Why Johnny Can't Make Money With His Contents: Pitfalls of Designing and Implementing Content Delivery Apps

Sze Yiu Chau
Purdue University
schau@purdue.edu

Bincheng Wang
The University of Iowa
bincheng-wang@uiowa.edu

Jianxiong Wang
Purdue University
wang1822@purdue.edu

Omar Chowdhury
The University of Iowa
omar-chowdhury@uiowa.edu

Aniket Kate
Purdue University
aniket@purdue.edu

Ninghui Li
Purdue University
ninghui@purdue.edu

ABSTRACT

Mobile devices are becoming the default platform for multimedia content consumption. Such a thriving business ecosystem has drawn interests from content distributors to develop apps that can reach a large number of audience. The business-edge of content delivery apps crucially relies on being able to effectively arbitrate the purchase and delivery of contents, and govern the access of contents with respect to usage control policies, on a plethora of consumer devices. Content protection on mobile platforms, especially in the absence of Trusted Execution Environment (TEE), is a challenging endeavor where developers often have to resort to ad-hoc deterrence-based defenses. This work evaluates the effectiveness of content protection mechanisms embraced by vendors of content delivery apps, with respect to a hierarchy of adversaries with varying real-world capabilities. Our analysis of 141 vulnerable apps uncovered that, in many cases, due to developers' unjustified trust assumptions about the underlying technologies, adversaries can obtain unauthorized and unrestricted access to contents of apps, sometimes without even needing to reverse engineer the deterrence-based defenses. Some weaknesses in the apps can also severely impact app users' security and privacy. All our findings have been responsibly disclosed to the corresponding app vendors.

CCS CONCEPTS

• **Security and privacy** → *Digital rights management*; *Cryptanalysis and other attacks*; *Access control*; *Authorization*; *Mobile and wireless security*; *Software security engineering*; • **Applied computing** → *E-commerce infrastructure*;

ACM Reference Format:
Sze Yiu Chau, Bincheng Wang, Jianxiong Wang, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2018. Why Johnny Can't Make Money With His Contents: Pitfalls of Designing and Implementing Content Delivery Apps . In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, Article 4, 16 pages. https://doi.org/10.1145/3274694.3274752

ACSAC '18, December 3–7, 2018, San Juan, PR, USA

ACM ISBN 978-1-4503-6569-7/18/12…$15.00
https://doi.org/10.1145/3274694.3274752

1 INTRODUCTION

The ubiquity of mobile devices has encouraged *content owners* (e.g., publishing houses and record labels) to tap into the online business ecosystem in an attempt to reach a larger number of audience. As a result, they often retain the service of app-developing *content distributors* to adapt to this emerging trend of customer engagement. The role of content distributors is focused on developing mobile apps tailor-made to fit the form of the contents and the business model of content owners, and providing continuous technical support in updating and distributing digitized contents (e.g., magazines and music). For maintaining their business edge it is crucial for the content distributors to ensure that the end users cannot easily have unfettered access to the raw, high-quality reproduction of contents in their devices, even in the cases where the digital contents can be consumed without Internet connectivity (e.g., offline playback). *The overarching goal of this paper is to systematically identify (and, in the process, educate developers about) design weaknesses in content delivery apps that can grant users unauthorized and unrestricted access to the underlying content.*

At a first glance, it may seem that the design of an effective *content protection* mechanism boils down to effectively enforcing *Digital Rights Management* (*DRM*). We however argue that there is a subtle distinction between the two. DRM enforcement is concerned about regulating user's access to the content *after* the content (and, the corresponding usage control policy and other bootstrapping information) has been securely delivered to the user's device. On the other hand, content protection mechanisms, especially in the context of online content distribution, *also* have to guarantee reliable receipt of payments and secure delivery of contents (and the accompanying control policies) to the user device.

In this paper, we demonstrate that, in many cases, an adversary (the device owner in our context) can modify the enforcement policy [1] while it is being delivered to the device during bootstrapping, to achieve unfettered access to raw contents.

The challenges of effective content protection enforcement is further exacerbated by the fact that content distributors, to increase their audience reach, often need to support a plethora of (legacy) devices running various (legacy) versions of operating system (OS). Even after successful bootstrap, effectively enforcing DRM, without considering the *analog loophole problem*—the Achilles' heel of

[1] Readers may question the rationale of delivering the usage control policies during app bootstrap, instead of having them hardcoded inside the apps. Delivering policies during bootstrap allows for flexible customization of various aspects of the policies (for instance, number of free trials allowed), to better fit the business decisions.

any DRM systems–is a challenging ordeal and requires content distributors to conceal secret states in a potentially hostile execution environment. In different systems, secret states manifest in various aspects of the underlying mechanisms, for example the content encryption keys, authorization tokens, subscription status, or even the raw content itself. The general consensus is that effective DRM enforcement is feasible on a device equipped with a trusted execution environment (TEE). Technologies like the ARM TrustZone have been available on the hardware architecture level for some years now. Nevertheless, various system-on-chip (SoC) vendors have come up with different TEE implementations that do not seem to conform to the same API standard [20]. Together with the fact that TEE vendors often adopt a tight admission control model, currently it is still somewhat difficult to develop widely deployable apps that uses TEE for DRM needs, especially on relatively low-end and legacy devices that do not have the trusted images preloaded. Due to this lack of a generic secure solution that is applicable to all ⟨device, OS⟩ pairs, developers often resort to a best-effort, *deterrence-based enforcement* of DRM—specialized for each ⟨device, OS⟩ pairs—where the main objective is to raise the bar for mounting successful bypass attacks instead of providing absolute enforcement guarantees. Since content protection is only as strong as the weakest links (*e.g.*, legacy ⟨device, OS⟩ pairs) in the ecosystem, this presents an interesting trade-off between audience reach and the strength of content protection.

Content Protection Enforcement Analysis. In this paper, we systematically identify the different attack surfaces a content delivery app may expose, and how those can be exploited by adversaries to bypass protection enforcement. In our analysis, we consider two abstract classes of adversaries, namely, the *network adversary* (who can observe and manipulate network traffic) and the *local adversary* (who can access internal states and possibly tamper with the execution environment). For each of the two classes, we further consider varying degree of adversary capabilities, ranging from a normal tech savvy user to more sophisticated ones like rooting the device and TLS interception capabilities. With respect to our hierarchy of adversaries, we present concrete attacks against 141 Android content delivery apps, including some high-profile ones like the *Amazon Music*, *Bloomberg Businessweek+*, and *Forbes Magazine* apps. Our evaluation reveals a bleak state of the affair. We observed that all the apps are susceptible to our attacks due to unjustified trust assumption on the underlying technologies, *e.g.*, insecure bootstrapping and policy delivery, and bad practices like client-side policy enforcement, and reuse of content encryption keys. Whenever possible, we further dissect the weaknesses that our attacks exploited and categorize them using the Common Weakness Enumeration (CWE) [2]. We believe that, with the patterns provided by our concrete analysis, this paper lays a solid foundation for further research on automated vulnerability detection and the development of more robust apps.

Findings. In our evaluation of publication apps, a somewhat uncharted territory for academic studies, many apps are not only falling short in terms of content protection, but contain weaknesses that allow remote exploits which threaten the app users' security and the privacy. Notable among our findings is the *purchase bypass attacks* against the Forbes Magazine and Mother Earth News apps which allows an adversary, with sufficient filesystem permission, to manipulate the purchase status for gaining unauthorized access to all issues for free. Another example of relying only on client-side enforcement of usage control policies was exhibited by the Bloomberg Businessweek+ app, for which a network adversary with TLS interception capability can rewrite the policies on the fly during app bootstrap, and obtain *virtually unlimited free previews* of its subscription-only articles. We observed that the service of a content distributor often gets retained by a diverse group of publishers. Since apps from the same distributor tend to be similar in their designs, our attacks affect a large number of different publications.

One popular choice for DRM enforcement is to employ a cryptographic cipher to encrypt the underlying content, in which case the robustness of the enforcement hinges on the concealment of the secret key, as the Kerckhoffs's principle mandates. This approach was embraced by the Amazon Music app, which was the most robust app analyzed in our evaluation. While the app seems to be programmed with the good practice of minimizing exposure time of cryptographic keys in memory, we were still able to devise a *key extraction attack* to extract the underlying content encryption keys by leveraging some non-complex binary instrumentation. To our surprise, a closer inspection of the app revealed that against best practices, the entire Amazon Music collection of 40 million songs seems to be encrypted under one single content encryption key, irrespective of accounts, device models, and subscription tiers. Our successful key extraction attack hence puts their entire collection in serious jeopardy.

Given that our findings could potentially affect the business of various stakeholders, we have engaged in responsible disclosure (Section 5.1), and careful ethical considerations have been taken during and after our experiments (Appendix A.1).

Contributions. In summary, this paper makes the following contributions:

1) We identify attack surfaces and practical adversaries with varying degree of sophistication that vendors of content distribution mobile apps should consider, in order to devise effective content protection mechanisms, especially in the absence of a trusted execution environment (TEE) support from the underlying platform.

2) We systematically evaluate 141 content distribution apps developed for Android with respect to our identified hierarchy of adversaries. Our evaluation uncovered that more often than not developers of these apps make unjustified trust assumptions about the underlying platform and design decisions that enable an adversary to circumvent these protection mechanisms without having to reverse engineer the apps to extract the underlying secret state.

3) We dissect and classify the weaknesses that our attacks exploited with respect to CWEs to help future developers avoid the same pitfalls. We have responsibly shared our analyses with the corresponding content distributors. With the understanding of various attack strategies, we discuss possible countermeasures, their trade-offs and implications.

2 SCOPE

We now discuss the attack surfaces, threat model, and platform that we consider in this paper.

[2] https://cwe.mitre.org/

2.1 Attack Surfaces

Without loss of generality, in the following discussions, we consider encryption is used to protect the underlying contents. The normal operation of a content distribution app can be roughly broken down into the following six phases: (1) Bootstrapping; (2) Storing authorization token; (3) Content transmission and storage on the device; (4) Playback preparation; (5) Content decryption; (6) Content playback. Each of the above steps presents opportunities for the attacker to bypass content protection and thus corresponds to one of the following attack surfaces. Depending on the actual implementation, certain steps might be skipped or merged, and events might happen in a slightly different order. In this paper we focus on **(AS1–4)**.

(AS1) Bootstrapping. In the bootstrapping phase, the app authenticates itself and the user to the content distributor's back-end server, obtains a list of available contents and their prices, as well as authorization to access them. This step may involve monetary transactions, if this is the first time that a user subscribes to the service, or purchases contents. Successful completion of the bootstrapping process may result in the back-end server returning an authorization token. Information inside the token can be as simple as URLs of contents, or it could contain rich policy enforcement details including expiration date and maximum playback times to allow granular control. In some cases, it may also contain the content decryption key to allow future consumption of contents. One might attack this surface in an attempt to get the content source URLs or to trick the app with more permissive policies by rewriting the authorization tokens.

(AS2) Authorization token storage. The token may need to be stored on the device's storage to accommodate content access, especially when the business model allows offline playback of contents (without Internet connectivity). An adversary with adequate storage access privilege might be able to retrieve and modify the authorization tokens on the device's storage and gain unauthorized access to contents.

