supported. *Issue:* App servers do not redirect incoming HTTP connections to HTTPS when legacy apps try to connect. *Symptom:* For an HTTP request, a server does not deliver an HTTP `3xx` redirect message which points to the corresponding HTTPS implementation of the web application. For an HTTPS request, a server delivers a HTTP redirect message.

2.6 Missing HSTS

HTTP header information is used to properly set up the connection by specifying various communication parameters, *e.g.*, the acceptable languages, the used compression, or the enforcement of HTTPS for future connection attempts, a feature which is called *HTTP Strict Transport Security (HSTS)*. HSTS provides protection against HTTPS to HTTP downgrading attacks, *i.e.*, when a user once accessed a web resource in a secure environment (at home or work), the client knows that the resource needs to be accessed *only* through HTTPS. If this is not possible, *e.g.*, at an airport at which an attacker tries to perform MITM attacks, the client will display a connection error. Hence, HSTS should be used in combination with HTTP to HTTPS redirects, because the HSTS header is only considered to be valid when sent over HTTPS connections. *Issue:* Servers either do not leverage the HSTS feature, or they do not use the recommended parameters. *Symptom:* A server does not deliver the HTTP HSTS header `Strict-Transport-Security: max-age=31536000; includeSubDomains` for an HTTPS request.

3 EMPIRICAL STUDY

For this empirical study we evaluated all URLs from the dataset according to the security smell symptoms described in the previous section. We collected the data twice: the initial download of HTTP headers and bodies was performed in June-2019 whereas additional data, *i.e.*, the authorization errors and up-to-dateness, was retrieved in August-2020. The duration of 14 months is arbitrary but long enough to ensure developers have to update their software infrastructure.

[4] https://doi.org/10.6084/m9.figshare.14981061

[5] https://www.cvedetails.com/vulnerability-list/vendor_id-74/product_id-128/version_id-183021/PHP-PHP-5.5.23.html

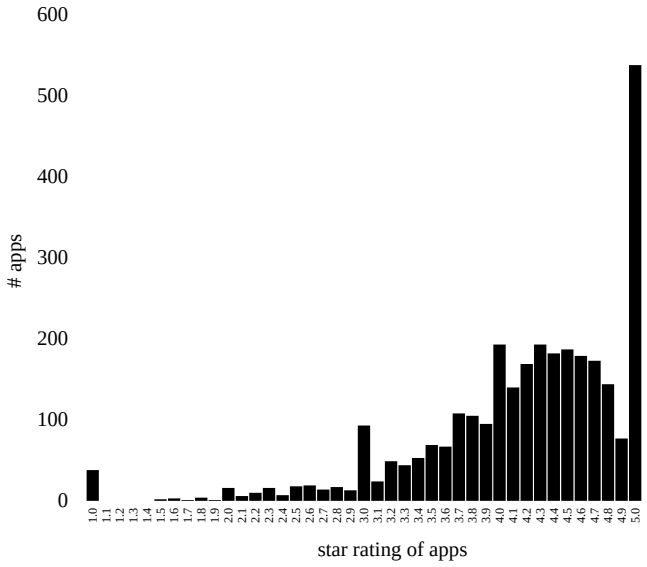

Figure 1: Star ratings for the Google Play apps in the dataset

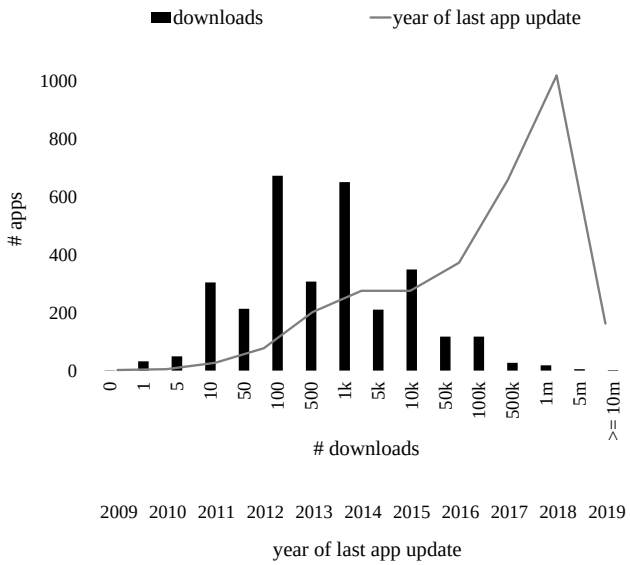

Figure 2: The popularity and developer support for the Google Play apps in the dataset

3.1 Dataset

We build on our previous work and dataset [3] in which we manually inspected Android apps to identify which APIs developers use to call web services, and how they are used. We then took advantage of this information to develop a tool to automatically extract and reconstruct string variables and the assigned values, the server URLs and their corresponding HTTP request headers statically from the apps. Using this information, we analyzed the reconstructed app server data and tried to establish connections to the corresponding servers from which they gathered additional information for analysis, *i.e.*, from HTTP response headers.

The apps from the dataset are randomly collected from those that use Android's internet permission. For closed-source apps we mined the free apps on the *Google Play* store, and for the open-source apps we relied on the *F-Droid* software repository.[6] For each app, we removed the duplicates, *i.e.*, apps with the same package identifier, but different version numbers, and kept only the most recent version of the app. We also included the partial results of the apps whose analysis was incomplete and could not finish in time, ultimately resulting in an analysis result for 303 open-source, and 3 073 closed-source apps in the dataset.

