JWTKey: Automatic Cryptographic Vulnerability Detection in JWT Applications

Bowen Xu[1,2], Shijie Jia[1,2(✉)], Jingqiang Lin[3], Fangyu Zheng[1,2], Yuan Ma[1,2], Limin Liu[1,2], Xiaozhuo Gu[1,2], and Li Song[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
jiashijie@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] School of Cyber Security, University of Science and Technology of China, Hefei, China

Abstract. JSON Web Token (JWT) has been widely adopted to increase the security of authentication and authorization scenarios. However, how to manage the JWT key during its lifecycle is rarely mentioned in the standards of JWT, which opens the door for developers with inadequate cryptography experience to implement cryptography incorrectly. Moreover, no effort has been devoted to checking the security of cryptographic usage in JWT applications. In this paper, we design and implement *JWTKey*, a static analysis detector leveraging program slicing technique to automatically identify cryptographic vulnerabilities in JWT applications. We derive 15 well-targeted cryptographic rules coupled with potential JWT key threats for the first time, and customized analysis entries and slicing criteria are identified accurately based on the observation of diversified JWT implementations, thus achieving balance between precise detection and overhead. Running on 358 open source JWT applications from GitHub, *JWTKey* discovered that 65.92% of the JWT applications have at least one cryptographic vulnerability. The comparative experiments with CryptoGuard demonstrate the effectiveness of our design. We disclose the findings to the developers and collect their feedback. Our findings highlight the poor cryptographic implementation in the current JWT applications.

Keywords: JWT · Key management · Cryptographic vulnerability

1 Introduction

JSON Web Token (JWT) is a compact claims representation format to be transferred between two parties [23]. As the advantages of being less verbose, more compact, smaller size, and easy to be parsed, JWT has been deeply coupled with various standard and non-standard access delegation and single sign-on (SSO) applications. Multiple indispensable parameters of standard protocols (e.g., OAuth [21,

G. Tsudik et al. (Eds.): ESORICS 2023, LNCS 14346, pp. 263–282, 2024.
https://doi.org/10.1007/978-3-031-51479-1_14

22] and OpenID Connect (OIDC) [43]) are either supported (e.g., *authorization grant*, *client credentials*, and *access token*) or explicitly mandatory required (e.g., *ID token*) to be transmitted in the format of JWT. Moreover, various JWT-based self-defined authentication/authorization schemes have been proposed in multifarious scenarios, such as web services [20], cloud SaaS applications [13], multi-agent systems [42] and software-defined networking [50].

The key insights of JWT are as follows: the claims need to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted in the format of JSON Web Signature (JWS) or JSON Web Encryption (JWE) [23]. This ensures the integrity and authenticity of the claims. According to Kerckhoff's principle [34], a cryptographic system should be designed to be secure, even if all its details, except for the key, are publicly known. Therefore, the security of JWT applications is based on the proper management of cryptographic keys throughout their lifecycle (e.g., generation, storage, transmission, and use). Otherwise, severe consequences could be provoked. For example, a vulnerability was disclosed in the "View As" feature of Facebook, making attackers could obtain the access tokens of the users [19]. This exposes the fact that there are many risks of token leaks and attacks in various application scenarios. Moreover, some vulnerabilities (e.g., CVE-2022-35540 [8] and CVE-2022-36672 [9]) have been disclosed in popular projects (e.g., NovelPlus and AgileConfig) due to hard-coded keys in their JWT usage, allowing attackers create a custom user session or gain administrator access.

Many efforts on the security of cryptographic applications have been proposed, while they are all limited in detecting cryptographic vulnerabilities of JWT applications. Firstly, previous JWT related studies either devote to extending the range of JWT applications [18,33,46] or try to check anti-protocol flaws at the protocol level (e.g., OAuth [2,14,41,53] and OIDC [15,18]), which only consider the correctness of the JWT-based parameters (e.g., *access token* and *ID token*), and do not consider the vulnerabilities from the aspect of JWT cryptographic usage. Secondly, the existing cryptographic misuse detectors commonly focus on the application security of APIs in underlying cryptographic libraries (e.g., JCA and OpenSSL) by using a program analysis method to check if an application respects the predefined cryptographic rules [31]. However, JWT applications typically do not invoke cryptographic libraries directly, but indirectly invoke them by third-party JWT libraries (e.g., java-jwt [4] and jose.4.j [6]), which encapsulate JWT cryptographic implementations by providing JWT generation and verification related APIs. Previous detectors treat JWT libraries as their "applications", whose diversified implementation details make the true upper applications (i.e., JWT applications) need to pass key-related parameters through multiple methods, classes, field variables, or conditional statements of the APIs in JWT libraries. The above complicated orthogonal invocations cannot be handled by the existing detectors as the refinements of clipping orthogonal explorations [40], thus resulting in prevalent false negatives [1].

In this paper, we tackle the above limitations and introduce *JWTKey*, a static analysis detector, which leverages static program slicing technique, achieving automated analysis of cryptographic vulnerabilities in JWT applications. Our

design relies on several key insights: 1) To provide precise cryptographic vulnerability detection in JWT applications, we derive 15 well-targeted rules by taking an in-depth analysis of the potential security threats throughout a JWT key lifecycle; 2) To obtain precise detection outcomes with acceptable overhead, we determine analysis entries and slicing criteria based on the observation of diversified implementations of JWT applications and JWT libraries; 3) Based on the specific stated cryptographic rules, both backward/forward program slicing and properties file feature analysis are performed to track the APIs and arguments in the intermediate representation of the detected applications.

To demonstrate the effectiveness of *JWTKey*, we carry out a large-scale evaluation based on 358 popular Java-based JWT applications crawled from GitHub. *JWTKey* report 417 alerts for the 358 applications, and our manual analysis confirms 410 alerts, achieving an accuracy of 98.32%. We identify that 65.92% of the JWT applications have at least one cryptographic vulnerability. The large number of alerts indicates a widespread misunderstanding of how to properly manage keys in JWT applications. We also utilize CryptoGuard [40] (the detector with the highest precision [1]) for comparative experiments, while it only report 49 cases of JWT cryptographic vulnerabilities, leaving 361 cases undiscovered. We report our security findings to the corresponding developers of JWT applications and JWT libraries, some of which have been indeed patched (see Sect. 7). In summary, we make the following contributions:

- We study the security of JWT applications from a brand-new perspective (i.e., cryptographic vulnerabilities). We conduct an in-depth study on the potential security threats throughout a JWT key lifecycle and derive 15 cryptographic rules coupled with JWT implementation cryptographic APIs. The purpose of this research is to cover as many vulnerabilities as possible to guide developers to use cryptography securely on large JWT applications.
- We design and implement a static analysis detector, *JWTKey*[1]. With accurate identification of analysis entries and slicing criteria based on the observation of diversified implementations of JWT applications and JWT libraries, *JWTKey* achieves balance between precise detection and overhead.
- Our evaluation on 358 JWT applications and the comparative experiments with CryptoGuard demonstrate the effectiveness of our design in discovering cryptographic vulnerabilities of JWT applications.

2 Background

2.1 JWT Structure

JWTs encode claims to be transmitted as a JSON object that is used as the payload of a JSON Web Signature (JWS) [25] or as the plaintext of a JSON Web Encryption (JWE) [26], enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted. A

[1] Available at https://github.com/JWTKeyIIE/JWTKey.

