Mind Your Keys? A Security Evaluation of Java Keystores

Riccardo Focardi
Università Ca' Foscari
Cryptosense
focardi@unive.it

Francesco Palmarini
Università Ca' Foscari
Yarix
palmarini@unive.it

Marco Squarcina
Università Ca' Foscari
Cryptosense
squarcina@unive.it

Graham Steel
Cryptosense
graham@cryptosense.com

Mauro Tempesta
Università Ca' Foscari
tempesta@unive.it

Abstract—**Cryptography is complex and variegate and requires to combine different algorithms and mechanisms in non-trivial ways. This complexity is often source of vulnerabilities. Secure key management is one of the most critical aspects, since leaking a cryptographic key vanishes any advantage of using cryptography. In this paper we analyze Java keystores, the standard way to manage and securely store keys in Java applications. We consider seven keystore implementations from Oracle JDK and Bouncy Castle, a widespread cryptographic library. We describe, in detail, how the various keystores enforce confidentiality and integrity of the stored keys through password-based cryptography and we show that many of the implementations do not adhere to state-of-the-art cryptographic standards. We investigate the resistance to offline attacks and we show that, for non-compliant keystores, brute-forcing can be up to three orders of magnitude faster with respect to the most compliant keystore. Additionally, when an attacker can tamper with the keystore file, some implementations are vulnerable to denial of service attacks or, in the worst case, arbitrary code execution. Finally we discuss the fixes implemented by Oracle and Bouncy Castle developers following our responsible disclosure.**

I. Introduction

Cryptography is a fundamental technology for IT security. Even if there are well established standards for cryptographic operations, cryptography is complex and variegated, typically requiring a non-trivial combination of different algorithms and mechanisms. Moreover, cryptography is intrinsically related to the secure management of cryptographic keys which need to be protected and securely stored by applications. Leaking cryptographic keys, in fact, diminishes any advantage of cryptography, allowing attackers to break message confidentiality and integrity, to authenticate as legitimate users or impersonate legitimate services. Quoting [51], "key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system".

In the recent years we have faced a multitude of flaws related to cryptography (*e.g.*, [16], [12], [36], [35]). Some of these are due to the intrinsic complexity of cryptography, that makes it hard to design applications that adopt secure combinations of mechanisms and algorithms. For example, in padding oracle attacks, the usage of some (standard) padding for the plaintext combined with a specific algorithm or mechanism makes it possible for an attacker to break a ciphertext in a matter of minutes or hours [54], [19], [12]. Most of the time this is not a developer fault as, unfortunately, there are well-known flawed mechanisms that are still enabled in cryptographic libraries. In other cases, the attacks are due to flaws in protocols or applications. The infamous Heartbleed bug allowed an attacker to get access to server private keys through a simple over-read vulnerability. Once the private key was leaked, the attacker could decrypt encrypted traffic or directly impersonate the attacked server [36].

Thus, breaking cryptography is not merely a matter of breaking a cryptographic algorithm: the attack surface is quite large and the complexity of low-level details requires abstractions. Crypto APIs offer a form of abstraction to developers that allows to make use of cryptography in a modular and implementation-independent way. The Java platform, for example, provides a very elegant abstraction of cryptographic operations that makes it possible, in many cases, to replace a cryptographic mechanism or its implementation with a different one without modifying the application code.

Crypto APIs, however, do not usually provide security independently of the low-level implementation: default mechanisms are often the weakest ones, thus developers have to face the delicate task of choosing the best mechanism available for their needs. For example, in the Java Cryptography Architecture (JCA), ECB is the default mode of operation for block ciphers [4] and PKCS#1 v1.5 is the default padding scheme for RSA [5], which is well know to be subject to padding oracle attacks [19]. Additionally, crypto APIs that promise to provide security for cryptographic keys have often failed to do so: in PKCS#11, the standard API to cryptographic tokens, it is possible to wrap a sensitive key under another key and then just ask the device to decrypt it, obtaining the value of the sensitive key in the clear [23], and violating the requirement that "sensitive keys cannot be revealed in plaintext off the token" [48].

In this paper we analyze in detail the security of key management in the Java ecosystem and, in particular, of Java keystores. Password-protected keystores are, in fact, the standard way to securely manage and store cryptographic keys in Java: once the user (or the application) provides the correct password, the keys in the keystore become available and can be

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23083
www.ndss-symposium.org

used to perform cryptographic operations, such as encryption and digital signature. The `KeyStore` Java class abstracts away from the actual keystore implementation, which can be either in the form of an encrypted file or based on secure hardware. As discussed above, this abstraction is very important for writing code that is independent of the implementation but developers are still required to select among the various keystore *types* offered by Java. Unfortunately, the information in the keystore documentation is not enough to make a reasoned and informed choice among the many alternatives. More specifically, given that the Java Keystore API does not provide control over the cryptographic mechanisms and parameters employed by each keystore, it is crucial to assess the security provided by the different implementations, which motivated us to perform the detailed analysis reported in this paper. In fact, our work is the first one studying the security of keystores for general purpose Java applications.

We have estimated the adoption rate and analyzed the implementation details of seven different Java keystores offered by the Oracle JDK and by Bouncy Castle, a widespread cryptographic library. Keystores are used by hundreds of commercial applications and open-source projects, as assessed by scraping the GitHub code hosting service including leading web applications servers and frameworks, *e.g.*, Tomcat [6], Spring [8], Oracle Weblogic [2]. Additionally, keystores have been found to be widespread among security-critical custom Java software for large finance, government and healthcare companies audited by the authors.

The security of keystores is achieved by performing a cryptographic operation C under a key which is derived from a password through a function F called Key Derivation Function (KDF). The aim of the cryptographic operation C is to guarantee confidentiality and/or integrity of the stored keys. For example, a Password-Based Encryption (PBE) scheme is used to protect key confidentiality: in this case C is typically a symmetric cipher, so that keys are encrypted using the provided password before being stored in the keystore. In order to retrieve and use that key, the keystore implementation will perform the following steps: (a) obtain the password from the user; (b) derive the encryption key from the password using F; (c) decrypt the particular keystore entry through C, and retrieve the actual key material. Notice that different passwords can be used to protect different keys and/or to achieve integrity. To prevent attacks, it is highly recommended that C and F are implemented using standard, state-of-the-art cryptographic techniques [40], [49].

Interestingly, we have found that the analyzed keystores use very diverse implementations for C and F and in several cases they do not adhere to standards or use obsolete and ad-hoc mechanisms. We show that, most of the time, keystores using weak or custom implementations for the key derivation function F open the way to password brute-forcing. We have empirically measured the speed-up that the attacker achieves when these flawed keystores are used and we show that, in some cases, brute-forcing is three orders of magnitude faster with respect to the keystores based on standard mechanisms. We even found keystores using the deprecated cipher RC2 that enables an attacker to brute-force the 40-bit long cryptographic key in a matter of hours using a standard desktop computer.

Our analysis has also pointed out problems related to availability and malicious code execution, which are caused by *type-flaws* in the keystore, *i.e.*, bugs in which an object of a certain type is interpreted as one of a different type. In particular, by directly tampering with the keystore file, an attacker could trigger denial of service (DoS) attacks or even arbitrary code execution. Interestingly, we also found that the use of standard key derivation functions can sometimes enable DoS attacks. These functions are parametrized by the number of internal iterations, used to slow down brute-forcing, which is stored in the keystore file. If the number of iterations is set to a very big integer, the key derivation function will hang, blocking the whole application.

Unless stated otherwise, our findings refer to Oracle JDK 8u144 and Bouncy Castle 1.57, the two latest releases at the time of the first submission of this work in August 2017.

Contributions: Our contributions can be summarized as follows:

(i) we define a general threat model for password-protected keystores and we distill a set of significant security properties and consequent rules that any secure keystore should adhere to;
(ii) we perform a thoughtful analysis of seven keystores, we report undocumented details about their cryptographic implementations and we classify keystores based on our proposed properties and rules;
(iii) we report on unpublished attacks and weaknesses in the analyzed keystores. For each attack we point out the corresponding violations of our proposed properties and rules and we provide a precise attacker model;
(iv) we empirically estimate the speed-up due to bad cryptographic implementations and we show that, in some cases, this allows to decrease the guessing time of three orders of magnitude with respect to the most resistant keystore, and four orders of magnitude with respect to NIST recommendations; interestingly, the attack on Oracle JKS keystore that we present in this paper, and we previously mentioned in a blog post [25], has been recently integrated into the Hashcat password recovery tool;
(v) we discuss the advancements on the security of Oracle and Bouncy Castle keystore implementations following our responsible disclosure. The Oracle Security Team acknowledged the reported issues by assigning two CVE IDs [37], [38] and released partial fixes in the October 2017 Critical Patch Update [44]. Other fixes are expected to be released in January 2018 [43]. Bouncy Castle developers patched some of the reported vulnerabilities in version 1.58. As of November 2017, remaining issues are being addressed in the development repository.

Paper Organization: We discuss related work in Section II; in Section III we define the security properties of interest, the rules for the design of secure keystores and the threat model; in Section IV we report on our analysis of seven Java keystores; in Section V we describe unpublished attacks on the analyzed keystores; in Section VI we make an empirical comparison of the password cracking speed among the keystores; in Section VII we discuss the improvements implemented by Oracle and Bouncy Castle following our responsible disclosure; finally, in Section VIII we draw some concluding remarks.

II. Related Work

Cooijmans *et al.* [24] have studied various key storage solutions in Android, either provided as an operating system service or through the Bouncy Castle cryptographic library. The threat model is very much tailored to the Android operating system and radically different from the one we consider in this paper. Offline brute-forcing, for example, is only discussed marginally in the paper. Interestingly, authors show that under a root attacker (*i.e.*, an attacker with root access to the device), the Bouncy Castle software implementation is, in some respect, more secure than the Android OS service using TrustZone's capabilities, because of the possibility to protect the keystore with a user-supplied password. Differently from our work, the focus of the paper is not on the keystore design and the adopted cryptographic mechanisms.