(AS3) Content transmission and storage. Upon receipt of an authorized request, the back-end distribution server sends the content over the Internet. If the content is not adequately protected in transit, an adversary might be able to intercept the communication and duplicate the raw content.

Once the content arrives on the user side, it may get consumed and removed almost immediately, or it might be stored for offline consumption, which most services tend to offer for better user experience, but opens up the possibility for adversaries with sufficient storage access privilege to conduct content extraction attack.

(AS4) Playback preparation. When a user initiates playback of an encrypted content from the device's storage, the app might perform certain control checks (*e.g.*, authorization expiration) and then load the relevant secret keys into the device's memory, so that content can be decrypted for consumption. This presents an opportunity for adversaries who can inspect the device's memory to perform key extraction attacks.

(AS5) Content decryption. During actual playback, (fragments of) the content would need to be decrypted in memory. An adversary who has the capability of inspecting the device's memory can exploit this opportunity to extract the content fragments in clear, and attempt to chain them back into the original content. An attack against this surface given video contents of high entropy has been demonstrated to be feasible [53]. Attacking this surface usually requires real-time effort, that is, to extract 2 hours worth of content, the attack needs to accommodate a 2-hour long playback.

(AS6) Analog loophole. One inevitable attack surface is the analog loophole, where analog signals of protected contents are recaptured during playback. For contents like publications and motion pictures, it can be quite costly to produce high quality replicas. Similar to **(AS5)**, such an attack is also real-time in nature.

2.2 Platform and Test Setup

In our studies, we focus on the Android platform because 1) it is the most popular operating system to date and is increasingly the platform where most multimedia contents are being consumed; and 2) there exists a wide range of legacy devices that lack new hardware-enforced isolation and are running old versions of Android. For the different levels of local adversary capabilities, we leverage rooted Android phones running Android version 4.4 (Kitkat) and 5.0 (Lollipop), whichever satisfies the minimum requirements of the studied apps. To emulate network adversaries, we leverage a Linux setup hosting a wireless AP and running MITMProxy [3]. Note that since this paper is not about the robustness of Android itself but the content delivery apps that run atop of Android, we deliberately choose older versions of Android devices that are representative of the "weakest link" in the business ecosystem, as would a rational attacker do. This allows us to demonstrate the perils of service providers not excluding such devices from accessing their content distribution services.

Since in all our evaluations, content distributors maintain the back-end distribution servers and develop the Android apps that interact with users and enforce content protection, in the rest of the paper we use *vendors*, *developers* and *content distributors* interchangeably. When we say *attacks*, we mean that an adversary is able to obtain contents in a manner that violates the control mechanisms in place.

2.3 Threat Model

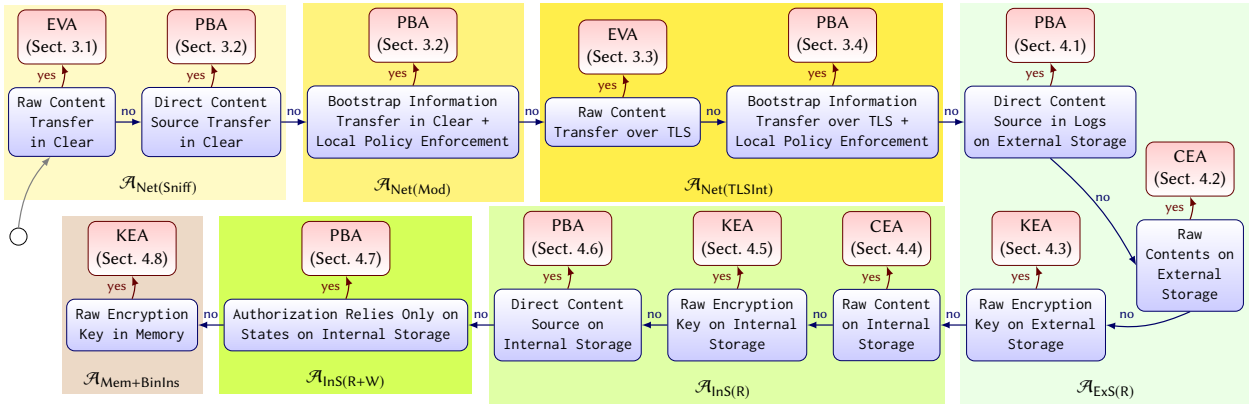
Meaningful discussion on robustness of access control and protection mechanisms requires a well-defined adversary model which bounds attackers' capabilities. Here we discuss the two categories of capabilities that we consider in this paper, focusing on software-only attacks. An enumeration of successful attacks and the corresponding adversary capabilities can be found in Figure 1.

2.3.1 *Network Adversaries.*

$\mathcal{A}_{\text{Net(Sniff)}}$ **(Passive Eavesdropping of Network Traffic)**. Such an adversary enjoys the capability of passively observing the network traffic between the device and the back-end servers serving the app. We also assume the adversary is capable of extracting payloads out of the network packets being observed and parsing messages of standard plaintext protocols (e.g., HTTP). This represents the lowest capability among all the $\mathcal{A}_{\text{Net}}$ adversaries.

[3] https://mitmproxy.org/

Figure 1: Enumeration of possible weaknesses and attacks under various adversary capabilities in attack tree form



§ KEA = Key Extraction Attacks; PBA = Purchase Bypass Attacks; CEA = Content Extraction Attacks; EVA = Eavesdropping Attacks. In this paper, KEA and EVA both imply CEA.

$\mathcal{A}_{\text{Net(Mod)}}$ **(Active Modification of Network Traffic)**. The adversary can modify and selectively block both incoming and outgoing network traffic, in order to change what the target apps receive. For plaintext protocols without strong integrity and authenticity guarantees, such adversary is also able to modify the content of protocol messages undetected. This is easily attainable by deploying a proxy server.

$\mathcal{A}_{\text{Net(TLSInt)}}$ **(Interception of TLS Traffic)**. This is an upgrade to $\mathcal{A}_{\text{Net(Mod)}}$ with the added capability of intercepting encrypted TLS traffic, as done quite frequently by anti-virus and parental control software [18], as well as middle-boxes in enterprise settings [21]. On top of a proxy setup, exactly how to attain this capability depends on the actual implementation. For target apps that trust the system CA store, it could be as simple as importing a new CA certificate into the trusted CA store as an unprivileged user. For apps that trust only their own CA stores or use key pinning, one might need the help of $\mathcal{A}_{\text{InS(R+W)}}$ or even $\mathcal{A}_{\text{Mem+BinIns}}$, both of which are discussed below.

2.3.2 *Local Adversaries.*

$\mathcal{A}_{\text{ExS(R)}}$ **(External Storage Read Any)**. This adversary capability can be achieved by a device user who has the minimum technical sophistication necessary for accessing and transferring files available on external storage of an Android device, which is "*world-readable*" [5] without any special modifications to the device. Storing large downloaded files on the external storage is a common practice in order to cope with devices that have internal storage of very limited size. On a side note, the two storage areas of Android are named *internal* and *external* due to historical reasons, and even on a device without actual physically removable media, the external storage area would still exist [5].

$\mathcal{A}_{\text{InS(R)}}$ **(Internal Storage Read Any)**. Such an adversary has the privilege to read arbitrary files on the internal storage of the device. A mobile OS such as Android [4] usually provides isolation so that an app can only read its own internal storage, and a normal user is by default not given direct access to the system's internal storage. Consequently, this capability is usually attained by "rooting" the device.

$\mathcal{A}_{\text{InS(R+W)}}$ **(Internal Storage Read Write Any)**. We consider this adversary to have the capability of reading and writing any files to any location of the internal storage. Though rooting the device would typically grant permissions to both read and write access to the internal storage, we make this fine-grained differentiation for the sake of generality, as each of these capabilities can enable different attacks.

$\mathcal{A}_{\text{Mem+BinIns}}$ **(Memory Inspection and Binary Instrumentation)**. The final adversary we consider is the most powerful one in the software domain without tampering hardware. This capability not only allows the inspection of the target app's internal execution state in memory but also the modification of the execution (control flow) of the app through binary instrumentation.

2.4 App Selection

Our evaluations start with manual analysis of some representative apps. Then, with the initial findings, we try to automate our attacks, and collect more apps that follow similar designs, and automatically test whether they are also vulnerable.

We chose the *Amazon Music* app because it is well-known and popular in the streaming business. At the time of writing, it has more than one hundred million installs and was one of the top 10 "*Music & Audio*" apps on the Google Play store. After successfully devising an attack, we then recreated it against the *Audible* app, which is another highly popular app also owned by Amazon, and the 2 apps happened to be using a very similar implementation.

We then focus on the publishing industry. We picked the *Forbes Magazine* and *Bloomberg Businessweek+* apps, as they are both well-known and popular business publications, which were coincidentally made by the same developer using 2 different designs. Having studied apps of US-based magazines, we then switched to look at their counterparts from the UK. We chose *Cosmopolitan* and *ELLE* as they are well-known magazines. We then collected many other publication apps that follow similar designs, to show that the

[4] Given that the user who did the installation get to choose the administrator/sudo password, conventional desktop operating systems like Windows and Linux don't have such a separation of storage space. In general, administrator/sudo privilege allows one to perform memory inspection and binary instrumentation.

weaknesses we found are indeed affecting a wide range of publishers and their publications. Finally we found a few publication apps that exhibit different weakness patterns on the lower-end of the spectrum, completing the study.

We give the full list of apps studied in this paper in Table 1 in Appendix. A vendor might use several different designs for its content distribution apps. In the rest of the paper, apps that are using similar designs (and hence susceptible to the same attacks) are grouped and discussed together.

3 APP WEAKNESSES & NETWORK ATTACKS

Here we present the weaknesses we found in the studied apps, as well as concrete network attacks that exploit them. We note that some weaknesses in this section also pose threats to the app users' security and privacy.

As an effort to systematize our findings, for most known weakness patterns, we map them to the relevant Common Weakness Enumerations (CWEs) in our analysis. A list of CWEs discussed in this paper can be found in Table 2 in Appendix.

3.1 Raw Content Transfer In Clear

If a content delivery app receives its contents in clear, an attacker with $\mathcal{A}_{\text{Net(Sniff)}}$ capability who can passively observe traffic exchanged between the device and the content distribution back-end server would be able to eavesdrop, extract and duplicate contents for free. We found that the *The MagPi*, *Business Money*, *Artists & Illustrators*, *My MS-UK*, *Popshot Magazine* apps (group-1 of Table 1) fall into this category. These are apps of magazines from different publishers, all made by a vendor called *Apazine*.

Eavesdropping Attacks. We note that in Apazine's design, contents are distributed based on `PDF`s, with each issue of the magazines and journals encapsulated in a single `PDF` file. Issues of publications can be purchased individually inside the apps, which would trigger a `PDF` download. However, because the apps and the back-end servers exchange data including the unencrypted content PDFs through HTTP (instead of HTTPS) [CWE-319], it is trivial for $\mathcal{A}_{\text{Net(Sniff)}}$, the weakest remote adversary we consider, to extract and duplicate the PDF files through the observed traffic. This is an attack against **(AS3)**. We have confirmed the feasibility of this attack in the aforementioned apps.

