RUHR-UNIVERSITÄT BOCHUM

# On the (in-)security of JavaScript Object Signing and Encryption

Dennis Detering

Master's Thesis – 18th November 2016.
Chair for Network and Data Security

Supervisor: Prof. Dr. Jörg Schwenk
Advisors: M. Sc. Christian Mainka, Dipl.-Ing. Vladislav Mladenov, Dr.-Ing. Juraj Somorovsky

## Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

## Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of High School.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

| | |
|---|---|
| Datum / Date | Unterschrift / Signature |

# Acknowledgements

I would like to thank many people who supported and helped me throughout the work on this thesis. First, I would like to thank the CSPi GmbH for the cooperation in writing this thesis. I learned a lot during this time and am happy to continue my employment after my graduation. A special thanks goes to my advisors at the Chair for Network and Data Security, Christian Mainka, Vladislav Mladenov and Juraj Somorovsky, for quickly responding to all of my questions and for their helpful input. Their explanations clarified complex issues and their enormous knowledge and years of practice ensured the practical relevance and necessary features of the developed Burp Suite extension. Many thanks to my proofreaders, Benedict and Alex, and to my dad Dirk for all the helpful comments and advices. Finally, another big thank you to my girlfriend Karoline for her support, patience and linguistic proofreading without even understanding a single word of the content.

*For every lock, there is someone out there trying to pick it or break it.*

— DAVID BERNSTEIN

# Abstract

JavaScript Object Signing and Encryption describes how to apply encryption and signing algorithms to JSON-based data structures. Despite their young age, the all together five new specifications have already been implemented in several major protocols, frameworks and applications. Those include Single Sign-on (SSO) protocols like OpenID Connect, the Automatic Certificate Management Environment (ACME) protocol, Apache's CXF Webservice Framework and the IBM DataPower Gateway solution.

This thesis investigates the security of these specifications, presents several practically applicable attacks on library level and introduces a newly developed Burp Suite extension to assist in performing security analyses on implementing applications. The attacks include the removal or faking of signatures to break the integrity of messages and the recovery of encrypted data containing symmetric keys to break the confidentiality of hidden contents. Apart from the attacks themselves, this thesis provides recommended countermeasures to the mentioned vulnerabilities. All libraries, which were found to be vulnerable during investigation, have been fixed in close communication with the maintainers.

KEYWORDS: JSON, JavaScript Object Signing and Encryption (JOSE), Digital Signature, Encryption, Burp Suite, Signature Exclusion, Key Confusion, Algorithm Substitution, Bleichenbacher Million Message Attack, PKCS#1 v1.5, Timing Attack

# Contents

# 1. Introduction

Many applications available on the World Wide Web rely on secure communication channels on lower layers, such as Internet Protocol Security (IPSec), Secure Sockets Layer (SSL) or Transport Layer Security (TLS). These mechanisms only provide end-to-end encryption in point-to-point scenarios, where the complete data is securely transported between two communication partners. In several more complex scenarios, like communication over untrusted third-party intermediates (proxy) or the need to only secure specific parts of a message, these technologies become insufficient. These cases demand additional security technologies on the application layer which provide the fundamental security concepts integrity, authenticity and confidentiality of arbitrary elements on message level.

In practice, the Extensible Markup Language (XML)-based secure object formats *XML Signature* and *XML Encryption* enjoy great popularity and have been adopted in many widely deployed protocols and systems, like in the Security Assertion Markup Language (SAML) [1]. The high complexity of the Extended Markup Language and its manifold features lead to an increased susceptibility to errors and potential vulnerabilities. XML parsers need to add support for namespacing, Document Type Definitions (DTD), different node types, canonicalization and more [2] which in certain scenarios might cause overhead issues, such as significant increases in processing time and the data size of the document content. In particular, in certain scenarios with constrained environments, XML-based formats do not satisfactorily fulfill the requirements. Furthermore, there exist several known attacks against XML Signatures and XML Encryption, abusing parsing issues and programming errors.

An alternative to XML-based formats is the JavaScript Object Notation (JSON) [3] format. JSON is a platform-independent data format which operates with only four primitive and two structured types [3]. The JSON data format is widely deployed and used, among other things, for databases, web services and configuration files. Its simplicity and small set of formatting rules [3] facilitate a developer's effort implementing JSON-based data structures and its lightweightness reduces network load. "With the increased usage of JSON in protocols [...] there is now a desire to offer security services which use encryption, digital signatures [and] Message Authentication Codes (MACs) algorithms, that carry their data in JSON format" [4].

This demand has been addressed by the JavaScript Object Signing and Encryption (JOSE) working group, which proposed all in all five new Request for Comments (RFC) specifications. These standards specify means of applying cryptographic mechanisms to JSON messages, in order to secure their integrity, authenticity and confidentiality.

## 1.1. Contribution

This thesis investigates the security of JavaScript Object Signing and Encryption and aims to support enthusiasts with further research. First, several vulnerabilities are analyzed for their practical exploitability and occurrence based on real world library implementations. *Signature Exclusion* and *Key Confusion* are addressing JSON Web Signature and used to force the receiving party to accept invalidly signed messages. The *Bleichenbacher Million Message Attack* is a well known vulnerability, but has not yet been investigated with a special focus on JSON Web Encryption implementations. This attack exploits specific system behavior to recover the encrypted plaintext message. *Timing attacks* on hash comparison are known security issues in basically all cryptographic implementations. It is analyzed whether implementing libraries follow best practices of cryptographic security for protection. During our investigation, the latest versions of several libraries have been identified to be vulnerable. All in all six Common Vulnerabilities and Exposures (CVE) identifiers have been assigned and all issues were fixed in cooperation with the maintainers.

Second, a Burp Suite extension has been developed, to support penetration testers and library maintainers to test implementations for their resistance of the examined attacks. Apart from the attacks, the extension provides several features to assist in manual testing and is easily extensible to add further discovered attacks and checks. During the development two difficulties in working with the Burp Suite Application Programming Interface (API) occurred. As a result, a public feature request has been created and a second one updated. Both are depicted in the Appendix A.5.

## 1.2. Related Work

There already existed some attempts in the past to develop mechanisms for JSON security on message level, all of which influenced the work on the JOSE specifications. The Google employees Panzer, Laurie and Balfanz drafted the *Magic Signatures* mechanism [5] to digitally sign nearly arbitrary messages, including JSON objects, while the Protivity Government Services and the Nomura Research Institute worked on a comparable design called *JSON Simple Sign* [6] and the related *JSON Simple Encryption* mechanism [7]. In 2009, the Microsoft employees Hardt and Goland proposed a standardization draft called *Simple Web Token* [8] to the IETF, whose ideas have been included into the JSON Web Token specification. Furthermore, certain concepts of the *JavaScript Message Security Format* were incorporated into JWS and JWE [9].

### XML Security

XML Signature and XML Encryption implementations suffered from several vulnerabilities and attacks in the past. Many of them were investigated by Juraj Somorovsky in his dissertation *On The Insecurity Of XML Security* [10] and related published papers like *How To Break XML Encryption* [11], *How To Break XML Encryption – Automatically* [12], *Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption* [13] and *One Bad Apple: Backwards Compatibility*

*Attacks on State-of-the-Art Cryptography* [14]. Further attacks on XML Signatures were researched by James Forshaw in his whitepaper *Exploiting XML Digital Signature Implementations* [15]. Due to similar supported cryptographic algorithms and use-cases, those publications were of special interest to this thesis.

### Security Research on JOSE

Tim McLean is a researcher, who focused on a security analysis of JSON Web tokens and discovered the *Signature Exclusion* and *Key Confusion* vulnerabilities [16]. Many JOSE implementations were fixed after his disclosure. Apart from the theoretical threat description, no proof-of-concept or testing tool has been published.

### Burp Suite Extension

Even if the standards for XML Signature and XML Encryption already exist for some years, the only two publicly available Burp Suite plugins appeared just recently - in the mid/end of 2015. The *SAML Raider* [17] "is a Burp Suite extension for testing SAML infrastructures [with the] two core functionalities: Manipulating SAML messages and managing X.509 certificates" [17]. It comes with a preset of common *XML Signature Wrapping* attacks to test against services. The *Extension for Processing and Recognition of Single Sign-On Protocols*, short EsPReSSO [18], focuses on Single Sign-On protocols and is the first plugin supporting the recognition and manipulation of messages containing JSON Web Tokens. At the time of writing, there exists no extension for the Burp Suite aiding security analyses of JSON Web Signature and JSON Web Encryption implementations, and offering a preset of known attacks.

## 1.3. Outline

Chapter 2 gives an overview of the JSON technology and introduces the five JOSE specifications. Additionally, it shortly describes the Burp Suite and its components, and delimits the term *library level*. These are the necessary fundamentals to understand our research and the related attacks. Chapter 3 is the main part of this thesis and explains the investigated vulnerabilities and applied attacks. Apart from the attack itself, at least one library found to be vulnerable is analyzed in detail and the resulting developed test cases are described. Each attack section finishes with possible and recommended countermeasures. Chapter 4 presents the developed Burp Suite extension by explaining the User Interface (UI) and its internal structure. The thesis concludes with a short summary and prospects for future research in Chapter 5.

# 2. Fundamentals

The following chapter introduces the concepts and specifications behind JOSE and its related JSON format. They are relevant to this thesis and provide the reader with the necessary basics to comprehend the developed test cases, understand the user interface and implementations of the Burp Suite extension and facilitate further research. Further, the Burp Suite platform and its components are shortly introduced. The last section gives a delimitation of the term *library level* as understood within this thesis. Readers familiar with these standards and technologies can safely skip this chapter.

## 2.1. JavaScript Object Notation

The JavaScript Object Notation is a "lightweight, text-based, language-independent data interchange format [...] derived from the ECMAScript Programming Language Standard" [3]. While originally specified by Douglas Crockford [19], the specification has been improved by the Internet Engineering Task Force (IETF) to an Internet Standards Track document [3]. JSON's design goals were to be minimal, portable, textual, and a subset of JavaScript and can represent four primitive types (`strings`, `numbers`, `booleans`, and `null`) and two structured types (`objects` and `arrays`) [3].

### JSON Types

The terms *string*, *object* and *array* are understood the same way as described in the specification [3]:

- A *string* is a sequence of zero or more Unicode [20] characters. Within a string, the backslash (`\`) is used as control character to escape e. g. the double quote (`"`) or the backslash itself.
- An *object* is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array.
- An *array* is an ordered sequence of zero or more values.

### Structural Characters

The specification names six structural characters. The square brackets (`[`, `]`) define the beginning and end of an array, the curly brackets (`{`, `}`) delimit an object, the colon (`:`) acts as separator in a name-value pair and the comma (`,`) separates different elements in an array or a value from a following name. Furthermore, insignificant whitespace, represented by a normal ASCII whitespace, horizontal tab (`\t`), line feed (`\n`) or carriage return (`\r`), is allowed before or after any of the six structural characters.

The following listing shows an example of a JSON array containing two objects and all previously mentioned primitive and structural types.

The *owner* member of both objects is another object itself, containing two values: A string (name) and an integer (age). The *registered* value is of the type boolean and the *favorite food* member is an array of strings.

```json
[{
    "name": "Rex",
    "species": "dog",
    "owner": {
      "Name": "Dennis Detering",
      "Age":  26
    },
    "favorite food": ["cracker", "cheese"],
    "registered": true
  },
  {
    "name": "Chang Miao",
    "species": "cat",
    "owner": {
      "Name": "John Doe",
      "Age":  null
    },
    "favorite food": ["milk"],
    "registered": false
}]
```

Listing 2.1: Example of a JSON array containing two objects

## 2.2. JavaScript Object Signing and Encryption

In May 2015, the JSON Web Signature (JWS) [21] and JSON Web Encryption (JWE) [22] mechanisms received the RFC status *Proposed Standard* by the IETF as a result of the proposals by the JavaScript Object Signing and Encryption (JOSE) working group [4]. Along with the related new RFC standard proposals JSON Web Key (JWK) [23], JSON Web Algorithm (JWA) [24] and JSON Web Token (JWT) [25][1] they build a new approach to represent secured content “intended for space constrained environments such as HTTP Authorization headers and URI query parameters [...] using JavaScript Object Notation (JSON) data structures” [26]. As a lightweight and

[1]When referring to all five RFC specifications as a group the abbreviation JOSE will be used in the following.

URL-safe[2] alternative to XML-based data structures it has early been integrated into several major protocols, frameworks and applications. Those include SSO protocols like OpenID Connect, an identity layer on top of the OAuth 2.0 protocol [27], the ACME protocol [28], exemplarily used by *Let's Encrypt* [29], *Atlassian Connect*, where it is used as an authentication layer for add-ons [30], the IBM DataPower Gateway solution [31] and Apache's CXF Webservice Framework [32].

#### Base64 vs. Base64url

Throughout the JOSE specifications, most components are represented in a `base64url` encoded format. "The Base 64 encoding is designed to represent arbitrary sequences of octets in a form that allows the use of both upper- and lowercase letters but that need not be human readable" [33, Section 4] and consists of an alphabet containing 64 characters. For using this encoding in constrained environments, such as an URL or HTTP header, the character set must not contain structural characters. Specifically, this refers to character 62 (`+`), 63 (`/`) and the padding character (`=`) of the original alphabet. Section 5 of RFC 4648 defines an alternative character set called *base 64 encoding with URL and filename safe alphabet* [33, Section 5], also shortly referred to as `base64url`. The main difference is the substitution of the `+` to the character `-` (minus) and the `/` to the character `_` (underline). Additionally, the appended padding is omitted. A short illustration of the difference is shown in Appendix A.1. Appendix C of the JWS specification gives an example of how to implement `base64url` encoding without padding [21].

### 2.2.1. JSON Web Algorithm

The JSON Web Algorithm is specified in RFC 7518 and enumerates cryptographic algorithms and identifiers represented in JSON-based data structures. Intended to be used with the JSON Web Signature, JSON Web Encryption and JSON Web Key specifications, it describes the semantics and operations that are specific to these algorithms and key types [24].

#### Cryptographic Algorithms for Digital Signatures and MACs

"JWS uses cryptographic algorithms to digitally sign or create a MAC of the contents of the JWS Protected Header and the JWS Payload" [24]. The available algorithms for use with JWS are listed in table 2.1.

#### Cryptographic Algorithms for Key Management

"JWE uses cryptographic algorithms to encrypt or determine the Content Encryption Key (CEK)" [24]. The available algorithms for use with JWE in order to encrypt the CEK, produce the JWE encrypted key or to use key agreement to agree upon the CEK are listed in table 2.2.

[2]In its *Compact Serialization* which will be described in Subsection 2.2.3.

Table 2.1.: List of available algorithms for use with JSON Web Signature including their `alg` header parameter value and implementation requirements according to RFC 7518 [24].

| `alg` Param Value | Digital Signature or MAC Algorithm | Implementation Requirements |
|---|---|---|
| `HS256` | `HMAC` using `SHA-256` | Required |
| `HS384` | `HMAC` using `SHA-384` | Optional |
| `HS512` | `HMAC` using `SHA-512` | Optional |
| `RS256` | `RSASSA-PKCS1-v1_5` using `SHA-256` | Recommended |
| `RS384` | `RSASSA-PKCS1-v1_5` using `SHA-384` | Optional |
| `RS512` | `RSASSA-PKCS1-v1_5` using `SHA-512` | Optional |
| `ES256` | `ECDSA` using `P-256` and `SHA-256` | Recommended |
| `ES384` | `ECDSA` using `P-384` and `SHA-384` | Optional |
| `ES512` | `ECDSA` using `P-521` and `SHA-512` | Optional |
| `PS256` | `RSASSA-PSS` using `SHA-256` and `MGF1` with `SHA-256` | Optional |
| `PS384` | `RSASSA-PSS` using `SHA-384` and `MGF1` with `SHA-384` | Optional |
| `PS512` | `RSASSA-PSS` using `SHA-512` and `MGF1` with `SHA-512` | Optional |
| `none` | No digital signature or `MAC` performed | Optional |

Table 2.2.: List of available algorithms for use with JSON Web Encryption in order to encrypt or determine the `CEK`, including their `alg` header parameter value and implementation requirements according to RFC 7518 [24].

| `alg` Param Value | Key Management Algorithm | Implementation Requirements |
|---|---|---|
| `RSA1_5` | `RSAES-PKCS1-v1_5` | Recommended |
| `RSA-OAEP` | `RSAES OAEP` using default parameters | Recommended |
| `RSA-OAEP-256` | `RSAES OAEP` using `SHA-256` and `MGF1` with `SHA-256` | Optional |
| `A128KW` | `AES` Key Wrap with default initial value using 128-bit key | Recommended |
| `A192KW` | `AES` Key Wrap with default initial value using 192-bit key | Optional |
| `A256KW` | `AES` Key Wrap with default initial value using 256-bit key | Recommended |
| `dir` | Direct use of a shared symmetric key as the `CEK` | Recommended |
| `ECDH-ES` | Elliptic Curve Diffie-Hellman Ephemeral Static key agreement using `Concat KDF` | Recommended |
| `ECDH-ES+A128KW` | `ECDH-ES` using `Concat KDF` and `CEK` wrapped with `A128KW` | Recommended |
| `ECDH-ES+A192KW` | `ECDH-ES` using `Concat KDF` and `CEK` wrapped with `A192KW` | Optional |
| `ECDH-ES+A256KW` | `ECDH-ES` using `Concat KDF` and `CEK` wrapped with `A256KW` | Recommended |
| `A128GCMKW` | Key wrapping with `AES GCM` using 128-bit key | Optional |
| `A192GCMKW` | Key wrapping with `AES GCM` using 192-bit key | Optional |
| `A256GCMKW` | Key wrapping with `AES GCM` using 256-bit key | Optional |
| `PBES2-HS256+A128KW` | `PBES2` with `HMAC` `SHA-256` and `A128KW` wrapping | Optional |
| `PBES2-HS384+A192KW` | `PBES2` with `HMAC` `SHA-384` and `A192KW` wrapping | Optional |
| `PBES2-HS512+A256KW` | `PBES2` with `HMAC` `SHA-512` and `A256KW` wrapping | Optional |

#### Cryptographic Algorithms for Content Encryption

"JWE uses cryptographic algorithms to encrypt and integrity-protect the plaintext and to integrity-protect the Additional Authenticated Data" [24]. The available algorithms for use with JWE in order to encrypt and integrity-protect the content are listed in table 2.3.