Sabt *et al.* [50] have recently found a forgery attack in the Android KeyStore service, an Android process that offers a keystore service to applications and is out of the scope of our work. However, similarly to our results, the adopted encryption scheme is shown to be weak and not compliant to the recommended standards, enabling a forgery attack that make apps use insecure cryptographic keys, voiding any benefit of cryptography.

Li *et al.* [33] have analyzed the security of web password managers. Even if the setting is different, there are some interesting similarities with keystores. In both settings a password is used to protect sensitive credentials, passwords in one case and keys in the other. So the underlying cryptographic techniques are similar. However the kind of vulnerabilities found in the paper are not related to cryptographic issues. Gasti *et al.* [27] have studied the format of password manager databases. There is some similarity with our paper for what concerns the threat model, *e.g.*, by considering an attacker that can tamper with the password database. However, the setting is different and the paper does not account for cryptographic weaknesses and brute-forcing attacks.

Many papers have studied password resistance to guessing, e.g., [31], [21], [55], [57]. While this is certainly a very important subject, our paper takes a complementary perspective: we analyze whether Java keystores provide a sufficient resistance to brute-forcing, compared to existing standards and recommendations. Of course, using a tremendously weak password would make it possible for the attacker to guess it, independently of the keystore implementation. Similarly, if the password is very long and with high entropy, the guess would be infeasible anyway. However, when a password is reasonably strong, the actual implementation makes a difference: brute-force is prevented only when key derivation is done accordingly to recommendations.

Kelsey *et al.* introduced the notion of *key stretching*, a mechanism to increase the time of brute-forcing for low entropy keys [32]. The basic idea is that key derivation should iterate the core derivation function l times so to multiply the computational cost of brute-forcing by l and make it equivalent to the cost of brute-forcing a password with additional $log_2 l$ bits. Intuitively, through this strategy, brute-forcing each password requires the same time as brute-forcing l passwords. Combined with standard random salting (to prevent precomputation of keys), key stretching effectively slows down brute-forcing, and prevents guessing the password even when its complexity is not very high. This idea is at the base of modern, state-of-the-art key derivation functions. In [10], [56], [15], this mechanism has been formalized and analyzed, providing formal evidence of its correctness. Standard key derivation functions are all based on key stretching and salting to slow down brute-forcing [40], [49]. In our paper we advocate the use of these standard mechanisms for keystores security.

III. Security Properties and Threat Model

In this section, we identify a set of fundamental security properties that should be guaranteed by any keystore (Section III-A). We then distill rules that should be followed when designing a keystore in order to achieve the desired security properties (Section III-B). Finally, we introduce the threat model covering a set of diverse attacker capabilities that enable realistic attack scenarios (Section III-C).

A. Security Properties

We consider standard security properties such as confidentiality and integrity of keys and keystore entries. Breaking confidentiality of sensitive keys allows an attacker to intercept all the encrypted traffic or to impersonate the user. Breaking integrity has similar severe consequences as it might allow an attacker to import fake CA certificates and old expired keys. Additionally, since the access to a keystore is mediated by a software library or an application, we also consider the effect that a keystore has on the execution environment. Thus, we require the following properties:

P1 Confidentiality of encrypted entries
P2 Integrity of keystore entries
P3 System integrity

Property **P1** states that the value of an encrypted entry should be revealed only to authorized users, who know the correct decryption password. According to **P2**, keystore entries should be modified, created or removed only by authorized users, who know the correct integrity password, usually called *store password*. Property **P3** demands that the usage of a keystore should always be tolerated by the environment, *i.e.*, interacting with a keystore, even when provided by an untrusted party, should not pose a threat to the system, cause misbehaviours or hang the application due to an unsustainable performance hit.

A keystore file should be secured similarly to a password file: the sensitive content should not be disclosed even when the file is leaked to an attacker. In fact, it is often the case that keystores are shared in order to provide the necessary key material to various corporate services and applications. Thus, in our threat model we will always assume that the attacker has read access to the keystore file (*cf.* Section III-C). For this reason we require that the above properties hold even in the presence of offline attacks. The attacker might, in fact, brute-force the passwords that are used to enforce confidentiality and integrity and, consequently, break the respective properties.

B. Design Rules

We now identify a set of core rules that should be embraced by the keystore design in order to provide the security guarantees of Section III-A:

R1 Use standard, state-of-the-art cryptography
R2 Choose strong, future-proof cryptographic parameters, while maintaining acceptable performance
R3 Enforce a typed keystore format

Rule **R1** dictates the use of modern and verified algorithms to achieve the desired keystore properties. It is well-known that the design of custom cryptography is a complex task even for experts, whereas standard algorithms have been carefully analyzed and withstood years of cracking attempts by the cryptographic community [13]. In this context, the National Institute of Standards and Technology (NIST) plays a prominent role in the standardization of cryptographic algorithms and their intended usage [14], engaging the cryptographic community to update standards according to cryptographic advances. For instance, NIST declared SHA1 unacceptable to use for digital signatures beginning in 2014, and more recently, urged all users of Triple-DES to migrate to AES for encryption as soon as possible [9] after the findings published in [17]. The KDF function recommended by NIST [52] is PBKDF2, as defined in the PKCS#5 standard, which supersedes the legacy PBKDF1. Another standard KDF function is defined in PKCS#12, although it has been deprecated for confidentiality purposes in favour of PBKDF2.

Key derivation functions combine the password with a randomly generated salt and iteratively apply a pseudorandom function (*e.g.*, a hash function) to produce a cryptographic key. The salt allows the generation of a large set of keys corresponding to each password [56], while the high number of iterations is introduced to hinder brute-force attacks by significantly increasing computational times. Rule **R2** reflects the need of choosing parameters to keep pace with the state-of-the-art in cryptographic research and the advances in computational capabilities. The latest NIST draft on Digital Identity Guidelines [28] sets the minimum KDF iteration count to 10,000 and the salt size to 32 bits. However, such lower bounds on the KDF should be significantly raised for critical keys according to [52] which suggests to set the number of iterations as high as can be tolerated by the environment, while maintaining acceptable performance. For instance, Apple iOS derives the decryption key for the device from the user password using a KDF with an iteration count calculated by taking into account the computational capabilities of the hardware and the impact on the user experience [11].

Finally, rule **R3** states that the keystore format must provide strong typing for keystore content, such that cryptographic objects are stored and read unambiguously. Despite some criticism over the years [29], the PKCS#12 standard embraces this principle providing precise types for storing many cryptography objects. Additionally, given that keystore files are supposed to be accessed and modified by different parties, applications parsing the keystore format must be designed to be robust against malicious crafted content.

Interestingly, not following even one of the aforementioned rules may lead to a violation of confidentiality and integrity of the keystore entries. For instance, initializing a secure KDF with a constant or empty salt, which violates only **R2**, would allow an attacker to precompute the set of possible derived keys and take advantage of *rainbow tables* [41] to speed up the brute-force of the password. On the other hand, a KDF with strong parameters is useless once paired with a weak cipher, since it is easier to retrieve the encryption key rather than brute-forcing the password. In this case only **R1** is violated.

Additionally, disrespecting Rule **R3** may have serious consequences on system integrity (breaking property **P3**), which range from applications crashing due to parsing errors while loading a malicious keystore to more severe scenarios where the host is compromised. An attacker exploiting type-flaw bugs could indirectly gain access to the protected entries of a keystore violating the confidentiality and integrity guarantees. System integrity can additionally be infringed by violating Rule **R2** with an inadequate parameter choice, *e.g.*, an unreasonably high iteration count value might hang the application, slow down the system or prevent the access to cryptographic objects stored in a keystore file due to an excessive computational load. In Section V we show how noncompliance to these rules translate into concrete attacks.

C. Threat Model

In our standard attacker model we always assume that the attacker has read access to the keystore file, either authorized or by means of a data leakage. We also assume that the attacker is able to perform offline brute-force attacks using a powerful system of her choice.

We now present a list of interesting attacker settings, that are relevant with respect to the security properties defined in Section III-A:

S1 Write access to the keystore
S2 Integrity password is known
S3 Confidentiality password of an entry is known
S4 Access to previous legitimate versions of the keystore file

Setting **S1** may occur when the file is shared over a network filesystem, *e.g.*, in banks and large organizations. Since keystores include mechanisms for password-based integrity checks, it might be the case that they are shared with both read and write permissions, to enable application that possess the appropriate credentials (*i.e.*, the integrity password) to modify them. We also consider the case **S2** in which the attacker possesses the integrity password. The password might have been leaked or discovered through a successful brute-force attack. The attacker might also know the password as an insider, *i.e.*, when she belongs to the organization who owns the keystore. Setting **S3** refers to a scenario in which the attacker knows the password used to encrypt a sensitive object. Similarly to the previous case, the password might have been accessed either in a malicious or in honest way. For example, the password of the key used to sign the `apk` of an Android application [1] could be shared among the developers of the team.

In our experience, there exists a strong correlation between **S2** and **S3**. Indeed, several products and frameworks use the same password both for confidentiality and for integrity, *e.g.*, Apache Tomcat for TLS keys and IBM WebSphere for LTPA authentication. Additionally, the standard utility for Java keystores management (`keytool`) supports this practice when creating a key: the tool invites the user to just press the `RETURN` key to reuse the store password for encrypting the entry.

To summarize, our standard attacker model combined with **S1**-**S3** covers both reading and writing capabilities of the attacker on the keystore files together with the possibility of passwords leakage. On top of these settings, we consider the peculiar case **S4** that may occur when the attacker has access to backup copies of the keystore or when the file is shared over platforms supporting version control such as *Dropbox*, *ownCloud* or *Seafile*.