3.2 Bootstrap Information Transfer in Clear

It is often necessary for publication apps to communicate with the back-end servers to get bootstrapped with information regarding what issues and subscription tiers are available at what price. We note that many apps we studied receive their bootstrap information in clear through HTTP, which is another instance of [CWE-319]. This leads to 2 different attacks on **(AS1)**, given varying levels of adversary capabilities.

Purchase Bypass Attacks with $\mathcal{A}_{\text{Net(Sniff)}}$. The 5 group-1 apps discussed in Section 3.1 can again serve as examples, as they are all susceptible to this attack. From Apazine's back-end server they receive bootstrap information in `JSON` format, which contains details of each issue. Specifically the URLs for downloading the unencrypted content PDF files of each issue can be found there as Base64 encoded ASCII strings. Since given those URLs, the back-end content distribution server does not enforce further authentication and authorization before serving the PDF files [CWE-425], an $\mathcal{A}_{\text{Net(Sniff)}}$ adversary can observe and parse the `JSON`, decode the URLs, and get unrestricted direct access to the unencrypted content PDFs. We verified the feasibility of this attack by observing the traffic generated by the 5 aforementioned apps.

Purchase Bypass Attacks with $\mathcal{A}_{\text{Net(Mod)}}$. There exist other possibilities for exploits even if the bootstrap information does not contain direct content sources. For concrete examples, we turn to the 70 publication apps (group-3 of Table 1, *e.g.*, *Forbes Magazine*) made by a developer called *Maz Systems*, which is reported to have an annual revenue of several million US dollars [22]. The design for these apps seems to rely on the apps to construct the content source URLs, based on the bootstrap information received in `XML` format and the unique IDs of each issue. The back-end server hosted on Amazon S3 requires some level of API key authentication before serving the contents. However, since the price of each issue is directly given by the bootstrapping `XML` received through plain HTTP without much integrity and authenticity guarantees [CWE-354], we found that an $\mathcal{A}_{\text{Net(Mod)}}$ adversary can rewrite the price of all the issues into zero, and the 70 apps we tested all trusted their corresponding altered `XML`, and offered magazine issues for free. The adversary can then use the apps to download the publications without paying. Additionally, some publishers offer subscriptions to their publications in the apps (*e.g.*, $29.99 per year for Forbes Magazine), the price of which was also received from the same bootstrapping `XML`. We have confirmed that an $\mathcal{A}_{\text{Net(Mod)}}$ adversary can also rewrite the prices of subscription plans into zero, then subscribe (for free) and get access to all the issues available within the subscription period.

These findings suggest that the price of purchase is enforced by the apps locally on client-side [CWE-603] without involving the back-end servers after the initial bootstrap.

3.3 Raw Content Transfer over TLS

Also for those 70 group-3 apps, after a purchase has been confirmed, it receives the contents, in the form of a `ZIP` file, from some back-end server hosted on Amazon S3 through TLS. Despite using encrypted connections, it does not mean one cannot attack **(AS3)**.

Eavesdropping Attacks. Specifically, we found that for establishing a TLS session, those apps trust the system CA store for signing certificates and do not seem to be using any forms of key/certificate pinning. As the result of which, it was trivial to attain $\mathcal{A}_{\text{Net(TLSInt)}}$, without the need to leverage other advanced local capabilities. Together with the fact that the `ZIP` files were not passphrase-protected, an $\mathcal{A}_{\text{Net(TLSInt)}}$ adversary can extract contents out of the passively observed `ZIP` files with ease.

3.4 Bootstrap Information Transfer over TLS

Even if apps receive bootstrap information over encrypted TLS connections, without additional integrity and authenticity guarantees, an $\mathcal{A}_{\text{Net(TLSInt)}}$ adversary can still abuse such information for his/her own gains. As concrete examples, we look at *a*) the 34 publication apps (group-4–6 of Table 1) exemplified by the *Bloomberg Businessweek+*, *Entrepreneur Magazine* and *Men's Health Magazine*

apps, which were coincidentally also developed by *Maz Systems*, under designs different from the group-3 apps; and *b*) the 30 publication apps (group-7–8 of Table 1) exemplified by *ELLE UK* and *The Independent*, developed by a vendor called *Pugpig*.

Purchase Bypass Attacks. Similar to the apps discussed in Section 3.3, these 34 group-4–6 apps all trust the system CA store, so attaining the $\mathcal{A}_{\text{Net(TLSInt)}}$ capability was straightforward. For the group-5–6 apps that offer periodicals, they receive detailed information regarding what issues are available at what price through some bootstrapping JSON over TLS. The description of each issue comes with a boolean indicating whether it is locked (require payment) or not. We have found that using the $\mathcal{A}_{\text{Net(TLSInt)}}$ capability, one can rewrite all instances of `"locked": true` into `"locked": false` in the bootstrapping JSON, and have all the issues unlocked for free.

On the other hand, the group-4 apps employ a different, article-centric subscription-based business model, which allows its users to read k number of articles for free every j days as trial. Likewise, the value of both k and j are retrieved from some bootstrapping JSON transferred over TLS. We have confirmed that an $\mathcal{A}_{\text{Net(TLSInt)}}$ adversary can rewrite the value of k and j in the bootstrapping JSON, as shown in Figure 2, to trick the apps into granting everyday a number of free articles so large that it is virtually like having a paid subscription.

Figure 2: Rewrite bootstrapping JSON with $\mathcal{A}_{\text{Net(TLSInt)}}$ to gain free articles in the Bloomberg Businessweek+ app

Original JSON Snippet

```
... ...,
"metering": {
    "freeViews": 4,
    "resetAfter": 28,
    "registerAfter": 2,
    "registerRequired": false
}, ... ...
```

Snippet After Rewrite

```
... ...,
"metering": {
    "freeViews": 400,
    "resetAfter": 1,
    "registerAfter": 300,
    "registerRequired": false
}, ... ...
```

Additionally, the 30 group-7–8 apps receive from their back-end server a series of XML files describing available issues and their pages. With $\mathcal{A}_{\text{Net(TLSInt)}}$, we found that one can parse the XML files, stitch various metadata components into the actual content source URLs and download magazine pages directly without paying.

All these findings suggest that the access control enforcement (*e.g.*, locked contents and free trial previews) are done locally on the client-side without involving the back-end servers [CWE-603].

3.5 Threats to User Security and Privacy

For the 75 group-1 and group-3 apps discussed in Sections 3.1 and 3.2, since their bootstrap information are sent in clear without strong integrity guarantees [CWE-354], any Man-In-The-Middle (MITM) can easily tamper with what is being transferred. This not only allows one to bypass purchase and extract contents, but also poses threats to the app users. For example, one might be able to increase the price of each issue to induce financial losses on the user. One can also remove specific issues in the bootstrap information to implement censorship. Rewriting URLs can also trick the users to visit some potentially malicious websites. Additionally, given known vulnerabilities about the libraries that the apps uses (*e.g.*, MuPDF [39], Zip [51]), one can potentially change the URLs in the bootstrap information to point to some maliciously crafted input files to attack the user's device.

Furthermore, for many of the 70 group-3 apps discussed in Sections 3.2 and 3.3, we have observed that during and after the bootstrap, some tracking data are being sent to the back-end over HTTP in clear. The exchanged data contains the device unique identifier and model name, along with some session ID and publication ID. A passive eavesdropper might try to extrapolate who is reading what magazines, which could be quite revealing given that some publications are related to medical conditions, musical instruments and specific industries, posing threats to the app users' privacy.

4 APP WEAKNESSES & LOCAL ATTACKS

Here we present more weaknesses of the studied apps, with a focus on local attacks. Similar to the previous section, we map our findings to the relevant CWEs whenever possible.

4.1 Log File Leakage

Another possible weakness is leakage of secrets through log files, similar to what had previously been observed in some Android mobile banking apps [43].

Purchase Bypass Attacks. As discussed previously in Section 3.1, those 5 group-1 apps use direct content source URLs for fetching contents. Our inspection revealed that those same apps leave some debugging log files on the external storage which contain both the direct URLs of publication PDF files hosted on their back-end servers and the identifiers of each of the issues available for purchase [CWE-532]. This allows an $\mathcal{A}_{\text{ExS(R)}}$ adversary to retrieve those URLs, and by replacing the appropriate portion of the URLs with the issue identifiers, one can enumerate the different published issues and download their corresponding unencrypted PDF files directly [CWE-425], effectively getting unlimited unauthorized access without having to purchase, mounting an attack against **(AS2)**.

4.2 Raw Content on External Storage

If the apps leave their contents on the External Storage, it would allow for an easy attack on **(AS3)** that both the apps and the publishers would lose control of the contents.

Content Extraction Attacks. We have found that the 9 group-6 apps serve contents in the form of PDF and put their PDF files on the device's external storage. Given the $\mathcal{A}_{\text{ExS(R)}}$ capability, one can easily get those files and make copies of them. This can be applied to the various free trial issues offered in the apps, as a user is allowed only several minutes of free preview before needing to pay to continue reading, but one can simply workaround this restriction by copying the full PDFs from the external storage and open them using a different reader.

4.3 Raw Encryption Key on External Storage

Even if an app employs encryption as the means for content protection, if the secret key is left in a place that is accessible by an adversary, one can attack **(AS2)** and strip the encryption.

Key Extraction Attacks. As examples, we again look at the 5 group-1 magazine apps discussed in Section 3.1. After purchasing a specific issue, those apps would download the content file and put it in the external storage of the device. With the $\mathcal{A}_{\text{ExS(R)}}$ capability, we can see that the content files retain the `.pdf` extension but the

contents are actually scrambled. Since not even the PDF metadata are comprehensible, we deduce that this is most likely due to the use of a whole file encryption. Together with the findings from Sections 3.1 and 3.2, this suggests that the content encryption was done locally on the client device after download.

While navigating through the files created by the apps on external storage with the $\mathcal{A}_{\text{ExS(R)}}$ capability, we found that there exists a serialized Java object outside the directory that contains the encrypted PDF files, adjacent to the log files discussed in Section 4.1. A quick inspection revealed that this serialized object is of the class `javax.crypto.spec.SecretKeySpec`, which turns out to contain the secret key used to encrypt the PDF files. That object also revealed that the encryption algorithm used was AES, though the exact block cipher mode remains unclear. This is tantamount to leaving one's house key under the doormat outside the house, a known weakness pattern described by [CWE-313] and [CWE-921].

After identifying the key, decrypting the content PDF files was somewhat straightforward. With around 200 lines of Java code and some trial-and-error to determine that the apps were using the Electronic Codebook (ECB) mode of AES, we confirm that the contents can be decrypted using the suspected secret key. We have verified this attack with a paid purchase of a recent issue in the My MS-UK app, and free trial issues in the Business Money, Artists & Illustrators, The MagPi and Popshot Magazine apps.

4.4 Raw Content on Internal Storage

Given that the official Android development training material claims files stored on internal storage are "*accessible by only your app*" and "*neither the user nor other apps can access your file*" [5], it is perhaps unsurprising that some apps are making strong assumptions about the confidentiality guarantees provided by the internal storage. Such assumptions, however, can be invalidated with $\mathcal{A}_{\text{InS(R)}}$.