JWT is represented as a sequence of URL-safe parts separated by period ('.') characters. Each part contains a base64url-encoded value. Specifically, a JWS consists of three parts (i.e., JSON object signing and encryption (JOSE) header, payload and signature) and a JWE consists of five parts (i.e., JOSE header, encrypted key, initialization vector, ciphertext and authentication tag).

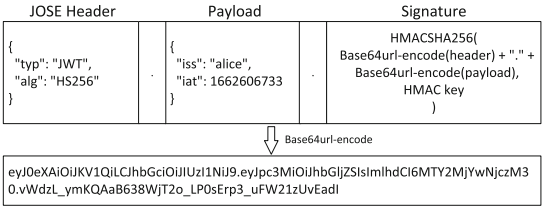

Fig. 1. A simple JWT instance with JWS structure.

Figure 1 presents a simple JWT instance with JWS structure. The first part is a JOSE header, whose contents describe the cryptographic operations applied to the corresponding JWT claims set. The "alg" (algorithm) parameter, which is required in the JOSE header, identifies the cryptographic algorithm (e.g., HMAC, ECDSA, AES-128) used to secure the JWT. Several optional parameters are also provided in the JOSE header, such as "jwk" (JSON web key), which is the public key that corresponds to the key used to sign the JWS or encrypt the JWE; "jku" (JWK set URL), which is a URI that refers to a resource for a set of JWK public keys; "kid" (key ID), which is a hint indicating which key was used to secure the JWS/JWE; "x5c" (X.509 certificate chain), which is the X.509 public key certificate or certificate chain corresponding to the key used to sign the JWS or encrypt the JWE; "x5u" (X.509 URL), which is a URI that refers to a resource for the public key certificate or certificate chain; "p2s" (PBES2 salt input), which is a salt value used for password-based encryption (PBE); "p2c" (PBES2 count), which is an iteration count used for PBE.

2.2 Static Program Slicing

Static program slicing is a decomposition technique that extracts program parts from program statements relevant to a particular computation [56]. As a static method, static program slicing does not require the execution of programs (e.g., JWT applications). They scale up to a large number of programs, cover a wide range of security rules, and are unlikely to have false negatives [40]. A program slice is the result of slicing which consists of a subset statements of a program that potentially affect or be affected by the slicing criterion. A slicing criterion consists of a pair $\langle p, V \rangle$. Specifically, p is a program point and V is a subset of program variables. Commonly speaking, there are two main types of static program slicing methods: backward slicing and forward slicing. Backward slicing is used to compute the set of instructions that affect the variables in V at program

point p, and forward slicing is used to compute the set of instructions affected by the variables in V at program point p. It is worth to mention that a system dependence graph (i.e., SDG) is commonly built to identify the minimal set of program statements and variables which are necessary to produce a particular output. In particular, a SDG is a directed graph that represents the program statements as nodes and represents the dependencies between them as edges. Control dependence edges represent the flow of control in the program, while data dependence edges represent the flow of data between different statements.

3 Related Work

In recent years, researchers have shown significant interest in the problem of vulnerability detection on various kinds of applications. However, how to detect cryptographic vulnerabilities in JWT applications is still an open question.

JWT-Related Studies. Since JWT is deeply coupled with authentication and authorization protocols, many works related to protocol security have been proposed (e.g., OAuth [2,14,41,53] and OIDC [15,18]). In the case of OAuth, Wang et al. [53] proposed a tool that combines static analysis and network analysis to identify OAuth bugs on Android platform. Fett et al. [14] carried out a formal analysis of the OAuth 2.0 standard in an expressive web model. Rahat et al. [2] developed OAUTHLINT, which incorporates a query-driven static analysis to check anti-protocol programs of OAuth client-side. More recently, Rahat et al. [41] proposed Cerberus, which aims to find logical flaws and identify vulnerabilities in the implementation of OAuth service provider libraries. In the case of OIDC, Fett et al. [15] developed a formal model of OIDC based on the Dolev-Yao style model of the web infrastructure (FKS model). Ghasemisharif et al. [18] investigated the security implications of SSO, offered an analysis of account hijacking on the modern Web and proposed an extension to OIDC based on JWT revocation token. However, all the above previous researches mainly focus on checking anti-protocol flaws at the protocol level, and do not consider the diversified cryptographic implementation details of JWT applications. Differently, in this work, we aim to provide deeper insights into how to securely implement JWT from a foremost cryptographic security perspective.

Cryptographic Misuse Detectors. Recently, many cryptographic misuse detectors have been proposed. Their key insights are using static methods (e.g., program slicing and taint analysis) or dynamic methods (e.g., fuzzing and logging) to check if an application respects the predefined cryptographic rules [31]. For example, in Java, Egele et al. [12] proposed CryptoLint, which performed a study to measure cryptographic misuse in Android applications with only 6 rules. Krüger et al. [30] developed CrySL, which provided higher precision than CryptoLint [30] by designing a set of cryptographic rules for the JCA library. Wen et al. [54] proposed MUTAPI to discover API misuse patterns via mutation analysis. However, it cannot discover misuse patterns which require specific values

(e.g., *javax.crypto.Cipher("DES")*). More recently, CryptoGuard [40] performed data-flow analysis based on forward/backward slicing methods for Java Projects. Piccolboni et al. [38] implemented CryLogger to detect cryptographic misuses dynamically. Ami et al. [3] presented the MASC framework, which enables a systematic and data-driven evaluation of cryptographic detectors using mutation testing. In C/C++, Rahaman et al. [39] proposed TaintCrypt, which adopted taint analysis to check projects issues using C/C++ cryptographic libraries (e.g., OpenSSL). Zhang et al. [57] implement CryptoRex to identify cryptographic misuse of firmware in IoT devices. In Python, Wickert et al. [55] designed LICMA, which adopted hybrid static analysis to discover cryptographic misuses in applications written in Python and C. Frantz et al. [16] proposed Cryptolation, which is a static analysis tool to discover Python cryptographic misuses.

The above detectors commonly focus on the library-level cryptographic implementation issues in cryptographic libraries (e.g., JCA, JCE and JSSE in Java, OpenSSL in C/C++, M2Crypto and PyCrypto in Python). Differently, in this work, we aim to fill the gap of inaccurate identification of cryptographic misuses at JWT applications by proposing JWT-oriented cryptographic rules and detection methods.

4 Threat Model and Cryptographic Rules

4.1 Threat Model

In this paper, we focus on the cryptographic vulnerabilities of JWT application during the lifecycle of key management. We assume that even developers with rich cryptography experience may bring insecure behaviors when handling JWT related keys. Adversaries aim to obtain resources or privileges of JWT applications by constructing forged JWTs with obtained key information. We consider two types of adversaries in this work. First, for the system adversaries who have physical or remote access to the physical devices (e.g., client or server devices), they could obtain or infer the symmetric key or private key of JWT applications during key generation, storage or use stages by reverse engineering [37] or privilege escalation [49]. Second, for the network adversaries, they can intercept and modify network traffics between JWT issuers and verifiers. They can also launch common TLS attacks (e.g., TLS stripping, rogue TLS certificate) to decrypt the HTTPS traffic [52] where the JWT application implements TLS problematically. Therefore, the network adversaries could exploit the insecure behaviors during key transmission or key use stages to infer the JWT key, or mislead the JWT verifiers to accept a forged JWT with a replaced key [11].