IV. Analysis of Java Keystores

The Java platform exposes a comprehensive API for cryptography through a *provider*-based framework called Java Cryptography Architecture (JCA). A provider consists of a set of classes that implement cryptographic services and algorithms, including keystores. In this section, we analyze the most common Java software keystores implemented in the Oracle JDK and in a widespread cryptographic library called Bouncy Castle that ships with a provider compatible with the JCA. In particular, since the documentation was not sufficient to assess the design and cryptographic strength of the keystores, we performed a comprehensive review of the source code exposing, for the first time, implementation details such as on-disk file structure and encoding, standard and proprietary cryptographic mechanisms, default and hard-coded parameters.

For reader convenience, we provide a brief summary of the cryptographic mechanisms and acronyms used in this section: Password-Based Encryption (PBE) is an encryption scheme in which the cryptographic key is derived from a password through a Key Derivation Function (KDF); a Message Authentication Code (MAC) authenticates data through a secret key and HMAC is a standard construction for MAC which is based on cryptographic hash functions; Cipher Block Chaining (CBC) and Counter with CBC-MAC (CCM) are two standard modes of operation for block ciphers, the latter is designed to provide both authenticity and confidentiality.

A. Oracle Keystores

The Oracle JDK offers three keystore implementations, namely JKS, JCEKS and PKCS12, which are respectively made available through the providers SUN, SunJCE and SunJSSE [42]. While JKS and JCEKS rely on proprietary algorithms to enforce both the confidentiality and the integrity of the saved entries, PKCS12 relies on open standard format and algorithms as defined in [47].

JKS: Java KeyStore (JKS) is the first official implementation of a keystore that appeared in Java since the release of JDK 1.2. To the time, it is still the *default* keystore in Java 8 when no explicit choice is made. It supports encrypted private key entries and public key certificates stored in the clear. The file format consists of a header containing the magic file number, the keystore version and the number of entries, which is followed by the list of entries. The last part of the file is a digest used to check the integrity of the keystore. Each entry contains the type of the object (key or certificate) and the label, followed by the cryptographic data.

Private keys are encrypted using a custom stream cipher designed by Sun, as reported in the OpenJDK source code. In order to encrypt data, a keystream W is generated in

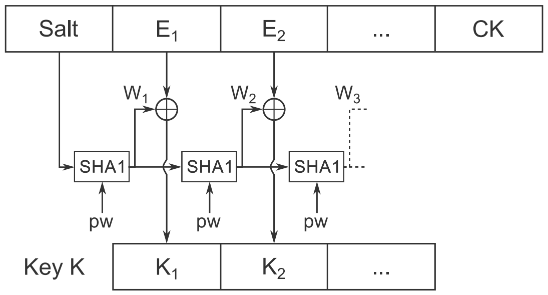


Fig. 1: Decryption in the custom stream cipher used by JKS.

20-bytes blocks with W_0 being a random salt and $W_i = \mathit{SHA1}(\mathit{password}||W_{i-1})$. The encrypted key E is computed as the XOR of the private key K with the keystream W, hence K and E share the same length. The ciphertext is then prepended with the salt and appended with the checksum $CK = \mathit{SHA1}(\mathit{password}||K)$. The block diagram for decryption is shown in Figure 1.

The integrity of the keystore is achieved through a custom hash-based mechanism: JKS computes the SHA1 hash of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and the keystore content. The result is then checked against the 20 bytes digest at the end of the keystore file.

JCEKS: Java Cryptography Extension KeyStore (JCEKS) has been introduced after the release of JDK 1.2 in the external Java Cryptography Extension (JCE) package and merged later into the standard JDK distribution from version 1.4. According to the Java documentation, it is an alternate proprietary keystore format to JKS "that uses much stronger encryption in the form of Password-Based Encryption with Triple-DES" [4]. Besides the improved PBE mechanism, it allows for storing also symmetric keys.

The file format is almost the same of JKS with a different magic number in the file header and support for the symmetric key type. The integrity mechanism is also borrowed from JKS.

JCEKS stores certificates as plaintext, while the PBE used to encrypt private keys, inspired by PBES1 [40], is based on 20 MD5 iterations and a 64 bits salt. Given that Triple-DES is used to perform the encryption step, the key derivation process must be adapted to produce cipher parameters of the adequate size. In particular, JCEKS splits the salt in two halves and applies the key derivation process for each of them. The first 192 bits of the combined 256 bits result are used as the Triple-DES key, while the remaining 64 bits are the initialization vector.

PKCS12: The PKCS12 keystore supports both private keys and certificates, with support for secret keys added in Java 8. Starting from Java 9, Oracle replaced JKS with PKCS12 as the default keystore type [7].

The keystore file is encoded as an ASN.1 structure according to the specification given in [47]. It contains the version number of the keystore, the list of keys and the certificates. The last part of the keystore contains an HMAC (together with the parameters for its computation) used to check the integrity of the entire keystore by means of a password.

The key derivation process, used for both confidentiality and integrity, is implemented as described in the PKCS#12 standard [47] using SHA1 as hashing function, 1024 iterations and a 160 bit salt. Private keys and secret keys (when supported) are encrypted using Triple-DES in CBC mode. Certificates are encrypted as well in a single encrypted blob, using the RC2 cipher in CBC mode with a 40-bit key. While each key can be encrypted with a different password, all the certificates are encrypted reusing the store password.

B. Bouncy Castle Keystores

Bouncy Castle is a widely used open-source crypto API. As of 2014, it provides the base implementation for the crypto library used in the Android operating system [24]. It supports four different keystore types via the BC provider: BKS, UBER, BCPKCS12 and the new FIPS-compliant BCFKS. Similarly to the Oracle keystores, all the BC keystores rely on passwords to enforce confidentiality over the entries and to verify the integrity of the keystore file.

BKS: The Bouncy Castle Keystore (BKS) allows to store public/private keys, symmetric keys and certificates. The BKS keystore relies on a custom file structure to store the entries. The file contains the version number of the BKS keystore, the list of stored cryptographic entries and an HMAC, along with its parameters, computed over the entries as integrity check.

Only symmetric and private keys can be encrypted in BKS, with Triple-DES in CBC mode. The key derivation schema is taken from PKCS#12 v1.0, using SHA1 as hashing function, a random number of iterations between 1024 and 2047 which is stored for each entry and a 160 bit salt.

The integrity of the keystore is provided by an HMAC using the same key derivation scheme used for encryption and applied to the integrity password. For backward compatibility, the current version of BKS still allows to load objects encrypted under a buggy PBE mechanism used in previous versions of the keystore[1]. If the key is recovered using an old mechanisms, it is immediately re-encrypted with the newer PBE scheme.

UBER: UBER shares most of its codebase with BKS, thus it supports the same types of entries and PBE. Additionally, it provides an extra layer of encryption for the entire keystore file, which means that all metadata around the keys and certificates are encrypted as well. The PBE mechanism used for encrypting the file is Twofish in CBC mode with a key size of 256 bits. The KDF is PKCS#12 v1.0 with SHA1 using a 160 bits salt and a random number of iterations in the range 1024 and 2047.

The integrity of the keystore is checked after successful decryption using the store password. The plaintext consists of the keystore entries followed by their SHA1 checksum. UBER recomputes the hash of the keystore and compares it with the stored digest.

[1]https://github.com/bcgit/bc-java/blob/master/prov/src/main/java/org/bouncycastle/jce/provider/BrokenPBE.java

BCFKS: BCFKS is a new FIPS-compliant [53] keystore introduced in the version 1.56 of Bouncy Castle[2] offering similar features to UBER. This keystore provides support for secret keys in addition to asymmetric keys and certificates.

The entire keystore contents is encrypted using AES in CCM mode with a 256 bits key, so to provide protection against introspection. After the encrypted blob, the file contains a block with a HMAC-SHA512 computed over the encrypted contents to ensure the keystore integrity. The store password is used to derive the two keys for encryption and integrity.

All key derivation operations use PBKDF2 with HMAC-SHA512 as pseudorandom function, 512 bits of salt and 1024 iterations. Each key entry is separately encrypted with a different password using the same algorithm for the keystore confidentiality, while this possibility is not offered for certificates.

BCPKCS12: The BCPKCS12 keystore aims to provide a PKCS#12-compatible implementation. It shares the same algorithms and default parameters for key derivation, cryptographic schemes and file structure of the Oracle JDK version detailed in Section IV-A. Compared to Oracle, the Bouncy Castle implementation lacks support for symmetric keys and the possibility to protect keys with different passwords, since all the entries and certificates are encrypted under the store password. The BC provider also offers a variant of the PKCS#12 keystore that allows to encrypt certificates using the same PBE of private keys, that is Triple-DES in CBC mode.

C. Keystores Adoption

We have analyzed 300 Java projects supporting keystores that are hosted on Github to estimate the usage of the implementations examined in this paper. Applications range from amateur software to well-established libraries developed by Google, Apache and Eclipse.

We searched for occurrences of known patterns used to instantiate keystores in the code of each project. We have found that JKS is the most widespread keystore with over 70% of the applications supporting it. PKCS12 is used in 32% of the analyzed repositories, while JCEKS adoption is close to 10%. The Bouncy Castle keystores UBER and BCPKCS12 are used only in 3% of the projects, while BKS can be found in about 6% of the examined software. Finally, since BCFKS is a recent addition to the Bouncy Castle library, none of the repositories is supporting it.

D. Summary

In Table I we summarize the features and the algorithms (rows) offered by the keystore implementations (columns) analyzed in this section. Table Ia does not contain the row "Store Encryption" since none of the JDK keystores provides protection against introspection.

To exemplify, by reading Table Ia we understand that the JCEKS keystore of the SunJCE provider relies on a custom PBE mechanism based on MD5 using only 20 iterations to derive the Triple-DES key for the encryption of keys. The ✓ mark shows that the keystore supports secret keys, while ✗ denotes that certificates cannot be encrypted.

[2]https://github.com/bcgit/bc-java/commit/80fd6825

TABLE I: Summary of the keystores.

(a) Oracle JDK 8u144 and below.