The apps in the dataset come from 48 different Google Play store categories. Most of them belong to `EDUCATION` (317 apps) and `TOOLS` (292 apps), however, a majority (574) have a `GAMES`-related tag. Interestingly, work-related apps are common in the dataset (335 apps). The top five categories whose apps contain the largest number of distinct URLs are `EDUCATION` (1 555 URLs), `LIFESTYLE` (1 027 URLs), `BUSINESS` (995 URLs), `ENTERTAINMENT` (704 URLs), and `PRODUCTIVITY` (619 URLs). As shown in Figure 1, almost 94% of the apps received a star rating of 3.0 or higher. Surprisingly, apps with a five star rating are more prevalent than apps in any other category. The apps have an average star rating of 4.2 stars and a median rating of 4.3 stars. Figure 2 presents the number of app downloads and the timeliness of app updates. The y-axis denotes the number of apps in each category. In contrast, the primary x-axis with the bars indicates the app downloads, and the secondary x-axis with the line indicates the time of the last app update. We can see that most apps achieved between 100 and 1 000 downloads, and barely any app was downloaded more than 1 million times. Regarding the app updates, most of the apps received an update in 2018. Therefore, we see that most vendors update their apps only a few times a year, because we collected the statistics separately in 2019.

We then exercised every URL in the dataset and collected the HTTP header and body of each server response. Eventually, we processed 1 230 open-source URLs and 8 486 closed-source URLs. We realized that many app servers do not leverage JSON, but instead they use, for example, XML or plain HTTP communication. Because we were interested whether there exist any differences for data-centric app servers, we split the results into four different groups. We report our findings based on closed-source and open-source apps, and we also separate between JSON and non-JSON app servers. We favored the JSON data format, because it was much more commonly used for communication than the others. Therefore, we partitioned the open-source URLs into 1 171 non-JSON URLs and 59 JSON URLs. Accordingly, we partitioned the closed-source URLs into 7 997 non-JSON URLs and 489 JSON URLs.

We were particularly interested in information such as operating system identifiers, used software modules, and version numbers. Hence, we crafted a number of search queries to detect occurrences of such features. The relevant features, *i.e.*, security smells, and the results are part of the discussion in the subsequent subsections.

[6] https://f-droid.org

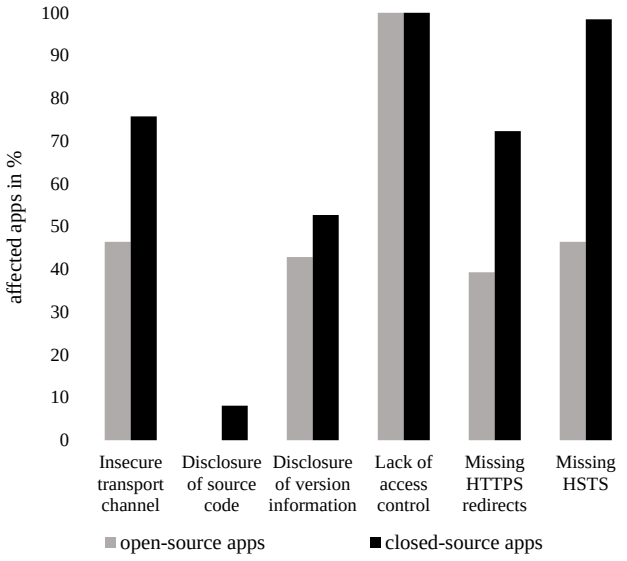

Figure 3: Prevalence of app server smells in apps considering JSON communication

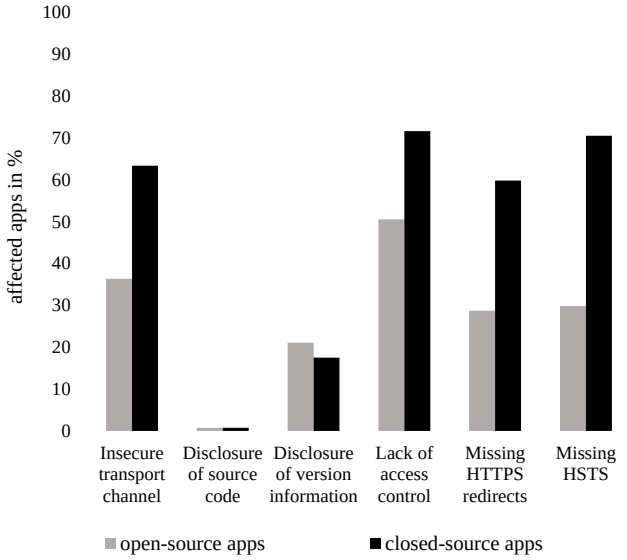

Figure 4: Prevalence of app server smells in apps considering non-JSON communication

3.2 Prevalence of Security Smells

This subsection answers **RQ**$_1$: *What is the prevalence of the server side security smells in web communication?* In Figure 3 and Figure 4 we report on the relative prevalence of app server security smells in apps for JSON and non-JSON web services, respectively. In Figure 3, the vertical axis indicates the percentage of apps that suffer from a specific app server security smell. In the following, we discuss the findings from different perspectives, *i.e.*, security smell categories, software development model, and technology.