Table 2.3.: List of available algorithms for use with JSON Web Encryption in order to encrypt and integrity-protect the content, including their `enc` header parameter value and implementation requirements according to RFC 7518 [24].

| `enc` Param Value | Content Encryption Algorithm | Implementation Requirements |
|---|---|---|
| `A128CBC-HS256` | `AES_128_CBC_HMAC_SHA_256` authenticated encryption algorithm | Required |
| `A192CBC-HS384` | `AES_192_CBC_HMAC_SHA_384` authenticated encryption algorithm | Optional |
| `A256CBC-HS512` | `AES_256_CBC_HMAC_SHA_512` authenticated encryption algorithm | Required |
| `A128GCM` | `AES GCM` using 128-bit key | Recommended |
| `A192GCM` | `AES GCM` using 192-bit key | Optional |
| `A256GCM` | `AES GCM` using 256-bit key | Recommended |

### 2.2.2. JSON Web Key

JSON Web Key is specified in RFC 7517 and represents a cryptographic key in a JSON data structure as used in the JWS and JWE specifications. Additionally, a *JWK Set* is defined as a set of JWKs to represent multiple keys in a single JSON object [23]. The specification lists and describes several parameters registered in the Internet Assigned Numbers Authority (IANA) registry for use with JWKs. Four of them, which are mainly used, are explained in the following. Refer to Table A.3 in the appendix for a full list and to [23] for full details.

- The `kty` (key type) parameter which "identifies the cryptographic algorithm family used with the key, such as RSA or EC" [23, Section 4.1]. This is the only *required* parameter to be present in a JWK.

- The `use` (public key use) parameter which "identifies the intended use of the public key [and] is employed to indicate whether a public key is used for encrypting data or verifying the signature on data" [23, Section 4.2]. This parameter is *optional*. Specified values are: `sig` (signature) and `enc` (encryption).

- The `alg` (algorithm) parameter which "identifies the algorithm intended for use with the key" [23, Section 4.4]. This parameter is *optional*.

- The `kid` (key ID) parameter which "is used to match a specific key [and] is used, for instance, to choose among a set of keys within a JWK Set during key rollover" [23, Section 4.5]. The structure of the value is not specified. This parameter is *optional*.

Listing 2.2 shows an example of a JWK Set containing three different cryptographic keys represented as JWKs. One symmetric key designated as being for use with the AES key wrap algorithm, a public key using an elliptic curve algorithm and a second public key using an RSA algorithm.

```
{
  "keys":
  [{
    "kty":"oct",
    "alg":"A128KW",
    "k":"GawgguFyGrWKav7AX4VKUg"
  },
  {
    "kty":"EC",
    "crv":"P-256",
    "x":"MKBCTNIcKUSDii11ySs3526iDZ8AiTo7Tu6KPAqv7D4",
    "y":"4Etl6SRW2YiLUrN5vfvVHuhp7x8PxltmWWlbbM4IFyM",
    "use":"enc",
    "kid":"1"
  },
  {
    "kty":"RSA",
    "n": "0vx7agoebGcQSuuPiLJXZptN9nndrQmbXEps2aiAFbWhM78LhWx
          4cbbfAAtVT86zwu1RK7aPFFxuhDR1L6tSoc_BJECPebWKRXjBZCiFV4n3oknjhMs
          tn64tZ_2W-5JsGY4Hc5n9yBXArwl93lqt7_RN5w6Cf0h4QyQ5v-65YGjQR0_FDW2
          QvzqY368QQMicAtaSqzs8KJZgnYb9c7d0zgdAZHzu6qMQvRL5hajrn1n91CbOpbI
          SD08qNLyrdkt-bFTWhAI4vMQFh6WeZu0fM4lFd2NcRwr3XPksINHaQ-G_xBniIqb
          w0Ls1jF44-csFCur-kEgU8awapJzKnqDKgw",
    "e":"AQAB",
    "alg":"RS256",
    "kid":"2011-04-29"
  }]
}
```

Listing 2.2: Example JWK Set containing one symmetric and two public keys represented as JWKs[3]

### 2.2.3. JSON Web Signature

JSON Web Signature is specified in RFC 7515 and represents integrity protected content secured with digital signatures or MACs using JSON-based data structures [21]. There are several registered header parameter names defined in the specification, a full list is printed in Table A.2 in the appendix. The only *required* of them is the `alg` (algorithm) parameter, used to identify the applied cryptographic algorithm to secure the JWS.

[3]Examples taken from RFC 7517 [23]

### JWS Header Types

The JWS specification differentiates between three types of headers: *JOSE Header*, *JWS Protected Header* and *JWS Unprotected Header*. The *JOSE Header* is a "JSON object containing the parameters describing the cryptographic operations and parameters employed" [21, Section 2]. The *JWS Protected Header* is a JSON object containing the header parameter which is integrity protected by the JWS signature. By using the *JWS Unprotected Header*, no integrity protection is used for the header parameter. A *JWS Unprotected Header* can only be present if the *JWS JSON Serialization* is used. Within the *JWS Compact Serialization* only one header is present, which means in this case, the *JOSE Header* and the *JWS Protected Header* are the same, and the integrity protection comprises the entire header.

The difference between both serializations is explained in the following section.

### JWS Serialization

The JWS specification defines two types of serialization to represent a JWS. The *JWS Compact Serialization* is a compact and URL-safe string representation and supports only a single signature or MAC. As already depicted in the previous section, this type of serialization does not provide any syntax to contain a *JWS Unprotected Header* and treats the *JOSE Header* and the *JWS Protected Header* as being equal. One example is depicted in Listing 2.3, showing the three `base64url`-encoded and concatenated result strings. Basically all samples used in this thesis are shown in its *JWS Compact Serialization* representation.

```
eyJhbGciOiJSUzI1NiJ9                                                # Header
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt      # Payload
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7      # Signature
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB-f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO-xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

Listing 2.3: JSON Web Signature in its JWS Compact Serialization representation[4]

The second type is called *JWS JSON Serialization* and further differentiates between a *general* and a *flattened* variant. The *JWS JSON Serialization* is "a representation of the JWS as a JSON object [and] enables multiple digital signatures and/or MACs to be applied to the same content" [21, Section 2]. Unlike the *JWS Compact Serialization*, "this representation is neither optimized for compactness nor URL-safe" [21, Section 2]. Listing 2.4 presents an example using the *general*

[4]Example taken from [21, Appendix A.2.1]

JWS JSON Serialization syntax and “demonstrates the capability for conveying multiple digital signatures and/or MACs for the same payload” [21, Appendix A.6]. The first digital signature has been generated with the RSA algorithm and the second one by using ECDSA.

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
              tcGxlLmNvbS9pc19yb290IjpOcnVlfQ",
  "signatures":[{
    "protected": "eyJhbGciOiJSUzI1NiJ9",
    "header": {"kid":"2010-12-29"},
    "signature": "cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZ
                  mh7AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjb
                  KBYNX4BAynRFdiuB-f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHl
                  b1L07Qe7K0GarZRmB_eSN9383LcOLn6_dO-xi12jzDwusC-eOkHWEsqtFZES
                  c6BfI7noOPqvhJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AX
                  LIhWkWywlVmtVrBp0igcN_IoypGlUPQGe77Rw"
  },{
    "protected": "eyJhbGciOiJFUzI1NiJ9",
    "header": {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
    "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
                  lSApmWQxfKTUJqPP3-Kg6NU1Q"
  }]
}
```

Listing 2.4: JSON Web Signature in its General JWS JSON Serialization representation[5]

Listing 2.5 illustrates an example using the *flattened* JWS JSON Serialization syntax which, as the name implies, reduces the amount of nesting within the JSON object and does not allow for multiple signatures or MACs.

```
{
  "payload": "eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGF
              tcGxlLmNvbS9pc19yb290IjpOcnVlfQ",
  "protected": "eyJhbGciOiJFUzI1NiJ9",
  "header": {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
  "signature": "DtEhU3ljbEg8L38VWAfUAqOyKAM6-Xx-F4GawxaepmXFCgfTjDxw5djxLa8IS
                lSApmWQxfKTUJqPP3-Kg6NU1Q"
}
```

Listing 2.5: JSON Web Signature in its Flattened JWS JSON Serialization representation[6]

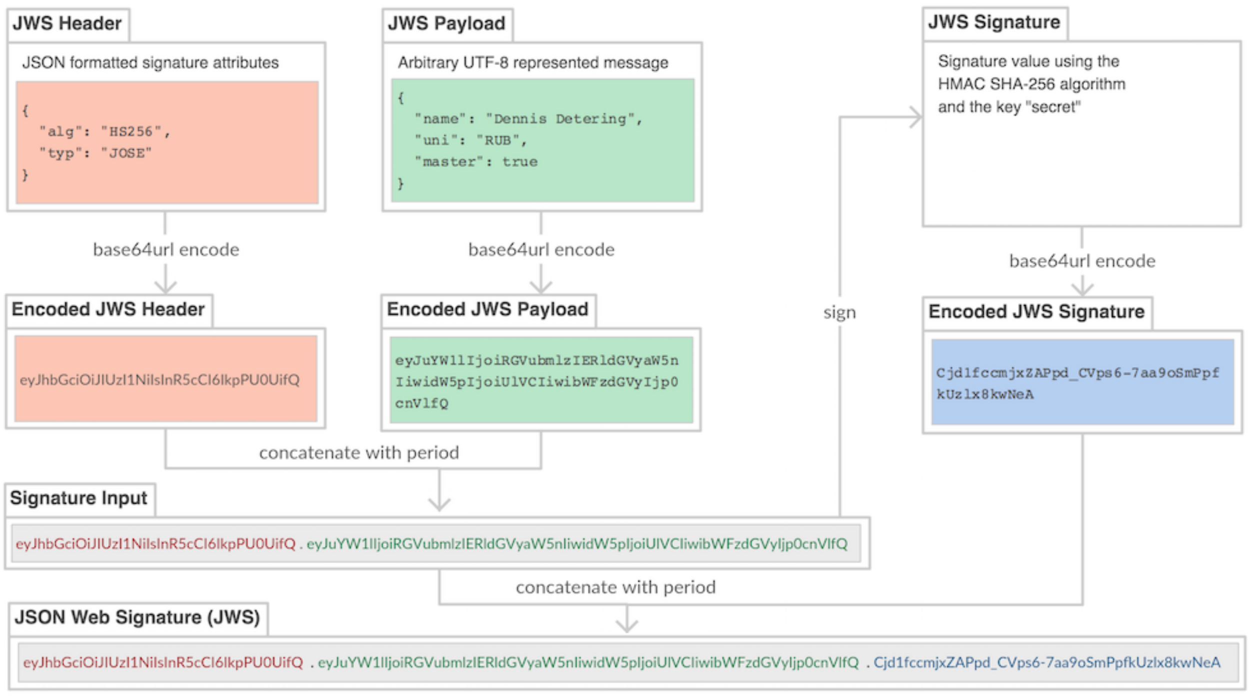
[5]Example taken from [21, Appendix A.6.4]
[6]Example taken from [21, Appendix A.7]

### Signature Computation

To create a JWS, the following steps have to be performed. “The order of the steps is not significant in cases where there are no dependencies between the inputs and outputs of the steps” [21, Section 5.1]. Some steps slightly vary if the JWS JSON Serialization is used. A graphical illustration of the generation process is shown in Figure 2.1.

**Steps:**

1. Create the JSON object containing the desired header parameters and compute the encoded header value by using `BASE64URL(UTF8(JWS Protected Header))` *(red)*
2. Compute the encoded payload value by using `BASE64URL(JWS Payload)` *(green)*
3. Compute the JWS signature in the manner defined for the particular algorithm by using the previously generated encoded values, concatenated with a dot, as input:
   `ASCII(BASE64URL(UTF8(JWS Protected Header)) || ’.’ || BASE64URL(JWS Payload))` *(white)*
4. Compute the encoded signature value by using `BASE64URL(JWS Signature)` *(blue)*
5. Create the JWS Compact Serialization output by concatenating the three encoded values with a dot: `BASE64URL(UTF8(JWS ProtectedHeader)) || ’.’ || BASE64URL(JWS Payload) || ’.’ || BASE64URL(JWS Signature)`



```
{
  "alg": "HS256",
  "typ": "JOSE"
}
```

```
{
  "name": "Dennis Detering",
  "uni": "RUB",
  "master": true
}
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpPU0UifQ
```

```
eyJuYW1lIjoiRGVubmlzIERldGVyaW5n
IiwidW5pIjoiUlVCIiwibWFzdGVyIjp0
cnVlfQ
```

```
Cjd1fccmjxZAPpd_CVps6-7aa9oSmPpf
kUzlx8kwNeA
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpPU0UifQ . eyJuYW1lIjoiRGVubmlzIERldGVyaW5nIiwidW5pIjoiUlVCIiwibWFzdGVyIjp0cnVlfQ
```

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpPU0UifQ . eyJuYW1lIjoiRGVubmlzIERldGVyaW5nIiwidW5pIjoiUlVCIiwibWFzdGVyIjp0cnVlfQ . Cjd1fccmjxZAPpd_CVps6-7aa9oSmPpfkUzlx8kwNeA
```

Figure 2.1.: Process of generating a JSON Web Signature

### 2.2.4. JSON Web Encryption

JSON Web Encryption is specified in RFC 7516 and provides authenticated encryption to ensure the confidentiality and integrity of an arbitrary sequence of octets using JSON-based data structures [22]. The registered header parameters defined by the specification are listed in Table A.3 in the appendix. For JWE, there are two *required* parameters to be implemented for compliance, namely the `alg` (algorithm) parameter and the `enc` (encryption algorithm) parameter. The `alg` parameter identifies the cryptographic algorithm or method used to transmit the value of the CEK, while the `enc` parameter holds the identifier of the content encryption algorithm used to perform authenticated encryption on the plaintext [22].

#### Basic Structure

The basic structure of a JWE consists of six logical components:

1. The *JOSE Header*, a JSON object containing the cryptographic meta-data, just like described in the previous section 2.2.3 for the JWS header types.
2. The *JWE Encrypted Key* which is the value of the Content Encryption Key, if necessary as defined by the used algorithm and encrypted with the key encryption algorithm specified in the `alg` header parameter [22].
3. The *JWE Initialization Vector* which is the Initialization Vector (IV) used for the encryption of the plaintext, for specific algorithms requiring an IV [22].
4. The *JWE Additional Authenticated Data* which is an integrity protected but not encrypted input to an AEAD[7] operation [22]. This might, for instance, be the *JWE Protected Header*.
5. The *JWE Ciphertext* which is the ciphertext value resulting from the authenticated encryption.
6. The *JWE Authentication Tag* which is one of the two outputs of an AEAD operation and used to ensure the integrity of the inputs [22]. Only available with specific algorithms.

#### JWE Serialization

The JWE specification defines two types of serialization which are closely related to the serializations for JWS. A compact variant for constrained environments, called *JWE Compact Serialization*, and the *JWE JSON Serialization* which represents JWEs as JSON objects and allows for encryption

[7]An Authenticated Encryption with Associated Data (AEAD) algorithm "is one that encrypts the plaintext, allows Additional Authenticated Data to be specified, and provides an integrated content integrity check over the ciphertext and Additional Authenticated Data. AEAD algorithms accept two inputs, the plaintext and the Additional Authenticated Data value, and produce two outputs, the ciphertext and the Authentication Tag value" [22, Section 2]. One of such algorithms is the AES Galois/Counter Mode (GCM).

of the same content to multiple parties [22]. Listing 2.6 depicts one example of a JWE in its *JWE Compact Serialization* representation, showing the five `base64url`-encoded and concatenated result strings. In this example, taken from [22, Appendix A.2.7], the plaintext "Live long and prosper." is encrypted using the *RSA PKCS#1 v1.5* algorithm for key encryption and *AES-128 CBC with HMAC SHA-1* for the content encryption.

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0                  # Header
.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOK1nN94nHPoltGRhWhw7Zx0-kFm      # Encrypted Key
1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIF
NPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0_F8
rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv
-B3oWh2TbqmScqXMR4gp_A
.
AxY8DCtDaGlsbGljb3RoZQ                                                # IV
.
KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY                           # Ciphertext
.
9hH0vgRfYgPnAHOd8stkvw                                                # Auth Tag
```

Listing 2.6: JSON Web Encryption in its JWE Compact Serialization representation[8]

### Message Encryption

The process of generating a JWE strongly depends on the used algorithms and contains up to 19 steps, described in detail in [22, Section 5.1]. One example of how a JWE might be generated is graphically illustrated in Figure 2.2. The first step is to create the JSON objects building the header(s) and encode them using `base64url` *(red)*. Second, the value of the CEK is determined which might include random generating, computing based on key agreement methods, asymmetrically or symmetrically encrypting to the recipient or simply directly adding the shared key or the empty octet sequence [22] *(purple)*. For JWE, there exist five specified *Key Management Modes*, defining how the CEK is determined and transferred:

1. *Key Encryption*, "a Key Management Mode in which the CEK value is encrypted to the intended recipient using an asymmetric encryption algorithm" [22, Section 2].