		JKS	JCEKS	PKCS12
Provider		Sun	SunJCE	SunJSSE
Support for secret keys		✗	✓	✓*
Keys PBE	KDF	Custom (SHA1)	Custom (MD5)	PKCS12 (SHA1)
	Salt	160b	64b	160b
	Iterations	-	20	1024
	Cipher	Stream cipher	3DES (CBC)	3DES (CBC)
	Key size	-	192b	192b
Certificates PBE	KDF	✗	✗	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Cipher			RC2 (CBC)
	Key size			40b
Store Integrity	KDF	SHA1 with password	SHA1 with password	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024
	Mechanism			HMAC (SHA1)

* since Java 8

(b) Bouncy Castle 1.57 and below.

		BKS	UBER	BCFKS	BCPKCS12
Provider		Bouncy Castle	Bouncy Castle	Bouncy Castle	Bouncy Castle
Support for secret keys		✓	✓	✓	✗
Keys PBE	KDF	PKCS12 (SHA1)	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b	160b	512b	160b
	Iterations	1024–2047	1024–2047	1024	1024
	Cipher	3DES (CBC)	3DES (CBC)	AES (CCM)	3DES (CBC)
	Key size	192b	192b	256b	192b
Certificates PBE	KDF	✗	✗	✗	PKCS12 (SHA1)
	Salt				160b
	Iterations				1024
	Cipher				RC2 / 3DES (CBC)
	Key size				40b / 192b
Store Encryption	KDF	✗	PKCS12 (SHA1)	PBKDF2 (HMAC-SHA512)	✗
	Salt		160b	512b	
	Iterations		1024–2047	1024	
	Cipher		Twofish (CBC)	AES (CCM)	
	Key size		256b	256b	
Store Integrity	KDF	PKCS12 (SHA1)	SHA1 after decrypt	PBKDF2 (HMAC-SHA512)	PKCS12 (SHA1)
	Salt	160b		512b	160b
	Iterations	1024–2047		1024	1024
	Mechanism	HMAC (SHA1)		HMAC (SHA512)	HMAC (SHA1)

V. Attacks

In the previous section, we have shown that the analyzed keystores use very diverse key derivation functions and cryptographic mechanisms and, in several cases, they do not adhere to standards or use obsolete and ad-hoc mechanisms. We now discuss how this weakens the overall security of the keystore and enables or facilitates attacks. In particular, we show that keystores using weak or ad-hoc implementations for password-based encryption or integrity checks open the way to password brute-forcing. During the in-depth analysis of keystores, we have also found security flaws that can be exploited in practice to mount denial of service and code execution attacks.

Attacks in this section are organized according to the security properties violated, as defined in Section III-A. For each attack we provide a detailed description discussing the attacker settings and the rules that are not followed by the keystore implementation (*cf.* Section III-B). We conclude with some general security considerations that are not specific to any particular attack.

Table II provides a high-level overview of the properties which are guaranteed by the analyzed keystores with respect to the attacks presented in this section. We consider versions of Oracle JDK and Bouncy Castle before and after disclosing our findings to the developers. Specifically, we refer to JDK 8u144 and 8u152 for Oracle, while version 1.57 of Bouncy Castle is compared against the development repository as of November 28, 2017.[3] We use the symbol → to point out improvements in newer versions. Details of the changes are listed in Section VII. The ✓✓ symbol denotes that a property is satisfied by the keystore under any attacker setting and the implementation adhere to the relevant design rules listed in Section III-B. We

[3]https://github.com/bcgit/bc-java/tree/8ed589d

Algorithm 1 JKS 1-block Crack

```
1:  procedure JKS_1BLOCKCRACK($Salt, E_{1..n}, CK$)
2:      $known\_plaintext \leftarrow$ 0x30 $\|$ $length(E)$
3:      $test\_bytes \leftarrow known\_plaintext \oplus E_1$
4:      for $password$ in passwords do
5:          $W_1 \leftarrow$ SHA1($password \| Salt$)
6:          if $W_1 = test\_bytes$ then
7:              $K \leftarrow$ DECRYPT($Salt, E, password$)
8:              $checksum \leftarrow$ SHA1($password \| K$)
9:              if $CK = checksum$ then
10:                 return $password$
```

use ✓ when no clear attack can be mounted but design rules are not completely satisfied, *e.g.* a legacy cipher like Triple-DES is used. The ✗ symbol indicates that the property is broken under the standard attacker model. When a property is broken only under a specific setting **Sx**, we report it in the table as ✗$_{\mathbf{Sx}}$. If a more powerful attack is enabled by additional settings, we clarify in the footnotes.

As an example, consider the system integrity property (**P3**) in the JCEKS keystore: up to JDK 8u144 included, write capabilities (**S1**) allow to DoS the application loading the keystore; when integrity and key confidentiality passwords are known (**S2** and **S3**), the attacker can also achieve arbitrary code execution on the system (*cf.* note 3 in the table). The rightmost side of the arrow indicates that JDK 8u152 does not include mitigations against the code execution attack.

A. Attacks on Entries Confidentiality (P1)

JKS Password Cracking: The custom PBE mechanism described in Section IV-A for the encryption of private keys is extremely weak. The scheme requires only one SHA1 hash and a single XOR operation to decrypt each block of the encrypted entry resulting in a clear violation of rule **R1**. Since there is no mechanism to increase the amount of computation needed to derive the key from the password, also rule **R2** is neglected.

Despite the poor cryptographic scheme, each attempt of a brute-force password recovery attack would require to apply SHA1 several times to derive the whole keystream used to decrypt the private key. As outlined in Figure 1, a successful decryption is verified by matching the last block (CK) of the protected entry with the hash of the password concatenated with the decrypted key. For instance, a single password attempt to decrypt a 2048 bit RSA private key entry requires over 60 SHA1 operations.

We found that such password recovery attack can be greatly improved by exploiting the partial knowledge over the plaintext of the key. Indeed, the ASN.1 structure of a key entry enables to efficiently test each password with a single SHA1 operation. In JKS, private keys are serialized as DER-encoded ASN.1 objects, along the PKCS#1 standard [39]. For instance, an encoded RSA key is stored as a sequence of bytes starting with byte `0x30` which represent the ASN.1 type `SEQUENCE` and a number of bytes representing the length of the encoded key. Since the size of the encrypted key is the same as the size of the plaintext, these bytes are known to the attacker. On average, given n bytes of the plaintext it is necessary

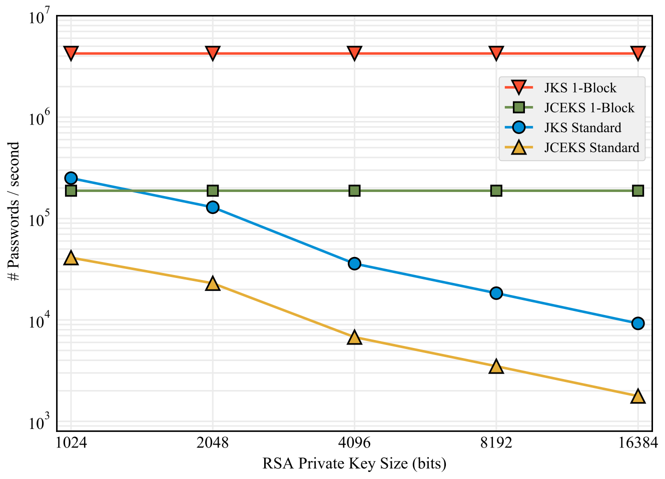


Fig. 2: Performance comparison of password cracking for private RSA keys on JKS and JCEKS using both the standard and the improved 1-block method on a Intel Core i7 6700 CPU.

to continue decryption beyond the first block only for one password every 256^n attempts.

The pseudocode of the attack is provided in Algorithm 1, using the same notation introduced in Section IV-A. We assume that the algorithm is initialized with the salt, all the blocks of the encrypted key and the checksum. The XOR operation between the known plaintext and the first encrypted block (line 3) is performed only once for all the possible passwords. As a halt condition, the result is then compared against the digest of the salt concatenated to the tested password (lines 5-6). To further verify the correctness of the password, a standard decrypt is performed.

A comparison between the standard cracking attack and our improved version is depicted in Figure 2. From the chart it is possible to see that the cost of the single block attack (referred to as 1-block) is independent from the size of the encrypted entry, while the number of operations required to carry out the standard attack is bound to the size of the DER-encoded key. As an example, for a 4096 bit private RSA key, the 1-block approach is two orders of magnitude faster than the standard one.

Based on our findings, that we previously mentioned in a blog post [25], this attack has been recently integrated into Hashcat 3.6.0[4] achieving a speed of 8 billion password tries/sec with a single NVIDIA GTX 1080 GPU.

JCEKS Password Cracking: The PBE mechanism discussed in Section IV-A uses a custom KDF that performs 20 MD5 iterations to derive the encryption key used in the Triple-DES cipher. This value is three orders of magnitude lower than the iteration count suggested in [28], thus violating both rules **R1** and **R2**. Given that keys are DER-encoded as well, it is possible to speed up a brute-force attack using a technique similar to the one discussed for JKS. Figure 2 relates the standard cracking speed to the single block version. Notice

[4]https://hashcat.net/forum/thread-6630.html

TABLE II: Properties guaranteed by keystores with respect to attacks, before and after updates listed in Section VII.