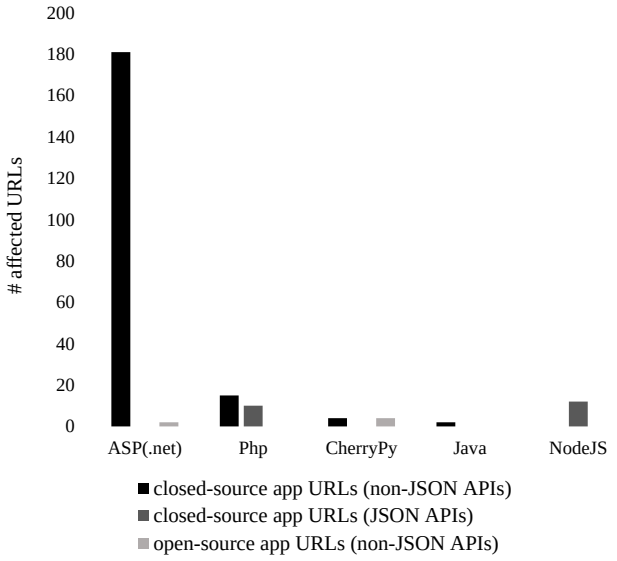

Figure 5: Frameworks that caused code leaks

3.2.1 By Security Smell Category. We report findings and provide actionable advice to mitigate the issue for each security smell.

Insecure transport channel. Communication through an insecure transport channel is prone to data leaks and manipulation, *e.g.*, an adversary could alter conversations. Hence, practitioners should avoid HTTP and instead focus on the secure HTTPS. Third-party libraries that require HTTP should be replaced with ones that support secure communication. With respect to URLs from open-source apps, we found that 582 non-JSON app servers (50%) did not use protected communication. Fortunately, this is not the case for JSON app servers: only six JSON app servers (10%) used plain text communication. We found worse results in closed-source communication. Secure communication was usually unavailable, *i.e.*, 5 639 non-JSON app servers (71%) used HTTP. A total of 245 JSON app servers were not protected (50%).

Disclosure of source code. Leaked code is valuable for adversaries to plot their attacks, or for competitors to glimpse into the source code and the architecture. Therefore, administrators should disable verbose error messages on production environments and review the default settings. We could identify stack traces from five different server frameworks, *i.e.*, ASP(.net), CherryPy, Java, NodeJS, and Php. As we can see in Figure 5, URLs from closed-source applications suffer the most from code leaks, *i.e.*, we found 225 instances (2.7%) where 182 instances can be assigned to the ASP(.net) framework. Considering URLs used in open-source software, we only found six instances (0.5%) primarily caused by ASP(.net) and CherryPy.

Disclosure of version information. The knowledge of what exact software runs on a server is crucial for successful attacks. Consequently, administrators should disable the self-promotion of services and review their default settings. In Figure 6, we present the

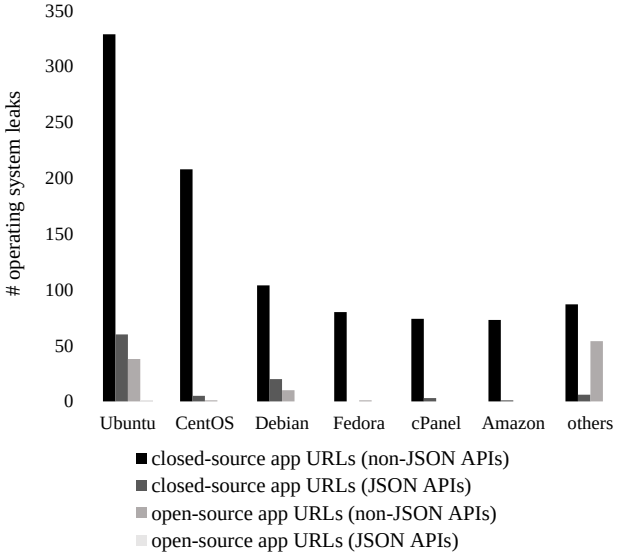

Figure 6: Disclosure of operating system information

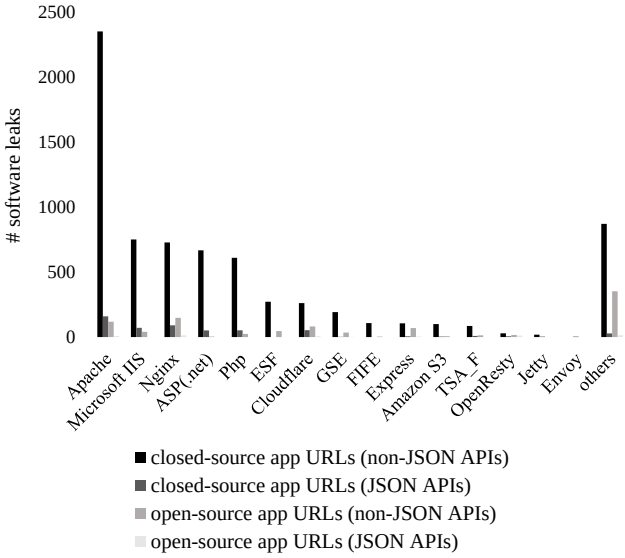

Figure 7: Disclosure of service information

found operating system leaks in app servers, where the y-axis denotes the number of leaks we found. We found 1 155 operating system leaks in our dataset. Ubuntu and Debian are the most prevalent operating systems for JSON app servers, and CentOS is rather used for non-JSON app servers. Customized Linux distributions, *i.e.*, cPanel and Amazon, are less commonly used among web application developers. In Figure 7, we present the found service leaks in app servers, where the y-axis denotes the number of leaks we found. We found 8 707 service leaks in our dataset, including servers that pack up to three leaks into a single HTTP response. Open-source