Content Extraction Attacks. For concrete examples, we again look at the 70 group-3 apps discussed in Sections 3.2 and 3.3. In their designs, each page of the publication is a `JPEG` image of about 0.7 megapixel. After downloading the content `ZIP` file of an authorized issue, the app extracts from it the content images and have them stored on the app's internal storage. The app then acts like an image viewer for displaying each page for the user to read. As the images of each issue are left inside the internal storage without further scrambling [CWE-313], an $\mathcal{A}_{\text{InS(R)}}$ adversary can easily access and make copies of the magazine issues, attacking **(AS3)**.

Through the in-app free previews, we found that the 16 group-8 apps also has each page of an issue saved as a `JPEG` image on the internal storage. In fact, we found that even though the free previews should allow only a small number of pages, all the other pages of the selected issue are already downloaded. Consequently, with $\mathcal{A}_{\text{InS(R)}}$, one can easily bypass the preview limit and access the saved pages directly.

4.5 Raw Encryption Key on Internal Storage

Similarly, developers might put encryption keys on the internal storage assuming confidentiality [CWE-313], however, in the face of the $\mathcal{A}_{\text{InS(R)}}$ capability, such a design manifests into an exploitable weakness on **(AS2)**.

Key Extraction Attacks. We use the Counter Intelligence Plus app (group-2) as an example, which is also made by *Apazine*. Interestingly, despite being older than the other group-1 apps discussed before, the Counter Intelligence Plus app appears to be doing a slightly better job in terms of hiding the secret key used in content encryption. In this case, instead of putting it on the "world-readable" external storage, the key is stored on the device's internal storage. However, with the $\mathcal{A}_{\text{InS(R)}}$ capability, we have managed a key extraction attack similar to what is described in Section 4.3.

4.6 Direct Content Source on Internal Storage

Leaving direct links to contents that do not enforce authentication and authorization [CWE-452] on Internal Storage is another exploitable weakness on **(AS2)**.

Purchase Bypass Attacks. With the exception of the *The Rebel Media* app, all the other 24 group-4–5 apps that offer articles (*e.g.*, the *Bloomberg Businessweek+* app) or video clips (*e.g.*, the *Outside TV Features* app), leave direct URLs to their corresponding contents on the apps' Internal Storage, organized by the different issues, allowing an $\mathcal{A}_{\text{InS(R)}}$ adversary to easily crawl for those and access contents without paying.

4.7 Client-Side Authorization

The assumptions on internal storage indeed presents an interesting attack vector. In addition to confidentiality, one might also assume that the internal storage provides strong integrity guarantees. Such an assumption can be invalidated with $\mathcal{A}_{\text{InS(R+W)}}$.

Purchase Bypass Attacks. Each of the 70 group-3 apps discussed in Section 3.2 also keeps a local database of published and available issues on the app's internal storage. For each issue, the database keeps a record about the name and date of the issue, a brief description, price, and some other metadata, including the purchase status. With the $\mathcal{A}_{\text{InS(R+W)}}$ capability, we have verified that one can modify the database and replace the default value of the purchase status column with some appropriate values [CWE-642] to trick the app into granting access to magazine issues that were not paid for.

This shows that the authorization of those apps is localized and done unilaterally on the client's device, and does not involve the back-end content distribution server [CWE-603]. Consequently, the robustness of such authorization mechanism hinges on the assumption that the internal storage guarantees integrity [CWE-654], which does not hold given an $\mathcal{A}_{\text{InS(R+W)}}$ adversary.

4.8 Raw Encryption Key in Memory

Even if raw secret keys are not left in the clear on permanent storages, they might be loaded into the memory, which presents another opportunity for attacking **(AS4)**. As a concrete example, we look at the Amazon Music app (version 6.5.3), which offers both streaming and offline playback of music to its subscribers. There are two tiers of subscription: Amazon Prime and Amazon Music Unlimited, with the only difference being the size of the collection accessible (2 million versus 40 million songs). Both tiers allow subscribers to download music available from their corresponding collections for offline playback. The downloaded songs are stored on the external storage of the Android device.

Storage Inspection. A quick inspection of the downloaded files shows that regardless of the subscription tier, songs appear to be encrypted and contain a human readable `PlayReadyHeader` XML object in their metadata [37], suggesting that this entire streaming service is using the Microsoft PlayReady DRM framework. With the contents already available on the external storage, we choose to focus on devising a key extraction attack.

We first leverage the $\mathcal{A}_{\mathsf{InS(R)}}$ capability to inspect the internal storage and see if one can attack **(AS2)**. As there exists quite a few secret key candidates (*e.g.*, Base64 strings that decode into binary values of various lengths), we soon run into the problem of not knowing how to verify whether a key candidate is the right one for content decryption.

Key Verification Oracle. Fortunately, the limited documentation publicly available regarding the `PlayReadyHeader` [37] turns out to be quite useful. The `PlayReadyHeader` metadata object contains the key ID, content encryption algorithm (in this case AES CTR mode) and the key length (16-byte), so we know what we are searching for. Better yet, it also contains a checksum used by the framework to protect against mismatched keys. According to the documentation, this is meant to prevent the case where decryption is done with an incorrect key, the subsequent output of which might damage audio equipments during playback. Since the checksum is simply the first 8 bytes of encrypting the 16-byte key ID with the key in AES ECB mode, which is easily computable, we now have an oracle for verifying key extraction correctness, without having to rely on decrypting the contents themselves.

A quick trial-and-error showed that none of our initial suspects were the right content encryption key. The publicly available documentation regarding the PlayReady framework [36] suggests that the content decryption keys are contained inside licenses. We then turned our attention into finding the license instead. We realized that there exists a `.hds` file, which according to some discussion about Silverlight [1], seems to contain the licenses. Without documentations on how to parse and interpret this file, the format of licenses, and whether this file is obfuscated or encrypted, key extraction from it seems could be quite complicated.

Key Extraction Attacks. Instead, we switch our focus to attack surface **(AS4)**. The intuition is that, even if the local license store on internal storage has complex protection mechanisms in place, the content encryption key would still need to be read from the license store and might be loaded into the memory in clear. Hence we upgrade the adversary capability to $\mathcal{A}_{\mathsf{Mem+BinIns}}$, by using the Frida [5] dynamic instrumentation framework. While we were able to trace the app's file read operations by hooking the `read()` system calls with $\mathcal{A}_{\mathsf{Mem+BinIns}}$, including those that reads from the license store, it remains unclear how to interpret the bytes being read. Since tracing and interpreting the preparation phase seems to be quite messy, we take a slightly different approach. With the intuition that sensitive information (*e.g.*, content encryption key), if they were indeed loaded into the memory, might exhibit some recognizable structure and would need to be released at some point (*e.g.*, after content playback), we try to hook deallocation functions instead. While it might also work to hook the `free()` function calls, a quick inspection of the native libraries used by the app shows that they are exporting some functions that seem to be used for deallocating sensitive information. Intercepting the entrance to some of those functions, tracing appropriate pointers, and then dumping a large enough portion of memory pointed by those, we successfully extracted the content encryption key, which verifies against the oracle discussed earlier.

Unlike in previous work where PlayReady protected video contents were reported to be partially encrypted [53], in this case we have observed that the entire original content is encapsulated in a so-called envelope file. Though we were unable to obtain documentations regarding the metadata (besides the `PlayReadyHeader` object), based on some header files publicly available on Github [35], we were able to guess and parse the metadata correctly. In the end it took us about 260 lines of Java code to parse the envelope file and decrypt with AES CTR using the extracted secret key to get the raw audio tracks out.

One might also wonder why $\mathcal{A}_{\mathsf{Mem}}$ alone is not strong enough. We note that the key extraction problem has a two-dimensional search space, spanning memory layout and time. While ultimately it is the memory inspection that gets us the key, without binary instrumentation, however, it is difficult to pinpoint the exact timing, especially if the implementation tries to minimize key exposure by actively releasing and overwriting memory regions containing the secret key, a practice also recommended by previous work [27]. Not knowing when to dump the memory would make it hard for $\mathcal{A}_{\mathsf{Mem}}$ to extract the key. Interestingly, in this app, we have observed a behavior that sensitive information deallocation happens as early as the actual music playback starts. Our speculation is that, since the CTR mode generates keystream blocks by encrypting the next counter values, after a long enough keystream has been generated to allow decryption of the entire content, the app removes the key from memory as soon as possible to minimize exposure time. This is exactly why $\mathcal{A}_{\mathsf{Mem+BinIns}}$ has an advantage in reducing uncertainties along the time dimension.

We further found that the Audible app (version 2.25.0) is also susceptible to this attack. Specifically, members are offered premium podcasts that can be downloaded for offline playback, in which case they are encrypted in the same manner as songs in Amazon Music. It appears that the PlayReady implementations in the 2 apps are quite similar, as our key extraction attack also worked. Since the downloaded Audible podcast tracks are partially encrypted `isma` files, instead of using our decryption code for Amazon Music, we used the Bento4 tookit [6] for successful decryption.

Key Scanning Heuristics. Curious readers might wonder how we recognized the 16-byte key from memory dumps. As explained in previous work, besides loading the raw secret key into memory, many cryptographic implementations speed up computation by precomputing the key schedules made of the different round keys [27]. This is because typical block ciphers, including AES, go through multiple rounds of operations to encrypt/decrypt a block, and each round involves a round key derived from the raw secret key. Having to repeatedly expand the raw key into the same key schedules for each block could be quite inefficient. It turns out that the key schedule observation also applies to this particular PlayReady implementation. Using the `keyfind` program [27], we

[5] https://www.frida.re

[6] https://www.bento4.com/

were able to confirm the mathematical relation between the suspected raw key and the derived round keys, strongly suggesting that those bytes found in the memory dump indeed constitute a key schedule.

One key to rule them all. Based on the handful of songs that we sampled in our proof-of-concept experiments, Amazon Music appears to be reusing content encryption keys across songs and different accounts, despite the fact that the PlayReady framework allows a much more granular key binding (*e.g.*, per individual item), as noted in a previous work [53]. We speculate that this is to lower the load and management overhead on the back-end servers, though we are not sure whether the entire ecosystem uses only one single key, or are keys different across data centers in various locations. Consequently, songs made available for offline playback from many different albums across artists, regardless of which tier of subscription and user accounts they came from, might all be decrypted with the same key. This puts the whole collection of 40 million songs available on Amazon Music in excessive risk.

While the attack we presented is agnostic to key granularity and can be performed over and over again to exhaust all the possible keys, however, a more fine-grained key binding (*e.g.* per album or even per song) would have at least required more effort from an attacker, and hinder automatic mass decryption of a large number of songs. Sharing keys across many accounts does not seem to be a good practice, as it is easy to have a single key leaked (*e.g.*, by an insider) and cause large damage to the ecosystem. Interestingly, unlike Amazon Music, Audible seems to be much more granular with its content encryption keys. It appears that for each podcast track, a new key is used for encryption.

5 DISCUSSIONS

5.1 Responsible Disclosure and Aftermath

We have notified the content distributors of our findings and provided them with sufficient details to understand and reproduce the attacks. In all cases, we have given the vendors more than 90 days before this paper is made public.