	JKS	JCEKS	PKCS12	BKS	UBER	BCFKS	BCPKCS12
(**P1**) Entries confidentiality	✗	✗ → ✓	✓[1]	✓	✓	✓ → ✓✓	✓[1]
(**P2**) Keystore integrity	✗[2]	✗[2]	✓ → ✓✓	✓	✓	✓ → ✓✓	✓ → ✓✓
(**P3**) System integrity	✓✓	$✗^{3}_{\textbf{S1}}$ → $✗_{\textbf{S1-3}}$	$✗_{\textbf{S1}}$ → ✓✓	$✗_{\textbf{S1}}$	✓✓	✓✓	$✗_{\textbf{S1}}$ → ✓✓

Legend:

✓✓	property is always satisfied
✓	no clear attacks but rules not completely satified
✗	property is broken in the standard attacker model
$✗_{\textbf{Sx}}$	property is broken under a attacker setting **Sx**

[1] only confidentiality of certificates can be violated
[2] under additional settings **S1** or **S4** it might be possible to use rainbow tables
[3] under additional settings **S2** and **S3** it is possible to achieve arbitrary code execution on JDK $\leq$ 8u152

that the cost of a password-recovery attack is one order of magnitude higher than JKS in both variants due to the MD5 iterations required by the custom KDF of JCEKS.

PKCS#12 Certificate Key Cracking: Oracle PKCS12 and BCPKCS12 keystores allow for the encryption of certificates. The PBE is based on the KDF defined in the PKCS#12 standard paired with the legacy RC2 cipher in CBC mode with a 40 bit key, resulting in a clear violation of rule **R1**. Due to the reduced key space, the protection offered by the KDF against offline attacks can be voided by directly brute-forcing the cryptographic key. Our serialized tests, performed using only one core of an Intel Core i7 6700 CPU, show that the brute-force performance is $8{,}300\,passwords/s$ for password testing (consisting of a KDF and decryption run), while the key cracking speed is $1{,}400{,}000\,keys/s$. The worst-case scenario that requires the whole 40-bits key space to be exhausted, requires about 9 days of computation on our system. This time can be reduced to about 1 day by using all eight cores of our processor. We estimate that a modern high-end GPU should be able to perform this task in less than one hour.

Notice, however, that although finding the key so easily makes the encryption of certificates pointless, an attacker cannot use the key value to reduce the complexity of cracking the integrity password since the random salt used by the KDF makes it infeasible to precompute the mapping from passwords to keys.

B. Attacks on Keystore Integrity (*P2*)

JKS/JCEKS Integrity Password Cracking: The store integrity mechanism used by both JKS and JCEKS (*cf.* Section IV-A) only relies on the SHA1 hash digest of the integrity password, concatenated with the constant string "`Mighty Aphrodite`" and with the keystore data. In contrast with rule **R1**, this technique based on a single application of SHA1 enables to efficiently perform brute-force attacks against the integrity password. Section VI reports on the computational effort required to attack the integrity mechanism for different sizes of the keystore file.

Additionally, since SHA1 is based on the Merkle-Damgård construction, this custom approach is potentially vulnerable to extension attacks [26]. For instance, it may be possible for an attacker with write access to the keystore (**S1**) to remove the original digest at the end of the file, extend the keystore content with a forged entry and recompute a valid hash without knowing the keystore password. Fortunately, this specific attack is prevented in JKS and JCEKS since the file format stores the number of entries in the keystore header.

JKS/JCEKS Integrity Digest Precomputation: The aforementioned construction to ensure the integrity of the keystore suffers from an additional problem. Assume the attacker has access to an empty keystore, for example when an old copy of the keystore file is available under a file versioning storage (**S4**). Alternatively, as special case of **S1**, the attacker may be able to read the file, but the interaction with the keystore is mediated by an application that allows to remove entries without disclosing the store password. This file consists only of a fixed header followed by the SHA1 digest computed using the password, the string "`Mighty Aphrodite`" and the header itself. Given that there is no random salting in the digest computation, it would be possible to mount a very efficient attack to recover the integrity password by exploiting precomputed hash chains, as done in rainbow tables [41].

C. Attacks on System Integrity (*P3*)

JCEKS Code Execution: A secret key entry is stored in a JCEKS keystore as a Java object having type `SecretKey`. First, the key object is serialized and wrapped into a `SealedObject` instance in an encrypted form; next, this object is serialized again and saved into the keystore.

When the keystore is loaded, all the serialized Java objects stored as secret key entries are evaluated. An attacker with write capabilities (**S1**) may construct a malicious entry containing a Java object that, when deserialized, allows her to execute arbitrary code in the application context. Interestingly, the attack is not prevented by the integrity check since keystore integrity is verified only after parsing all the entries.

The vulnerable code can be found in the `engineLoad` method of the class `JceKeyStore` implemented by the SunJCE provider.[5] In particular, the deserialization is performed on lines 837-838 as follows:

```
// read the sealed key
try {
    ois = new ObjectInputStream(dis);
    entry.sealedKey =
        (SealedObject) ois.readObject();
    ...
```

Notice that the cast does not prevent the attack since it is performed after the object evaluation.

To stress the impact of this vulnerability, we provide three different attack scenarios: i) the keystore is accessed by

[5]http://hg.openjdk.java.net/jdk8u/jdk8u/jdk/file/5534221c23fc/src/share/classes/com/sun/crypto/provider/JceKeyStore.java

multiple users over a shared storage. An attacker can replace or add a single entry of the keystore embedding the malicious payload, possibly gaining control of multiple hosts; ii) a remote application could allow its users to upload keystores for cryptographic purposes, such as importing certificates or configuring SSL/TLS. A crafted keystore loaded by the attacker may compromise the remote system; iii) an attacker may even forge a malicious keystore and massively spread it like a malware using email attachments or instant messaging platforms. Users with a default application associated to the keystore file extension (*e.g.*, keystore inspection utilities such as KSE[6]) have a high probability of being infected just by double clicking on the received keystore. Interestingly, all the malicious keystores generated during our tests did not raise any alert on antivirus tools completing a successful scan by *virustotal.com*.

We checked the presence of the vulnerability from Java 6 onwards. We were able to achieve arbitrary command execution on the host with JDK $\leq$ 7u21 and JDK $\leq$ 8u20 by forging a payload with the tool `ysoserial`.[7] Newer versions are still affected by the vulnerability, but the JDK classes exploited to achieve code execution have been patched. Since the deserialization occurs within a Java core class, the classpath is restricted to bootstrap and standard library classes. However, by embedding a recursive object graph in a JCEKS entry, an attacker can still hang the deserialization routine consuming CPU indefinitely and thus causing a DoS in the target machine. We were able to mount this attack on any version of the Oracle JDK $\leq$ 8u144.

The implementation choice for storing secret keys in JCEKS is a clear violation of Rule **R3**, since these entities are essentially stored as Java code. The correct approach is to adopt standard formats and encodings, such as the PKCS#8 format used in the PKCS12 keystore.

JCEKS Code Execution After Decryption: When the attacker knows the integrity password and the confidentiality password of a secret key entry (**S2**, **S3**) in addition to **S1**, the previous attack can be further improved to achieve arbitrary command execution even on the latest, at the time of writing, Java 8 release (8u152). This variant of the attack assumes that the application loading the JCEKS keystore makes use of one of the widespread third-party libraries supported by `ysoserial`, such as *Apache Commons Collections* or the *Spring* framework: such libraries have been found [3] to contain vulnerable gadget chains that can be exploited by the malicious payload.

When a `SealedObject` wrapping a secret key is successfully loaded and decrypted, an additional deserialization call is performed over the decrypted content. The `SealedObject` class extends the classpath to allow the deserialization of any class available in the application scope, including third-party libraries. By exploiting this second deserialization step, an attacker may construct more powerful payloads to achieve command execution.

The exploitation scenarios are similar to the ones already discussed in the previous variant of the attack. Additionally, we point out that even an antivirus trained to detect deserialization signatures would not be able to identify the malicious content since the payload is stored in encrypted form in the keystore.

DoS by Integrity Parameters Abuse: Many keystores rely on a keyed MAC function to ensure the integrity of their contents. The parameters of the KDF used to derive the key from the store password are saved inside the file. Thus, an attacker with write capabilities (**S1**) may tamper with the KDF parameters to affect the key derivation phase that is performed before assessing the integrity of the keystore file. In particular, the attacker may set the iteration count to an unreasonably high value in order to perform a DoS attack on applications loading the keystore.

We found that Oracle PKCS12, BKS and BCPKCS12 implementations are affected by this problem. Starting from valid keystore files, we managed to set the iteration count value to $2^{31}-1$. Loading such keystores required around 15 minutes at full CPU usage on a modern computer. According to [52] the iteration count should not impact too heavily on the user-perceived performance, thus we argue that this is a violation of Rule **R2**.

D. Bad Design Practices

During our analysis we found that some of the keystores suffered from bad design decisions and implementation issues that, despite not leading to proper attacks, could lead to serious security consequences.

Our review of the Oracle PKCS12 keystore code showed that the KDF parameters are not treated uniformly among MAC, keys and certificates. During a store operation, the Oracle implementation does not preserve the original iteration count and salt size for MAC and certificates that has been found at load time in the input keystore file. Indeed, iteration count and salt size are silently set to the hard-coded values of 1024 and 20 byte, respectively. Since this keystore format is meant to be interoperable, this practice could have security consequences when dealing with keystores generated by third-party tools. For instance, PKCS12-compatible keystores generated by OpenSSL default to 2048 iterations: writing out such keystore with the Oracle JDK results in halving the cost of a password recovery attack.

The Bouncy Castle BCPKCS12 implementation suffers a similar problem: in addition to MAC and certificate parameters, also the iteration count and the salt size used for private keys are reverted to default values when the keystore is saved to disk. Following our report to the Bouncy Castle developers, this behaviour is currently being addressed in the next release by preserving the original parameters whenever possible.[8]

Lastly, the construction of the integrity mechanism for the UBER keystore could cause an information leakage under specific circumstances. After a successful decryption using the store password, UBER recomputes the hash of the keystore and compares it with the stored digest. This MAC-then-encrypt approach is generally considered a bad idea, since it can lead to attacks if, for example, there is a perceptible difference in behavior (an error message, or execution time) between

[6]http://keystore-explorer.org

[7]https://github.com/frohoff/ysoserial

[8]https://github.com/bcgit/bc-java/commit/ebe1b25a

a decryption that fails because the padding is invalid, or a decryption that fails because the hash is invalid (a so-called padding oracle attack [54]).