4.2 Cryptographic Rules

Based on our threat model, we take an in-depth analysis on the arguments of APIs in the JWT libraries, and summarize all the potential threats of a key throughout its lifecycle discussed by JWT specifications [24–26] and current security best practices [3,44], thus concluding the cryptographic rules in Table 1.

Table 1. JWT-oriented cryptographic rules derived form the APIs of JWT libraries and applications throughout a key lifecycle. (⇐: backward slicing; ⇒: forward slicing; ◇: feature analysis in properties files)

ID	Cryptographic Rules	Key Lifecycle	Method	Reference
R-01	Do not use insecure PRNG or predictable PRNG seeds	Generation	⇐, ⇒	[29,39]
R-02	Do not use keys with insufficient length		⇐, ⇒	[5,24]
R-03	Do not use hard-coded symmetric/private key	Storage	⇐	[5,40]
R-04	Do not use predictable/constant passwords for PBE		⇐	[5,24]
R-05	Do not use predictable/constant passwords for KeyStore		⇐, ⇒	[5,40]
R-06	Do not store symmetric/private key in properties files		⇐, ◇	[5,10]
R-07	Do not store private key in plaintext files		⇐, ◇	[5,10]
R-08	Do not use HTTP URL connections for key transmission	Transmission	⇐, ◇	[25,26]
R-09	Do not verify certificates or host names in SSL/TLS in trivial ways during key transmission		⇐, ⇒	[31,40]
R-10	Do not use "jwk" and "jku" for key transmission		⇐	[43,45]
R-11	Do not use JWT public key before validating the certificate or certificate chain	Use	⇐, ⇒	[25,26]
R-12	Do not use static IVs in cipher operation modes (e.g., CBC and GCM)		⇐	[7,40]
R-13	Do not use PBE with fewer than 1,000 iterations		⇐	[24,40]
R-14	Do not use static salts for PBE		⇐	[24,40]
R-15	Do not use deprecated insecure APIs		⇒	[11,24]

Vulnerabilities in Key Generation. HMAC, digital signature algorithms and symmetric/asymmetric encryption algorithms may be used in JWS or JWE (Sect. 2.1). *R-01* requires that the keys used in the cryptographic algorithms should be derived from a cryptographically secure random number generator. Insecure pseudo-random number generator (PRNG) should not be used (e.g., *java.util.Random*), meanwhile, predictable seeds should not be provided to the PRNG for key generation [39]. Otherwise, the generated key may be easily reversed by an attacker [29]. *R-02* emphasizes the security of information protected by cryptography directly depends on the strength of the keys [5]. However, keys with insufficient length are allowed to be supplied to the cryptographic algorithm APIs of JWT libraries, which may result in inputting keys with insufficient length for developers and thus are vulnerable to brute force attacks [28,40]. For example, RSA-1024 (with less than 112 bits security strength) should not be used to generate a JWT [24], and the key length of HS256 (i.e., HMAC using SHA-256) should be equal or over 256 bits.

Vulnerabilities in Key Storage. Confidentiality shall be provided for all secret key information [5], otherwise, the adversary may easily obtain or infer the key of JWT. However, there are various insecure key storage methods in JWT applications. *R-03* prohibits using hard-coded keys in the program codes. *R-04* and *R-05* forbid using predictable/constant passwords for password-based encryption (PBE) [24] and KeyStore, respectively. For example, though an application can only access its own KeyStore in Android, privilege escalation attacks can bypass this restriction if an insecure password is utilized [40,49]. The following two rules focus on the insecure ways of storing the key information at rest, i.e., in properties files (*R-06*) and plaintext files (*R-07*) [10].

Vulnerabilities in Key Transmission. There are many cases of key transmission in JWT applications, for example, the symmetric HMAC key and the public key in the case of JWS, the key encryption key (KEK) and the public key in the case of JWE. The protocol used to transmit the key must provide security protection, and thus secure TLS must be used for key transmission [25,26]. Correspondingly, *R-08* forbids using HTTP URL connections for key transmission. *R-09* requires JWT applications to properly verify certificates and host names in SSL/TLS during key transmission to avoid man-in-the-middle attacks [31,40]. *R-10* forbids using "jwk" and "jku" for key transmission, and additional verification is required (e.g., by matching the URL to a whitelist of allowed locations and ensuring no cookies are sent in the GET request) [43]. This is because JWT provides "jwk" and "jku" header parameters to refer to a resource for a set of public keys, however, blindly trusting the header parameters, which may contain an arbitrary URL, could result in server-side request forgery (SSRF) attacks [45].

Vulnerabilities in Key Use. Before using the public key to which the JWE is encrypted (or the key used to verify the JWS), the recipient must validate the corresponding certificate or certificate chain [25,26]. *R-11* focuses on the validity of the obtained certificates (e.g., in the "x5c" or "x5u") in JWT, and the public key should be considered as invalid if any validation failure occurs, otherwise, SSRF attacks may occur [45]. As inappropriate settings of other parameters (e.g., IV, salts) may also pose a threat to JWT applications, we involve the following rules in *JWTKey*. Specifically, as JWT supports AES CBC mode and GCM mode to encrypt JWT content, *R-12* emphasizes that static IVs should not be used in CBC mode (to avoid chosen-plaintext attacks [40]) and GCM mode (to avoid forbidden attacks [7]). *R-13* focuses on that at least 1,000 iterations are required for PBE and *R-14* forbids using static salts for PBE, which may result in dictionary attacks [24,40]. At last, *R-15* prohibits the use of deprecated insecure APIs of JWT libraries. For example, in java-jwt library, both private and public keys are allowed to simultaneously transmit to the *require* API in *Verification* class to verify tokens. Moreover, as RSAES-PKCS1-v1_5 algorithm may be vulnerable to certain attacks (e.g., Bleichenbacher million message attack [11]), thus it is not recommended for new applications [24].

5 Design

5.1 Overview

Figure 2 shows an overview of *JWTKey*, which leverages static program slicing [56] to detect cryptographic vulnerabilities of JWT applications. *JWTKey* takes Java source files (maven or gradle project), *.class* files, *.jar* files, *.war* files or *.apk* files as input, and outputs reports with the identified vulnerabilities.

5.2 Construct System Dependence Graph

Given a JWT application program, *JWTKey* first converts Java source code or byte-code into an intermediate representation (IR) format (i.e., Jimple) by

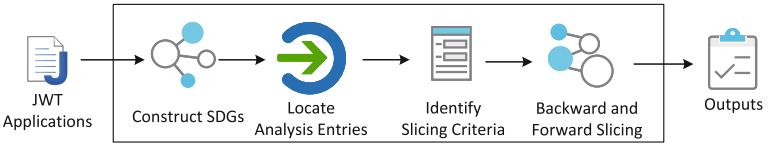

Fig. 2. Overview of *JWTKey*.

taking advantage of Soot [47] framework. Moreover, we use Soot to construct intra-procedural data-dependencies, which can be used to build system dependence graphs (SDGs). For inter-procedural analysis, *JWTKey* uses class-hierarchy analysis to determine the calling relationship between all the methods, thus building a caller-callee relationship of all the methods of the application. The calling relationship can be used to build control dependency edges in the SDG. Moreover, by processing the *AssignStmt* statements defined in Jimple, *JWTKey* adds field influence as a control dependency to the SDG for inter-procedural analysis.