2. *Key Wrapping*, "a Key Management Mode in which the CEK value is encrypted to the intended recipient using a symmetric key wrapping algorithm" [22, Section 2].

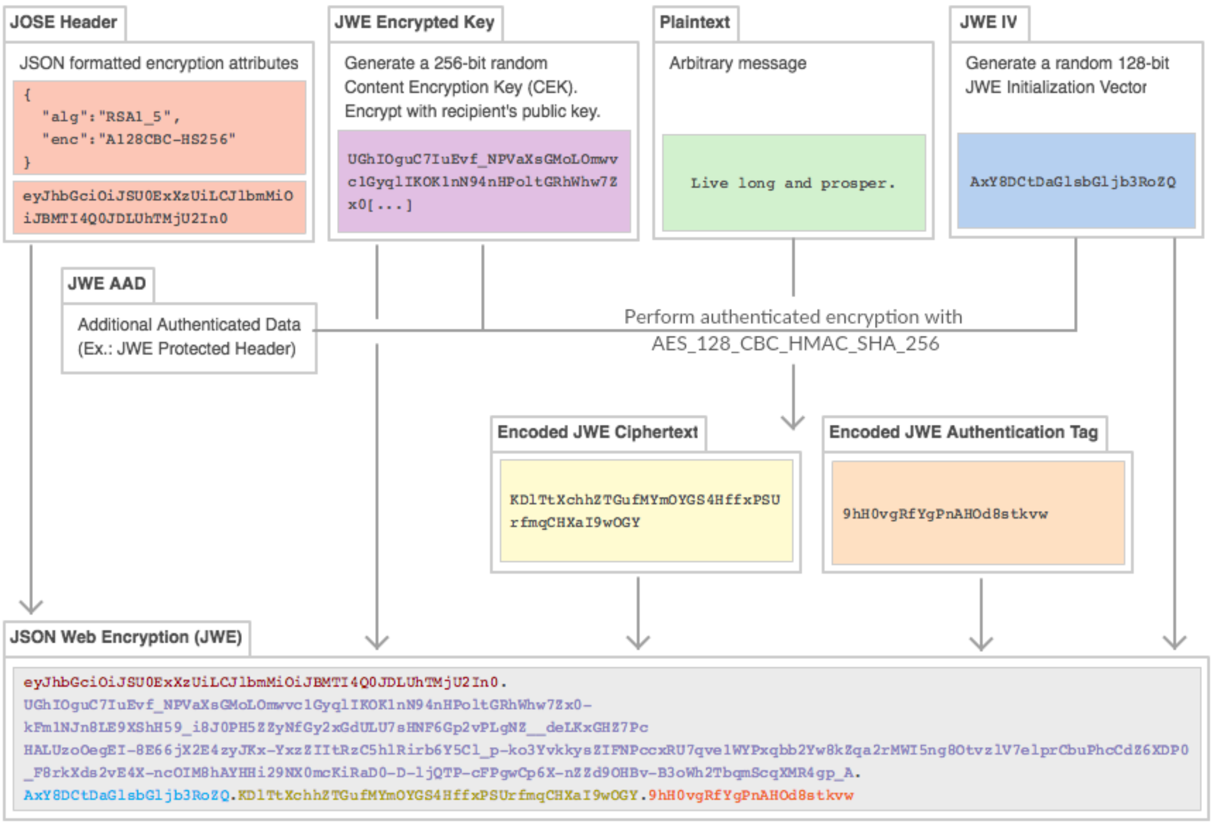
3. *Direct Key Agreement*, "a Key Management Mode in which a key agreement algorithm is used to agree upon the CEK value" [22, Section 2].

[8]Example taken from [22, Appendix A.2.7]

4. *Key Agreement with Key Wrapping*, "a Key Management Mode in which a key agreement algorithm is used to agree upon a symmetric key used to encrypt the CEK value to the intended recipient using a symmetric key wrapping algorithm" [22, Section 2].

5. *Direct Encryption*, "a Key Management Mode in which the CEK value used is the secret symmetric key value shared between the parties" [22, Section 2].

Next, the CEK value is encoded using `base64url`. After that, if required for the specific algorithm, an IV is randomly generated and `base64url` encoded *(blue)*. If compression is enabled, the plaintext will be compressed using the selected algorithm.


JOSE Header
JSON formatted encryption attributes
JWE Encrypted Key
Generate a 256-bit random Content Encryption Key (CEK). Encrypt with recipient's public key.
Plaintext
Arbitrary message
JWE IV
Generate a random 128-bit JWE Initialization Vector
JWE AAD
Additional Authenticated Data (Ex.: JWE Protected Header)
Perform authenticated encryption with AES_128_CBC_HMAC_SHA_256
Encoded JWE Ciphertext
Encoded JWE Authentication Tag
JSON Web Encryption (JWE)

```
{
  "alg":"RSA1_5",
  "enc":"A128CBC-HS256"
}
```

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiO
iJBMTI4Q0JDLUhTMjU2In0
```

```
UGhIOguC7IuEvf_NPVaXsGMoLOmwv
c1GyqlIKOKlnN94nHPoltGRhWhw7Z
x0[...]
```

```
Live long and prosper.
```

```
AxY8DCtDaGlsbGljb3RoZQ
```

```
KDlTtXchhZTGufMYmOYGS4HffxPSU
rfmqCHXaI9wOGY
```

```
9hH0vgRfYgPnAHOd8stkvw
```

```
eyJhbGciOiJSU0ExXzUiLCJlbmMiOiJBMTI4Q0JDLUhTMjU2In0.
UGhIOguC7IuEvf_NPVaXsGMoLOmwvc1GyqlIKOKlnN94nHPoltGRhWhw7Zx0-
kFm1NJn8LE9XShH59_i8J0PH5ZZyNfGy2xGdULU7sHNF6Gp2vPLgNZ__deLKxGHZ7Pc
HALUzoOegEI-8E66jX2E4zyJKx-YxzZIItRzC5hlRirb6Y5Cl_p-ko3YvkkysZIFNPccxRU7qve1WYPxqbb2Yw8kZqa2rMWI5ng8OtvzlV7elprCbuPhcCdZ6XDP0
_F8rkXds2vE4X-ncOIM8hAYHHi29NX0mcKiRaD0-D-ljQTP-cFPgwCp6X-nZZd9OHBv-B3oWh2TbqmScqXMR4gp_A.
AxY8DCtDaGlsbGljb3RoZQ.KDlTtXchhZTGufMYmOYGS4HffxPSUrfmqCHXaI9wOGY.9hH0vgRfYgPnAHOd8stkvw
```

Figure 2.2.: Process of generating a JSON Web Encryption

The plaintext value *(green)* is encrypted using the CEK and IV values of the previous steps, together with the Additional Authenticated Data (AAD) *(white)*, if present. The encryption is based on the selected content encryption algorithm and outputs the ciphertext *(yellow)* and an authentication tag *(orange)*. These values are all encoded using `base64url` and compound to the desired serialization output.

### 2.2.5. JSON Web Token

JSON Web Token is specified in RFC 7519 and used to transfer claims[9] in a compact, URL-safe representation using the *JWS/JWE Compact Serialization* between two parties [25]. “The claims in a JWT are encoded as a JSON object that is used as the payload of a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or integrity protected with a Message Authentication Code (MAC) and/or encrypted” [25, Abstract].

The specification defines seven *Claim Names* that are registered in the IANA registry. None of them is intended to be mandatory to use or implement, but rather seen as “a starting point for a set of useful, interoperable claims” [25, Section 4.1].

- The `iss` (issuer) claim, which “identifies the principal that issued the JWT” [25, Section 4.1.1].
- The `sub` (subject) claim, which “identifies the principal that is the subject of the JWT, [as] claims in a JWT are normally statements about the subject” [25, Section 4.1.2].
- The `aud` (audience) claim, which “identifies the recipients that the JWT is intended for. [...] If the principal processing the claim does not identify itself with a value in the `aud` claim when this claim is present, then the JWT MUST be rejected” [25, Section 4.1.3].
- The `exp` (expiration time) claim, which “identifies the expiration time on or after which the JWT MUST NOT be accepted for processing” [25, Section 4.1.4].
- The `nbf` (not before) claim, which “identifies the time before which the JWT MUST NOT be accepted for processing” [25, Section 4.1.5].
- The `iat` (issued at) claim, which “identifies the time at which the JWT was issued [...] can be used to determine the age of the JWT” [25, Section 4.1.6].
- The `jti` (JWT ID) claim, which “provides a unique identifier for the JWT” [25, Section 4.1.7] and can be used to protect against replay attacks.

As the claims of a JWT are either used as payload of a JWS or plaintext of a JWE, the generation and verification processes follow the exact same steps as described in the previous sections. Listing 2.7 shows an exemplary payload containing six claims that are used to determine a user’s session and administrative role.

[9] A *claim* is “a piece of information asserted about a subject [and] is represented as a name/value pair consisting of a *Claim Name* and a *Claim Value*” [25, Section 2]

```json
{
  "iss": "dety.eu",
  "iat": 1478452995,
  "nbf": 1478452935,
  "exp": 1483228800,
  "username": "ddety",
  "is_admin": true
}
```

Listing 2.7: Example of a JWT payload containing six claims

## 2.3. Burp Suite

Burp Suite is "an integrated platform for performing security testing of web applications" [34] developed by PortSwigger Ltd. and enjoys great popularity amongst security testers. It allows for a combination of advanced manual and fully automated security testing of web-based services and is extensible through a built-in modular plugin system. Burp Suite consists of the following key components:

**Intercepting Proxy** "An intercepting proxy, which [enables a user to] inspect and modify traffic between [the] browser and the target application" [34] by operating as a man-in-the-middle. The proxy tool gives full control over any request with the ability to *forward*, *modify* or *drop* them. Apart from listing interesting details, such as the response *length*, *status code*, *MIME type*, etc., the *HTTP history* enables the user to review all previously recorded requests and responses.

**Spider** "An application-aware spider, for crawling content and functionality" [34]. This tool aims to aid in the reconnaissance of a test by passively compiling a list of URLs found in HTTP responses, thus creating a comprehensive site map of the target's application, with the additional information of its actual reachability.

**Web Application Scanner** "An advanced web application scanner, for automating the detection of numerous types of vulnerability" [34]. The vulnerability scanner is only available in the *professional* version of the Burp Suite and actively or passively scans the target for a large list of known security issues[10].

**Intruder** "An intruder tool, for performing powerful customized attacks to find and exploit unusual vulnerabilities" [34]. Its main characteristic is to inspect specific entry points, such as parameters or headers, by performing, for instance, brute force, fuzzing[11] or enumeration checks.

[10]See: `https://portswigger.net/KnowledgeBase/Issues/` for a full list of Burp Suite's issue types.

[11]"Fuzz testing or Fuzzing is a Black Box software testing technique, which basically consists in finding implementation bugs using malformed/semi-malformed data injection in an automated fashion." [35]

**Repeater** "A repeater tool, for manipulating and resending individual requests" [34]. With this tool, the user is able to easily test for replay attacks or manually manipulate certain parts of a request.

**Sequencer** "A sequencer tool, for testing the randomness of session tokens" [34]. By collecting a list of samples for session, CSRF or other security-relevant tokens, this tool estimates the degree of randomness and analyzes its quality – including the standard FIPS[12] tests.

**Extender** An extension API "allowing [one] to easily [develop custom] plugins, to perform complex and highly customized tasks within Burp" [34]. Burp Suite in general is closed source and only exposes certain interfaces and functions for public access[13]. Its functionality can be extended with plugins developed in Java, as Burp Suite itself, in Python using JPython or Ruby using JRuby. Developers have the opportunity to publish their bundled extensions to the *BApp Store*[14], Burp Suite's own application store containing extensions written by its community.

## 2.4. Library Level

All tests of the attacks against implementations of the JOSE specifications explained in this thesis have been performed on the library level.

Apart from auditing the publicly available source codes, a minimal testing environment has been set up as HTTP wrapper to process incoming requests, call the corresponding functions of the JOSE library to be tested and generate a JSON-formatted response. The necessary configurations, function calls and arguments have been taken from the library's documentation or examples. For Python libraries the microframework *Flask*[15] and for PHP libraries a combination of the web server *nginx* and the *PHP-FPM*[16] FastCGI implementation are used. With this setup, the developed Burp Suite extension could be tested and the practical applicability of the attacks could be verified.

Limiting the test setup this way allowed for a controlled context, as higher level services, frameworks or applications, like e.g. OpenID Connect or Atlassian Connect, usually perform additional operations on received messages and may have specific countermeasures in place.

[12]Federal Information Processing Standards (FIPS) are standards and guidelines developed by the National Institute of Standards and Technology (NIST) [36].
[13]See `https://portswigger.net/burp/extender/api/index.html` for Burp Suite's extender API documentation
[14]URL: `https://portswigger.net/bappstore/`
[15]URL: `http://flask.pocoo.org/`
[16]URL: `https://php-fpm.org/`

# 3. Attacks

The following chapter describes attacks against JWT, JWS and JWE. For each attack, the basic idea and underlying problem is explained first. Afterwards, at least one specific library is analyzed in order to understand how such vulnerabilities occur and look like in real-world implementations, following an explanation of the developed test cases for the Burp Suite extension. Each section finishes with possible countermeasures to mitigate the attack. All tests of the attacks and source code audits were performed at the library level and not against actual implementations in applications.

## 3.1. Signature Exclusion

*Signature Exclusion* is an attack, where an adversary is able to remove the signature of a signed message and to trick the application into falsely accepting this message as valid. The JWA specification [24] defines the algorithm type *none*, intended for use in "contexts where the payload is secured by means other than a digital signature or MAC value, or need not be secured" [24]. These so called *Unsecured JWSs* are of the exact same format as other JWSs, with the only difference of using the *empty octet sequence* as its JWS Signature value [24]. Up to and including draft 15 of the RFC 7518 specification [37], the *none* algorithm was one of the two required algorithms to be implemented for compliance.

It was first discovered by Tim McLean that many libraries do not adequately check if *Unsecured JWSs* are allowed, and that they treat them as a valid token with a correct signature [16]. An attacker might easily abuse this to craft a valid JWS or JWT with arbitrary content by replacing the `alg` header value with *none* and removing the signature, thus performing arbitrary actions on a system or impersonating other users.

### 3.1.1. Library Analysis

To understand how such a vulnerability occurs and how to build appropriate test cases for the Burp Suite extension, old versions of the PHP JOSE library by Alessandro Nadalin (aka Namshi) [38] have been analyzed, which are known to be vulnerable against Signature Exclusion.

In order to check whether a given algorithm is supported and to load its related class, the PHP JOSE library relies on the mandatory `alg` header value and checks the existence of a class with this name on a certain location using namespace definition. Listing 3.1 shows the implementation of the `getSigner()` function, which performs this check (l. 4), and will return a new instance if the class exists (l. 5). If not, an `InvalidArgumentException` is thrown, stating that the given algorithm is not supported (l. 7).