E. Security Considerations

We now provide general considerations on the security of Java keystores. The first one is about using the same password for different purposes. If the integrity password is also used to ensure the confidentiality of encrypted entries, then the complexity of breaking either the integrity or the confidentiality of stored entries turns out to be the one of attacking the weakest mechanism. For instance, we consider a keystore where cracking the integrity password is more efficient than recovering the password used to protect sensitive entries: as shown in Section VI, this is the case of PKCS12 and BCPKCS12 keystores. Under this setting, sensitive keys can be leaked more easily by brute-forcing the integrity password.

Although this is considered a bad practice in general [33], all the keystores analyzed permit the use of the same password to protect sensitive entries and to verify the integrity of the keystore. This practice is indeed widespread [27] and, as already stated in Section III-C, prompted by `keytool` itself. Furthermore, our analysis found that the BCPKCS12 keystore forcibly encrypts keys and certificates with the store password. For these reasons, we argue that using the same password for integrity and confidentiality is not a direct threat to the security of stored keys when both mechanisms are resistant to offline attacks and a strong password is used. Still the security implications of this practice should be seriously considered.

The second consideration regards how the integrity of a keystore is assessed. Indeed, a poorly designed application may bypass the integrity check on keystores by providing a null or empty password to the Java `load()` function. All the Oracle keystores analyzed in the previous section and BouncyCastle BKS are affected by this problem. On the other hand, keystores providing protection to entries inspection, such as UBER and BCFKS, cannot be loaded with an empty password since the decryption step would fail. Lastly, BCPKCS12 throws an exception if an attempt of loading a file with an empty password is made. Clearly, if the integrity check is omitted, an attacker can trivially violate Property **P2** by altering, adding or removing any entry saved in the clear. Conversely, the integrity of encrypted sensitive keys is still provided by the decryption mechanism that checks for the correct padding sequence at the end of the plaintext. Since the entries are typically encoded (*e.g.*, in ASN.1), a failure in the parse routine could also indicate a tampered ciphertext.

We also emphasize that the 1-block cracking optimization introduced in V-A is not limited to JKS and JCEKS. Indeed, by leveraging the structure of saved entries, all the analyzed keystores enable to reduce the cost of the decrypt operation to check the correctness of a password. However, excluding JKS and JCEKS, this technique only provides a negligible speed-up on the remaining keystores given that the KDF is orders of magnitude slower than the decrypt operation.

Finally, we point out that the current design of password-based keystores cannot provide a proper key-revocation mechanism without a trusted third-party component. For instance, it may be the case that a key has been leaked in the clear and subsequently substituted with a fresh one in newer versions of a keystore file. Under settings **S1** and **S4**, an attacker may replace the current version of a keystore with a previously intercepted valid version, thus restoring the exposed key. The integrity mechanism is indeed not sufficient to distinguish among different versions of a keystore protected with the same store password. For this reason, the store password must be updated to a fresh one every time a rollback of the keystore file is not acceptable by the user, which is typically the case of a keystore containing a revoked key.

VI. Estimating Brute-Force Speed-Up

We have discussed how weak PBEs and integrity checks in keystores can expose passwords to brute-forcing. In this section we make an empirical comparison of the cracking speed to bruteforce both the confidentiality and integrity mechanisms in the analyzed keystores. We also compute the speed-up with respect to BCFKS, as it is the only keystore using a standard and modern KDF, *i.e.*, PBKDF2, which provides the best brute-forcing resistance. Notice, however, that the latest NIST draft on Digital Identity Guidelines [28] sets the minimum KDF iteration count to 10,000 which is one order of magnitude more than what is used in BCFKS (*cf.* Table I). Thus all the speed-up values should be roughly multiplied by 10 if compared with a baseline implementation using PBKDF2 with 10,000 iterations.

It is out of the scope of this paper to investigate brute-forcing strategies. Our tests only aim at comparing, among the different keystores, the actual time to perform the key derivation step and the subsequent cryptographic operations, including the check to assess key correctness. Our study is independent of the actual password guessing strategy adopted by the attacker.

A. Test Methodology

We developed a compatible `C` implementation of the key decryption and the integrity check for each keystore type. Each implementation is limited to the minimum steps required to check the correctness of a test password. This procedure is then executed in a timed loop to evaluate the cracking speed. Algorithms 2 and 3 show the pseudocode of our implementations. Note that, in both algorithms, we set the password length to 10 bytes because it is an intermediate value between trivial and infeasible. Similarly, since the iteration count in BKS and UBER is chosen randomly in the range 1024 and 2047, we set it to the intermediate value 1536.

Confidentiality: The confidentiality password brute-forcing loop (Algorithm 2) is divided into three steps: key derivation, decryption and a password correctness check. The last step is included in the loop only to account for its computational cost in the results. Both PBES1 (PKCS#5) and PKCS#12 password-based encryption schemes, used in all keystores but BCFKS, require to run the KDF twice to derive the decryption key and the IV. On the other hand, in BCFKS the initialization vector is not derived from the password but simply stored with the ciphertext. During our tests we set *encrypted_entry* to a fixed size to resemble an on-disk entry containing a 2048 bits RSA key. However, in Section V-A we have shown how the partial knowledge of the plaintext structure of a JKS key entry can

Algorithm 2 Confidentiality password cracking benchmark

```
1: procedure BenchConfidentiality(test_duration)
2:     $encrypted\_entry \leftarrow (B_1, ..., B_{2000})$
3:     $passwords \leftarrow (pw_1, ..., pw_n)$                ▷ all 10-bytes passwords
4:     $salt \leftarrow$ constant
5:     $counter \leftarrow 0$
6:     while ElapsedTime $<$ test_duration do
7:         $password \leftarrow$ next$(passwords)$
8:         key $\leftarrow$ KDF$_{key}$(password, salt)
9:         iv $\leftarrow$ KDF$_{iv}$(password, salt)          ▷ not in JKS, BCFKS
10:        plaintext $\leftarrow$ DecryptBlock(encrypted_entry, key, iv)
11:        VerifyKey(plaintext)
12:        counter $\leftarrow$ counter $+ 1$
13:    return $counter$
```

Algorithm 3 Integrity password cracking benchmark

```
1: procedure BenchIntegrity(test_duration)
2:     keystore_content$_{small} \leftarrow (B_1, ..., B_{2048})$
3:     keystore_content$_{medium} \leftarrow (B_1, ..., B_{8192})$
4:     keystore_content$_{large} \leftarrow (B_1, ..., B_{16384})$
5:     $passwords \leftarrow (pw_1, ..., pw_n)$                ▷ all 10-bytes passwords
6:     $salt \leftarrow$ constant
7:     $counter_{(small,medium,large)} \leftarrow 0$
8:     for all $keystore\_content, counter$ do
9:         while ElapsedTime $<$ test_duration do
10:            $password \leftarrow$ next$(passwords)$
11:            key $\leftarrow$ KDF$_{mac}$(password, salt)   ▷ not in JKS, JCEKS
12:            mac $\leftarrow$ MAC(keystore_content, key)
13:            VerifyMAC(mac)
14:            counter $\leftarrow$ counter $+ 1$
15:    return $counter_{(small,medium,large)}$
```

be leveraged to speed-up brute-forcing. This shortcut can be applied to all the analyzed keystores in order to decrypt only the first block of *encrypted_entry*. For this reason, the key size becomes irrelevant while testing for a decryption password.

Integrity: Similarly, the integrity password cracking code (Algorithm 3) is divided into three steps: key derivation, a hash/MAC computation and the password correctness check. The key derivation step is run once to derive the MAC key in all keystores, with the exception of JKS and JCEKS where the password is fed directly to the hash function (*cf.* Section IV-A). As described later in this section, the speed of KDF plus MAC calculation can be highly influenced by the keystore size, thus we performed our tests using a *keystore_content* of three different sizes: 2048, 8192 and 16384 bytes.

Test configuration: We relied on standard implementations of the cryptographic algorithms to produce comparable results: the OpenSSL library (version 1.0.2g) provides all the needed hash functions, ciphers and KDFs, with the exception of Twofish where we used an implementation from the author of the cipher.[9] All the tests were performed on a desktop computer running Ubuntu 16.04 and equipped with an Intel Core i7 6700 CPU; source code of our implementations has been compiled with GCC 5.4 using `-O3 -march=native` optimizations. We run each benchmark on a single CPU core because the numeric results can be easily scaled to a highly parallel systems. To collect solid and repeatable results each benchmark has been run for 60 seconds.

[9]https://www.schneier.com/academic/twofish/download.html

B. Results

The charts in Figure 3 show our benchmarks on the cracking speed for confidentiality (Figure 4a) and integrity (Figure 4b). On the x-axis there are the 7 keystore types: we group together different keystores when the specific mechanism is shared among the implementations, *i.e.*, PKCS12/BCPKCS12 for both confidentiality and integrity and JKS/JCEKS for integrity. On the y-axis we report the number of tested passwords per second doing a serial computation on a single CPU core: note that the scale of this axis is logarithmic. We stress that our results are meant to provide a relative, inter-keystore comparison rather than an absolute performance index. To this end, a label on top of each bar indicates the speed-up relative to the strongest BCFKS baseline. Absolute performance can be greatly improved using both optimized parallel code and more powerful hardware which ranges from dozens of CPU cores or GPUs to programmable devices such as FPGA or custom-designed ASICs [30], [22], [34].