As the original SDG is obtained by stitching all control flow graphs from the program methods, to improve the efficiency and accuracy, *JWTKey* confines the analysis by keeping track of the methods relevant to JWT implementations. Namely, we track the cryptographic related information to pinpoint a sub-callgraph during the whole key lifecycle. Since the sub-callgraph is typically small, its corresponding SDG will also be small.

5.3 Analysis Entries and Slicing Criteria

JWTKey adopts static program slicing to identify cryptographic vulnerabilities in JWT applications. In particular, we take the APIs, which are provided by the JWT libraries to generate or verify JWT, as analysis entries.

Analysis Entries. *JWTKey* locates the starting point of key related operations in JWT libraries and divides the analysis-entry-APIs from JWT libraries into the following two categories. First, the key specification APIs for JWT generation. JWT libraries provide APIs to set key related parameters for JWT generation and support developers to specify the algorithm and the corresponding key for signing or encrypting. *JWTKey* uses this kind of APIs as analysis entries to detect rules related to key generation (*R-01* and *R-02*), key storage (*R-03* to *R-07*), and key use (*R-12* to *R-15*). For example, *JWTKey* utilizes the *signWith (Key key, SignatureAlgorithm alg)* method in *JWTBuilder* provided by jjwt library [27] as analysis entry and takes the key as the parameter of interest to perform slice analysis for different rules detection. Second, another category of analysis entries comprises the key specification APIs for JWT verification. For example, the *setVerificationKey(Key key)* method in the *JwtConsumerBuilder* class of Jose4j library [6]. *JWTKey* takes the key as the parameter of interest to perform analysis of key storage (*R-03* and *R-06*), key transmission (*R-08* to

R-10) and key use (*R-11*) related detection rules. As the number of detected analysis entries of *JWTKey* is quite large, for convenience, we show two representative analysis entries of each popular Java JWT library in Table 2.

Table 2. The representative analysis entries APIs of the top 7 most popular Java JWT libraries (i.e., java-jwt, jjwt, jose.4.j, Nimbus-JOSE-JWT, Spring Security OAuth, FusionAuth JWT and Vert.x Auth JWT).

No.	Analysis Entry
1.1	com.auth0.jwt.algorithms.Algorithm HMAC256(**java.lang.String**)
1.2	com.auth0.jwt.algorithms.Algorithm RSA256(**java.lang.String**)
2.1	io.jsonwebtoken.JwtBuilder signWith(io.jsonwebtoken.SignatureAlgorithm,**byte**[])
2.2	io.jsonwebtoken.JwtParser setSigningKey(**java.lang.String**)
3.1	org.jose4j.keys.HmacKey void < init>(**byte**[])
3.2	org.jose4j.keys.AesKey void <init>(**byte**[])
4.1	com.nimbusds.jose.crypto.MACSigner void <init>(**byte**[])
4.2	com.nimbusds.jose.crypto.DirectEncrypter void <init>(**javax.crypto.SecretKey**)
5.1	org.springframework.security.jwt.crypto.sign.RsaSigner: void <init>(**java.lang.String**)
5.2	org.springframework.security.jwt.crypto.sign.EllipticCurveVerifier: void <init> (**java.security.interfaces.ECPublicKey**, java.lang.String)
6.1	io.fusionauth.jwt.hmac.HMACSigner: void <init>(io.fusionauth.jwt.domain.Algorithm,**byte**[],java.lang.String, io.fusionauth.security.CryptoProvider)
6.2	io.fusionauth.jwt.rsa.RSASigner: void <init>(io.fusionauth.jwt.domain.Algorithm,**java.security.PrivateKey**, java.lang.String, io.fusionauth.security.CryptoProvider)
7.1	io.vertx.ext.auth.PubSecKeyOptions setPublicKey(**java.lang.String**)
7.2	io.vertx.ext.auth.PubSecKeyOptions setSecretKey(**java.lang.String**)

Slicing Criteria. Note that insecure key storage behaviours (i.e., *R-03*, *R-06* and *R-07*) can be detected by the corresponding analysis entries, while the others need more rounds of program slicing. For the other rules, starting from the analysis entries, the propagation paths of the parameter of interest tracks with the slicing criterion APIs. Similarly, as the number of slicing criterion APIs is quite large, we show partial representative APIs in Table 5 (in the Appendix). The slicing criterion APIs can be divided into two groups: APIs in JWT libraries and APIs in Java cryptographic libraries. First, the APIs in JWT libraries are used to detect the rules related to JWT-specific properties. For example, in *R-08*, *JWTKey* determines the APIs used to retrieve keys from URI provided by JWT libraries (e.g., the *retrieveKeysFromJWKS* method provided by FusionAuth JWT [17]). Then *JWTKey* uses these APIs as slicing criteria to perform a new round of backward slicing to determine whether there is an insecure HTTP connection. For *R-10, R-13, R-14*, *JWTKey* detects the header parameter setting and getting methods provided by JWT libraries, and performs backward slicing on the parameters of these related methods to analyze the insecure use of "jwk", "jku", "p2c", "p2s" header parameters. *JWTKey* detects IV setting methods in JWT libraries for *R-12*. Second, as only a few of JWT libraries provide APIs to generate random values (e.g., key, random number, salt value), thus

a large part of JWT application developers still tend to use the APIs of cryptographic libraries to generate random values. Therefore, *JWTKey* also detects the misuse of the underlying Java cryptographic libraries APIs for different algorithms based on the slicing result of the analysis entries. For example, to detect *R-02*, *JWTKey* identifies the use of the *initialize* method in *KeyPairGenerator* class and the *init* method in *SecretKeySpec* class to check the key size.

5.4 Execute Specific Slicing for Different Rules

We break down the detection of rules into one or more steps and perform backward and/or forward slicing for each step (the fourth column of Table 1).

Backward Slicing. Backward slicing in *JWTKey* consists of intra-procedural and inter-procedural analysis. *JWTKey* leverages the Soot framework to perform def-use analysis to compute slices for intra-procedural backward slicing. For inter-procedural backward analysis, *JWTKey* analyzes the upward inter-propagation of the slicing criterion such as method invocation and indirect field access realized by orthogonal methods based on the intra-procedural backward slicing. For example, in *R-03*, *JWTKey* executes a single round of backward slicing based on key parameters from the analysis entries and locates hard-coded keys by analyzing the assignment statement and *Constant* object (defined by Soot) included in invocation statements of slices. *JWTKey* also checks the length of the hard-coded key for *R-02*. For the rules related to the misuse of APIs in cryptographic libraries (e.g., *R-01*, *R-05*), *JWTKey* performs one or more rounds of backward slicing. *JWTKey* first locates the detected APIs, and then determines the relationship between the APIs and the key lifecycle by analyzing the SDG and the slicing results of the analysis entry. For *R-01*, we detect PRNGs that contain a hard-coded seed value. When *JWTKey* analyzes the call of an insecure API (e.g., *init(byte[])* method in *SecureRandom* class) and determines that the API is related to JWT key generation, it will use the API as a slicing criterion, and execute backward slicing with the *byte[]* parameter. For *R-05*, we detect predictable KeyStore password by performing backward slicing with the parameters of *load*, *store*, *getKey* methods in *java.security.KeyStore* class and *getKeyPair* method in *KeyStoreKeyFactory* class provided by Spring Security OAuth [48] as the slicing criteria.