```
1 protected function getSigner() {
2     $signerClass = sprintf('Namshi\\JOSE\\Signer\\%s\\%s',
3         $this->encryptionEngine, $this->header['alg']);
4     if (class_exists($signerClass)) {
5         return new $signerClass();
6     }
7     throw new InvalidArgumentException(sprintf("The algorithm '%s' is not
          supported for %s", $this->header['alg'], $this->encryptionEngine));
8 }
```

Listing 3.1: PHP JOSE `getSigner()` function (version 2.1.3)[1].

With version 2.1.3, an `allowUnsecure` flag has been introduced and set to `false` by default (Listing 3.2, l. 2) in order to mitigate any unexpected use of an Unsecured JWS. An additional condition checks whether the algorithm value of the header is `None` and if yes, whether `allowUnsecure` is permitted (Listing 3.2, ll. 4-6).

```
1 - public static function load($jwsTokenString)
2 + public static function load($jwsTokenString, $allowUnsecure = false)
3 [...]
4 + if ($header['alg'] === 'None' && !$allowUnsecure) {
5 +   throw new InvalidArgumentException(sprintf('The token "%s" cannot be
    validated in a secure context, as it uses the unallowed "none" algorithm',
    $jwsTokenString));
6 + }
```

Listing 3.2: Commit diff excerpt of the PHP JOSE library showing the changes to disable the *none* algorithm by default[2].

The problem with this amendment is how the algorithm value is checked: Apart from the fact that `None` is not the correct spelling as defined by the specification – that would be all lowercase: `none` – an attacker might use different capitalization to bypass this check. The algorithm value is used to verify that a class with this name exists by using the `class_exists()` function and creating an instance with the help of its namespace definition (Listing 3.1, ll. 3-5). The class name though "is matched in a case-insensitive manner"[3], leaving this version of the library open to the signature exclusion vulnerability even with the introduced `allowUnsecure` flag.

This issue has been detected and fixed with version 5.0.2 by using the native `strtolower()` function to perform a case-insensitive check of the algorithm value. Listing 3.3 shows the diff of the related file.

[1]See: https://github.com/namshi/jose/blob/master/src/Namshi/JOSE/JWS.php for full file.
[2]See: https://github.com/namshi/jose/commit/127b4415e66d89b1fcfb5a07933db0b5ff5cd636 for full commit.
[3]PHP.net, *class_exists function*, URL: http://php.net/manual/en/function.class-exists.php

```
-  if ($header['alg'] === 'None' && !$allowUnsecure) {
+  if (strtolower($header['alg']) === 'none' && !$allowUnsecure) {
```

Listing 3.3: Commit diff excerpt of the PHP JOSE library showing the changes to fix the case-sensitivity of the algorithm value[4].

### 3.1.2. Test Cases

Testing an implementation for signature exclusion is quite straightforward and does not require additional input from the tester. Only a single original message is required without any further prerequisites. Based on the original message, the signature value is removed to fulfill the *empty octet sequence* requirement and the `alg` value is modified to the *none* algorithm in four spelling variations – the payload and other header values remain unchanged. Based on the JWA specification in RFC 7518 [24] and the analyzed PHP JOSE library by Alessandro Nadalin [38], the following four spelling variations have been chosen: `None`, `none`, `NONE`, `nOnE` (Table 3.1).

Table 3.1.: Available vectors for the Signature Exclusion attack

| Vector | Action |
|---|---|
| Lowercase | Replace algorithm header value with: `none`; Remove signature string. |
| Capitalized | Replace algorithm header value with: `None`; Remove signature string. |
| Uppercase | Replace algorithm header value with: `NONE`; Remove signature string. |
| Mixed | Replace algorithm header value with: `nOnE`; Remove signature string. |

### 3.1.3. Countermeasures

Due to the fact that the *none* algorithm has intentionally been added to the JWA specification to support Unsecured JWSs in certain scenarios, it is necessary for full compliance to support this algorithm and implement other mechanism to prevent any abuse. The *none* algorithm has been introduced in draft 01 of the RFC 7518 and was one of the two required algorithms to be implemented for compliance up until draft 15. Since draft 16, the *none* algorithm is set to being *optional*. The *Unsecured JWS Security Considerations* section of the JWA specification [24, section 8.5] provides two possible example means of prevention:

1. Add an additional boolean `acceptUnsecured` parameter to the `verify()` method of the library to indicate whether the *none* algorithm is an acceptable algorithm to use. This approach has been implemented by the analysed PHP JOSE library of the previous section.

2. Amend the `verify()` method of the library to take a list of acceptable algorithms as an additional parameter and reject all JWS values that are not listed.

[4]See: `https://github.com/namshi/jose/commit/be2db86f5224cc7d34ef98f9a315c6b45bc4fc4e` for full commit.

Apart from these two means of prevention named in the RFC, two other approaches have been discovered by evaluating different implementations of the JOSE specifications:

3. Every algorithm related function is individually implemented in its own class according to the specific algorithm. In case of the *none* algorithm, the `verify()` function always returns false, with the intention in mind that if *Unsecured JWSs* are used on purpose, there is no need to validate any signature. This approach is, for example, taken by the *PyJWT*[5] python library by José Padilla.

4. A similar approach, taken for example by the *node-jsonwebtoken*[6] library by Auth0, Inc., would be to throw an error on verification for tokens using the *none* algorithm or not containing a signature, *if* a secret key was provided.

All of the given approaches are satisfactory to circumvent the explained *Signature Exclusion*, but do not solve the underlying problem that attackers control the choice of the algorithm. This leaves implementations open to other possible pitfalls, one of which is examined in the following section.

## 3.2. Key Confusion

*Key Confusion*, also known as *Algorithm Substitution*, is an attack, where an adversary is able to trick the application into using a specific known cryptographic key for an unexpected algorithm. This is problematic in cases where both, symmetric and asymmetric, algorithms are supported. Symmetric algorithms use a shared secret to sign a given message and verify its related signature, whereas asymmetric algorithms use a secret private key to generate a signature and the corresponding (published) public key to verify its validity. The JWA specification defines four different cryptographic algorithms with different key sizes for digital signatures and MACs. The only symmetric Keyed-Hash Message Authentication Code (HMAC) algorithm is *required* for compliant implementations and the asymmetric algorithms based on RSA and ECDSA are *recommended*, thus the probability of symmetric and asymmetric algorithms being implemented (and used) alongside is considered realistic.

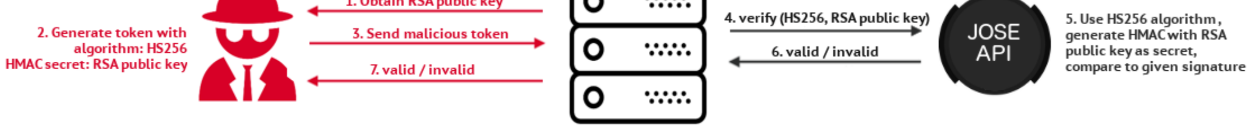

2. Generate token with algorithm: HS256 HMAC secret: RSA public key
1. Obtain RSA public key
3. Send malicious token
7. valid / invalid
4. verify (HS256, RSA public key)
6. valid / invalid
JOSE API
5. Use HS256 algorithm, generate HMAC with RSA public key as secret, compare to given signature

Figure 3.1.: Attack scenario for the Key Confusion Attack

[5] José Padilla, *JSON Web Token implementation in Python*, URL: `https://github.com/jpadilla/pyjwt`
[6] Auth0, Inc., *JsonWebToken implementation for node.js*, URL: `https://github.com/auth0/node-jsonwebtoken`

Tim McLean discovered that many libraries solely rely on the user-controlled algorithm header parameter `alg` to distinguish which algorithm to use for verification [16]. In his research, he recognized that most implementing libraries use the same basic structure for the `verify()` function:

```
verify(string token, string verificationKey)
```

Depending on the implementing system and which algorithm is used, the verification function is either called with the shared HMAC secret key or with the server's (e.g. RSA) public key:

```
# System using HMAC
verify(clientToken, serverHMACSecretKey)

# System using an asymmetric algorithm (e.g. RSA)
verify(clientToken, serverRSAPublicKey)
```

The vulnerability will occur if the system is expecting a token signed with one of the asymmetric algorithms. An attacker might abuse the structure of the verification API to craft an HMAC signature by using the server's public key as shared secret. On the server side, the system passes the token and the RSA public key to the `verify()` function to check its validity. The underlying JOSE library, however, bases its verification decision on the `alg` header – which in this case is HMAC. Thus, it generates a new HMAC with the given *public key* as secret and compares it to the provided signature – which has been generated equally by the adversary. An exemplary attack workflow is illustrated in Figure 3.1.

### 3.2.1. Library Analysis

Tim McLean put much effort in informing the public and especially the maintainers of many JOSE libraries via blog post, Twitter, proposal to the JOSE working group and direct mail[7]. Apart from media attention[8], corresponding security considerations have been added to the JWS specification in RFC 7515 [21].

This is the reason why almost all of the analyzed libraries were not vulnerable to the key confusion attack (anymore). Nevertheless, one library could be identified which implemented a countermeasure, but disabled it by default.

**Responsible Disclosure.** The maintainer of the jose-php library by Nov Matake & Gree, Inc. [39] has been informed that its implemented mitigation is deactivated by default, thus likely to be still vulnerable on many usages. This issue has been assigned the CVE identifier CVE-2016-5431.

[7]Blog post: URL: `https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html`; Twitter: URL: `https://twitter.com/McLean0/status/578281292237815808`; JOSE working group mailing list: URL: `http://www.ietf.org/mail-archive/web/jose/current/msg05036.html`; Exemplary direct mail: URL: `https://bitbucket.org/b_c/jose4j/wiki/04-01-15-Transparency`

[8]Threatpost, The Kaspersky Lab security news service, *Critical vulnerabilities affect JSON Web Token libraries*, URL: `https://threatpost.com/critical-vulnerabilities-affect-json-web-token-libraries/111943/`

One of the recommended countermeasures is to pass an additional parameter with a list of allowed algorithms to the `verify()` function (see Section 3.2.3). This mitigation technique has been implemented by the jose-php library [39], but set to `null` by default:

```php
private function _verify($public_key_or_secret, $expected_alg = null)
```

Listing 3.4: Signature of the verify function of jose-php[9]

The problem is that a developer is not forced to specify the expected algorithm and that old vulnerable code is still working when upgrading to the new version without any warning or notice. The maintainer of the library reacted with the fix shown in Listing 3.5, which will raise an `Exception` if one of the HMAC algorithms is used *and* if the algorithm is implicitly determined by the `alg` header.

```diff
if (!$expected_alg) {
-    # NOTE: might better to warn here
     $expected_alg = $this->header['alg'];
+    $using_autodetected_alg = true;
}

switch ($expected_alg) {
     case 'HS256':
     case 'HS384':
     case 'HS512':
-    return $this->signature === hash_hmac($this->digest(),
    $signature_base_string, $public_key_or_secret, true);
+    if ($using_autodetected_alg) {
+        throw new JOSE_Exception_UnexpectedAlgorithm(
+            'HMAC algs MUST be explicitly specified as $expected_alg'
+        );
+    }
+    $hmac_hash = hash_hmac($this->digest(), $signature_base_string,
    $public_key_or_secret, true);
+    return hash_equals($this->signature, $hmac_hash);
```

Listing 3.5: Commit diff of the php-jose library addressing the disabled countermeasure to the key confusion attack[10]

Even if the given commit fixes this specific case of the key confusion vulnerability, it was recommended to change the signature from `$expected_alg = null` to `$expected_alg` to force a developer to explicitly set the expected algorithms. With the given fix, it is still possible to ma-

[9]URL: https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/src/JOSE/JWS.php#L121

[10]URL: https://github.com/nov/jose-php/commit/1cce55e27adf0274193eb1cd74b927a398a3df4b

nipulate the algorithm header when other algorithms are used, which might, for instance, lead to possible downgrade attacks.

### 3.2.2. Test Cases

The most challenging part from an attacker's perspective is to use the exact same string representation of the public key as used by the verifying system. Usually, (RSA) public keys are stored in Privacy Enhanced Mail (PEM) formatted files or as JWK data set. Both formats are supported by the developed *JOSEPH* Burp Suite extension.

Table 3.2.: Available vectors for the Key Confusion attack

| **Vector** | **Action** |
|---|---|
| Original | Original PEM string. |
| OriginalNoHeaderFooter | Remove PEM header and footer. |
| OriginalNoLF | Remove any line feeds. |
| OriginalNoHeaderFooterLF | Remove PEM header and footer. Remove any line feeds. |
| PKCS1 | Remove first 24 bytes. |
| PKCS1NoHeaderFooter | Remove first 24 bytes. Remove PEM header and footer. |
| PKCS1NoLF | Remove first 24 bytes. Remove any line feeds. |
| PKCS1NoHeaderFooterLF | Remove first 24 bytes. Remove PEM header and footer. Remove any line feeds. |
| PKCS8 | Original PKCS8 PEM string. |
| PKCS8WithHeaderFooter | Add PEM header and footer. |
| PKCS8WithLF | Add PEM specific line feeds. |
| PKCS8WithHeaderFooterLF | Add PEM specific line feeds. Add PEM header and footer. |
| PKCS8WithHeaderFooterLFEndingLF | Add PEM specific line feeds. Add PEM header and footer. Add ending line feed. |

Various string operations are performed on the given input, as depicted in Table 3.2, in order to increase the probability of finding the correct string representation used by the server. Basically, several different variations with/without the PEM header and footer and with/without line feeds and spaces are generated. Additionally, some vectors remove the first 24 bytes, which is the most simple textual conversion from a PKCS#8 formatted public key to a PKCS#1 format, specifically used for RSA keys. All vectors are tested with all three available key sizes of the HMAC algorithm, to cover even more variations.

### 3.2.3. Countermeasures

In his blog post [16], Tim McLean suggested adding an `algorithm` parameter to the signature of the verification function:

```
verify(string token, string algorithm, string verificationKey)
```

With the intention that the receiving party should know its expected algorithm, any dissent would be easily detectable. Nevertheless, just passing a list of expected algorithms to the verification function is not sufficient for systems supporting multiple algorithms. The JWS specification defines the `kid` (key ID) header parameter for the purpose of selecting the correct key for the corresponding algorithm [21, section 4.1.4]. Not only does this evade an attacker to control the manner of which key is used for which algorithm, it additionally facilitates phased *key rotation*[11] and "allows originators to explicitly signal a change of key to recipients" [21, Section 4.1.4]

Apart from that, the more basic underlying problem with the given verification API is the typical case of *primitive obsession* by representing a cryptographic key as a primitive data type – as in this case a string. As depicted by Pedro Félix, a cryptographic key is a "potentially composed object (e.g. two integers in the case of a public key for RSA-based schemes) with associated metadata" [40], such as the related algorithm(s) it applies for and the usage context (encryption, signing). It is recommended to use corresponding objects, such as `RSAPublicKeySpec`[12] in Java, to represent a cryptographic key.

In addition to these countermeasures, one might add an additional field containing the algorithm value to the payload of the JWS. This integrity protected value can then be compared to the provided algorithm in the header during the verification process to detect any dissent. This mitigation method is more likely to be used on application level, though.

## 3.3. Bleichenbacher Million Message Attack

In 1998, Daniel Bleichenbacher published a novel adaptive chosen ciphertext attack against protocols based on the RSA encryption standard PKCS#1 [41] at the International Cryptology Conference CRYPTO. Bleichenbacher exemplarily applied his attack to the SSL v3.0 protocol with experimental results of recovering an encrypted message within between 300 thousand and 2 million chosen ciphertexts. Due to an average of roughly 1 million necessary messages, this attack is referred to as the *Million Message Attack (MMA)*. Since then, the attack has been tested on other protocols, implementations and devices and further, has been optimized to improve the necessary amount of oracle queries in certain scenarios [42] [43].

In 2002, the W3C consortium published the XML Encryption standard [44]. Up until today, the RSA with PKCS#1 v1.5 padding algorithm[13] is one of the two mandatory key transport mechanisms to be implemented for compliance. Despite the known vulnerability found by Bleichenbacher, the RSA PKCS#1 v1.5 algorithm is used in practice for years, but the XML Encryption standard does not even contain a related security considerations section. In 2012, the researchers Jager, Schinzel and Somorovsky published a paper describing several attacks against the PKCS#1 v1.5 key transport mechanism, based on the known Bleichenbacher attack [46]. They were able to "re-

[11]Key rotation is the process of systematically replacing one cryptographic key with a new one without interruption of ongoing operations.

[12]Oracle, *Class RSAPublicKeySpec*, `URL: https://docs.oracle.com/javase/7/docs/api/java/security/spec/RSAPublicKeySpec.html`

[13]When mentioning `RSA PKCS#1 v1.5` in the context of encryption, the `RSAES-PKCS1-v1_5` algorithm as specified in RFC 3447 is meant, which combines the defined `RSAEP` and `RSADP` primitives with the `EME-PKCS1-v1_5` encoding method [45].

cover the secret key used to encrypt [the] transmitted payload data [by exploiting] differences in error messages and in the timing behavior" [46].

Starting with the very first draft of the JWA specification in 2012, the RSA PKCS#1 v1.5 algorithm was one of the listed key management algorithms for JWE. From the introduction of the *implementation requirements* definition for algorithms in draft 03, up to and including draft 33, RSA PKCS#1 v1.5 was the *only* mandatory algorithm for key management. With draft 34 in 2015, the implementation requirement has been demoted to *Recommended-*[14], but is still one of the recommended algorithms and thus likely to be implemented in JOSE libraries.

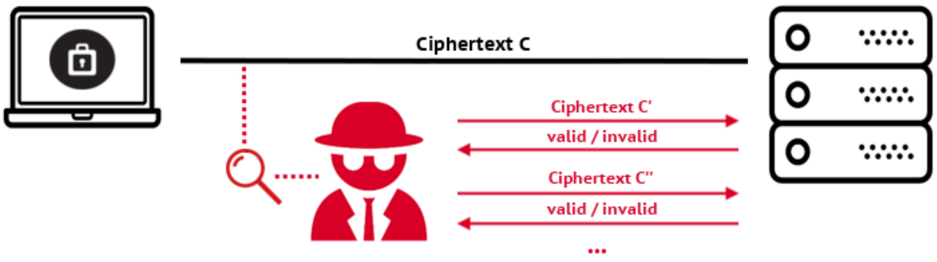


Figure 3.2.: Attack scenario for the Bleichenbacher Million Message Attack

The basic idea of the Bleichenbacher attack is to send several chosen ciphertexts to the server and observe its response. If the attacker is able to distinguish between a validly and invalidly padded message – based on detailed error messages, measurable timing differences or other side channels – he will learn sensitive information about the encrypted plaintext. Abusing an involved party as an *oracle* for the PKCS#1 v1.5 padding, classifies this attack as *Padding Oracle Crypto Attack* as specified in CAPEC-463 of the Common Attack Pattern Enumeration and Classification (CAPEC) database [47]. "An attacker is able to efficiently decrypt data without knowing the decryption key if a target system leaks data on whether or not a padding error happened while decrypting the ciphertext" [47]. The only prerequisites to apply this attack is that an attacker is able to capture a single ciphertext and has the ability to send arbitrary ciphtertexts to the intended receiver, as illustrated in Figure 3.2.

**PKCS#1 v1.5 Encryption Padding.** The PKCS#1 encryption padding version 1.5 is specified in RFC 2313 [48] and used to pad the data to be encrypted using the RSA public-key cryptosystem out to the length of the modulus $N$. This is done by concatenating a randomly generated padding string $PS$ to the given message $k$, before applying the RSA encryption function $m \mapsto m^e \ mod \ N$. The PKCS#1 v1.5 conforming RSA input message $m$ is of the following format and interpreted as integer, such that $0 < m < N$:

$$m := 00||02||PS||00||k$$

[14]The "-" indicates that the requirement strength is likely to be decreased in a future version of the specification.

The leading zero byte `0x00` "ensures that the encryption block, converted to an integer, is less than the modulus" [48, Section 8.1] and the second byte `0x02` specifies the *block type* as public-key encryption operation[15]. The random padding string $PS$ is of the length $l - 3 - |k|$ with a minimum length of $|PS| \geq 8$ and does not contain any zero byte `0x00`. $l$ in this case denotes the byte-length of the modulus $N$. The subsequent zero byte `0x00` is used to separate the padding string and the data $k$.

**Attack Description.** The Bleichenbacher MMA exploits the malleability of the RSA encryption scheme, which allows the following binding of a (randomly generated) integer $s$ [41]:

$$c' \equiv (c \cdot s^e) \, mod \, N = (m^e \cdot s^e) \, mod \, N = (ms)^e \, mod \, N$$

Given an oracle $\vartheta(c')$ responding with `true` or `false` according to the PKCS#1 v1.5 conformity, an attacker will learn that the first two bytes of $ms$ are `0x00` and `0x02` if the response is `true`. Mathematically, this leads to $2B \leq ms \, mod \, N < 3B$, where $B = 2^{8(l-2)}$ [41]. By incrementing the value $s$ and querying the oracle, the adversary learns on every positive result that

$$2B \leq ms - rN < 3B$$

for some computed $r$, which allows him to reduce the set of possible solutions to

$$\frac{2B + rN}{s} \leq m < \frac{3B + rN}{s}$$

and gradually narrowing down the interval containing the original $m$ value, until only one solution in the interval is left [49].

A more detailed description of the algorithm is shown in Section 3.3.2, for full details refer to the original paper [41].

### 3.3.1. Library Analysis

In this section the results of two analyzed libraries are described, which latest versions were found to be vulnerable to the Million Message Attack.

**Responsible Disclosure.** The maintainers of all listed libraries have been informed about the MMA vulnerability and all libraries were fixed in cooperation with the developers. The following

[15]Other block types are `0x00` and `0x01` for private-key and signing operations. Refer to [48] for further details.

CVE identifiers have been assigned: CVE-2016-6298 for JWCrypto by Simo Source [50] and CVE-2016-5430 for jose-php by Nov Matake & Gree, Inc. [39]. Additionally, the json-jwt ruby library by Nov Matake [51] and the José C library by latchset [52] have been fixed[16] based on our disclosure.

The jose-php library structurally outsources every single step of the decryption process – decryption of the CEK, derivation of the encryption and MAC keys, the actual decryption of the ciphertext and the integrity check with the authentication tag – into its own functions[17]. The success of each step is checked within its function and immediately throws an `Exception` on failure, giving precise information about which part failed. Listing 3.6 shows the three relevant parts and their occurrence in the code.

```
throw new JOSE_Exception_DecryptionFailed('Master key decryption failed');
# https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/
    src/JOSE/JWE.php#L193

throw new JOSE_Exception_DecryptionFailed('Encryption/Mac key derivation failed');
# https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/
    src/JOSE/JWE.php#L214

throw new JOSE_Exception_DecryptionFailed('Payload decryption failed');
# https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/
    src/JOSE/JWE.php#L239
```

Listing 3.6: Exceptions thrown during the decryption process in the jose-php library

This behavior can be used by an attacker to successfully build an error-based *Padding Oracle* to distinguish between an invalid PKCS#1 v1.5 padding ("Master key decryption failed") or a valid one. Even if an implementing developer does not directly pass the `Exception` messages to the end user, immediately throwing an `Exception` causes distinguishable timing difference in the processing [49] – offering the ability to create a time-based validity oracle. However, it was not possible to programmatically exploit this vulnerability to apply the MMA attack by using Bleichenbacher's original algorithm. Further investigation revealed that the underlying *phpseclib* library[18] does not strictly validate the PKCS#1 v1.5 format as defined in the specification [45].

```
if (ord($em[0]) != 0 || ord($em[1]) > 2) {
  user_error('Decryption error');
  return false;
}
```

Listing 3.7: Excerpt of phpseclib's `_rsaes_pkcs1_v1_5_decrypt()` function[19]

[16]See: `https://github.com/nov/json-jwt/commit/c46f8133bedd48e3d24a97a5f45df02ebf5923d2` and `https://github.com/latchset/jose/pull/11` for the related commit / pull request.

[17]See: `https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/src/JOSE/JWE.php#L47`

[18]*PHP Secure Communications Library*, URL: `http://phpseclib.sourceforge.net`

The relevant excerpt is shown in Listing 3.7. Apart from the desired prefix `0x00 02`, the phpseclib only checks whether the second byte is not $> 2$, which leads to messages beginning with `0x00 00` and `0x00 01` also being treated as valid. According to a describing comment within the source code[20], this deviation has been added for compatibility reasons with PKCS#1 v2.1. The original Bleichenbacher algorithm is not able to correctly deal with false positives, which resulted in an endless loop of searching for compliant messages in our tests.

Nonetheless, the given vulnerability in the jose-php library can be practically exploited with a modified version of Bleichenbacher's algorithm. In [49], the researchers had to cope with a similar problem. They were able to amend the original algorithm to work with a much weaker oracle, which responded with `true` if a decrypted message started with `0x?? 02`, where `0x??` represents an arbitrary byte [49]. Such modification could also be used for the jose-php library, but has been set out of scope for this thesis.

The JWCrypto library had the same issue of exposing detailed information of specific failing steps during the decryption process. The most relevant `Exceptions` are depicted in Listing 3.8.

```
raise InvalidJWEKeyLength(keylen, len(cek))
# https://github.com/latchset/jwcrypto/blob/v0.3.1/jwcrypto/jwe.py#L178

raise InvalidJWEData('Decryption Failed')
# https://github.com/latchset/jwcrypto/blob/v0.3.1/jwcrypto/jwe.py#L274

raise InvalidJWEData('Failed to verify MAC')
# https://github.com/latchset/jwcrypto/blob/v0.3.1/jwcrypto/jwe.py#L699
```

Listing 3.8: Exceptions raised during the decryption process in the JWCrypto library

The *Decryption Failed* `Exception` indicated an invalid first or second byte of the PKCS#1 v1.5 format and could be successfully used to build an error-based padding oracle to apply the Bleichenbacher attack. In all test cases with different key sizes of 512, 1024 and 2048 bit and all specified encryption algorithms[21], the *Content Encryption Key* could be recovered within less than 100.000 requests to the server and used to decrypt the hidden message.

### 3.3.2. Test Cases

Testing an implementation for the Million Message Attack requires the public key of the receiving party as additional input by the tester and is performed in two steps:

1. **Testing for the existence of a Padding Oracle.** The first step is to perform several checks to determine whether an oracle exists that exposes information about the PKCS#1 v1.5

[19]See: `https://github.com/phpseclib/phpseclib/blob/2.0.4/phpseclib/Crypt/RSA.php#L2538`
[20]See: `https://github.com/phpseclib/phpseclib/blob/2.0.4/phpseclib/Crypt/RSA.php#L2496`
[21]Refer to Table 2.3

conformity of a given ciphertext. This is performed by using various different test vectors to generate encrypted messages with the provided public key, listed in Table 3.3.

Table 3.3.: List of PKCS#1 v1.5 vectors to test the availability of a Padding Oracle.

| **Vector** | **Characteristic** |
|---|---|
| NoNullByte | Generated a PKCS1 padded message with no separating `0x00` byte. |
| NullByteInPadding | Generated a PKCS1 padded message with a `0x00` byte in the padding. |
| NullByteInPkcsPadding | Generated a PKCS1 padded message with a `0x00` byte in the PKCS1 padding. |
| SymmetricKeyOfSize8 | Generated a PKCS1 padded symmetric key of size 8. |
| SymmetricKeyOfSize16 | Generated a PKCS1 padded symmetric key of size 16. |
| SymmetricKeyOfSize24 | Generated a PKCS1 padded symmetric key of size 24. |
| SymmetricKeyOfSize32 | Generated a PKCS1 padded symmetric key of size 32. |
| SymmetricKeyOfSize40 | Generated a PKCS1 padded symmetric key of size 40. |
| WrongFirstByte | Generated a PKCS1 padded message with a wrong first byte (`!= 0x00`). |
| WrongSecondByte | Generated a PKCS1 padded message with a wrong second byte (`!= 0x02`). |

The responses have to be manually reviewed by the tester and assigned to a list of responses indicating valid messages, due to the fact that response and error messages might vary a lot, depending on the used implementation (and underlying programming language) and thus, rendering a fully automated solution imprecise and inflexible.

2. **Decryption of the ciphertext with the Padding Oracle.** For the actual decryption process using the previously identified and adjusted Padding Oracle, the original algorithm from Bleichenbacher is used [41], which is shown in a reduced form in the following:

   **Step 1: Blinding.** Given a ciphertext $c$ as integer, choose different random integers $s_0$ and check whether $c(s_0)^e \, mod \, N$ is PKCS conforming by accessing the oracle. This step can be skipped if $c$ already is PKCS conforming, which should always be the case if an attacker captured a JWE and attempted to recover its plaintext.

   **Step 2: Searching for PKCS conforming messages.**

   **Step 2a: Starting the search.** If this is the first iteration ($i = 1$), search for the smallest positive integer $s_1 > N/(3B)$, such that the ciphertext $c_0(s_1)^e \, mod \, N$ is PKCS conforming.

   **Step 2b: Searching with more than one interval left.** If $i > 1$ and the number of remaining intervals is $\geq 2$, search for the smallest integer $s_i > s_{i-1}$, such that the ciphertext $c_0(s_i)^e \, mod \, N$ is PKCS conforming.

   **Step 2c: Searching with one interval left.** If only one interval is left, choose small integer values $r_i$ and $s_i$ based on the formulas (1) and (2) of the original paper, until the ciphertext is PKCS conforming.

   **Step 3: Narrowing the set of solutions.** After $s_i$ has been found, compute the new interval set $M_i$ with formula (3) of the original paper.

**Step 4: Computing the solution.** If the interval set $M_i$ contains only one interval of length 1, return the solution $m \leftarrow a(s_0)^{-1}\, mod\, N \equiv c^d\ (mod\, N)$. Otherwise, increment $i$ and go to step 2.

The Chair of Network and Data Security of the Ruhr University Bochum[22] and the Hackmanit GmbH[23] developed a modular and open source framework for web services penetration testing called *WS-Attacker*[24] [12], which includes a Java implementation of the Bleichenbacher attack on XML Encryption. Main parts of this implementation have been reused and adapted for the developed *JOSEPH* Burp Suite extension.

### 3.3.3. Countermeasures

Due to the high impact and relevance of the Bleichenbacher algorithm, the RFC 3218 [53] has been developed to draw attention to the known vulnerability and provide three basic countermeasures.

One of the listed countermeasures is to use the alternative *Optimal Asymmetric Encryption Padding (OAEP)* technique – also known as PKCS#1 version 2.1[25] – which is specified in RFC 3447 [45] and not vulnerable to the MMA. Nevertheless, one must take special care using this solution if PKCS#1 v1.5 is also implemented and usable alongside. In the paper *One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography* [14] the researchers proved that certain circumstances allow the decryption of OAEP-ciphertexts by mounting the Bleichenbacher attack. The only requirements are that a "PKCS#1 v1.5-validity oracle is given" and that "the PKCS#1 v1.5 encryption scheme and the attacked cryptosystem use the same RSA-key" [14].

The second countermeasure states that *sufficiently careful checking* might be satisfactory to mitigate the success of the MMA [53]. "If the receiving implementation also checks the length of the CEK and the parity bits (if available) AND responds identically to all such errors, the chances of a given M' being properly formatted are substantially decreased" [53, Section 2.3.1]. This technique, however, does explicitly not entirely eliminate the attack, but intends to increase the number of necessary messages to such an amount that renders the attack impractical.

The third countermeasure is called *Random Filling* and simply "treat[s] misformatted messages as if they were properly PKCS-1 formatted" [53, Section 2.3.2]. Instead of instantly returning an error message if an improperly formatted message is detected, the recipient substitutes the message with a new, randomly generated message and continues the decryption process. "Eventually this will result in a decryption or signature verification error but this is exactly what would have happened if M' happened to be properly formatted but contained an incorrect CEK. This approach also prevents the attacker from distinguishing various failure cases via timing since all failures return roughly the same timing behavior." [53, Section 2.3.2]. Even if the additional time required to generate the random bytes can be considered negligible, special care must be taken to not open

[22]`http://nds.rub.de`
[23]`http://hackmanit.de`
[24]RUB NDS, *WS-Attacker*, URL: `https://github.com/RUB-NDS/WS-Attacker`
[25]Which obsoletes the previous version 2.0 [54]

further side-channels. In [49], the researchers identified measurable timing differences up to 20 microseconds between validly and invalidly padded messages due to improper (internal) `Exception` handling in Java's PKCS#1 implementation. The countermeasure of *unified error messages* did not leak any distinguishable information, but non-compliant messages provoked an *additional* internal `Exception` and thus, needed more processing time, due to the fact that `Exception` handling is a time consuming task.

The best countermeasure would be to completely implement the decryption in constant-time. This would include reimplementing all used native functions to be constant-time and would need a lot of effort and analysis to prevent any leakage. A sufficient alternative is shown in Listing 3.9, which is the implemented fix by Simo Source for CVE-2016-6298. Apart from the *Random Filling* protection mechanism, an `Exception` is *always* thrown – irrespective of the success or failure of the operation. This prevents the previously mentioned possible new side channel of improper `Exception` handling.