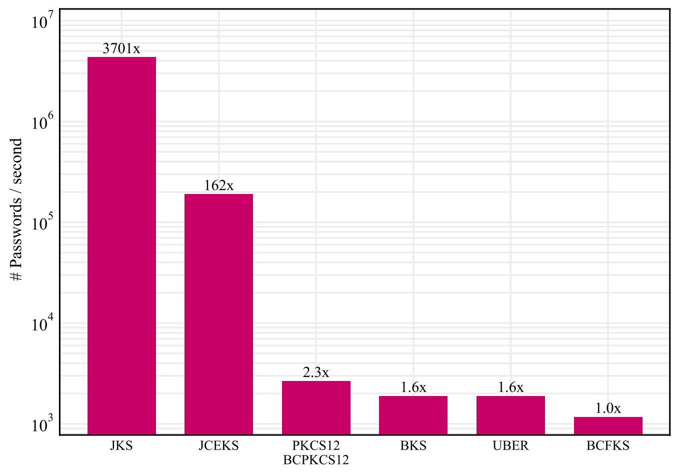
Confidentiality: From the attack described in Section V-A, it follows that cracking the password of an encrypted key contained in JKS - the default Java keystore - is at least three orders of magnitude faster than in BCFKS. Even without a specific attack, recovering the same password from JCEKS is over one hundred times faster due to its low (20) iteration count. By contrast, the higher value (1024 or 1024-2047) used in PKCS12, BKS and UBER translates into a far better offline resistance as outlined in the chart.

Integrity: Similar considerations can be done for the integrity password resistance. Finding this password in all keystores but JKS is equivalent, or even faster than breaking the confidentiality password. Moreover, the performance of these keystores is influenced by the size of the file due to the particular construction of the MAC function (*cf.* Section IV-A). The speed gain (w.r.t. confidentiality) visible in PKCS12, BKS and UBER is caused by the missing IV derivation step which, basically, halves the number or KDF iterations. Interestingly, in BCFKS there is no difference between the two scores: since the whole keystore file is encrypted, we can reduce the integrity check to a successful decryption, avoiding the computation overhead of the HMAC on the entire file.

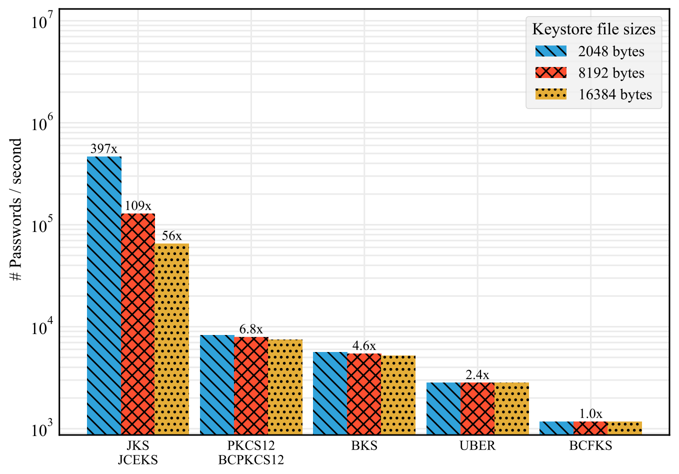
VII. Disclosure and Security Updates

We have timely disclosed our findings to Oracle and Bouncy Castle developers in May 2017. The Oracle Security Team has acknowledged the reported issues with CVE IDs [37], [38] and has released most of the fixes in the October 2017 Critical Patch Update (CPU) [44]. In the following list, we summarize the changes already published by Oracle:

- `keytool` suggests to switch to PKCS12 when JKS or JCEKS keystores are used;
- improved KDF strength of the PBE in JCEKS by raising the iteration count to 200,000. Added a ceiling value of 5 millions to prevent parameter abuse;
- in PKCS12 the iteration count has been increased to 50,000 for confidentiality and 100,000 for integrity. The same upper bound as in JCEKS is introduced;
- fixed the first JCEKS deserialization vulnerability described in Section V-C by checking that the object being deserialized is of the correct type, *i.e.*,



(a) Speed comparison of password recovery attack for key encryption (confidentiality).



(b) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

Fig. 3: Comparison of keystores password cracking speed. Bar labels indicate the speed-up to the strongest BCFKS baseline.

`SealedObjectForKeyProtector`, and by imposing a recursion limit to prevent infinite loops.

Additionally, Oracle informed us that a fix for the second JCEKS deserialization vulnerability is planned for release in the January 2018 CPU [43].

In version 1.58 of the library, Bouncy Castle developers fixed the parameter abuse vulnerability of BCPKCS12 by adding an optional Java system property that imposes an upper bound for the KDF iteration count. Moreover, they have committed in the development repository the following changes that will appear in version 1.59:

- in BCFKS, the iteration count is raised to 51,200 for both confidentiality and integrity;
- in BCPKCS12, the iteration count is increased to 51,200 and 102,400 for confidentiality and integrity, respectively.

Table II outlines the improved security guarantess offered by keystore implemenations following the fixes released by Oracle and Bouncy Castle. Additionally, in Figure 4 we show the updated results of the brute-force resistance benchmarks to reflect the improved KDF parameters. JCEKS and BCFKS now offer the best resistance to offline brute-force attacks of the confidentiality password. However, JCEKS still provides the weakest integrity mechanism. Thus, if the same password is used both for key encryption and for keystore integrity, then the increased protection level can easily be voided by attacking the latter mechanism. On the other hand, both the confidentiality and the integrity mechanisms have been updated in PKCS12. This keystore, which is now the default in Java 9, offers a much higher security level with respect to the previous release.

VIII. Conclusion

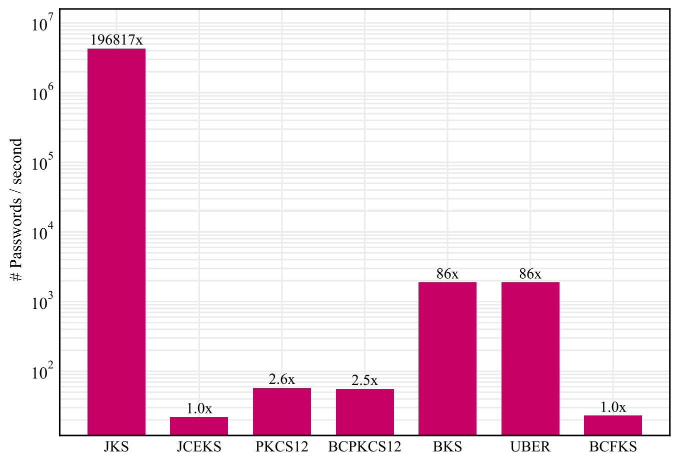
Keystores are the standard way to store and manage cryptographic keys and certificates in Java applications. In the literature there is no in-depth analysis of keystore implementations and the documentation does not provide enough information to evaluate the security level offered by each keystore. Thus, developers cannot make a reasoned and informed choice among the available alternatives.

In this paper we have thoroughly analyzed seven keystore implementations from the Oracle JDK and the Bouncy Castle library. We have described all the cryptographic mechanisms used to guarantee standard security properties on keystores, including offline attacks. We have pointed out that several implementations adopt non-standard mechanisms and we have shown how this can drastically speed-up the brute-forcing of the keystore passwords. Additionally, we reported new and unpublished attacks and defined a precise threat model under which they may occur. These attacks range from breaking the confidentiality of stored keys to arbitrary code execution on remote systems and denial of service. We also showed how a keystore can be potentially weaponized by an attacker to spread malware.

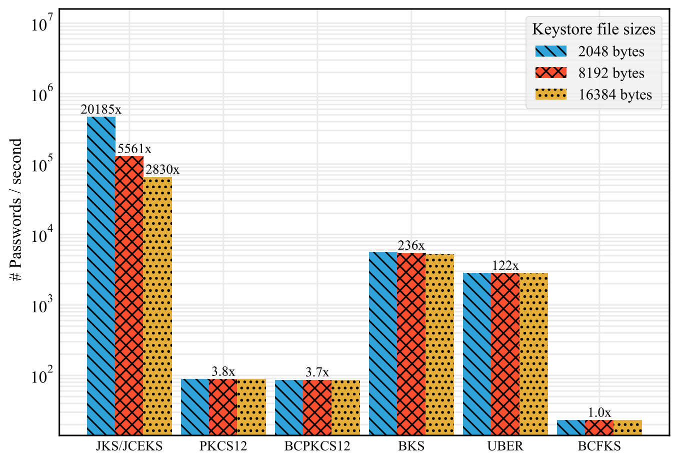
We have reported the security flaws to Oracle and Bouncy Castle. Most of the issues in the Oracle JDK have been fixed in the October 2017 Critical Patch Update [44] following CVE IDs [37], [38]. Similarly, Bouncy Castle developers committed changes to address several problems discussed in this paper.

Following our analysis and succeeding fixes, it appears evident that the security offered by JKS, the default keystore in Java 8 and previous releases, is totally inadequate. Its improved version JCEKS still uses a broken integrity mechanism. For these reasons, we favorably welcome the decision of Oracle to switch to PKCS12 as the default keystore type in the recent Java 9 release. After the previously discussed updates this keystore results quite solid, although certificate protection is bogus and key encryption relies on legacy cryptographic primitives.

Alternatives provided by Bouncy Castle have been found to be less susceptible to attacks. Among the analyzed keystores, the updated BCFKS version clearly sets the standard from a security standpoint. Indeed, this keystore relies on mod-



(a) Speed comparison of password recovery attack for key encryption (confidentiality).



(b) Speed comparison of password recovery attack for keystore integrity, considering different keystore sizes.

Fig. 4: Revised password cracking benchmarks after library updates.

ern algorithms, uses adequate cryptographic parameters and provides protection against introspection of keystore contents. Moreover, the development version of Bouncy Castle includes preliminary support for *scrypt* [45], [46] in BCFKS, a *memory-hard* function that requires significant amount of RAM. Considering the steady nature of keystore files, we argue that in addition to approved standard functions, it would be advisable to consider future-proof cryptographic primitives so to be more resistant against parallelized attacks [18], [20].

Acknowledgments

The authors would like to thank Stefano Calzavara and the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

This work has been partially supported by CINI Cybersecurity National Laboratory within the project FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks (www.filierasicura.it) funded by CISCO Systems Inc. and Leonardo SpA.