Forward Slicing. For example, we detect *getPublicKey* method in *X509Certificate* class. As shown in Listing 5.1, r12 is an object of *Certificate* class, where its fields are accessed indirectly with the orthogonal method (i.e., *getPublicKey*), and r13 is an object of *PublicKey* class. We design a forward slicing for *R-11* to detect the object of *PublicKey* class, which is used to verify a JWT. In addition, we perform inter-procedural forward analysis on methods invoking r12 object, and analyze whether these methods include certificate validity verification methods (e.g., *verify* method and *checkValidity* method

in *java.security.cert.X509Certificate*) and certificate chain verification methods (e.g., *validate* method in *java.security.cert.CertPathValidatorResult* class).

```
1  r12 = (java.security.cert.X509Certificate) $r11;
2  ......
3  r13 = virtualinvoke r12.<java.security.cert.X509Certificate: java.security.PublicKey
       getPublicKey()>();
```

Listing 5.1. A converted IR of getting the public key from certificate in *R-11*.

Properties File Analysis. A myriad of JWT applications are based on Spring framework [48], which simplifies authentication and authorization implementations [32]. We observe that such applications commonly use properties file to externalize user configuration (e.g., key, password and URI). As static analysis methods commonly cannot capture the full semantics of features (e.g., the injection from properties file) [41], therefore, we conduct a special analysis on the processing of properties files in Spring framework. To detect the symmetric/private key, private key path and insecure HTTP URL in properties file (e.g., *R-06*, *R-07* and *R-08*), *JWTKey* analyzes the annotation of the assignment statement for specifying value, path, or URL of key used to process JWT, searches the *@Value* annotation and parses the corresponding property name, and then retrieves the properties files to obtain the corresponding property value. We analyze the property value injection methods provided by the Spring framework, and add corresponding analysis of *@Configuration*, *@EnableAuthorizationServer*, and *@Autowired* annotations, which are used to configure JWT related values from properties file.

6 Implementation and Evaluation

6.1 Implementation

We implemented *JWTKey* with around 12,403 lines of Java code. *JWTKey* is realized based on the Soot framework [47], which provides compilation and analysis for Java applications. We use the intra-procedural data-flow and def/use analysis provided by Soot to construct SDGs and utilize the worklist algorithm of Soot to process the orthogonal method during slicing. In *JWTKey*, we select and support the top seven most popular JWT libraries (i.e., java-jwt [4], jjwt [27], Nimbus-JOSE-JWT [35], jose.4.j [6], FusionAuth JWT [17], Vert.x Auth JWT [51] and Spring Security OAuth [48]). Note that 99% Java-based JWT applications that we crawled from GitHub are developed based on the above selected libraries.

6.2 Experimental Setup

We select our data-set from GitHub as follows: 1) we determined several keywords (e.g., authorization, authentication, OAuth, OIDC, microservice and

mobile application) as the topics where JWT is most widely used to obtain popular open-source applications with JWT usage; 2) in each topic, we filtered by language Java and sorted by the number of stars that are more than 50 stars. Finally, we crawled 358 open source Java-based JWT applications in total. We deployed *JWTKey* on a PC with Intel Core i7-4500U (1.80GHZ CPU and 12GB RAM). The average runtime was 1.34 s per thousand LoC (Line of Code).

6.3 Security Findings in JWT Applications

The detailed evaluation results are shown in Table 3. *JWTKey* reported a total of 417 alerts for the 358 detected JWT applications. Out of the 358 applications, 236 applications (65.92%) have at least one JWT cryptographic vulnerability and 116 applications (32.40%) have at least two related vulnerabilities. Our careful manual source-code analysis (conducted by two Ph.D. students under the guidance of a professor from the area of applied cryptography) confirmed that 410 alerts are true positives, resulting in the accuracy as 98.32%. The 7 false positives are due to the path insensitivity and clipping detection depth when dealing with some complex semantic cases (See Sect. 7). Note that in terms of precision, CryptoGuard outperforms CrySL [1], and CrySL outperforms CryptoLint [30], therefore, we prove the effectiveness of *JWTKey* by making comparative experiments with CryptoGuard [40]. Note that *JWTKey* adopts JWT-oriented cryptographic rules, while some of the rules (e.g., *R-06*, *R-07*, *R-10*, *R-11* and *R-15*) are not supported by CryptoGuard, thus resulting in its direct false negatives.

Table 3. Accuracy comparison between *JWTKey* and CryptoGuard of 358 JWT applications. (TP: True Positives; FN: False Negatives; "-": Not supported. Note that the TP and FN of CryptoGuard are counted with the cryptographic rules of *JWTKey*.)

ID	JWTKey				CryptoGuard [40]	
	# Applications	# Alerts	# TP	Accuracy	# TP	# FN
R-01	1(0.28%)	1(0.24%)	1	100.00%	1	0
R-02	118(32.96%)	125(29.98%)	120	96.00%	4	116
R-03	105(29.33%)	109(26.14%)	109	100.00%	28	81
R-04	1(0.28%)	1(0.24%)	1	100.00%	1	0
R-05	17(4.75%)	17(4.08%)	17	100.00%	13	4
R-06	98(27.37%)	126(30.22%)	126	100.00%	-	126
R-07	3(0.84%)	3(0.72%)	3	100.00%	-	3
R-08	14(3.91%)	17(4.08%)	17	100.00%	1	16
R-09	0(0.00%)	0(0.00%)	0	0.00%	0	0
R-10	1(0.28%)	1(0.24%)	1	100.00%	-	1
R-11	11(3.07%)	11(2.64%)	9	81.82%	-	9
R-12	0(0.00%)	0(0.00%)	0	0.00%	0	0
R-13	1(0.28%)	1(0.24%)	1	100.00%	1	0
R-14	1(0.28%)	1(0.24%)	1	100.00%	1	0
R-15	4(1.12%)	4(0.96%)	4	100.00%	-	4
Total	N/A	417	410	98.32%	50	360

Vulnerabilities in Key Generation. For *R-01*, both *JWTKey* and CryptoGuard identified 1 case (in GCAuth) using insecure PRNG (i.e., *random* method in *java.lang.Math* class) to generate a JWT key. For *R-02*, *JWTKey* reported 125 alerts, and we confirmed 120 of them. The false positives mainly come from path insensitivity. For example, in wetech-admin, the generated short HMAC key will be combined with other parameters (e.g., user ID) as the final HMAC key. As CryptoGuard detected insecure asymmetric ciphers (e.g., RSA and ECC), thus it can identify the 4 cases of insecure RSA short key pairs (e.g., spring-boot-api-seedling), while it cannot identify the rest 116 cases of short HMAC key (e.g., in restheart and micronaut-microservice).