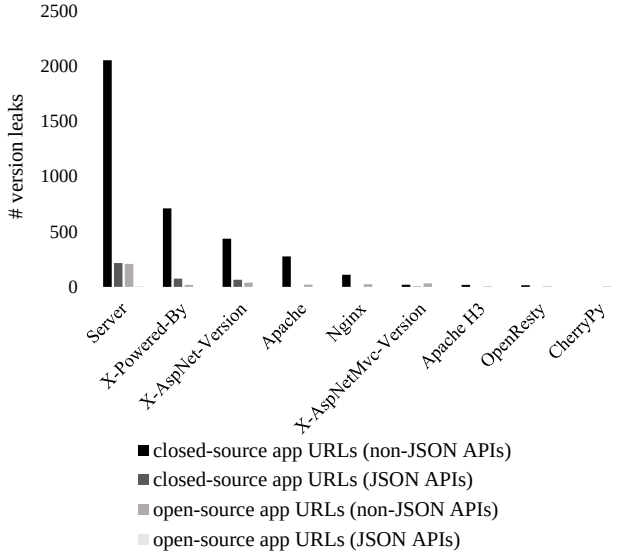

Figure 8: Disclosure of version information

and closed-source software behave similarly, *i.e.*, Apache and Nginx are among the top three web application gateway servers used, but Microsoft services, *i.e.*, Microsoft IIS and ASP(.net), remain a preferred choice for closed-source developers. Interestingly, the web security provider Cloudflare is used not only for numerous closed-source apps, but also for open-source apps, as we expect, due to their free plans. Furthermore, the service leaks indicate that most of the app servers do not use the Google Cloud API (ESF) or storage services such as Amazon S3. In Figure 8, we present the found version leaks in app servers, where the y-axis denotes the number of leaks we found. We found 3 992 closed-source and 359 open-source software leaks in our dataset. Most version leaks occur for both closed-source and open-source app servers in the HTTP header field `Server`, followed by `X-Powered-By`, and `X-AspNet-Version`. The leaks in HTTP bodies, *i.e.*, `Apache`, `Nginx`, `Apache H3`, `OpenResty`, and `CherryPy` are less prevalent than those found in the headers.

Lack of access control. Unprotected information can be accessed by everyone on the internet. Since apps usually provide experiences tailored to each user, their servers should use well known authentication schemes to prevent leaks of personal data. We encountered 53 HTTP authentication errors for closed-source non-JSON app servers, and 28 errors for open-source non-JSON app servers. We did not find any such errors for open-source or closed-source JSON app servers. However, there exist JSON web applications that returned arbitrary authorization errors in the JSON format, *e.g.*, using OAuth instead of the HTTP mechanism.

Missing HTTPS redirects. Missing redirects leave flawed or outdated clients vulnerable to eavesdropping. Redirects should always be set in place, if a server has ever been accessible through the insecure HTTP protocol. Redirects can be chained, but they should be used sparingly. As shown in Figure 9, we found server responses with missing HTTPS redirects in the URLs from 4 961 closed-source apps and from 387 open-source apps. Fortunately, we did not find

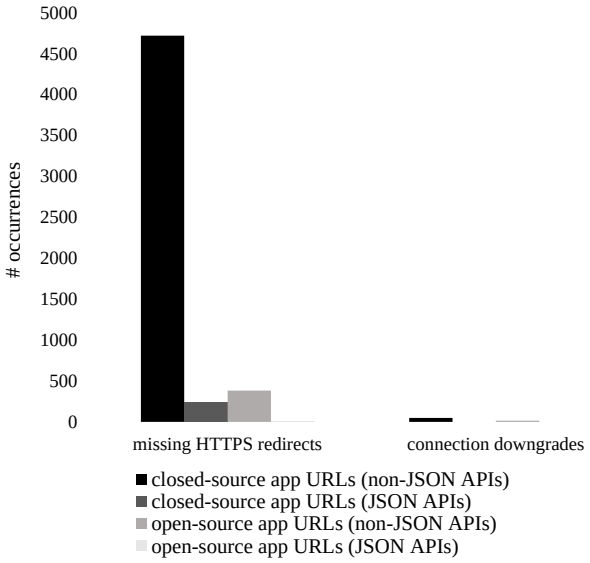

Figure 9: Missing HTTPS redirects in app servers

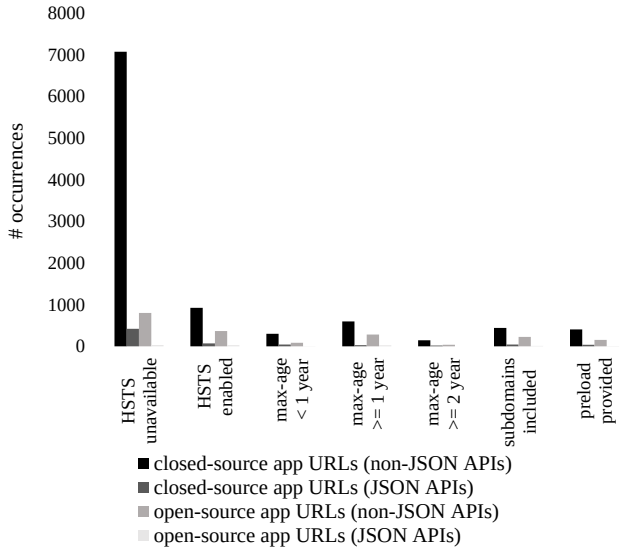

Figure 10: Missing HSTS protection for app servers

any HTTPS to HTTP connection downgrades in JSON app servers, but we found 48 for closed-source non-JSON app servers and 15 in open-source non-JSON app servers. Concerning forwarded requests, closed-source app servers forwarded the requests on average 1.3 times, open-source non-JSON app servers 1.5 times, and open-source JSON app servers once. We found two request loops, *i.e.*, infinite redirects from a destination to itself, in each open-source and closed-source app servers. Without the request loops, open-source app servers redirected a request up to three times, and closed-source app servers up to seven times.