In response to our initial report sent in Feb 2018 regarding bypass attacks with $\mathcal{A}_{\mathsf{InS(R+W)}}$ (Section 4.7), developers at Maz Systems implemented the use of encrypted database on newer versions of some of the group-3 apps. This is a solution that we do not endorse, as it does not change the client-only nature of the authorization mechanism, so the pattern of [CWE-603] still holds. We followed up with reports on group-4–6 apps in Jun 2018. They have expressed gratitude to our efforts, and are working on app improvements.

We sent several reports to Apazine in Feb, Apr and Sep 2018 regarding weaknesses in their group-1–2 apps. They have replied in Sep 2018 suggesting that the magazines are in public domain and do not contain any sensitive or valuable information, so our implied expectation of robustness is not relevant. We point out that if the contents indeed have no market values then not using encryption can improve user experience, and that some publications in print (*e.g.*, Business Money and My MS-UK) do not seem to be available for free.

Developers at Pugpig have been notified in Jul 2018. They have since acknowledged and confirmed our findings, and replied that they are aware of the weaknesses, and that the apps are designed that way by choice to accommodate anonymous sharing of magazine pages, a feature requested by their clients (publishers).

Amazon has been notified about the key extraction attack against the Amazon Music app in Jan 2018. They responded to our report with several new versions of their music app, implementing new obfuscation strategies and offline playback restrictions on rooted devices. However, despite our recommendation of considering the secret key compromised and switching over to a new key, as of Jun 2018, we have noticed that recent new releases are still encrypted using the same old key. The KEA against Audible was reported to Amazon in Jun 2018, and new versions implementing various obfuscation strategies have since been released.

5.2 Possible Countermeasures and Challenges

Bilateral Policy Enforcement. While a stateless sever allows for a more simplistic deployment, as [CWE-603] has noted, a client-only authorization is weak and can potentially be bypassed, especially on an environment where execution/code can be reverse-engineered and tampered with. Since local adversaries are not able to directly tamper with the execution state of a remote server, considering the threats of $\mathcal{A}_{\mathsf{InS(R+W)}}$ and $\mathcal{A}_{\mathsf{Mem+BinIns}}$, policy enforcement can be done more robustly involving the back-end servers. For example, to avoid the attacks discussed in Sections 3.2, 3.4 and 4.7. the authorization logic should be shifted to the back-end server. Then client-side modification of prices and purchase status would result in detectable discrepancies with records on the server, and the latter can refuse to serve contents in such cases.

Direct Content Sources. To hinder the attacks discussed in Sections 3.2, 4.1 and 4.6, content source URLs should not be left in a log file [CWE-532] and also not on a storage that an adversary has unlimited access to [CWE-921]. Instead of explicitly saving the URLs, it would perhaps be better to have them constructed dynamically during runtime, and the servers should request extra authentication and authorization. An $\mathcal{A}_{\mathsf{Mem+BinIns}}$ adversary might still be able to figure out the URLs and the accompanying parameters, but it would be an improvement comparing to the current deployments.

Certificate Pinning. To hinder the $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ attacks discussed in Sections 3.3 and 3.4, instead of trusting the system CA store, the apps could adopt some forms of key/certificate pinning [7]. Even though $\mathcal{A}_{\mathsf{InS(R+W)}}$ and $\mathcal{A}_{\mathsf{Mem+BinIns}}$ might still be used in tandem to defeat pinning, it would at least make $\mathcal{A}_{\mathsf{Net(TLSInt)}}$ more difficult to achieve than in the current implementations.

Denying services to rooted devices. One might propose for the apps to stop providing services on rooted devices, as on unrooted ones the adversary capabilities are greatly limited. This approach however has its own challenges. First, while a concrete global number of rooted devices is not available, it has been suggested that the number could be quite high in certain communities [6, 17]. Various rooting tools have reported millions of downloads [14, 33], and the superuser access management app has hundreds of millions of installs [3]. A content distributor denying service on rooted devices risks losing these customers. Second, determining whether a device is "rooted" is an on-going arms race. Depending on the heuristics

[7] https://www.owasp.org/index.php/Certificate_and_Public_Key_Pinning

used and how the checks were implemented, binary instrumentation might be able to bypass those as well [44]. We have observed that, as of version 7.5.4, the new Amazon Music restriction of no offline playback on rooted devices can be bypassed with RootCloak [34], a popular system modification module.

Google has since introduced the SafetyNet service for developers to detect if a device has been tampered with. Android Pay, Netflix, and Pokemon Go are some examples that would deny service if SafetyNet finds the devices is rooted. However, the cat-and-mouse game between SafetyNet and the Magisk systemless rooting technique in 2017 has been well documented [45, 46], and there are reports suggesting that on *legacy Android versions* various bypass and attacks against SafetyNet are possible [15].

Anti-Debugging and Anti-Instrumentation. Another possibility is to implement anti-debugging and anti-instrumentation techniques in the apps to hinder analysis, potentially on even rooted systems. Depending on what heuristics are being used, some might still be bypassable [41]. With the advancements of artifact detection [38], anti-instrumentation [23, 32], and transparent debugging [19, 42], this line of defense appears to be an on-going cat-and-mouse-game, similar to root detection.

Obfuscating keys in memory. While relying solely on obscurity for security lacks robustness [CWE-656], however, in the case where content encryption keys must be inevitably loaded into the memory, one possibility is to make it harder for key scanning heuristics to identify secret keys in memory dumps.

Heuristics used in identifying memory regions of interests (*e.g.*, those that contain content fragments and cryptographic keys) typically assume their targets to occupy a contiguous region of memory [27, 53]. Additionally, they might also leverage the mathematical relation between the raw secret key and its derived key schedules to pinpoint the targets [27]. It remains to be seen whether obfuscations can be used to defeat these assumptions and make it more difficult for an attacker to recover secret keys from the memory.

Watermarking. An orthogonal line of protection is to use watermarking to make the origin of piracy traceable. Over the years, there are techniques developed to watermark multimedia like audios [11] and motion pictures [10], as well as textual contents [26, 52]. In some cases, however, attackers can remove trivially detectable watermarks. Resilience against detection and removal remains the main objectives of watermarking research.

Another weakness of relying on watermarking for piracy tracking is that detection often relies on the content being leaked and shared on the Internet, and it remains difficult to detect offline sharing and contents that are stolen but not shared at all.

Trusted Execution Environments. A potential game changer is the use of Trusted Execution Environments (TEEs), Since the traditional execution environment could potentially be under adversarial control, TEE vendors typically leverage separation mechanisms enforced by the hardware platform to create an isolated execution environment, the internal execution state of which not even the OS can inspect, though depending on implementations, cryptographic code running inside an isolated environment like Intel SGX might still be susceptible to cache timing attacks [13, 25].

Various TEE implementations have been made available in recent years, especially on mobile platforms, where it has been reported that there are multiple vendors offering various TEE solutions that do not seem to conform to the same API standard [20], which might have made it hard for developers to implement a one-size-fits-all solution that is universally deployable across brands of devices, potentially hindering adoption.

In the TEE trust model, the vendors would typically serve as the root of trust, and they often employ a tight admission control model which might require potentially costly licensing and non-disclosure agreements prior to app development and deployment. This could potentially create a market where only big companies can afford to compete in, shutting out small businesses and individual developers. For some, this also presents a concern for consumer rights. Since the trust on TEE vendors who enforce admission control on what can be executed in the isolated environment is *transitive* (users trust the vendor, which in turn trust the TEE licensees to provide opaque but non-malicious software), the lack of transparency makes it difficult to detect subtle attacks (*e.g.*, spying and tracking) and hold the vendors accountable. While for cloud service providers, the ability to create and attest isolated execution environments might add appeals to their customers, it remains unclear how, if given the choice, consumers would be willing to pay for a hardware technology that they cannot control and cannot opt-out, instead of choosing the low cost devices without these hassles that are more customizable and configurable.

6 RELATED WORK

App Weakness Analysis. Reaves et al. [43] have carried out an analysis, that shares the same spirit as ours, for 7 branchless mobile banking applications, and have uncovered weak design and implementation practices including inadequate authentication and authorization checks, weak (or, non-standard) cryptographic primitive usage, predictable key usage, and sensitive information leakage.

Memory Dumping Attacks. From the perspective of attempting to bypass DRM content protection, the closest work comparing to ours is perhaps the Steal This Movie paper [53]. While their attack on **(AS5)** is focused on identifying data paths of cryptographic operations and the dumping and reconstruction of decrypted content streams, we are more concerned about the overall app life cycles **(AS1–4)**, especially on commodity mobile devices that are lacking in hardware protection capabilities.

Cryptographic Attacks. Biryukov et al. [8] present a cryptanalysis of the weak cipher (PC1) employed by Kindle for DRM protection. The authors have shown that due to the lack of avalanche effect in PC1, one can extract the key using known plaintext and ciphertext attacks. Crosby et al. [16] present a cryptanalysis for High Bandwidth Digital Content Protection (HDCP) scheme, an identity-based cryptosystem used for communication in the Digital Visual Interface (DVI) bus, in which they identify that if an adversary has access to 40 public/private key pairs they can essentially break all the security guarantees promised by the scheme.

Side Channel Attacks. Depending on their implementations, cryptographic code might leak secrets through various side channels, which can sometimes be exploited. Several implementations of AES are known to be vulnerable to side channel attacks [7, 12] Side channels might still exist even if one uses a TEE like SGX [13, 25].

Implementation Challenges. From a purely technical perspective, a cryptographic enforcement of DRM has similarity with an offline password managers and a cryptographic ransomware, though the three face different challenges. Although offline password managers try to prevent illegal access to user-passwords [4, 24, 50]—analogous to contents under DRM protection—there is a major difference in the adversarial assumptions: the underlying user is completely trusted in the context of the password managers but is not the case for DRM. Such an assumption allows a neat key concealment approach in which the secret key is not required to be stored on the device, and can be derived at runtime from the user's master password using a key derivation function, making password-manager somewhat easier to implement. On the other hand, cryptographic ransomware shares the same view of adversary as in DRM [28–31, 47, 48]—resisting recovery of contents—but because of the relaxation on the decryption correctness guarantees, ransomware can simply delete the secret key, while DRM systems, particular those that allow offline playback for better user experience, cannot.

Some have suggested that the large file size of high-definition multimedia contents can be considered as a natural DRM [9], for which we disagree with. While large file size slightly hinder Internet sharing, DRM has a variety of other objections like copy control, license expiration check and authorization that are beyond the scope of Internet sharing, especially in the era that the subscription-based stream business is dominant.

App Security Standard. OWASP recently released version 1 of its Mobile Application Security Verification Standard [40], which attempts to standardize security requirements and verification levels that fit different application and threat scenario. Interestingly, for Intellectual Property protection, it recommends verification level `L1+R`. While `R` requires resiliency against reverse engineering, `L1` does not require key/certificate pinning, which as discussed in Sections 3.3 and 3.4, allows for relatively easy TLS interception and potential content protection bypass.

7 CONCLUSION

In this paper, we shed light on the current practices and weaknesses of content delivery apps on mobile platforms, with concrete attacks on 141 apps. Due to some unjustified trust assumptions and weak design patterns, given the right adversary capability, it is often possible to bypass the content protection mechanism in place to achieve unrestricted access to raw contents. Feasibility of such attacks might have contributed to the conventional printed media's struggle for revenue. Content owners should evaluate the robustness of the app design before retaining the service of a developer.