Vulnerabilities in Key Storage. For *R-03*, *JWTKey* identified 109 cases, 2 of them are hard-coded AES symmetric keys and the rest 107 cases are hard-coded HMAC keys. For example, both litemall (6.7k forks, 17.3k stars) and lamp-cloud (1.2k forks, 4.4k stars) utilize a hard-coded HMAC key to sign and verify JWTs. We also detected that a hard-coded HMAC key is used in Novel-Plus, which has been defined as CVE-2022-36672 [9]. For *R-04*, both *JWTKey* and CryptoGuard reported 1 case of using hard-coded PBE password (i.e., azure-activedirectory). For *R-05*, *JWTKey* identified 17 cases of hard-coded KeyStore password. As CryptoGuard adopts refinements after clipping orthogonal explorations, it missed 81 cases and 4 cases in *R-03* and *R-05*, respectively. The accurate identification of analysis entries and slicing criteria of *JWTKey* shortens the analysis paths and avoids the false negatives of CryptoGuard. Moreover, *JWTKey* reported 126 alerts (e.g., eladmin and Sa-Token) and 3 alerts (e.g., stormpath-sdk-java) for *R-06* and *R-07*, respectively, however, CryptoGuard missed the above two cases due to lacking of feature analysis in properties files.

Vulnerabilities in Key Transmission. For *R-08*, *JWTKey* identified 17 cases of insecure HTTP URL for key transmission (e.g., happyride). However, CryptoGuard missed 16 cases due to a lack of feature analysis in properties files. For *R-09*, note that a JSSE provider (i.e., SunJSSE) is preinstalled and preregistered with JCA, which provides an implementation of secure SSL/TLS protocols [36] for key transmission without the need of self-configuration, thus no alert is reported by *JWTKey* and CryptoGuard. For *R-10*, *JWTKey* identified 1 case of using "jwk" parameter for key transmission in devdojo-microservices.

Vulnerabilities in Key Use. For *R-11*, *JWTKey* identified 11 cases of insecure certificate/chain validation, while we confirmed 9 of them. The 2 false positives come from path insensitivity or clipping detection depth when dealing with self-implemented certificate verification methods with inter-procedural forward analysis. For example, as shown in Listing 6.1, *JWTKey* raised an alert for *R-11* because the application obtains a public key from "x5c" parameter without certificate validity verification. However, this is a false positive due to the *setX5C* is set as false, making the path from line 4 to line 6 unreachable. As a path-

insensitive static detector, *JWTKey* cannot handle such conditional execution paths.

```
1  boolean sendX5C;
2  ......//Setting sendX5C as false
3  if(setX5C){
4      List<cert> certs = new ArrayList<>();
5      for (String cert : credential.getEncodedPublicKeyCertificateChain()){
6          certs.add(new Base64(cert));}}
7  ......//cert get from KeyStore
```

Listing 6.1. A false positive example due to path insensitivity for *R-11*.

For *R-12*, no alert is reported by *JWTKey* and CryptoGuard due to the fact that rare applications are based on JWE in the wild. For *R-13* and *R-14*, both *JWTKey* and CryptoGuard reported 1 alert of fewer than 1,000 iterations and static salts for PBE (i.e., azure-activedirectory), respectively. For *R-15*, *JWTKey* identified 4 alerts of using deprecated insecure APIs (e.g., Dashboard uses *require* API to transmit both private and public keys to verify tokens). *JWTKey* identified no use case of RSAES-PKCS1-v1_5 algorithm in our data-set. Note that if a JWT application uses RSAES-PKCS1-v1_5 algorithm, it may not be directly vulnerable to an attack (e.g., Bleichenbacher attack [11]), thus more manual detection efforts should be taken to verify whether it is a vulnerability.

Table 4. Insecure functions provided in JWT libraries. (Lib 1–7 corresponds to java-jwt, jose.4.j, Nimbus-JOSE-JWT, jjwt, FusionAuth JWT, Vert.x Auth JWT and Spring Security OAuth, respectively. √: secure; ✕: insecure; √†: provide insecure functions in previous versions; \: not supported.)

Functions	Lib-1	Lib-2	Lib-3	Lib-4	Lib-5	Lib-6	Lib-7
HMAC key length restriction	✕	√	√	√†	✕	✕	√†
RSA key length restriction	✕	√	√	√	√	✕	√
PBE parameters restriction	\	✕	√	\	\	\	\
Restrict HTTP for key transmission	✕	✕	✕	\	✕	\	\
Certificate verification before obtaining JWT public key	\	✕	✕	\	✕	\	✕

6.4 Security Findings in JWT Libraries

As Table 4 shows, *JWTKey* also discovered several vulnerabilities which are due to the insecure functions provided in the JWT libraries. Specifically, some JWT libraries adopt loose policy of key length restriction. For example, java-jwt, FusionAuth JWT, Vert.x Auth JWT, jjwt (before v0.10.0) and Spring Security OAuth (before v2.5.1) allow the use of HMAC keys with insufficient length. Java-jwt and Vert.x Auth JWT allow the use of RSA key pairs with less than 2048 bits. In the case of PBE, certain JWT libraries do not provide PBE functions (e.g., java-jwt and jjwt), while jose.4.j allows users to use less than 1,000 iterations and provides an interface to specify static salts during JWT generation. In the case of key transmission, java-jwt, jose.4.j, Nimbus-JOSE-JWT and FusionAuth JWT provide APIs to obtain keys used to verify signatures from an insecure

HTTP URL. At last, jose.4.j, Nimbus-JOSE-JWT, FusionAuth JWT and Spring Security OAuth provide APIs to obtain the public key from a certificate. However, none of the JWT libraries provide certificate validity verification functions before obtaining JWT public key, leaving this burden to the developers of JWT applications.

7 Limitations and Discussion

As a static analysis detector, same with previous methods [31,40], there still exist some avenues for future improvements for *JWTKey*.

Accuracy. As a static analysis detector, *JWTKey* has several inherent limitations. Firstly, *JWTKey* currently focuses on the APIs provided by the top 7 most popular Java-based JWT libraries, which may incur false negatives in the case of invocation of API from other JWT libraries. Secondly, since the detection is based on data- and control-flow, the limited depth of clipping detection makes it unable to handle certain complex semantics (e.g., determining the final HMAC key length, certificate verification). Thirdly, *JWTKey* can only cover the data stored in program files and properties files. However, if a vulnerable cryptographic argument (e.g., key, and IV) is generated dynamically (e.g., by manual inputting), *JWTKey* cannot detect such cases.

Responsible Disclosure. We contacted 236 developers of JWT applications with cryptographic vulnerabilities to disclose all the confirmed alerts reported in Table 3. We respected the disclosure policies of the companies we contacted. Unfortunately, only 42 developers provided useful feedback on our findings. Specifically, 32 developers acknowledged and fixed the JWT key security management vulnerabilities (e.g., *R-02*, *R-03* and *R-06*) in their applications (e.g., Solon, Admin3, JWT, and Sureness). Some developers (e.g., practical-microservices-architectural-patterns) explained that they support the insecure certificate verification option (*R-11*) by the reason for simplifying implementations. 3 vulnerabilities (e.g., hard-coded key) from 3 JWT applications (i.e., Lilishop, Saas_IHRM, and CWA_Warn) have been declared as non-issue, declaring that the current implementation will not cause specific attacks on their entire systems. We also contacted the developers of JWT libraries to disclose the security findings reported in Table 4, three of them provided feedback on key length and PBE parameter restriction. Firstly, the developer of java-jwt declared that the insufficient key length in HMAC and RSA will not have a specific effect on the users. Secondly, the developer of jose.4.j approved our disclosure, and promised to add a check of iteration count. Thirdly, the developer of FusionAuth JWT explained that the responsibility to ensure secure implementations belongs to the users rather than the library, and the restrictions on key length may limit the applicability of the JWT library. We have submitted a series of CVE ID requests to disclose the acknowledged and fixed vulnerabilities (e.g., *R-03*, *R-06*, *R-11* and *R-13*) in popular JWT applications. More details of the responsible disclosure will be provided on the homepage of *JWTKey*.