```python
def unwrap(self, key, bitsize, ek, headers):
    self._check_key(key)
    # Address MMA attack by implementing RFC 3218 - 2.3.2. Random Filling
    # provides a random cek that will cause the decryption engine to
    # run to the end, but will fail decryption later.

    # always generate a random cek so we spend roughly the
    # same time as in the exception side of the branch
    cek = _randombits(bitsize)
    try:
        cek = super(_Rsa15, self).unwrap(key, bitsize, ek, headers)
        # always raise so we always run through the exception handling
        # code in all cases
        raise Exception('Dummy')
    except Exception:
        return cek
```

Listing 3.9: Fixed version of JWCrypto[26]

## 3.4. Timing Attack on Hash Comparison

The timing attack on hash comparison is an attack of the type *Covert Timing Channel*, specified in CWE-385 of the Common Weakness Enumeration (CWE) database [55]. "Covert timing channels convey information by modulating some aspect of system behavior over time, so that the program receiving the information can observe system behavior and infer protected information"[55]. In this specific case, the time needed for validating different HMAC signatures is observed in order to gain information about its validity at a specific position of a character.

[26]https://github.com/latchset/jwcrypto/blob/v0.3.2/jwcrypto/jwe.py#L190

In order to explain how this vulnerability occurs and how an attacker might exploit it, one has to understand the underlying functionality of a regular (string) comparison operation – exemplarily shown on the PHP programming language. Listing 3.10 shows an excerpt of the `zend_is_identical()` function from PHP's source code, which handles the actual comparison. Apart from a preliminary check on identity (l. 2), it is first checked whether both strings are of equal length (l. 3). If this is the case, both strings are passed to the `memcmp()` C function (l. 4), which is displayed in Listing 3.11.

```
1 case IS_STRING:
2     return (Z_STR_P(op1) == Z_STR_P(op2) ||
3         (Z_STRLEN_P(op1) == Z_STRLEN_P(op2) &&
4          memcmp(Z_STRVAL_P(op1), Z_STRVAL_P(op2), Z_STRLEN_P(op1)) == 0));
```

Listing 3.10: Excerpt of PHP's `zend_is_identical()` function[27]

The `memcmp()` function simultaneously iterates over all characters of both strings and checks each character at the same position of both strings for equality. If two compared characters differ, the function will return the difference *immediately*.

```
int memcmp(const void *s1, const void *s2, size_t n)
{
    unsigned char u1, u2;

    for ( ; n-- ; s1++, s2++) {
        u1 = * (unsigned char *) s1;
        u2 = * (unsigned char *) s2;
        if ( u1 != u2) {
            return (u1-u2);
        }
    }
    return 0;
}
```

Listing 3.11: Implementation of the `memcmp()` C function[28]

Basically all programming languages implement their functions with respect to performance, for which such a comparison construction makes perfectly sense. In a security, or more specifically a cryptographic, context, such an implementation does not meet the necessary conditions.

An adversary with the ability to send arbitrary messages to the server might abuse the premature return on the first difference by gradually iterating over every possible character/byte and measuring the timing needed for processing. Listing 3.12 gives an example of possible results[29] when searching for the first character of a given but unknown hash.

[27]`https://github.com/php/php-src/blob/php-7.0.12/Zend/zend_operators.c#L2006`
[28]`http://opensource.apple.com//source/tcl/tcl-3.1/tcl/compat/memcmp.c`
[29]See Appendix A.3 for details on how this numbers were created.

```
    (Original) 73702ca3cf26f97b69d6891bff8f93a06d0bcc68
0.0000013119118 0000000000000000000000000000000000000000
0.0000013932216 1000000000000000000000000000000000000000
0.0000016285241 2000000000000000000000000000000000000000
0.0000015981053 3000000000000000000000000000000000000000
0.0000013780439 4000000000000000000000000000000000000000
0.0000014892939 5000000000000000000000000000000000000000
0.0000012960518 6000000000000000000000000000000000000000
0.0000021717087 7000000000000000000000000000000000000000
0.0000014749665 8000000000000000000000000000000000000000
0.0000016347097 9000000000000000000000000000000000000000
0.0000016016298 a000000000000000000000000000000000000000
0.0000016616338 b000000000000000000000000000000000000000
0.0000016631538 c000000000000000000000000000000000000000
0.0000015649927 d000000000000000000000000000000000000000
0.0000014099265 e000000000000000000000000000000000000000
0.0000016158320 f000000000000000000000000000000000000000
```

Listing 3.12: Example showing measurable timing differences on native string comparison[29]

This way, an attacker might determine step-by-step the correct signature to an arbitrary message, without knowing the secret.

### 3.4.1. Library Analysis

Although remote timing attacks are applicable in practice [56] [57], it is a challenging task to reliably measure timing differences over a network, due to the noise and jitter induced by network components and other clients [58]. Setting up a working test environment requires full control over the measuring machine to, for instance, disable CPU halting and CPU frequency scaling or use specific hardware [49] for sufficient results. Controlling such configuration is not possible from within a Burp Suite extension and thus, out of scope for this thesis. Nevertheless, several timing attack vulnerabilities have been identified in different libraries by analyzing their public source code.

**Responsible Disclosure.** The maintainers of all listed libraries have been informed about the timing attack vulnerability and all libraries were fixed in cooperation with the developers. The following CVE identifiers have been assigned: CVE-2016-5429 for jose-php by Nov Matake & Gree, Inc. [39], CVE-2016-7037 for JWT by Malcolm Fell [59] and CVE-2016-7036 for python-jose by Michael Davis [60].

All of the three libraries follow the same structure: Given a signature and a message, a new signature for the message is calculated and the resulting hash is compared to the provided signature. The comparison itself is performed as a native string comparison. Listings 3.13, 3.14 and 3.15 show the relevant excerpts of the examined libraries.

```
private function _verify($public_key_or_secret, $expected_alg = null) {
    [...]
    return $this->signature === hash_hmac($this->digest(),
        $signature_base_string, $public_key_or_secret, true);
    [...]
}
```

Listing 3.13: Excerpt of the vulnerable `_verify()` function of the jose-php library by Nov Matake & Gree Inc.[30]

```
public function verify($value, $signature) {
    return $this->algorithm->compute($value) === $signature;
}
```

Listing 3.14: Vulnerable `verify()` function of the JWT library by Malcolm Fell[31]

```
def verify(self, msg, sig):
    return sig == self.sign(msg)
```

Listing 3.15: Vulnerable `verify()` function of the python-jose library by Michael Davis[32]

### 3.4.2. Countermeasures

The issue of timing attacks on cryptographic operations is well known and addressed in the JWA specification, explicitly stating that "the comparison of the computed HMAC value to the JWS Signature value MUST be done in a constant-time manner to thwart timing attacks" [24, Section 3.2]. Within the *Security Considerations* chapter of the JWS specification, a more general description has been used to draw attention to timing differences on successful or unsuccessful operations of cryptographic algorithm implementations, but without mentioning any mitigation strategies [21, Section 10.9].

There exist basically two defensive strategies to prevent any leakage of sensitive information by exploiting measurable timing differences in the comparison of two hashes:

**Constant Time Comparison.** The constant-time comparison strategy ensures that every single byte of both inputs is compared, regardless of previously recognized differences. Most of the programming languages nowadays offer native functions to perform such timing-safe operations,

[30]URL: https://github.com/nov/jose-php/blob/a7fa2b3a02ce62f1edc1804dd93bc81e7cb59f8c/src/JOSE/JWS.php#L132
[31]URL: https://github.com/emarref/jwt/blob/1.0.2/src/Encryption/Symmetric.php#L59
[32]URL: https://github.com/mpdavis/python-jose/blob/1.3.1/jose/jwk.py#L162

like `hash_equals` in PHP[33], `hmac.compare_digest` in Python[34] or `MessageDigest.isEqual` in Java[35]. Listing 3.16 shows the source code of the `isEqual` implementation in Java 1.7, which might also be used as reference if native functions do not exist. On every byte of both inputs the bitwise *exclusive* OR operation is performed and added to the `result` variable with the bitwise *inclusive* OR operation. This way, any difference at any position can be recognized by checking whether `result` is unequal zero at the end. One might notice that constant-time comparison functions are meant to compare two inputs of equal length (see e.g. ll. 9-11 of Listing 3.16). This is not considered problematic, as constant-time comparison is usually used for values of publicly known length – such as hashes. It is quite the reverse: If the verification function did not check the length of the input and the first argument `digesta` is controlled by the user, an adversary might simply send a zero-length string, resulting in the `for`-loop never being executed, thus the input being accepted as valid.

```
 1 /**
 2  * Compares two digests for equality. Does a simple byte compare.
 3  *
 4  * @param digesta one of the digests to compare.
 5  * @param digestb the other digest to compare.
 6  * @return true if the digests are equal, false otherwise.
 7  */
 8 public static boolean isEqual(byte[] digesta, byte[] digestb) {
 9     if (digesta.length != digestb.length) {
10         return false;
11     }
12     int result = 0;
13     // time-constant comparison
14     for (int i = 0; i < digesta.length; i++) {
15         result |= digesta[i] ^ digestb[i];
16     }
17     return result == 0;
18 }
```

Listing 3.16: Source code of the `MessageDigest.isEqual()` function in Java 1.7[36]

**Double HMAC Verification.** The second strategy is called *Double HMAC Verification* [61]. This method makes use of an additional random key to render the comparison operation non-deterministic and thus, truly blind the side-channel (under the assumption that a cryptographically secure random number generator is used) [62]. Listing 3.17 shows an exemplary use of this strategy:

[33] PHP.net, *hash_equals function*, URL: http://php.net/manual/de/function.hash-equals.php

[34] Python Software Foundation, *hmac.compare_digest function*, URL: https://docs.python.org/3/library/hmac.html#hmac.compare_digest

[35] Oracle, *MessageDigest.isEqual function*, URL: https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html#isEqual(byte[],%20byte[])

[36] http://www.docjar.com/html/api/java/security/MessageDigest.java.html

Given two input strings `a` and `b`, a random key is generated and used to generate an HMAC value for input `a` and an HMAC value for input `b`. The return value of this function is not the direct comparison of the two input values, but the comparison of their related HMAC values. The use of a randomly generated key ensures that sending the same message twice does not yield the same time duration, rendering this function non-deterministic.

```php
/**
 * Compare two strings with non-deterministic blinding
 *
 * @param string $a
 * @param string $b
 * @return bool
 */
function hmac_compare($a, $b)
{
    $compare_key = random_bytes(32);
    return hash_hmac('sha256', $a, $compare_key) === hash_hmac('sha256', $b,
       $compare_key);
}
```

Listing 3.17: Exemplary use of the double HMAC strategy[37]

Although both of the countermeasures perfectly prevent disclosing any sensitive timing information, the *Constant Time Comparison* is more likely to be found in existing implementations, probably due to a slightly better performance compared to the *Double HMAC Verification* technique.

[37]Example taken from: https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy

# 4. Implementation

As part of this thesis, a Burp Suite extension with the name *JOSEPH* has been developed. JOSEPH is an abbreviation for *JavaScript Object Signing and Encryption Pentesting Helper* and aims to support penetration testers, developers and analysts to test implementations of the JOSE specifications. Its newly designed logo is pictured in Figure 4.1.

Figure 4.1.: Logo of the *JOSEPH* Burp Suite extension

This chapter gives an overview of the developed Burp Suite extension and its features. It starts with short information about how to compile the project from source and which versions JOSEPH has been tested with. Apart from the following explanation of the UI to understand necessary workflows and how to use this extension in general, this chapter also serves as a documentation and describes the underlying internal structure and extensibility.

## 4.1. System Setup

The Burp Suite extension is written in *Java* and uses the build management tool *Apache Maven*[1] to organize its dependencies, run the included unit tests and compile the project to a standalone `JAR`-file. The minimal required Java version to work with is 1.7, in order to support a variety of operating systems not yet running Java 1.8.

### Compiling the Extension

To compile the *JOSEPH* Burp Suite extension from source, it is necessary to have *Apache Maven* installed and to run the command shown in Listing 4.1 below from within the project directory.

```
$ mvn clean package
```

Listing 4.1: Command to compile the JOSEPH Burp Suite extension from source

This command attempts to `clean` any files and directories generated during previous builds to start with a fresh environment, automatically compiles the files, runs all included unit tests and

[1]The Apache Software Foundation, *Apache Maven*, URL: `https://maven.apache.org/`

takes the compiled code to `package` it to its distributable `JAR` format. To skip the (unit) tests, the additional argument `-DskipTests` can be used.

One *important notice* is that if Oracle's Java Development Kit (JDK) is used, it is not possible to ship the Burp Suite extension as single `JAR`-file. The Java Cryptography Extension (JCE) requires additional providers, in this case the Bouncy Castle[2] library, to be signed – due to security considerations to ensure its integrity. Repackaging it into a comprehensive JOSEPH `JAR` will remove its signature and will break its inclusion.

When performing the *Bleichenbacher* attack without Bouncy Castle being correctly loaded, the following error will occur:

```
[BleichenbacherPkcs1Info]: Error during key encryption: Cannot find any provider
    supporting RSA/NONE/NoPadding
```

If this issue arises, one needs to perform the following step(s)[3]:

1. Copy the Bouncy Castle `JAR`-file `bcprov-jdk15on-1.54.jar` from JOSEPH's *lib* folder into the `/[PATH_TO_JVM]/jre/lib/ext` directory.

2. In some cases, it is necessary to additionally amend the `/[PATH_TO_JVM]/jre/lib/security/java.security` file and add the following line (preferably directly below the other provider definitions):
   `security.provider.9=org.bouncycastle.jce.provider.BouncyCastleProvider`.
   The `9` in this case specifies the priority and should be adjusted to fit into existing definitions.

This workaround should not be necessary when the extension is delivered through the *BApp Store*. BApp Store extensions may contain a library folder which will be automatically loaded by the Burp Suite.

### Testing Environment

The JOSEPH Burp Suite extension has been tested with both Java versions `1.7.0` and `1.8.0` (Oracle JDK and OpenJDK), using the latest stable release of Burp Suite's free edition in version `1.7.10`[4]. JOSEPH has been developed on an OSX 10.11.6 (El Capitan) operating system and has been additionally tested on a Linux Mint 17.3 system. A quick basic test has also been performed on a Windows 10 operating system. For building and dependency management, Apache Maven in version 3.3.9 has been used.

[2]Legion of the Bouncy Castle Inc., *Bouncy Castle Crypto APIs*, URL: https://www.bouncycastle.org/

[3]See https://docs.oracle.com/cd/E19830-01/819-4712/ablsc/index.html for a similar solution described in the *Sun Java System Application Server Platform Edition 8.2 Administration Guide*

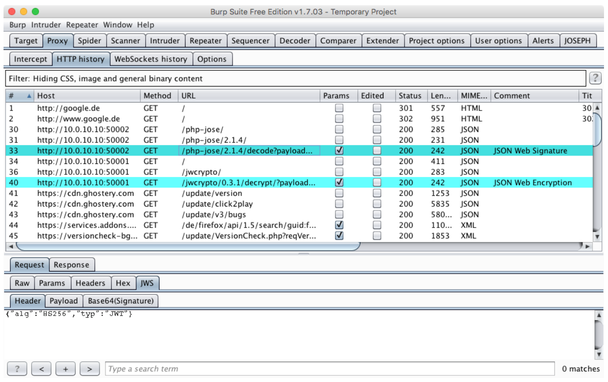
[4]Last checked on 2016-11-15

## 4.2. User Interface

The look-and-feel of JOSEPH's graphical user interface is adapted to the other parts of the Burp Suite. The basic idea is to create a familiar environment to quickly get along with its features and to reduce the need of any previous training. Its goal is to follow the principle of simplicity while still offering as much flexibility as possible.

### Proxy and Editors

The *HTTP history* tab of the Burp Suite proxy lists all processed HTTP messages and enables the user to review the performed requests and recorded responses. By enabling the JOSEPH extension, the functionality of the HTTP history is amended to search for JWS and JWE patterns and to highlight matching messages with a *cyan* colored background, alongside with a specific comment. Figure 4.2(a) shows a screenshot of the HTTP history tab listing several messages, where two requests were detected to contain a JOSE value.

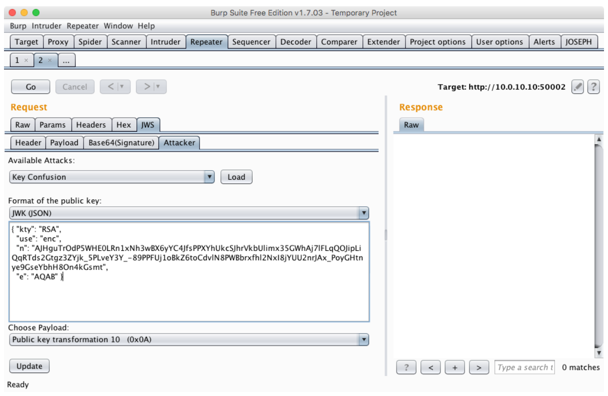

Burp Suite Free Edition v1.7.03 - Temporary Project
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts JOSEPH
Intercept HTTP history WebSockets history Options
Request Response
Raw Params Headers Hex JWS
Header Payload Base64(Signature)
Type a search term
0 matches

Filter: Hiding CSS, image and general binary content

| # | Host | Method | URL | Params | Edited | Status | Len... | MIME... | Comment | Tit |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | http://google.de | GET | / | ☐ | ☐ | 301 | 557 | HTML | | 30 |
| 2 | http://www.google.de | GET | / | ☐ | ☐ | 302 | 951 | HTML | | 30 |
| 30 | http://10.0.10.10:50002 | GET | /php-jose/ | ☐ | ☐ | 200 | 285 | JSON | | |
| 31 | http://10.0.10.10:50002 | GET | /php-jose/2.1.4/ | ☐ | ☐ | 200 | 231 | JSON | | |
| 33 | http://10.0.10.10:50002 | GET | /php-jose/2.1.4/decode?payload... | ✓ | ☐ | 200 | 242 | JSON | JSON Web Signature | |
| 34 | http://10.0.10.10:50001 | GET | / | ☐ | ☐ | 200 | 411 | JSON | | |
| 36 | http://10.0.10.10:50001 | GET | /jwcrypto/ | ☐ | ☐ | 200 | 283 | JSON | | |
| 40 | http://10.0.10.10:50001 | GET | /jwcrypto/0.3.1/decrypt/?payload... | ✓ | ☐ | 200 | 242 | JSON | JSON Web Encryption | |
| 41 | https://cdn.ghostery.com | GET | /update/version | ☐ | ☐ | 200 | 1253 | JSON | | |
| 42 | https://cdn.ghostery.com | GET | /update/click2play | ☐ | ☐ | 200 | 5835 | JSON | | |
| 43 | https://cdn.ghostery.com | GET | /update/v3/bugs | ☐ | ☐ | 200 | 580... | JSON | | |
| 44 | https://services.addons.... | GET | /de/firefox/api/1.5/search/guid:f... | ✓ | ☐ | 200 | 110... | XML | | |
| 45 | https://versioncheck-bg... | GET | /update/VersionCheck.php?reqVer... | ✓ | ☐ | 200 | 1853 | XML | | |

```
{"alg":"HS256","typ":"JWT"}
```

(a) Detection and marking of JOSE requests


Burp Suite Free Edition v1.7.03 - Temporary Project
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts JOSEPH
1 2 ...
Go Cancel < >
Request
Raw Params Headers Hex JWS
Header Payload Base64(Signature) Attacker
Load
Update
Response
Raw
Type a search t
0 matches
Ready

Target: http://10.0.10.10:50002

Available Attacks: Key Confusion

Format of the public key: JWK (JSON)

```
{ "kty": "RSA",
  "use": "enc",
  "n": "AJHguTrOdP5WHE0LRn1xNh3wBX6yYC4JfsPPXYhUkcSJhrVkbUlimx35GWhAj7lFLqQOJipLi
QqRTds2Gtgz3ZYjk_5PLveY3Y_-89PPFUj1oBkZ6toCdvlN8PWBbrxfhl2NxI8jYUU2nrJAx_PoyGHtn
ye9GseYbhH8On4kGsmt",
  "e": "AQAB" }
```

Choose Payload: Public key transformation 10 (0x0A)

(b) Attacker tab of the JWS editor

Figure 4.2.: Screenhots of the marking feature (a) and additional editor tabs (b)

In addition to the marking, the native request/response editors are supplemented to include a JWS/JWE tab with sub tabs for displaying the separate components of a discovered JOSE value. Where useful, the `base64url` encoded content is shown in its decoded ASCII format.
When used within the *Repeater* tool or during an active *interception*, the JWS/JWE editors are editable and may be used to modify the JOSE parameter value of the request before sending it. Furthermore, the JWS editor is extended by an additional *Attacker* tab, allowing a user to update the given request with attack related modifications. This feature is shown in Figure 4.2(b).

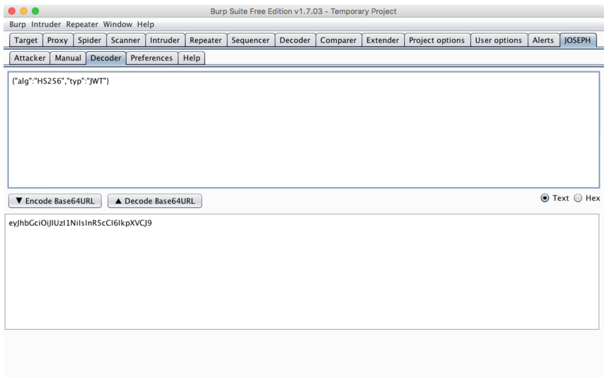
### JOSEPH Tab

If the extension is getting enabled, an extra *JOSEPH* tab will appear on the main navigation of Burp Suite. This tab contains different sections for the features of the extension, namely the

*Attacker*, a *Manual* tab, the *Decoder* and *Preferences*.

## Decoder and Preferences

The *Decoder*, depicted in Figure 4.3(a), is a simple helper utility to en-/decode `base64url` strings and display them in an ASCII or hexadecimal format. The Burp Suite itself has its own *Decoder* tool, but `base64url` is not one of the available encoding formats and the public API for extensions does not offer any possibility to add additional formats.

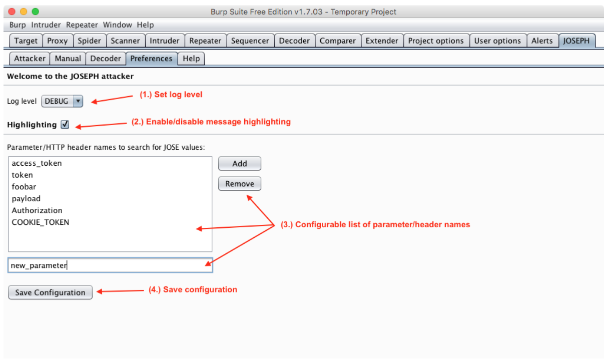

Burp Suite Free Edition v1.7.03 - Temporary Project
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts JOSEPH
Attacker Manual Decoder Preferences Help
▼ Encode Base64URL
▲ Decode Base64URL
Text Hex

```
{"alg":"HS256","typ":"JWT"}
```

(● Text ○ Hex)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

(a) The *Decoder* tab


Burp Suite Free Edition v1.7.03 - Temporary Project
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts JOSEPH
Attacker Manual Decoder Preferences Help
(1.) Set log level
(2.) Enable/disable message highlighting
Add
Remove
(3.) Configurable list of parameter/header names
Save Configuration
(4.) Save configuration

Welcome to the JOSEPH attacker

Log level: DEBUG

Highlighting ✓

Parameter/HTTP header names to search for JOSE values:

access_token
token
foobar
payload
Authorization
COOKIE_TOKEN

new_parameter

(b) The *Preferences* tab

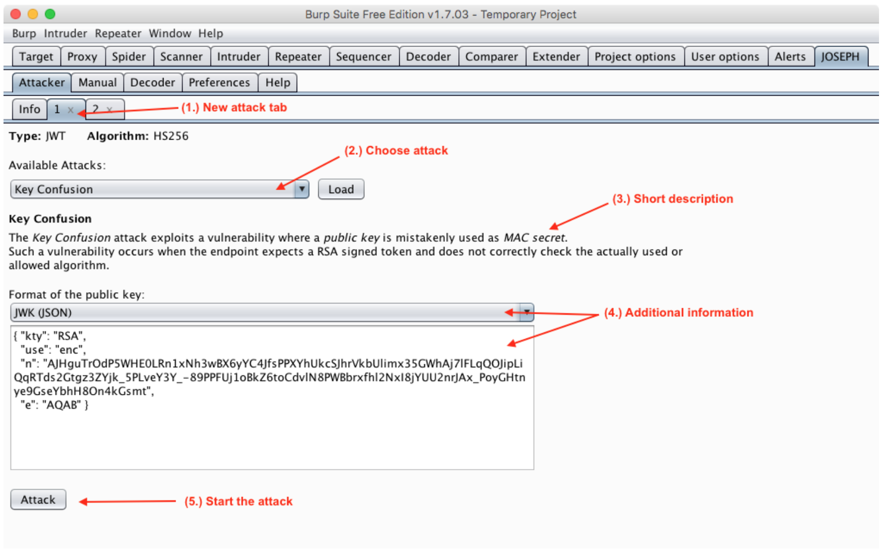
Figure 4.3.: Screenshots showing the decoder (a) and preferences (b) tabs

Figure 4.3(b) shows a screenshot of the *Preferences* tab. This tab serves to configure several options of JOSEPH's behavior. JOSEPH uses its own logging functions, for which the verbosity level can be set to *Debug*, *Info* or *Error* (1.). A second option (2.) allows to en-/disable the highlighting feature of messages containing JOSE values in the *HTTP history*. The third option (3.) aims to increase the flexibility of this extension. The user is able to dynamically maintain a list of names, which are used to search for JOSE values in parameters at different locations[5] and HTTP headers. By clicking the *Save Configuration* button (4.), the preferences are persistently saved into a configuration file on the hard disk to survive a restart or crash of the Burp Suite.

## Attack Workflow

One of the main features of JOSEPH is the *attack engine*, enabling the user to apply the discovered attacks of Chapter 3 and test implementations for their vulnerability. If a message is detected to contain a JOSE value, one is able to send it to the JOSEPH extension by right clicking on the message and selecting *Send to JOSEPH* from the context menu. Within the JOSEPH *Attacker*, shown in Figure 4.4, a new tab will be added (1.), showing some very basic information about the token and a list of available attacks (2.) to choose from. If a specific attack is loaded, a short description of the selected attack will be displayed (3.) and, if necessary as defined by the attack itself, additional

[5]Within the URL, body and cookies, as JSON, XML or multipart format.


Burp Suite Free Edition v1.7.03 - Temporary Project
Burp Intruder Repeater Window Help
Target Proxy Spider Scanner Intruder Repeater Sequencer Decoder Comparer Extender Project options User options Alerts JOSEPH
Attacker Manual Decoder Preferences Help
Info 1 × 2 ×
(1.) New attack tab
(2.) Choose attack
Load
(3.) Short description
(4.) Additional information
Attack
(5.) Start the attack

**Type:** JWT **Algorithm:** HS256

Available Attacks:

Key Confusion

**Key Confusion**

The *Key Confusion* attack exploits a vulnerability where a *public key* is mistakenly used as *MAC secret*.
Such a vulnerability occurs when the endpoint expects a RSA signed token and does not correctly check the actually used or allowed algorithm.

Format of the public key:

JWK (JSON)

```
{ "kty": "RSA",
  "use": "enc",
  "n": "AJHguTrOdP5WHE0LRn1xNh3wBX6yYC4JfsPPXYhUkcSJhrVkbUlimx35GWhAj7lFLqQOJipLi
QqRTds2Gtgz3ZYjk_5PLveY3Y_-89PPFUj1oBkZ6toCdvlN8PWBbrxfhl2NxI8jYUU2nrJAx_PoyGHtn
ye9GseYbhH8On4kGsmt",
  "e": "AQAB" }
```

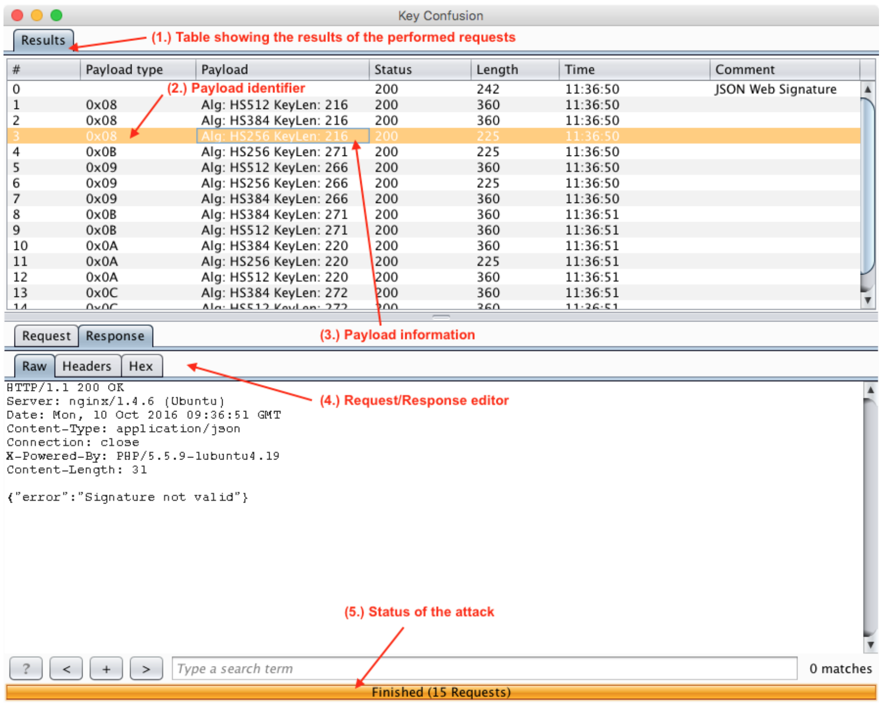
Figure 4.4.: Attack selection and configuration

form elements will appear requesting additional information needed to perform the attack (4.). This additional information can be, for instance, the recipient's public key, as shown in the given screenshot in Figure 4.4 where the *Key Confusion* attack is loaded. The public key might be of the *PEM* or *JWK* (Set) format, both are supported by JOSEPH. By clicking the *Attack* button at the bottom of this window, the user can start the attack. If any of the configuration is incorrect or format checks fail, an error popup will show up with a short `Exception` message, exemplarily shown in Figure 4.5. For the attack and its results a separate window is opened, in order to concurrently use other tools of the Burp Suite in case the attack takes a longer time. This new window, depicted in Figure 4.6, is basically structured similarly to Burp Suite's native *Intruder* tool. The upper part shows a table listing the performed requests (1.), with details such as the response status, length, time, an attack related *payload type* (2.) and short information about the payload itself (3.). The *payload type* is used to identify the exact payload, in order to select and use it for further tests within the *Repeater* tool or during active *interception*. On the lower part of the window, the *Request/Response* viewer and JOSEPH's additional JWS/JWE editors are shown (4.). The very bottom displays a progress bar (5.), indicating the current status of the attack by giving the amount of requests that have already been performed alongside with the number of all prepared requests.

Not a valid JWK format!

Figure 4.5.: Error popup

This structure and workflow is basically the same for all standard attacks. For Bleichenbacher's Million Message Attack though a slightly different workflow and UI was needed, as it is performed in two separate steps: The *testing for an oracle* and the actual *decryption of the hidden content*. The result table of the MMA (Figure 4.7(a)) has been expanded by an additional editable column (2.), which is used as a configuration step for the second part of the attack. According to the responses of



| # | Payload type | Payload | Status | Length | Time | Comment |
|---|---|---|---|---|---|---|
| 0 | | | 200 | 242 | 11:36:50 | JSON Web Signature |
| 1 | 0x08 | Alg: HS512 KeyLen: 216 | 200 | 360 | 11:36:50 | |
| 2 | 0x08 | Alg: HS384 KeyLen: 216 | 200 | 360 | 11:36:50 | |
| 3 | 0x08 | Alg: HS256 KeyLen: 216 | 200 | 225 | 11:36:50 | |
| 4 | 0x0B | Alg: HS256 KeyLen: 271 | 200 | 225 | 11:36:50 | |
| 5 | 0x09 | Alg: HS512 KeyLen: 266 | 200 | 360 | 11:36:50 | |
| 6 | 0x09 | Alg: HS256 KeyLen: 266 | 200 | 225 | 11:36:50 | |
| 7 | 0x09 | Alg: HS384 KeyLen: 266 | 200 | 360 | 11:36:50 | |
| 8 | 0x0B | Alg: HS384 KeyLen: 271 | 200 | 360 | 11:36:51 | |
| 9 | 0x0B | Alg: HS512 KeyLen: 271 | 200 | 360 | 11:36:51 | |
| 10 | 0x0A | Alg: HS384 KeyLen: 220 | 200 | 360 | 11:36:51 | |
| 11 | 0x0A | Alg: HS256 KeyLen: 220 | 200 | 225 | 11:36:51 | |
| 12 | 0x0A | Alg: HS512 KeyLen: 220 | 200 | 360 | 11:36:51 | |
| 13 | 0x0C | Alg: HS384 KeyLen: 272 | 200 | 360 | 11:36:51 | |
| 14 | 0x0C | Alg: HS512 KeyLen: 272 | 200 | 360 | 11:36:51 | |

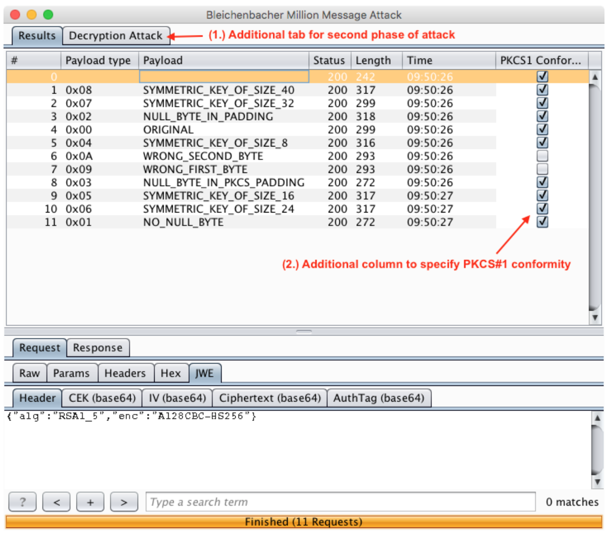
```
HTTP/1.1 200 OK
Server: nginx/1.4.6 (Ubuntu)
Date: Mon, 10 Oct 2016 09:36:51 GMT
Content-Type: application/json
Connection: close
X-Powered-By: PHP/5.5.9-1ubuntu4.19
Content-Length: 31