References

[1] "Android Studio User Guide: Sign Your App." [Online]. Available: https://developer.android.com/studio/publish/app-signing.html

[2] "WebLogic Integration 7.0: Configuring the Keystore." [Online]. Available: http://docs.oracle.com/cd/E13214_01/wli/docs70/b2bsecur/keystore.htm

[3] "Vulnerability Note VU#576313," 2015. [Online]. Available: https://www.kb.cert.org/vuls/id/576313

[4] "Java Cryptography Architecture (JCA) Reference Guide," 2016. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[5] "JDK 7 Security Enhancements," 2016. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/enhancements-7.html

[6] "Apache Tomcat 7 Documentation: SSL/TLS Configuration," 2017. [Online]. Available: https://tomcat.apache.org/tomcat-7.0-doc/ssl-howto.html

[7] "JDK 9 Early Access Release Notes," 2017. [Online]. Available: http://jdk.java.net/9/release-notes

[8] "Spring Crypto Utils Documentation: Keystore," 2017. [Online]. Available: http://springcryptoutils.com/keystore.html

[9] "Update to Current Use and Deprecation of TDEA," 2017. [Online]. Available: https://beta.csrc.nist.gov/News/2017/Update-to-Current-Use-and-Deprecation-of-TDEA

[10] M. Abadi and B. Warinschi, "Password-Based Encryption Analyzed," in *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, ICALP 2005*, 2005, pp. 664–676.

[11] Apple inc., "iOS Security Guide," Tech. Rep., 03 2017. [Online]. Available: https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[12] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J. Tsay, "Efficient Padding Oracle Attacks on Cryptographic Hardware," in *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*, 2012, pp. 608–625.

[13] E. Barker, "Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms," http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-175B.pdf, August 2016.

[14] E. Barker and A. Roginsky, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths (Rev. 1)," http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar1.pdf, November 2015.

[15] M. Bellare, T. Ristenpart, and S. Tessaro, "Multi-instance Security and Its Application to Password-Based Cryptography," in *Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, CRYPTO 2012*, 2012, pp. 312–329.

[16] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. K. Zinzindohoue, "A Messy State of the Union: Taming the Composite State Machines of TLS," in *Proceedings of the 36th IEEE Symposium on Security and Privacy, S&P 2015*, 2015, pp. 535–552.

[17] K. Bhargavan and G. Leurent, "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016*, 2016, pp. 456–467. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978423

[18] A. Biryukov, D. Dinu, and D. Khovratovich, "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications," in *Proceedings of the 1st IEEE European Symposium on Security and Privacy, EuroS&P 2016*, 2016.

[19] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '98*, 1998, pp. 1–12.

[20] D. Boneh, H. Corrigan-Gibbs, and S. Schechter, “Balloon Hashing: A Memory-Hard Function Providing Provable Protection Against Sequential Attacks,” in *Proceedings of the 22nd Annual International Conference on the Theory and Applications of Cryptology and Information Security, ASIACRYPT 2016*, 2016.

[21] W. E. Burr, D. F. Dodson, E. M. Newton, R. A. Perlner, W. T. Polk, S. Gupta, E. A. Nabbus, U. D. of Commerce, N. I. of Standards, and Technology, *Electronic Authentication Guideline: Recommendations of the National Institute of Standards and Technology - Special Publication 800-63-1*, 2012.

[22] R. Clayton and M. Bond, “Experience Using a Low-Cost FPGA Design to Crack DES Keys,” in *Proceedings of the 4th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2002*, 2002, pp. 579–592.

[23] J. Clulow, “On the Security of PKCS#11,” in *Proceedings of the 5th International Workshop on Cryptographic Hardware and Embedded Systems, CHES 2003*, 2003, pp. 411–425.

[24] T. Cooijmans, J. de Ruiter, and E. Poll, “Analysis of Secure Key Storage Solutions on Android,” in *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices, SPSM 2014*, 2014, pp. 11–20.

[25] Cryptosense S.A., “Mighty Aphrodite Dark Secrets of the Java Keystore,” 2016. [Online]. Available: https://cryptosense.com/mighty-aphrodite-dark-secrets-of-the-java-keystore/

[26] Y. Dodis, T. Ristenpart, and T. Shrimpton, “Salvaging Merkle-Damgård for Practical Applications,” in *Proceedings of the 28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, EUROCRYPT 2009*, 2009, pp. 371–388.

[27] P. Gasti and K. B. Rasmussen, “On the Security of Password Manager Database Formats,” in *Proceedings of the 17th European Symposium on Research in Computer Security, ESORICS 2012*, 2012, pp. 770–787.

[28] P. A. Grassi, J. L. Fenton, E. M. Newton, R. A. Perlner, A. R. Regenscheid, W. E. Burr, J. P. Richer, N. B. Lefkovitz, J. M. Danker, Y. Choong, K. K. Greene, and M. F. Theofanos, “Digital Identity Guidelines: Authentication and Lifecycle Management,” https://pages.nist.gov/800-63-3/sp800-63b.html#sec5, 2017.

[29] P. Gutmann, “Lessons Learned in Implementing and Deploying Crypto Software,” in *Proceedings of the 11th USENIX Security Symposium*, 2002, pp. 315–325. [Online]. Available: http://dl.acm.org/citation.cfm?id=647253.720291

[30] J. P. Kaps and C. Paar, “Fast DES Implementations for FPGAs and Its Application to a Universal Key-Search Machine,” in *Proceedings of the 5th Annual International Workshop in Selected Areas in Cryptography, SAC’98*, 1999, pp. 234–247.

[31] P. G. Kelley, S. Komanduri, M. L. Mazurek, R. Shay, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, and J. Lopez, “Guess Again (and Again and Again): Measuring Password Strength by Simulating Password-Cracking Algorithms,” in *Proceedings of the 33rd IEEE Symposium on Security and Privacy, S&P 2012*, 2012, pp. 523–537.

[32] J. Kelsey, B. Schneier, C. Hall, and D. Wagner, “Secure Applications of Low-Entropy Keys,” in *Proceedings of the 1st International Workshop on Information Security, ISW ’97*, 1997, pp. 121–134.

[33] Z. Li, W. He, D. Akhawe, and D. Song, “The Emperor’s New Password Manager: Security Analysis of Web-based Password Managers,” in *Proceedings of the 23rd USENIX Security Symposium*, 2014, pp. 465–479.

[34] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, “ASIC Clouds: Specializing the Datacenter,” in *Proceedings of the 43rd International Symposium on Computer Architecture, ISCA 2016*, 2016, pp. 178–190.

[35] MITRE, “CVE-2012-4929: CRIME attack,” http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-4929, September 2012.

[36] MITRE, “CVE-2014-0160: Heartbleed bug,” http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160, December 2013.

[37] MITRE, “CVE-2017-10345,” http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10345, October 2017.

[38] MITRE, “CVE-2017-10356,” http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-10356, October 2017.

[39] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, “PKCS#1: RSA Cryptography Specifications (Version 2.2),” https://www.ietf.org/rfc/rfc8017.txt, November 2016.

[40] K. Moriarty, B. Kaliski, and A. Rusch, “PKCS#5: Password-Based Cryptography Specification (Version 2.1),” https://www.ietf.org/rfc/rfc8018.txt, January 2017.

[41] P. Oechslin, “Making a Faster Cryptanalytic Time-Memory Trade-Off,” in *Proceedings of the 23rd Annual International Cryptology Conference on Advances in Cryptology, CRYPTO 2003*, 2003, pp. 617–630.

[42] Oracle Corporation, “Java Cryptography Architecture, Standard Algorithm Name Documentation for JDK 8,” http://docs.oracle.com/javase/8/docs/technotes/guides/security/StandardNames.html#KeyStore, 2014.

[43] Oracle Corporation, Private communication, October 2017.

[44] Oracle Corporation, “Critical Patch Updates, Security Alerts and Third Party Bulletin,” October 2017. [Online]. Available: http://www.oracle.com/technetwork/security-advisory/cpuoct2017-3236626.html

[45] C. Percival, “Stronger Key Derivation via Sequential Memory-Hard Functions,” May 2009.

[46] C. Percival and S. Josefsson, “The scrypt Password-Based Key Derivation Function,” https://tools.ietf.org/html/rfc7914, August 2016.

[47] RSA Laboratories, “PKCS#12: Personal Information Exchange Syntax Standard (Version 1.0),” June 1999.

[48] RSA Laboratories, “PKCS#11 v2.30: Cryptographic Token Interface Standard ,” April 2009.

[49] RSA Laboratories, “PKCS#12: Personal Information Exchange Syntax Standard (Version 1.1),” October 2012.

[50] M. Sabt and J. Traoré, “Breaking into the KeyStore: A Practical Forgery Attack Against Android KeyStore,” in *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016), Part II*, 2016, pp. 531–548.

[51] B. Schneier, *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., 1995.

[52] M. S. Turan, E. Barker, W. Burr, and L. Chen, “Recommendation for Password-Based Key Derivation. Part 1: Storage Applications,” http://csrc.nist.gov/publications/nistpubs/800-132/nist-sp800-132.pdf, December 2010.

[53] A. Vassilev, “Annex A: Approved Security Functions for FIPS PUB 140-2, Security Requirements for Cryptographic Modules,” http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf, April 2016.

[54] S. Vaudenay, “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...” in *Proceedings of the 21st International Conference on the Theory and Applications of Cryptographic Techniques Advances in Cryptology, EUROCRYPT 2002*, 2002, pp. 534–546.

[55] M. Weir, S. Aggarwal, M. Collins, and H. Stern, “Testing Metrics for Password Creation Policies by Attacking Large Sets of Revealed Passwords,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*, 2010, pp. 162–175.

[56] F. F. Yao and Y. L. Yin, “Design and Analysis of Password-Based Key Derivation Functions,” *IEEE Transactions on Information Theory*, vol. 51, no. 9, pp. 3292–3297, 2005.

[57] Y. Zhang, F. Monrose, and M. K. Reiter, “The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*, 2010.