8 Conclusion

We propose *JWTKey* to detect cryptographic vulnerabilities of JWT applications. *JWTKey* leverages static program slicing technique, along with 15 cryptographic rules strongly coupled with JWT key potential security threats. The evaluation results on 358 JWT applications and the comparative experiments with CryptoGuard demonstrate the effectiveness of our design. Our work highlights a lack of appreciation for the principle of key management in real-world cryptographic deployments, which brings to the surface weaknesses not only in JWT applications, but also in other cryptographic implementations. With respect to the future work, the cryptographic vulnerability detection efforts would be expanded to other typical cryptographic application areas (e.g., blockchain, PKI system, and industrial control system).

Acknowledgements. We would like to thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions. We are grateful to Prof. Juraj Somorovsky for helping us to improve our paper. This work is supported in part by National Natural Science Foundation of China No.62272457, National Key R&D Plan of China under Grant No.2020YFB1005800.

Appendix

Table 5. Partial representative slicing criterion APIs of JWT libraries and Java cryptographic libraries. (⇐: backward slicing; ⇒: inter-procedural forward slicing; ⇒∗: intra-procedural forward slicing)

No.	API	Method
1.1	java.util.Random: int nextInt(int)	⇒
1.2	java.util.Random: double nextDouble()	⇒
1.3	java.security.SecureRandom: void <init>(**byte**[])	⇐
1.4	java.security.SecureRandom: void setSeed(**byte**[])	⇐
2.1	java.security.KeyPairGenerator: KeyPairGenerator getInstance(java.lang.String)	⇒∗
2.2	java.security.KeyPairGenerator: KeyPairGenerator getInstance(String,String)	⇒∗
2.3	java.security.KeyPairGenerator: void initialize(**int**)	⇐
2.4	javax.crypto.spec.SecretKeySpec: void <init>(**byte**[],String)	⇐
4.1	org.jose4j.jwe.kdf.PasswordBasedKeyDerivationFunction2: byte[] derive(**byte**[],byte[],int,int)	⇐
4.2	com.nimbusds.jose.crypto.PasswordBasedEncrypter: void <init>(**java.lang.String**,int,int)	⇐
5.1	org.springframework.security.oauth2.provider.token.store.KeyStoreKeyFactory: void <init>(org.springframework.core.io.Resource,**char**[])	⇐
5.2	java.security.KeyStore: void load(InputStream,**char**[])	⇐
8.1	com.auth0.jwk.UrlJwkProvider: void <init>(**java.lang.String**)	⇐
8.2	org.jose4j.jwk.HttpsJwks: void <init>(**java.lang.String**)	⇐
8.3	com.nimbusds.jose.jwk.source.RemoteJWKSet: void <init>(**java.net.URL**)	⇐
8.4	io.fusionauth.jwks.JSONWebKeySetHelper: java.util.List retrieveKeysFromJWKs(**java.lang.String**)	⇐
9.1	javax.net.ssl.HostnameVerifier: boolean verify(String,SSLSession)	⇒
9.2	javax.net.ssl.SSLSocketFactory: SocketFactory getDefault()	⇒∗
10.1	com.nimbusds.jose.JWSHeader: com.nimbusds.jose.jwk.JWK getJWK()	⇐
10.2	com.nimbusds.jose.JWEHeader: java.net.URI getJWKURL()	⇐
11.1	java.security.cert.X509Certificate: void verify	⇒∗
11.2	java.security.cert.X509Certificate: void checkValidity	⇒∗
12.1	javax.crypto.spec.IvParameterSpec: void <init>(**byte**[])	⇐
12.2	javax.crypto.spec.IvParameterSpec: void <init>(**byte**[],int,int)	⇐
13.1	org.jose4j.jwe.kdf.PasswordBasedKeyDerivationFunction2: byte[] derive(byte[],byte[],**int**,int,java.lang.String)	⇐
13.2	com.nimbusds.jose.crypto.PasswordBasedEncrypter: void <init>(java.lang.String,int,**int**)	⇐
14.1	org.jose4j.jwe.kdf.PasswordBasedKeyDerivationFunction2: byte[] derive(byte[],**byte**[],int,int)	⇐
14.2	org.jose4j.jwe.kdf.PasswordBasedKeyDerivationFunction2: byte[] derive(byte[],**byte**[],int,int,java.lang.String)	⇐
15.1	com.auth0.jwt.interfaces.Verification require(com.auth0.jwt.algorithms.Algorithm)	⇒
15.2	com.nimbusds.jose.crypto.impl.RSA1_5: void <init>()	⇒

References

1. Afrose, S., Xiao, Y., Rahaman, S., Miller, B., Yao, D.D.: Evaluation of static vulnerability detection tools with java cryptographic API benchmarks. IEEE Trans. Softw. Eng. **49**, 485–497 (2022)
2. Al Rahat, T., Feng, Y., Tian, Y.: OAUTHLINT: an empirical study on OAuth bugs in android applications. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 293–304. IEEE (2019)
3. Ami, A.S., Cooper, N., Kafle, K., Moran, K., Poshyvanyk, D., Nadkarni, A.: Why crypto-detectors fail: a systematic evaluation of cryptographic misuse detection techniques. In: 2022 IEEE Symposium on Security and Privacy (SP), pp. 614–631. IEEE (2022)
4. Auth0: Java jwt. A Java implementation of JSON Web Token (JWT) - RFC 7519 (2017). https://github.com/auth0/java-jwt
5. Barker, E.: Nist special publication 800–57 part 1 revision 5, recommendation for key management, part 1-general. NIST Spec. Publ. **800–57**, 1–171 (2020)
6. Bitbucket: Welcome to jose4j (2015). https://bitbucket.org/b_c/jose4j/wiki/Home
7. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: {Nonce-Disrespecting} adversaries: practical forgery attacks on {GCM} in {TLS}. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16) (2016)
8. Cve-2022-35540 (2022). https://nvd.nist.gov/vuln/detail/CVE-2022-35540
9. Cve-2022-36672 (2022). https://nvd.nist.gov/vuln/detail/CVE-2022-36672
10. CWE-260: Password in configuration file (2022). https://cwe.mitre.org/data/definitions/260.html
11. Detering, D., Somorovsky, J., Mainka, C., Mladenov, V., Schwenk, J.: On the (in-) security of javascript object signing and encryption. In: Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium, pp. 1–11 (2017)
12. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 73–84 (2013)
13. Ethelbert, O., Moghaddam, F.F., Wieder, P., Yahyapour, R.: A JSON token-based authentication and access management schema for cloud SaaS applications. In: 2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud), pp. 47–53. IEEE (2017)
14. Fett, D., Küsters, R., Schmitz, G.: A comprehensive formal security analysis of OAuth 2.0. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1204–1215 (2016)
15. Fett, D., Küsters, R., Schmitz, G.: The web SSO standard openID connect: In-depth formal security analysis and security guidelines. In: 2017 IEEE 30th Computer Security Foundations Symposium (CSF), pp. 189–202. IEEE (2017)
16. Frantz, M., Xiao, Y., Pias, T.S., Yao, D.D.: Poster: Precise detection of unprecedented python cryptographic misuses using on-demand analysis. The Network and Distributed System Security (NDSS) Symposium (2022)
17. FusionAuth: Fusionauth jwt (2016). https://github.com/fusionauth/fusionauth-jwt
18. Ghasemisharif, M., Ramesh, A., Checkoway, S., Kanich, C., Polakis, J.: O single sign-off, where art thou? an empirical analysis of single sign-on account hijacking and session management on the web. In: 27th USENIX Security Symposium (USENIX Security 18), pp. 1475–1492 (2018)