Missing HSTS. App servers without proper support for HSTS expose users to eavesdropping due to possible HTTPS to HTTP connection downgrades. Therefore, servers should deploy this feature to every subdomain and request the client side caching of this setting for at least one year. Ultimately, the protected URLs should be added to the publicly available HSTS preload list that is included in all major browsers. As shown in Figure 10, we found 7 494 closed-source app servers and 833 open-source app servers that miss HSTS HTTP headers. Only a minority of the connections are protected, that is 397 (34%) of all open-source app servers and 992 (12%) of all closed-source app servers. Contrary to recommended practices,[7] 432 app servers use `max-age` values shorter than one year, and 785 do not use the preload feature. In other words, 31% of the app servers that support HSTS have not sufficiently configured the protection for subdomains, and 57% lack the preload feature that enforces security already for the first request.

3.2.2 By Software Development Model. We report findings for two different software development models, *i.e.*, the open-source and the closed-source software development model. We can clearly see in Figure 3 and Figure 4 that closed-source apps generally suffer from more security smells than open-source apps. Especially *Lack of access control* and *Missing HSTS* appear in the communication of almost all closed-source apps. Moreover, the three smells *Insecure transport channel*, *Missing HTTPS redirects* and *Disclosure of version information* are less frequent, but exist still in more than 52% of all closed-source apps and in more than 39% of all open-source apps. Interestingly, *Disclosure of source code* only emerges in closed-source app communication.

3.2.3 By Technology. We report our findings for two different technologies, *i.e.*, the JSON and non-JSON-based web communication. According to Figure 3, access control and unprotected HTTP communication constitute major threats for apps that use JSON web services. However, apps that do not rely on JSON communication are apparently more robust against security smells: such apps are on average about 19% less affected by them. Code leaks primarily occurred in JSON communication. For instance, *Disclosure of source code* only exists in less than 1% of the apps that use non-JSON web services, whereas it is more than 8% for the apps that use regular JSON web services. We only found code leaks in JSON app servers that use the Php or NodeJS framework, but in contrast, we found code leaks in non-JSON servers from almost every major framework.

3.2.4 Summary. App server security smells pose a severe threat. Most security smells exist in more than 25% of all apps, regardless whether the app is open-source or closed-source, and whether it uses a JSON or non-JSON app server. Particularly alarming is the finding that apps using JSON app servers suffer 1.5 times more from app server security smells than non-JSON apps, and even worse, closed-source applications suffer 1.6 times more compared to open-source applications.

More than 50% of the servers accessed by mobile apps use unprotected HTTP communication. Since smart devices are becoming

[7] Google Chrome HSTS preload list submission form, https://hstspreload.org/

rather personal assistants, they carry much sensitive information that needs adequate protection.

Misconfigured app servers cause code leaks. Although only little code is revealed at a time, an attacker can replay requests and alter parameters to reconstruct the architecture and logic behind the service. Such information eases the search for bugs in the code.

The leaked information is devastating. Although intended for publicity purposes, the currently leaked data reveals very often not only the operating system running on the server, but also the installed services and their version number. Such information can be entered into vulnerability databases to find suitable security issues that could be exploited.

Based on our results, access control for JSON app servers is currently not implemented with HTTP status codes, but instead with arbitrary replies. A standardized approach would help in creating more service independent apps, and at the same time default authorization templates could be used from back-end developers.

HTTPS redirects are usually inexistent for HTTP-based app servers. Even worse, some downgrade a HTTPS connection to an insecure HTTP connection. Moreover, redirect loops exist occasionally, and few redirect implementations use more than five redirects which is not recommended by RFC2068.[8]

Finally, HSTS is only set up for a minority of app servers, and for those it is common to have weak configurations.

In conclusion, we see that security smells are very prevalent in app servers. In fact, every app references on average more than three servers that suffer from at least one of these smells.

3.3 Maintenance of Server Infrastructure

In order to answer **RQ**$_2$: *What is the relationship between security smells and app server maintenance?*, we investigate maintenance operations performed on the servers used by mobile apps. In particular, we are interested whether app server administrators have updated their infrastructure within the time period of 14 months, and if we see a correlation between the number of identified security smells and the quality of server maintenance. The selected duration of more than a year covers multiple bug fixes including major releases of common server software, *e.g.*, Apache, Microsoft IIS, or PHP. We accessed the URLs by sending an HTTP GET request, and stored their HTTP header responses twice, *i.e.*, once in June-2019 and once in August-2020. We can only compare version numbers between the two datasets if we received some version information in the HTTP `Server` header. As a result, the data in this section are based on fewer responses, *i.e.*, from 309 open-source (JSON and non-JSON) app server URLs (25%) and 3 006 closed-source (JSON and non-JSON) app server URLs (35%).