Our findings present an interesting dilemma for content distributors to consider: either risk losing controls over contents by allowing untrustworthy devices to access their services, or risk losing customer reach. We hope that our work would bring awareness to the situation, and spark further research on identifying more app weaknesses. In particular, with more sophisticated frameworks like [49] and recent advancements in transparent debugging against anti-debugging and anti-instrumentation techniques [19, 42] we expect more apps can be reverse engineered and analyzed.

A Call to Arms. Another goal of this work is to summarize weakness patterns so that future developments of similar apps can benefit from the insights provided by this paper, take various attack strategies into consideration, and avoid similar pitfalls.

Penetration testing becomes especially important in the case when content distributors are unwilling to completely shut off their services to customers who own only low-end devices without TEE capabilities, making obfuscation the only feasible partial solution. In the absence of a generic framework for quantifying the complexity of obfuscation, penetration testing becomes perhaps the only way to empirically evaluate how difficult it would be to extract secret states. Companies who already have an in-house red team could perhaps leverage it for this purpose.

ACKNOWLEDGMENT

We would like to thank the app vendors for responding to our vulnerability reports, and in many cases collaborated with us regarding new releases. We thank the anonymous reviewers for their helpful comments. Special thanks to Mr. Horace Tse for drawing our attention to apps of medical journals, which ignited our interests on this topic. We are also very grateful to Prof. Gene Spafford for his advice on responsible disclosure and DMCA. This work was supported in part by NSF CRII 1657124 and ARO W911NF-16-1-0127.

REFERENCES

[1] 2010 (accessed Feb 07, 2018). *DRM License cacheable.* https://social.msdn.microsoft.com/Forums/silverlight/en-US/b0220e8a-0660-49aa-8353-18d12ae285dd/drm-license-cacheable.
[2] 2016 (accessed Feb 07, 2018). *DMCA security research exemption for consumer devices.* https://www.ftc.gov/news-events/blogs/techftc/2016/10/dmca-security-research-exemption-consumer-devices.
[3] 2017 (accessed Feb 07, 2018). *SuperSU - Android Apps on Google Play.* https://play.google.com/store/apps/details?id=eu.chainfire.supersu.
[4] Andrey Belenko and Dmitry Sklyarov. 2012. *"Secure Password Managers" and "Military-Grade Encryption" on Smartphones: Oh, Really?* https://www.elcomsoft.com/WP/BH-EU-2012-WP.pdf.
[5] Android Developers. (accessed Feb 02, 2018). *Saving Files - Choose Internal or External Storage.* https://developer.android.com/training/data-storage/files.html#InternalVsExternalStorage.
[6] Andy Boxall. 2015 (accessed Feb 06, 2018). *80% of Android phone owners in China have rooted their device.* http://www.businessofapps.com/80-android-phone-owners-china-rooted-device/.
[7] Daniel J Bernstein. 2005 (accessed Sep 19, 2018). Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
[8] Alex Biryukov, Gaëtan Leurent, and Arnab Roy. 2012. Cryptanalysis of the "kindle" cipher. In *International Conference on Selected Areas in Cryptography.* Springer, 86–103.
[9] A. Blaich and A. Striegel. 2009. Is High Definition a natural DRM?. In *Proceedings of 18th International Conference on Computer Communications and Networks.* 1–4.
[10] Jeffrey A Bloom and Christos Polyzois. 2004. Watermarking to track motion picture theft. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, Vol. 1. IEEE, 363–367.
[11] Laurence Boney, Ahmed H Tewfik, and Khaled N Hamdy. 1996. Digital watermarks for audio signals. In *Multimedia Computing and Systems, 1996., Proceedings of the Third IEEE International Conference on.* IEEE, 473–480.
[12] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems.* Springer, 201–215.
[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies.*
[14] Chainfire. 2012 (accessed Feb 07, 2018). *[CENTRAL] CF-Auto-Root.* https://forum.xda-developers.com/showthread.php?t=1980683.
[15] Collin Mulliner and John Kozyrakis. 2017 (accessed Feb 06, 2018). *Inside Android's SafetyNet Attestation.* https://www.blackhat.com/docs/eu-17/materials/eu-17-Mulliner-Inside-Androids-SafetyNet-Attestation.pdf.
[16] Scott Crosby, Ian Goldberg, Robert Johnson, Dawn Song, and David Wagner. 2001. A cryptanalysis of the high-bandwidth digital content protection system.

In *ACM Workshop on Digital Rights Management*. Springer, 192–200.
[17] David Ruddock. 2014 (accessed Feb 06, 2018). *[Weekend Poll] Is Your Primary Android Device Rooted?* http://www.androidpolice.com/2014/11/23/weekend-poll-is-your-primary-android-device-rooted-2/.
[18] X de Carné de Carnavalet and Mohammad Mannan. 2016. Killed by proxy: Analyzing client-end TLS interception software. In *Proc. Network and Distributed System Security Symposium (NDSS)*.
[19] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 289–298.
[20] doridori. 2017 (accessed Feb 06, 2018). *Android-Security-Reference/TEE.md*. https://github.com/doridori/Android-Security-Reference/blob/master/hardware/TEE.md.
[21] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. 2017. The security impact of HTTPS interception. In *Proc. Network and Distributed System Security Symposium (NDSS)*.
[22] Editors of Inc. 2017. MAZ Systems - New York, NY. (2017). https://www.inc.com/profile/maz-systems
[23] F Falcon and N Riva. 2012. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *Reverse Engineering Conference*.
[24] Paolo Gasti and Kasper B. Rasmussen. 2012. On the Security of Password Manager Database Formats. In *Computer Security – ESORICS 2012*. Springer, 770–787.
[25] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. 2017. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*. ACM.
[26] Christian Grothoff, Krista Grothoff, Ryan Stutsman, Ludmila Alkhutova, and Mikhail Atallah. 2009. Translation-based steganography. *Journal of Computer Security* 17, 3 (2009), 269–303.
[27] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. 2008. Lest we remember: cold boot attacks on encryption keys. In *17th USENIX Security Symposium*. 45–60.
[28] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. 2017. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. 2231–2244.
[29] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium*. 757–772.
[30] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In *12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2015)*. Springer, 3–24.
[31] Eugene Kolodenker, William Koch, Gianluca Stringhini, and Manuel Egele. 2017. PayBreak: Defense Against Cryptographic Ransomware. In *Proceedings of the 2017 Asia Conference on Computer and Communications Security (ASIA CCS '17)*. 599–611.
[32] Xiaoning Li and Kang Li. 2014. Defeating the transparency features of dynamic binary instrumentation. BlackHat US. (2014).
[33] Magisk Manager. 2018 (accessed Feb 06, 2018). *Download Magisk Manager Latest Version 5.5.5 For Android 2018.* https://magiskmanager.com/#Magisk_Root_Universal_Systemless_Interface.
[34] Jingran Wang Matt Joseph and contributors. 2016. RootCloak. (2016). https://github.com/devadvance/rootcloak
[35] mediaarcadmin. 2010 (accessed Jan 29, 2018). *ios_stack/Sources/Classes/DRM/drmenvelope.h.* https://github.com/mediaarcadmin/ios_stack/blob/master/Sources/Classes/DRM/drmenvelope.h.
[36] Microsoft Corporation. 2015 (accessed Feb 07, 2018). *Microsoft PlayReady Content Protection Technology.* https://www.microsoft.com/playready/documents/.
[37] Microsoft Corporation. 2017 (accessed Feb 07, 2018). *PlayReady Header Specification.* https://www.microsoft.com/playready/documents/.
[38] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *IEEE Symposium on Security and Privacy*.
[39] MITRE Corporation. 2018 (accessed Feb 05, 2018). *CVE-2018-5686.* https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5686.
[40] Bernhard Mueller and Sven Schleier. 2018. OWASP Mobile App Security Requirements and Verification v 1.0. (2018). https://github.com/OWASP/owasp-masvs/releases/download/1.0/OWASP_Mobile_AppSec_Verification_Standard_v1.0.pdf
[41] Nahuel Cayetano Riva. 2015 (accessed Feb 06, 2018). *Anti-instrumentation techniques: I know you're there, Frida!* https://crackinglandia.wordpress.com/2015/11/10/anti-instrumentation-techniques-i-know-youre-there-frida/.
[42] Zhenyu Ning and Fengwei Zhang. 2017. Ninja: Towards Transparent Tracing and Debugging on ARM. In *26th USENIX Security Symposium*. 33–49.
[43] Bradley Reaves, Nolen Scaife, Adam M Bates, Patrick Traynor, and Kevin RB Butler. 2015. Mo (bile) Money, Mo (bile) Problems: Analysis of Branchless Banking Applications in the Developing World.. In *USENIX Security Symposium*. 17–32.
[44] Rohit Salecha. 2017 (accessed Feb 06, 2018). *Pentesting Android Apps Using Frida.* https://www.notsosecure.com/pentesting-android-apps-using-frida/.
[45] Ryne Hager. 2017 (accessed Feb 06, 2018). *Google may have updated SafetyNet detection, breaking some root hiding.* http://www.androidpolice.com/2017/06/14/google-may-updated-safetynet-detection-breaking-root-hiding-methods/.
[46] Ryne Hager. 2017 (accessed Feb 06, 2018). *SafetyNet can detect Magisk again, but a fix is in the works.* http://www.androidpolice.com/2017/07/16/safetynet-can-detect-magisk-fix-works/.
[47] N. Scaife, H. Carter, P. Traynor, and K. R. B. Butler. 2016. CryptoLock (and Drop It): Stopping Ransomware Attacks on User Data. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 303–312.
[48] Daniele Sgandurra, Luis Muñoz-González, Rabih Mohsen, and Emil C Lupu. 2016. Automated dynamic analysis of ransomware: Benefits, limitations and use for detection. *arXiv preprint arXiv:1609.03020* (2016).
[49] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.
[50] David Silver, Suman Jana, Dan Boneh, Eric Chen, and Collin Jackson. 2014. Password Managers: Attacks and Defenses. In *23rd USENIX Security Symposium*. 449–464.
[51] Synk Security Team. 2018. Zip Slip Vulnerability. (2018). https://github.com/snyk/zip-slip-vulnerability
[52] Mercan Topkara, Umut Topkara, and Mikhail J Atallah. 2007. Information hiding through errors: a confusing approach. In *Security, Steganography, and Watermarking of Multimedia Contents IX*, Vol. 6505. International Society for Optics and Photonics.
[53] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2013. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services. In *USENIX Security Symposium*. 687–702.

A APPENDIX

A.1 Legal and Ethical Matters

First and foremost, this research is definitely not aimed at assisting piracy. We have not and will not distribute any code and other artifacts used in conducting the experiments.

As this research was done inside the United States, it is our understanding that the DMCA security research exemption [2] should be applicable. We believe what we did in this paper meets the four main requirements for the said exemption: 1) the apps and the device of which the apps were running on were all lawfully acquired; 2) the experiments were done *solely for the purpose of good-faith security research*; 3) the research was conducted *in a controlled setting designed to avoid harm to individuals or the public*; 4) the research did not begin before October 28, 2016.