19. Guy Rosen: Facebook security update (2018). https://about.fb.com/news/2018/09/security-update/
20. Haekal, M., et al.: Token-based authentication using JSON web token on SIKASIR restful web service. In: 2016 International Conference on Informatics and Computing (ICIC), pp. 175–179. IEEE (2016)
21. Hammer-Lahav, E.: RFC 5849: The OAuth 1.0 protocol. Tech. rep., Internet Engineering Task Force (2010)
22. Hardt, D.: RFC 6749: The OAuth 2.0 authorization framework. Tech. rep., Internet Engineering Task Force (2012)
23. Jones, M., Bradley, J., Sakimura, N.: RFC 7519: JSON web token (JWT). Tech. rep, Internet Engineering Task Force (2015)
24. Jones, M.: RFC 7518: JSON web algorithms (JWA). Tech. rep, Internet Engineering Task Force (2015)
25. Jones, M., Bradley, J., Sakimura, N.: RFC 7515: JSON web signature (JWS). Tech. rep, Internet Engineering Task Force (2015)
26. Jones, M., Hildebrand, J.: RFC 7516: JSON web encryption (JWE). Tech. rep, Internet Engineering Task Force (2015)
27. Jwtk: Jjwt. Java JWT: JSON Web Token for Java and Android (2016). https://github.com/jwtk/jjwt
28. Kleinjung, T., et al.: Factorization of a 768-bit RSA modulus. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 333–350. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_18
29. Krawczyk, H.: How to predict congruential generators. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 138–153. Springer, New York (1990). https://doi.org/10.1007/0-387-34805-0_14
30. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: CrySL: an extensible approach to validating the correct usage of cryptographic APIs. IEEE Trans. Software Eng. **47**(11), 2382–2400 (2019)
31. Li, W., Jia, S., Liu, L., Zheng, F., Ma, Y., Lin, J.: CryptoGO: automatic detection of go cryptographic API misuses. In: Annual Computer Security Applications Conference, pp. 318–331 (2022)
32. Meng, N., Nagy, S., Yao, D., Zhuang, W., Argoty, G.A.: Secure coding practices in java: challenges and vulnerabilities. In: Proceedings of the 40th International Conference on Software Engineering, pp. 372–383 (2018)
33. Michaelides, M., Sengul, C., Patras, P.: An experimental evaluation of MQTT authentication and authorization in IoT. In: WiNTECH'21: Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization, New Orleans, LA, USA, 4 February 2022, pp. 69–76. ACM (2021)
34. Mousa, A., Hamad, A.: Evaluation of the rc4 algorithm for data encryption. Int. J. Comput. Sci. Appl. **3**(2), 44–56 (2006)
35. Nimbusds: Nimbus-jose-jwt (2016). https://bitbucket.org/connect2id/nimbus-jose-jwt/src/master/
36. Oracle: Java secure socket extension (jsse) reference guide (2018). https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html#StandardAPI
37. OWASP: reverse engineering (2023). https://owasp.org/www-project-mobile-top-10/2016-risks/m9-reverse-engineering
38. Piccolboni, L., Di Guglielmo, G., Carloni, L.P., Sethumadhavan, S.: CRYLOGGER: detecting crypto misuses dynamically. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 1972–1989. IEEE (2021)

39. Rahaman, S., Cai, H., Chowdhury, O.H., Yao, D.D.: From theory to code: identifying logical flaws in cryptographic implementations in C/C++. IEEE Trans. Dependable Secure Comput. **19**, 3790–3803 (2021)
40. Rahaman, S., et al.: CryptoGuard: high precision detection of cryptographic vulnerabilities in massive-sized java projects. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2455–2472 (2019)
41. Rahat, T.A., Feng, Y., Tian, Y.: Cerberus: query-driven scalable vulnerability detection in oauth service provider implementations. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 2459–2473 (2022)
42. Sabir, B.E., Youssfi, M., Bouattane, O., Allali, H.: Authentication and load balancing scheme based on JSON token for multi-agent systems. Proced. Comput. Sci. **148**, 562–570 (2019)
43. Sakimura, N., Bradley, J., Jones, M., De Medeiros, B., Mortimore, C.: OpenID connect core 1.0. The OpenID Foundation, p. S3 (2014)
44. Sharif, A., Carbone, R., Sciarretta, G., Ranise, S.: Best current practices for OAuth/OIDC native apps: a study of their adoption in popular providers and top-ranked android clients. J. Inf. Secur. Appl. **65**, 103097 (2022)
45. Sheffer, Y., Hardt, D., Jones, M.: RFC 8725: JSON web token best current practices. Tech. rep, Internet Engineering Task Force (2020)
46. Singh, J., Chaudhary, N.K.: OAuth 2.0 : architectural design augmentation for mitigation of common security vulnerabilities. J. Inf. Secur. Appl. **65**, 103091 (2022)
47. Soot-oss: Soot - a framework for analyzing and transforming java and android applications (2022). https://soot-oss.github.io/soot/
48. Spring: Spring security OAuth (2016). https://github.com/spring-attic/spring-security-oauth
49. van der Veen, V., et al.: Drammer: deterministic Rowhammer attacks on mobile platforms. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1675–1689 (2016)
50. Varalakshmi, P., Guhan, B., Dhanush, T., Saktheeswaran, K., et al.: Improvising JSON web token authentication in SDN. In: 2022 International Conference on Communication, Computing and Internet of Things (IC3IoT), pp. 1–8. IEEE (2022)
51. Vertx: Vert.x jwt auth (2019). https://vertx.io/docs/vertx-auth-jwt/java/
52. Wang, H., Zhang, Y., Li, J., Gu, D.: The achilles heel of OAuth: a multi-platform study of OAuth-based authentication. In: Proceedings of the 32nd Annual Conference on Computer Security Applications, pp. 167–176 (2016)
53. Wang, H., et al.: Vulnerability assessment of OAuth implementations in android applications. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 61–70 (2015)
54. Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.C., Su, Z.: Exposing library API misuses via mutation analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 866–877. IEEE (2019)
55. Wickert, A.K., Baumgärtner, L., Breitfelder, F., Mezini, M.: Python crypto misuses in the wild. In: the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 1–6 (2021)
56. Xu, B., Qian, J., Zhang, X., Wu, Z., Chen, L.: A brief survey of program slicing. ACM SIGSOFT Softw. Eng. Notes **30**(2), 1–36 (2005)
57. Zhang, L., Chen, J., Diao, W., Guo, S., Weng, J., Zhang, K.: CryptoREX: large-scale analysis of cryptographic misuse in IoT devices. In: RAID, pp. 151–164 (2019)