During our manual analysis of the first 100 entries, we encountered eight different scenarios: i) no updates have been applied, *i.e.*, the software name and version remains identical, ii) the version has been downgraded, *i.e.*, the software name remains identical, but the version number decreased, iii) the version has been upgraded, *i.e.*, the software name remains, but the version number increased, iv) the version leak has been closed, *i.e.*, the software name remains, but the version number is not anymore available, v) the environment has changed, *i.e.*, the software has been replaced and it might use a different versioning scheme, vi) Cloudflare protection has been enabled, *i.e.*, the server has moved behind a Cloudflare protection gateway and does not anymore leak version information, vii) server spawned, *i.e.*, we received no software name in the first run, but we received one in the second run, viii) server shutdown, *i.e.*, we received a software name in the first run, but not anymore in the second run. We could not gather security-related changes for 1 254 app server URLs for several reasons: i) new server instances have been spawned without prior knowledge of software configurations, ii) existing server instances have been shutdown without the possibility to find any changes, or iii) the environment has changed using a different versioning scheme.

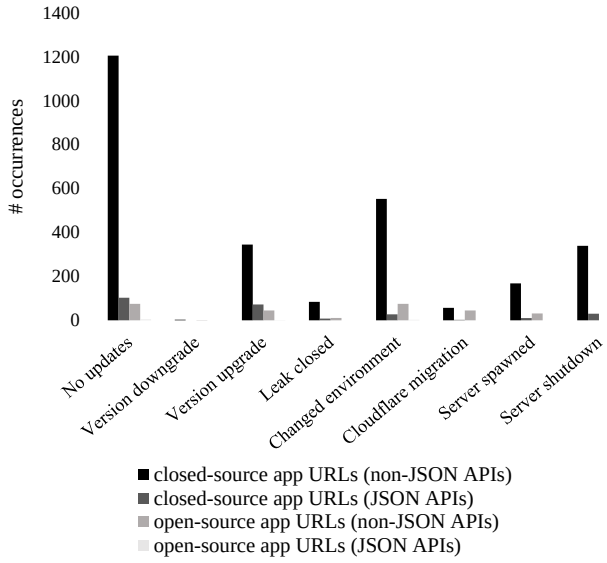

Figure 11: Configuration changes of app servers after 14 months

3.3.1 Configuration Changes. In Figure 11, we show the results. From the app servers that leaked versioning information, by far most closed-source non-JSON app servers did not undergo any changes to the server software. Closed-source JSON app server infrastructure seems to be updated more frequently, however the majority still do not provide any updates. The same is true for open-source software although less evident. Version downgrades occurred sparsely, *i.e.*, four times, and not for JSON app servers. Only a fraction of the leaking servers, *i.e.*, 103 (4%), has been configured to mitigate the leaks. Interestingly, environment changes occur more frequently for open-source non-JSON app servers than no updates at all. In other words, open-source developers seem to replace app servers rather then updating them. Moreover, Cloudflare support has been enabled for 104 app servers, *i.e.*, for 45 open-source URLs and for 59 closed-source URLs. Finally, more servers are shut down than spawned.

3.3.2 Correlation of Security Smells. Figure 12 shows the correlation between app server security smells and administrative configuration changes. For this figure, we consolidated all app server

[8] RFC2068,HTTP/1.1,https://tools.ietf.org/html/rfc2068#section-10.3

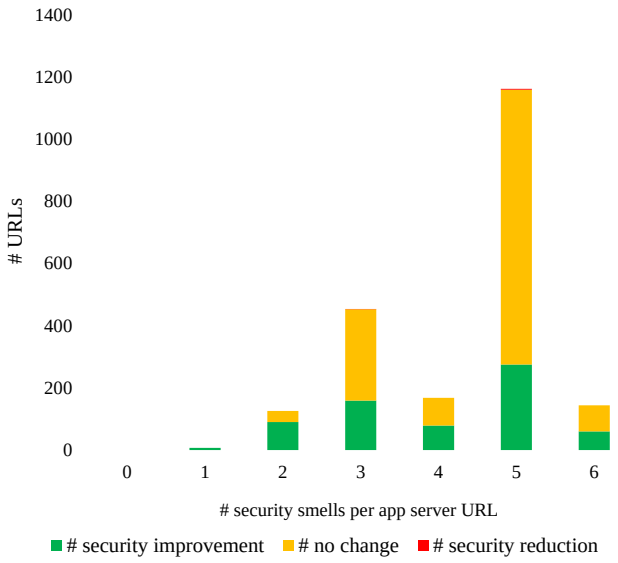

Figure 12: Correlation between app server security smells and configuration changes

categories, *i.e.*, open-source, closed-source, JSON, non-JSON due to the limited number of elements in some of them. The x-axis denotes the number of security smells from which a particular app server suffers, and the y-axis indicates how many such app servers exist in each category. Based on the versioning information from 2 061 URLs, we can see that app servers suffering from three or more smells are usually not well maintained, *i.e.*, they are set up once and then left alone. Although security improvements, *i.e.*, version upgrades, the removal of versioning information, and the migration to Cloudflare appear more frequently in instances that suffer from more than one smell, they only affect a minority. Security downgrades, *i.e.*, the change to a more dated version, appear only in app servers that massively suffer from security smells, *i.e.*, from three or more smells.

3.3.3 Summary. According to our findings, app servers are usually set up once and never touched again. This paradigm introduces severe security risks due to outdated software running on publicly accessible interfaces. Hence, sensitive user data could be exfiltrated when adversaries apply suitable exploits to such systems. Luckily, version upgrades are much more common than version downgrades, although they cannot at all compensate for the lack of change. We expect that downgrades were performed to circumvent new bugs or compatibility issues, because all downgrades considered only minor release changes, *e.g.*, from *nginx* release 1.14.1 to 1.12.1. Some developers shift to Cloudflare to protect their infrastructure especially for non-JSON app servers.