For ethical reasons, after an attack has been demonstrated to be working, we stop our experiments and did not perform mass content extraction for our personal gains. For example, in the case of Amazon Music, we tried content decryption on only four songs, in order to gain confidence that decryption with the same key would work on songs that are: 1) from different tiers of subscription; 2) stored on the same device but different albums; 3) stored on different devices. Similarly, for each of the group-7–8 apps, we only downloaded 2 random pages from 2 magazine issues. The extracted raw contents (*e.g.*, audio tracks and magazine pages) have been subsequently deleted. We have also engaged in responsible disclosure with the app vendors, demonstrating good-faith.

A.2 Table of Apps and CWEs

The complete list of publication apps that we studied can be found at the end of this paper in Table 1. The list of CWEs discussed in this paper can be found in Table 2.

Table 1: List of content distribution apps studied in this paper

App Name	Version	Publisher	† Latest In-App Issue Cover Date	Latest In-App Issue Price ($)	‡ Category	‡ Installs	Attacks and Adversaries Discussed ❖
Group-1 Apps (Vendor: Apazine)							
The MagPi	5.0.3	The Raspberry Pi Foundation	Apr, 2018	3.99	News & Magazines	50000+	EVA (Sect. 3.1, $\mathcal{A}_{\text{Net(Sniff)}}$) PBA (Sect. 3.2, $\mathcal{A}_{\text{Net(Sniff)}}$) PBA (Sect. 4.1, $\mathcal{A}_{\text{ExS(R)}}$) KEA (Sect. 4.3, $\mathcal{A}_{\text{ExS(R)}}$)
Business Money	5.0	Business Money Promotions	Mar, 2018	14.99	News & Magazines	1000+	
Artists & Illustrators	5.1	The Chelsea Magazine Company	Jun, 2018	5.99	Lifestyle	1000+	
My MS-UK	5.0.3	MS-UK	Jan/Feb 2018	3.99	News & Magazines	100+	
Popshot Magazine	5.2	The Chelsea Magazine Company	Spring, 2018	7.49	Lifestyle	50+	
Group-2 Apps (Vendor: Apazine)							
Counter Intelligence Plus	4.0.0	Communications International Group	Jan 05, 2017	Free	Medical	1,000+	KEA (Sect. 4.5, $\mathcal{A}_{\text{InS(R)}}$)
Group-3 Apps (Vendor: Maz Systems)							
Forbes Magazine Ψ	6.1.0	Forbes Media	May 31, 2018	5.99	Business	100,000+	PBA (Sect. 3.2, $\mathcal{A}_{\text{Net(Sniff)}}$) EVA (Sect. 3.3, $\mathcal{A}_{\text{Net(TLSInt)}}$) Privacy Threats (Sect. 3.5) CEA (Sect. 4.4, $\mathcal{A}_{\text{InS(R)}}$) PBA (Sect. 4.7, $\mathcal{A}_{\text{InS(R+W)}}$)
Harvard Business Review	4.6.15 4.6.56 #	Harvard Business Publishing	May 01, 2018	18.99	Business	10,000+	
Designs in Machine Embroidery	6.1.0	Designs in Machine Embroidery	May 01, 2018	4.99	Lifestyle	10,000+	
Diabetes Self-Management	6.1.0	Madavor Media	Jun 01, 2018	5.99	Health & Fitness	10,000+	
ForbesLife	6.1.0	Forbes Media	Nov 23, 2015	6.99	Lifestyle	10,000+	
Mother Earth News	6.1.0, 6.1.56 #	Ogden Publications, Inc.	Feb 01, 2018	5.99	Lifestyle	10,000+	
Boys' Life	6.1.4, 6.1.56 #	Boy Scouts of America	Jun 01, 2018	4.99	News & Magazines, Education	5,000+	
Craft Beer & Brewing Magazine	6.1.50	Unfiltered Media Group	May 09, 2018	9.99	Lifestyle	5,000+	
GRIT Magazine	6.1.0	Ogden Publications, Inc.	May 01, 2018	4.99	Lifestyle	5,000+	
Guitar World	6.1.15	NewBay Media	Jul 01, 2018	7.99	Music & Audio	5,000+	
Inside Lacrosse	6.1.0	American City Business Journals	May 01, 2018	4.99	Sports	5,000+	
Mother Earth Living	6.1.0	Ogden Publications, Inc.	May 01, 2018	5.99	Lifestyle	5,000+	
USA Today Sports Weekly	6.1.0, 6.1.59 #	Gannett Company	May 29, 2018	2.99	Sports	5,000+	
ABA Journal Magazine	6.1.0	American Bar Association	May 01, 2018	6.99	Business	1,000+	
American Cheerleader Magazine	6.1.0	Varsity Spirit	Mar 21, 2018	3.99	Sports	1,000+	
BirdWatching	6.1.0	Madavor Media	May 01, 2018	5.99	Lifestyle	1,000+	
BUST Magazine	6.1.0	Debbie Stoller and Laurie Henzel	Apr 01, 2018	4.99	Entertainment	1,000+	
Cake Central Magazine	6.1.0	Cake Central Media Corp.	Apr 01, 2017	5.99	Lifestyle	1,000+	
Scouting magazine	6.1.0	Boy Scouts of America	May 01, 2018	3.99	Lifestyle	1,000+	
Faerie Magazine	6.1.0	Faerie Magazine	Mar 15, 2018	4.99	Lifestyle	1,000+	
Grassroots Motorsports Mag	6.1.15	Tim Suddard	Jun 01, 2018	5.99	Lifestyle	1,000+	
Guitar Player Magazine++	6.1.15	NewBay Media	Jun 01, 2018	6.99	Music & Audio	1,000+	
JazzTimes	6.1.0	Madavor Media	Jun 01, 2018	3.99	Music & Audio	1,000+	
Joy of Kosher Magazine	6.1.0	Kosher Network International	Nov 24, 2017	3.99	Lifestyle	1,000+	
Kayak Angler	6.1.15	Rapid Media	Apr 01, 2018	3.99	Sports	1,000+	
Leatherneck Magazine	6.1.0	Marine Corps Association	Jun 01, 2018	4.99	News & Magazines	1,000+	
Marine Corps Gazette	6.1.0	Marine Corps Association	Jun 01, 2018	4.99	News & Magazines	1,000+	
Motorcycle Classics Magazine	6.1.0	Ogden Publications, Inc.	May 01, 2018	6.99	Entertainment	1,000+	

National Wildlife magazine	6.1.0	National Wildlife Federation	Apr 01, 2018	3.99	Education	1,000+	
New York Observer	6.1.0	Observer Media	Nov 14, 2016	1.99	News & Magazines	1,000+	
Paddling Mag	6.1.15	Rapid Media	Apr 01, 2018	2.99	Sports	1,000+	
Paleo Magazine	6.1.12	Paleo Magazine	Jul 05, 2018	2.99	Health & Fitness	1,000+	
The Writer	6.1.0	Madavor Media	Jul 01, 2018	5.99	Education	1,000+	
V Magazine	6.1.0	Visionaire	May 03, 2018	3.99	Entertainment	1,000+	
Volleyball Magazine	6.1.0	Volleyball World Wide	Apr 29, 2016	3.99	Sports	1,000+	
Adventure Kayak+ Magazine	6.1.15	Rapid Media	Jun 01, 2017	3.99	Sports	500+	
Art Photo Feature	6.1.0	APF Magazine	Feb 01, 2016	3.99	Photography	500+	
City Journal	6.1.0	Manhattan Institute for Policy Research	Apr 15, 2018	3.99	Business	500+	
Farm Collector	6.1.0	Ogden Publications, Inc.	Jun 01, 2018	4.99	Lifestyle	500+	
Gluten-Free Living	6.1.0	Madavor Media	May 01, 2018	6.99	Health & Fitness	500+	
Keyboard Magazine	6.1.0	NewBay Media	Jun 01, 2018	5.99	Music & Audio	500+	
Man of the World Magazine	6.1.0	Man of the World	Oct 17, 2016	9.99	Lifestyle	500+	
The Real Deal Magazine	6.1.0	Korangy Publishing	May 01, 2018	2.99	Business	500+	
Revolver Magazine	6.1.15	NewBay Media	Apr 01, 2018	6.99	Music & Audio	500+	
Art Business News	6.1.0	Redwood Media Group	Apr 01, 2016	0.99	Business	100+	
Animania Magazine	6.1.15	RSPCA NSW	Mar 01, 2018	3.99	Lifestyle	100+	
Pain-Free Living	6.1.0	Madavor Media	Jun 01, 2018	4.99	Health & Fitness	100+	
Bass Player+	6.1.15	NewBay Media	Jun 01, 2018	5.99	Music & Audio	100+	
Digital Video	6.1.15	NewBay Media	Jan 01, 2018	4.99	Productivity	100+	
Electronic Musician+	6.1.0	NewBay Media	Jun 01, 2018	5.99	Music & Audio	100+	
Guitar Aficionado	6.1.15	NewBay Media	Jan 01, 2018	7.99	Lifestyle	100+	
Hail Varsity Magazine	6.1.0	Hail Varsity	Feb 13, 2018	2.99	Sports	100+	PBA (Sect. 3.2, $\mathcal{A}_{\text{Net(Sniff)}}$) EVA (Sect. 3.3, $\mathcal{A}_{\text{Net(TLSInt)}}$) Privacy Threats (Sect. 3.5) CEA (Sect. 4.4, $\mathcal{A}_{\text{InS(R)}}$) PBA (Sect. 4.7, $\mathcal{A}_{\text{InS(R+W)}}$)
Inside Pitch	6.1.0	The American Baseball Coaches Association	May 01, 2018	1.99	Sports	100+	
Inside Weddings	6.1.0	Inside Weddings	Mar 13, 2018	5.99	Lifestyle	100+	
Multichannel News++	6.1.15	NewBay Media	May 14, 2018	6.99	Music & Audio	100+	
Mix Magazine+	6.1.0	NewBay Media	May 01, 2018	6.99	Music & Audio	100+	
MUSE Magazine	6.1.0	MUSE Magazine	Feb 19, 2018	3.99	Lifestyle	100+	
National Affairs	6.1.15	National Affairs, Inc.	Mar 21, 2018	3.99	Education	100+	
TWICE+	6.1.0, 6.1.56 #	NewBay Media	May 21, 2018	9.99	Music & Audio	100+	
AV Technology	6.1.12	NewBay Media	May 01, 2018	5.99	Business	50+	
Broadcasting & Cable++	6.1.8	NewBay Media	May 21, 2018	6.99	Music & Audio	50+	
Deli Business	6.1.0	Phoenix Media Network	Dec 01, 2017	14.99	Business	50+	
HeirlmGardnrMag	6.1.15	Ogden Publications, Inc.	Mar 01, 2018	9.99	Lifestyle	50+	
Sound Video Contractor	6.1.15	NewBay Media	May 01, 2018	6.99	Business	50+	
Digital Signage	6.1.15	NewBay Media	Apr 27, 2018	5.99	Business	10+	
Resident Sys	6.1.15	NewBay Media	Jun 01, 2018	4.99	Business	10+	
System Contractor News	6.1.15	NewBay Media	Jun 01, 2018	5.99	Business	10+	
Tech&Learning	6.1.15	NewBay Media	May 01, 2018	6.99	Education	10+	
Pro Sound News §	6.1.15	NewBay Media	May 01, 2018	5.99	Business (Amazon App Store)	Ranked 1250 in Business	
Revista La Fuente §	6.1.0	Revista La Fuente	Jun 01, 2018	2.99	Lifestyle (Amazon App Store)	Ranked 7367 in Lifestyle	
Group-4 Apps (Vendor: Maz Systems)							
Bloomberg Businessweek+	2.4.6	Bloomberg L.P.	Subscription-based	59.99 yearly	News & Magazines	10,000,000+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$) PBA (Sect. 4.6, $\mathcal{A}_{\text{InS(R)}}$)
Salon.com	2	Salon Media Group	Subscription-based	49.99 yearly	News & Magazines	1000+	