We conclude that app server security smells seem to be a good indicator for poor server maintenance. In fact, the more smells an app server has the more likely it is that server maintenance processes are broken.

4 THREATS TO VALIDITY

Completeness. A major threat to validity is the completeness of the used dataset built from Android apps. Although state of the art decompilation tools have been used, only about 37% of all closed-source Android apps could be successfully decompiled for the subsequent analysis. Of these decompiled apps, the analysis for 22% could not finish in time and might have led to incomplete results. Moreover, the analysis tool skipped the evaluation of bundled build scripts and XML resources that could have pointers to additional app servers. This threat cannot be mitigated entirely, however the rather large and diverse set of included apps ensures that the results can be generalized.

Accuracy. Another important threat represents the accuracy of the used dataset. According to the authors, the tool that has been used to build the dataset achieves a precision of 46% and a recall of 80%. However, this performance is the result of a manual analysis of decompiled code performed by the authors which included only ten open-source and ten closed-source apps that comprised 22 web API URLs. In particular, it reported several URLs unrelated to web APIs but to static HTML pages, and the tool occasionally reconstructed invalid requests. In this work, we do not depend on accurate requests, *i.e.*, the investigated response headers are identical even for malformed requests. In fact, most of the reconstructed requests contained placeholders that we could leverage to see whether the app servers leak sensitive information in case of errors.

Data collection. The collected data might contain duplicates or suffer from temporal issues. Some requests we generated from the URL might have reached identical servers which ultimately lead to duplicated connection information in the result set. Another problem is that of server side outages or configuration changes that temporarily cause unexpected or erroneous results. To mitigate these threats, we filtered the URL list for duplicates, and we used rather long timeouts and a high retry count when we accessed the servers.

Selection bias. The data used for the investigation of server maintenance represents only a subset of the original dataset. This is an immediate result of the many servers that do not leak any data. Even more, for the qualitative analysis we require two responses, each containing versioning information. In order to reduce the impact of these threats, we manually reviewed the first 100 server responses to ensure that we do not miss any version information. We then designed the value extraction process for the individual version numbers based on the results of this initial exploration.

Recency. The data set contains apps that have been downloaded in 2018, and the corresponding metadata has been collected in 2019. This might change the results due to improved development processes and tools. However, recent works still identified a lack of security in web communication [1, 6].

Security risks. The risks associated with the security smells are not necessarily severe. We do not know what and how much data the web services hoard, and many of the risks directly correlate with the confidentiality of the data. Since we cannot easily obtain this information, we follow a defensive strategy, *i.e.*, we assume that every server might host at least some sensitive data.

Construct validity. There is a threat to construct validity through potential bias in our expectancy.

5 RELATED WORK

Related work primarily pertains to app analyses that have been summarized by the concept of security code smells, data transmissions with a particular interest in web communication, and public service audits that improve the app server security. We present relevant literature in each of these three research areas in the remainder of this section.

5.1 Security Code Smells

The research about security code smells investigates the metamorphosis from unfavorable code that could become a security threat. Ghafari *et al.* collected 46 000 closed-source apps from the official Android market and investigated the nature and prevalence of common mistakes developers suffered. For that purpose, they introduced the notion of a *security code smell* and used it to identify 28 different security smells in five different categories [4]. They found that *XSS-like Code Injection*, *Dynamic Code Loading*, and *Custom Scheme Channel* are the most prevalent smells, many of them leveraging inter-component communication features of the Android operating system. As a result, Gadient *et al.* started to study the prevalence of *Inter-Component Communication (ICC)*-related security smells in more than 700 open-source apps, and found that security code smells that involve web communication prevail against others, and that such issues are often introduced with new feature updates of apps [2]. Moreover, the manual investigation of 100 apps demonstrated the usefulness of their tool, *i.e.*, about 43% of the reported smells were in fact vulnerabilities. Since many of the newly discovered smells relied on responses from web applications, they consequently began to investigate the web API communication of mobile apps [3]. The preliminary results of their static analysis tool, which has been used to mine security code smells were devastating: In 3 376 apps, they encountered credential leaks, excessive use of embedded languages such as SQL and JavaScript, insecure web communication including source-code and version information leaks to name a few. As a matter of fact, they found that unprotected web communication is seven times more prevalent in closed-source apps compared to open-source apps, and that embedded code is used in web communication in more than 500 different apps. Our work continues this research, *i.e.*, we investigate the server side prevalence of the reported security smells.

5.2 Web Communication

Web communication in apps is usually initiated by the client, *i.e.*, the app that sends a request to a specific server. Therefore, apps can reveal interesting features used to establish such a connection. For example, Zuo *et al.* analyzed 5 000 top-ranked apps in Google Play and identified 297 780 URLs that they fed to the VirusTotal URL screening service [16]. The service identified 8 634 harmful URLs of which the majority related to malware (43%), followed by malicious sites (37%), and phishing (23%). Mendoza *et al.* investigated the input validation constraints imposed by apps on outgoing requests to web API services from 10 000 popular free apps from the Google Play Store of which 46% suffered from inconsistencies that could be exploited by attackers [9]. Such inconsistencies allowed them to access app-related databases through various injection attacks, *e.g.*, they could misuse an app's email address field for an SQL injection attack, because its value did not receive additional server side validation. We found many similarities in the results of our work: advertisement services were omnipresent and proper authentication measures were barely implemented. For instance, access to personal information was protected by the sole use of a single attribute, *e.g.*, an email address or hardware-based identifier.

5.3 App Server Security

Finally, app server security focuses on server side problems, configuration or implementation. Zuo *et al.* found that 15 098 app servers are subject to data leakage attacks [17]. In particular, they suffer either from a broken key management, *i.e.*, the developers became confused about root and app keys, or from a broken permission configuration, *i.e.*, developers were overwhelmed when they had to choose appropriate permissions for their data. They assume that this is a direct consequence of the utterly complex interfaces to configure such services designed for developers. That is, Google even provides a language for developers to specify the desired user permissions. With respect to web servers, Lavrenovs *et al.* worked through responses of the top one million Alexa websites, and collected security-related information such as HSTS support, protection against cross site scripting, and other HTTP headers that might impose a security risk [7]. They found that website popularity is the major driver for security measures. In fact, the implementation rates compared against the Alexa ranking reveal an exponential decline pattern, *i.e.*, all of their analysed security headers started to be much more prevalent in the top 50 000 websites, and that ratio steeply increased towards the top websites. Moreover, Mendoza *et al.* found discrepancies between the use of such features in the mobile and desktop version of websites that enable various injection and spoofing attacks, although the affected websites remain in the realm of a few percent [8]. Although we can confirm these results, according to our study a lack of security is much more prevalent in apps that use JSON app servers, especially in closed-source apps that are 11% more susceptible to such issues than their open-source counterparts.

6 CONCLUSION

We analyzed the prevalence of six security smells in app servers and investigated the consequence of these smells from a security perspective. We used an existing dataset that includes 9 714 distinct URLs that were used in 3 376 Android mobile apps. We exercised the URLs twice over 14 months, and stored the HTTP headers and bodies. We realized that the top three smells exist in more than 69% of all tested apps, and that unprotected communication and server misconfigurations are very common. Particularly alarming is the finding that apps using JSON app servers suffer 1.5 times more from app server security smells than non-JSON apps, and even worse, closed-source applications suffer 1.6 times more compared to open-source applications. Moreover, source-code and version leaks, or the lack of update policies foster future attacks against these data centric systems. We found that app server security smells are omnipresent and they indicate poor app server maintenance.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assistance" (SNSF project No. 200020-181973, Feb. 1, 2019 - April 30, 2022).

REFERENCES

[1] Eman Salem Alashwali, Pawel Szalachowski, and Andrew Martin. 2020. Exploring HTTPS security inconsistencies: A cross-regional perspective. *Computers & Security* 97 (2020), 101975.

[2] Pascal Gadient, Mohammad Ghafari, Patrick Frischknecht, and Oscar Nierstrasz. 2018. Security Code Smells in Android ICC. *Empirical Software Engineering Special Issue* (2018). https://doi.org/10.1007/s10664-018-9673-y

[3] Pascal Gadient, Mohammad Ghafari, Marc-Andrea Tarnutzer, and Oscar Nierstrasz. 2020. Web APIs in Android through the Lens of Security. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 13–22.

[4] M. Ghafari, P. Gadient, and O. Nierstrasz. 2017. Security Smells in Android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 121–130. https://doi.org/10.1109/SCAM.2017.24

[5] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. 2015. Information flow analysis of Android applications in DroidSafe.. In *NDSS*, Vol. 15. 110.

[6] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. 2021. A large-scale analysis of HTTPS deployments: Challenges, solutions, and recommendations. *Journal of Computer Security* Preprint (2021), 1–26.

[7] Arturs Lavrenovs and F. Jesús Rubio Melón. 2018. HTTP security headers analysis of top one million websites. In *2018 10th International Conference on Cyber Conflict (CyCon)*. 345–370. https://doi.org/10.23919/CYCON.2018.8405025

[8] Abner Mendoza, Phakpoom Chinprutthiwong, and Guofei Gu. 2018. Uncovering HTTP Header Inconsistencies and the Impact on Desktop/Mobile Websites. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 247–256. https://doi.org/10.1145/3178876.3186091

[9] Abner Mendoza and Guofei Gu. 2018. Mobile application web API reconnaissance: Web-to-mobile inconsistencies & vulnerabilities. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 756–769.

[10] Andrea Possemato and Yanick Fratantonio. 2020. Towards HTTPS Everywhere on Android: We Are Not There Yet. In *29th USENIX Security Symposium (USENIX Security 20)*. 343–360.

[11] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure As Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 164–175. https://doi.org/10.1109/ICSE.2019.00033

[12] Marianna Rapoport, Philippe Suter, Erik Wittern, Ondřej Lhóták, and Julian Dolby. 2017. Who You Gonna Call?: Analyzing Web Requests in Android Applications. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, Piscataway, NJ, USA, 80–90. https://doi.org/10.1109/MSR.2017.11

[13] Longji Tang, Liubo Ouyang, and Wei-Tek Tsai. 2015. Multi-factor web API security for securing Mobile Cloud. In *2015 12th International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*. 2163–2168. https://doi.org/10.1109/FSKD.2015.7382287

[14] Erik Wittern, Annie T.T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. 2017. Statically Checking Web API Requests in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 244–254. https://doi.org/10.1109/ICSE.2017.30

[15] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting developer credentials in Android apps. In *WISEC*. 1–12.

[16] Chaoshun Zuo and Zhiqiang Lin. 2017. SMARTGEN: Exposing Server URLs of Mobile Apps With Selective Symbolic Execution. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 867–876. https://doi.org/10.1145/3038912.3052609

[17] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1296–1310. https://doi.org/10.1109/SP.2019.00009