Group-5 Apps (Vendor: Maz Systems)							
Entrepreneur Magazine	10020	Entrepreneur Media	Jun, 2018	4.99	Business	100,000+	
Forbes Magazine Ψ	10020	Forbes Media	Jun, 2018	5.99	Business	100,000+	
Diesel World	10010	Engaged Media	Aug, 2018	5.99	News & Magazines	10000+	
Knives Illustrated	10020	Engaged Media	Jul, 2018	4.99	News & Magazines	5000+	
Gun World	10020	Engaged Media	Jul, 2018	3.99	News & Magazines	5000+	
American Survival Guide	10020	Engaged Media	Jul, 2018	6.99	News & Magazines	5000+	
Ultimate Diesel Builders Guide	10020	Engaged Media	Jun, 2018	5.99	News & Magazines	5000+	
Outside TV Features	10021	Outside Television	Subscription-based	4.99 monthly	Sports	1,000+	
Inc. Must Reads and Magazine	10020	Mansueto Ventures	Jun, 2018	4.99	Business	1,000+	
The Nation Magazine	10020	The Nation Company	May 28, 2018	1.99	News & Magazines	1,000+	
Conceal & Carry	10020	Engaged Media	Summer, 2018	7.99	News & Magazines	1000+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$) PBA (Sect. 4.6, $\mathcal{A}_{\text{InS(R)}}$)
Cottages & Bungalow	10020	Engaged Media	Jun, 2018	8.99	News & Magazines	1000+	
The Rebel Media	10020	The Rebel News Network	Subscription-based	84.99 yearly	News & Magazines	1,000+	
Lion's Roar Magazine	10020	Lion's Roar Foundation	Subscription-based	23.99 yearly	Lifestyle	500+	
Buddhadharma	10020	Lion's Roar Foundation	Subscription-based	23.99 yearly	News & Magazines	500+	
Texas Monthly	10020	Texas Monthly	Jun, 2018	4.99	News & Magazines	500+	
All About Beer	10020	All About Beer	Mar, 2018	4.99	Food & Drink	100+	
Atomic Ranch	10020	Engaged Media	Summer, 2018	5.99	News & Magazines	100+	
FNF Coaches	10020	A.E. Engine	Apr, 2018	varies	Sports	100+	
Tread Magazine	10020	Engaged Media	May, 2018	7.99	Lifestyle	100+	
Vogue Knitting	1	Soho Publishing	Spring, 2018	5.99	News & Magazines	100+	
American Farmhouse Style	10020	Engaged Media	Summer, 2018	7.99	Lifestyle	10+	
Berko	10020	Pat Callinan Media	May 10, 2018	5.99	Lifestyle	10+	
Group-6 Apps (Vendor: Maz Systems)							
Men's Health Magazine	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	10,000+	
Runner's World	10020	Hearst Communications	Jul, 2018	4.99	Health & Fitness	1,000+	
Women's Health Mag	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	1,000+	
Prevention	10020	Hearst Communications	Jun, 2018	4.99	Health & Fitness	500+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$) CEA (Sect. 4.2, $\mathcal{A}_{\text{ExS(R)}}$)
Quilting	1	F+W Media	Jun, 2018	6.99	Education	500+	
Bicycling	10020	Hearst Communications	Jul, 2018	4.99	Health & Fitness	100+	
FNF | Friday Night Football	10020	A.E. Engine	2017	0.99	Sports	100+	
IW Knits	1	F+W Media	Summer, 2018	7.99	Education	50+	
ArtistsMag	1	F+W Media	Jul, 2018	6.99	Education	50+	
Group-7 Apps (Vendor: Pugpig)							
The Independent Daily Edition	4.5.1313.370	Independent Digital News & Media Limited	Wednesday 27 Jun, 2018	168.74 yearly	News & Magazines	50,000+	
Primal 9	1.1.3804.716	Hearst Magazines UK	2018	49.99	Health & Fitness	10,000+	
Cosmopolitan UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	9.99 yearly	Lifestyle	5,000+	
Glamour Magazine (UK)	1.7.2137.893	The Condé Nast Publications Limited	Spring/Summer, 2018	0.99	News & Magazines	5,000+	
Wired UK	33.3.187.893	The Condé Nast Publications Limited	May/Jun, 2018	17.99 yearly	News & Magazines	5,000+	
Condé Nast Traveler Magazine	1.2.1189.893	The Condé Nast Publications Limited	Jun, 2018	29.99 yearly	Lifestyle	1,000+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$)
GQ Style UK	1.2.3455.893	The Condé Nast Publications Limited	Spring/Summer, 2018	9.99 yearly	Lifestyle	1,000+	
Tatler	1.2.1189.893	The Condé Nast Publications Limited	Jul, 2018	29.99 yearly	Lifestyle	1,000+	
WH Transform	1.0.2982.490	Hearst Magazines UK	2018	54.99	Health & Fitness	1,000+	
Brides	1.2.1189.893	The Condé Nast Publications Limited	May/Jun, 2018	23.99 yearly	Lifestyle	500+	

House & Garden	1.2.1189.893	The Condé Nast Publications Limited	Jul, 2018	30.99 yearly	Lifestyle	500+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$)
Reveal UK	6.5.31.50.377	Hearst Magazines UK	Week 26, 2018	26.99 yearly	Entertainment	100+	
The World of Interiors	1.1.326.893	The Condé Nast Publications Limited	Jul, 2018	35.99 yearly	Lifestyle	100+	
QP Magazine	1.0.3390.1350	Hearst Magazines UK	Jun, 2018	12.99 yearly	News & Magazines	10+	
Group-8 Apps (Vendor: Pugpig)							
ELLE Magazine UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	5,000+	PBA (Sect. 3.4, $\mathcal{A}_{\text{Net(TLSInt)}}$) CEA (Sect. 4.4, $\mathcal{A}_{\text{InS(R)}}$)
Men's Health UK	1.1.3804.716	Hearst Magazines UK	Jul, 2018	29.99 yearly	Lifestyle	5,000+	
ELLE Decoration UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	34.99 yearly	Lifestyle	1,000+	
Esquire UK	6.5.1665.377	Hearst Magazines UK	Jul/Aug, 2018	19.99 yearly	Lifestyle	1,000+	
Good Housekeeping UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	1,000+	
Harper's Bazaar UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	35.99 yearly	Lifestyle	1,000+	
Inside Soap UK	6.5.1655.377	Hearst Magazines UK	Week 26, 2018	59.99 yearly	Entertainment	1,000+	
Runner's World UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	35.99 yearly	Health & Fitness	1,000+	
Women's Health UK	1.5.3696.377	Hearst Magazines UK	Jul, 2018	29.99 yearly	Lifestyle	1,000+	
House Beautiful UK	6.5.1655.377	Hearst Magazines UK	Aug, 2018	30.99 yearly	Lifestyle	500+	
Country Living UK	6.5.1655.377	Hearst Magazines UK	Jun, 2018	29.99 yearly	Lifestyle	100+	
Prima UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	28.99 yearly	Lifestyle	100+	
Real People UK	6.5.1655.377	Hearst Magazines UK	Week 27, 2018	26.99 yearly	Entertainment	100+	
Red Magazine UK	6.5.1655.377	Hearst Magazines UK	Jul, 2018	23.99 yearly	Lifestyle	100+	
Town & Country	6.5.1655.377	Hearst Magazines UK	Summer, 2018	18.99 yearly	Lifestyle	100+	
Best UK	6.5.1655.377	Hearst Magazines UK	Week 26, 2018	30.99 yearly	Entertainment	50+	
Group-9 Apps (Vendor: Amazon)							
Amazon Music	6.5.3	various	2-tier subscriptions		Music & Audio	100,000,000+	KEA (Sect. 4.8, $\mathcal{A}_{\text{Mem+BinIns}}$)
Audible	2.25.0	various	various subscription plans exist		Books & Reference	100,000,000+	

§ These 2 Apps were only available on Amazon App Store but not on Google Play Store. ‡ Information in these columns retrieved from Google Play Store in Jun, 2018.
# For these apps, the newer version uses an encrypted database. They are however still susceptible to PBAs through attack surface **(AS1)** as discussed in Section 3.2.
Ψ The new version was released during the course of our study to replace the old one. The varied designs are susceptible to different attacks, though the pricing stays the same.
† Some cover dates are in the future due to 1) some have long intervals between issues; 2) some offer digital access earlier than in print. ($) All prices are in US dollars.
❖ KEA = Key Extraction Attacks; PBA = Purchase Bypass Attacks; CEA = Content Extraction Attacks; EVA = Eavesdropping Attacks. In this paper, KEA and EVA both imply CEA.

Table 2: List of CWEs discussed in this paper

CWE ID	Name	Description
CWE-313	Cleartext Storage in a File or on Disk	The application stores sensitive information in cleartext in a file, or on disk.
CWE-319	Cleartext Transmission of Sensitive Information	The software transmits sensitive or security-critical data in cleartext in a communication channel that can be sniffed by unauthorized actors.
CWE-354	Improper Validation of Integrity Check Value	The software does not validate or incorrectly validates the integrity check values or "checksums" of a message. This may prevent it from detecting if the data has been modified or corrupted in transmission.
CWE-425	Direct Request ('Forced Browsing')	The web application does not adequately enforce appropriate authorization on all restricted URLs, scripts, or files.
CWE-454	External Initialization of Trusted Variables or Data Stores	The software initializes critical internal variables or data stores using inputs that can be modified by untrusted actors.
CWE-532	Information Exposure Through Log Files	Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.
CWE-603	Use of Client-Side Authentication	A client/server product performs authentication within client code but not in server code, allowing server-side authentication to be bypassed via a modified client that omits the authentication check.
CWE-642	External Control of Critical State Data	The software stores security-critical state information about its users, or the software itself, in a location that is accessible to unauthorized actors.
CWE-654	Reliance on a Single Factor in a Security Decision	A protection mechanism relies exclusively, or to a large extent, on the evaluation of a single condition or the integrity of a single object or entity in order to make a decision about granting access to restricted resources or functionality.
CWE-656	Reliance on Security Through Obscurity	The software uses a protection mechanism whose strength depends heavily on its obscurity, such that knowledge of its algorithms or key data is sufficient to defeat the mechanism.
CWE-921	Storage of Sensitive Data in a Mechanism without Access Control	The software stores sensitive information in a file system or device that does not have built-in access control.