{"error":"Signature not valid"}
```

Figure 4.6.: Attack result window

the different testing vectors, the user has to decide whether the format was considered PKCS#1 v1.5 conform or not and needs to assign the related message to a list of valid responses by clicking the checkbox. With at least a single valid response in the list, the *Decryption Attack* tab gets enabled, containing the UI to perform the second part of the attack. Within this tab (Figure 4.7(b)), the user is able to start (1.) the decryption attack, which takes the previously configured list of valid responses to query the server endpoint and use the system as padding oracle. The attack can be canceled at any time by clicking the *Cancel* button. By starting the attack, further UI elements get visible, displaying secondly updated information about the current status (2.). This includes the elapsed time, amount of performed requests and last found `s` value[6]. On completion and successful decryption of the hidden content, a new text box with the recovered *Content Encryption Key* in its hexadecimal, or alternatively `base64url` encoding appears (3.). The content is shown in its full PKCS#1 v1.5 conform representation, including prefix and padding string. The recovered CEK is then used to decrypt the actual protected message, based on the encryption algorithm of the `enc` JOSE header (4.). Both algorithm types, AES CBC with HMAC and AES GCM, with all three key sizes[7] are supported by JOSEPH.

[6]Refer to Section 3.3 for its meaning
[7]Refer to Table 2.3

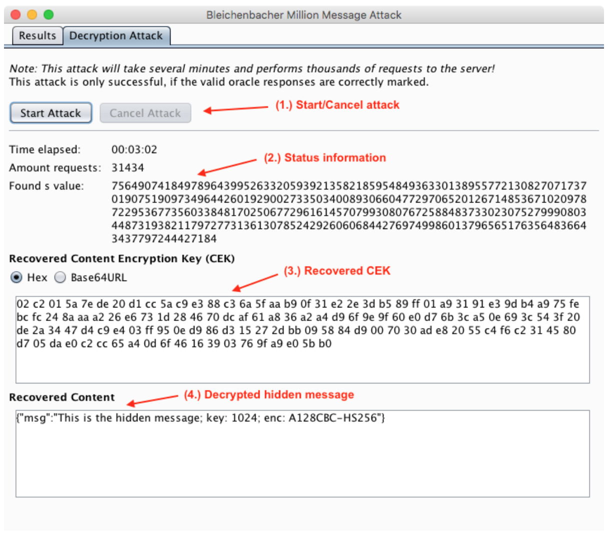


| # | Payload type | Payload | Status | Length | Time | PKCS1 Confor... |
|---|---|---|---|---|---|---|
| 0 | | | 200 | 242 | 09:50:26 | ✓ |
| 1 | 0x08 | SYMMETRIC_KEY_OF_SIZE_40 | 200 | 317 | 09:50:26 | ✓ |
| 2 | 0x07 | SYMMETRIC_KEY_OF_SIZE_32 | 200 | 299 | 09:50:26 | ✓ |
| 3 | 0x02 | NULL_BYTE_IN_PADDING | 200 | 318 | 09:50:26 | ✓ |
| 4 | 0x00 | ORIGINAL | 200 | 299 | 09:50:26 | ✓ |
| 5 | 0x04 | SYMMETRIC_KEY_OF_SIZE_8 | 200 | 316 | 09:50:26 | ✓ |
| 6 | 0x0A | WRONG_SECOND_BYTE | 200 | 293 | 09:50:26 | ☐ |
| 7 | 0x09 | WRONG_FIRST_BYTE | 200 | 293 | 09:50:26 | ☐ |
| 8 | 0x03 | NULL_BYTE_IN_PKCS_PADDING | 200 | 272 | 09:50:26 | ✓ |
| 9 | 0x05 | SYMMETRIC_KEY_OF_SIZE_16 | 200 | 317 | 09:50:27 | ✓ |
| 10 | 0x06 | SYMMETRIC_KEY_OF_SIZE_24 | 200 | 317 | 09:50:27 | ✓ |
| 11 | 0x01 | NO_NULL_BYTE | 200 | 272 | 09:50:27 | ✓ |

```
{"alg":"RSA1_5","enc":"A128CBC-HS256"}
```

Finished (11 Requests)

(a) Phase 1: Testing for a *Padding Oracle*



*Note: This attack will take several minutes and performs thousands of requests to the server!*
This attack is only successful, if the valid oracle responses are correctly marked.

```
Time elapsed:      00:03:02
Amount requests:   31434
Found s value:     7564907418497896439952633205939213582185954849363301389557721308270717 37
                   01907519097349644260192900273350340089306604772970652012671485367102097 8
                   72295367735603384817025067729616145707993080767258848373302307527999080 3
                   44873193821179727731361307852429260606844276974998601379656517635648366 4
                   3437797244427184
```

```
3437797244427184
```

**Recovered Content Encryption Key (CEK)**

[✓ Hex] Base64URL

```
02 c2 01 5a 7e de 20 d1 cc 5a c9 e3 88 c3 6a 5f aa b9 0f 31 e2 2e 3d b5 89 ff 01 a9 31 91 e3 9d b4 a9 75 fe
bc fc 24 8a aa a2 26 e6 73 1d 28 46 70 dc af 61 a8 36 a2 a4 d9 6f 9e 9f 60 e0 d7 6b 3c a5 0e 69 3c 54 3f 20
de 2a 34 47 d4 c9 e4 03 ff 95 0e d9 86 d3 15 27 2d bb 09 58 84 d9 00 70 30 ad e8 20 55 c4 f6 c2 31 45 80
d7 05 da e0 c2 cc 65 a4 0d 6f 46 16 39 03 76 9f a9 e0 5b b0
```

**Recovered Content**

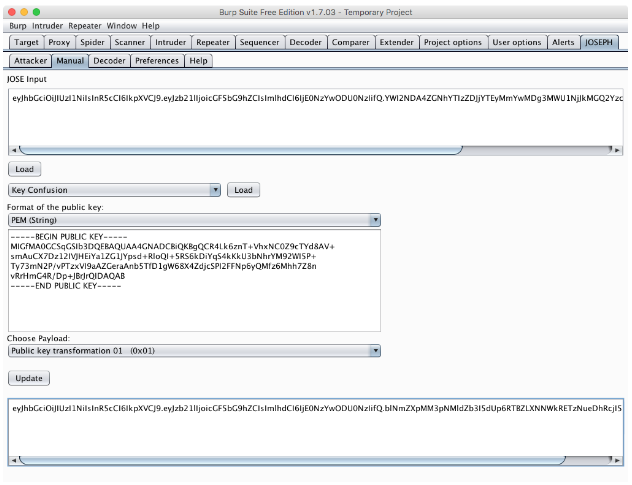
```
{"msg":"This is the hidden message; key: 1024; enc: A128CBC-HS256"}
```

(b) Phase 2: Decrypting the hidden content

Figure 4.7.: Workflow of the MMA attack: Testing for the existence of a *Padding Oracle* (a) and performing the actual decryption of the hidden content (b).

## Manual Tab

The *Manual* tab is some kind of fallback solution, for any special cases where non-standard JOSE implementations need to be tested and JOSEPH is not able to automatically recognize and handle it. In this cases, the user can manually copy and paste the JOSE token into the manual tab and is still able to apply the provided attacks to it. The UI of the manual tab, shown in Figure 4.8, is basically the same as for the repeater and attacker tabs of the editors, with only one additional textbox widget for the output.



JOSE Input

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCIsImlhdCI6IjE0NzYwODU0NzlifQ.YWI2NDA4ZGNhYTIzZDJjYTEyMmYwMDg3MWU1NjJkMGQ2Yzc
```

Key Confusion

Format of the public key: PEM (String)

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCR4Lk6znT+VhxNC0Z9cTYd8AV+
smAuCX7Dz12IVJHEiYa1ZG1JYpsd+RloQI+5RS6kDiYqS4kKkU3bNhrYM92WI5P+
Ty73mN2P/vPTzxVI9aAZGeraAnb5TfD1gW68X4ZdjcSPI2FFNp6yQMfz6Mhh7Z8n
vRrHmG4R/Dp+JBrJrQIDAQAB
-----END PUBLIC KEY-----
```

Choose Payload: Public key transformation 01 (0x01)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzb21lIjoicGF5bG9hZCIsImlhdCI6IjE0NzYwODU0NzlifQ.blNmZXpMM3pNMldZb3I5dUp6RTBZLXNNWkRETzNueDhRcjI5
```

Figure 4.8.: The *Manual* tab

## 4.3. Internal Structure

The JOSEPH extension consists of several packages separating the logical components, all being part of the `eu.dety.burp.joseph` namespace. An overview of the package structure is shown in Figure 4.9. In addition to this section, the source code itself is comprehensively documented and follows the *Javadoc*[8] syntax to enable the automatic generation of an API documentation.

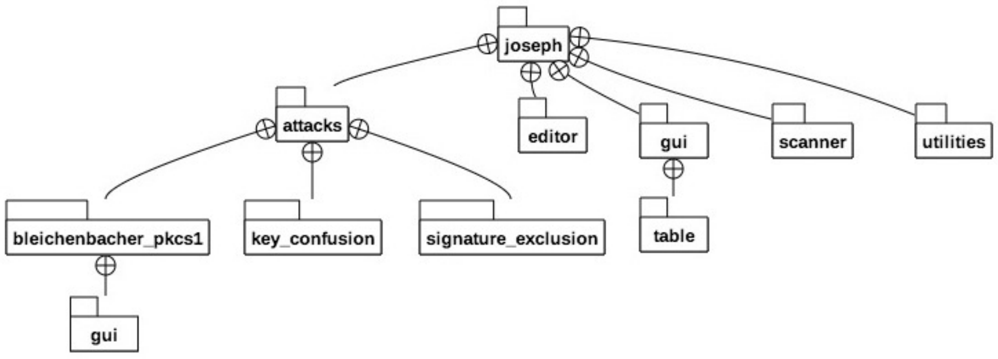


Figure 4.9.: Package structure overview

### Utilities Package

The `utilities` package contains helper classes and functions to provide an interface for recurring operations at different locations. This includes the following classes:

- The `Converter`, which helps on the transformation between a JWK object or PEM string and Java's `RSAPublicKey` type.
- A `Crypto` class, which provides methods to generate a MAC, decrypt an AES ciphertext in different modes and with different key sizes, or get the correct key size based on the selected algorithm.
- The `Decoder`, which aids to encode and decode a JOSE value from and to different representations.
- The `Finder`, which assists in finding and extracting JWS and JWE values based on regex pattern matching.
- A `JoseParameter` class, which is used to represent a JOSE parameter, irrespective of its source (header, URL, cookie, ...) and type (JWS or JWE).
- The own `Logger` for this extension, which enables the adjustability of different log levels and the use of a custom logging format.

[8]Oracle, *Javadoc Tool*, URL: `http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html`

### Scanner Package

The `scanner` package contains the recognition and marking logic, used within the HTTP proxy history. Its `Marker` class implements Burp Suite's `IHttpListener`[9] interface and gets registered as *HTTP listener*. The `Marker` is responsible for highlighting a message and adding a specific comment if any JWS or JWE pattern was found.

### GUI Package

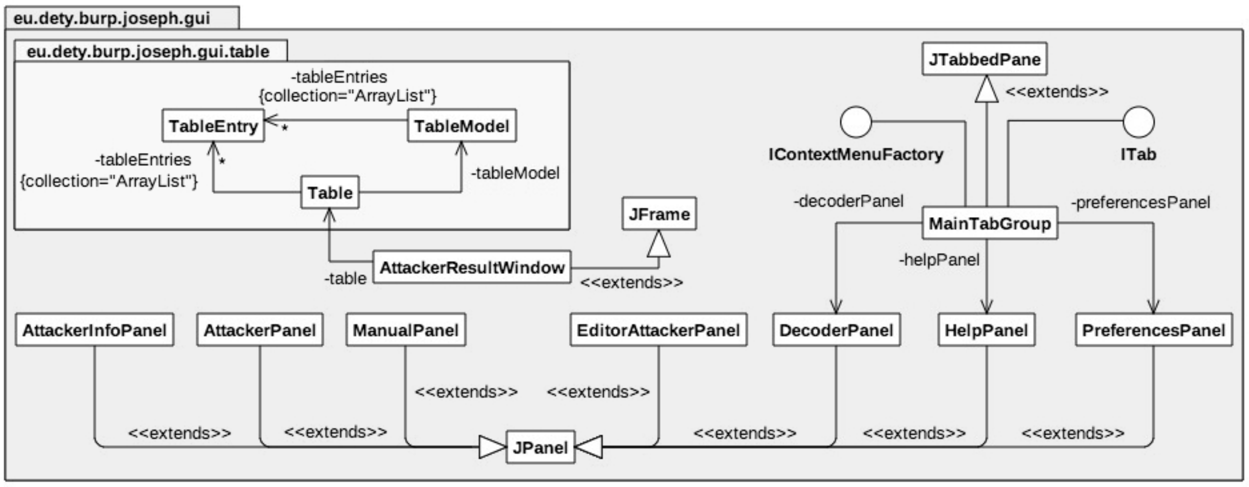


Figure 4.10.: The GUI package structure

Within the `gui` package, basically all visible UI components are defined. Figure 4.10 gives an overview of the containing classes. The `MainTabGroup` is JOSEPH's main UI component, which implements Burp Suite's `ITab`[10] interface and adds itself as new extension tab to the Burp Suite window. The `MainTabGroup` also serves as parent for the `Decoder`, `Preferences`, and `Help` panels and implements Burp Suite's `IContextMenuFactory`[11] interface to allow being registered as context menu within the HTTP proxy history. It additionally creates a new `JTabbedPane` instance, which holds the `AttackerInfo` panel and all created instances of the `Attacker` panel, which are added when a user sends a recorded message to the JOSEPH extension. The `Manual` panel provides a fallback solution to manually apply attack payloads to JOSE values. The `Help` panel is a static view, displaying information about the extension and how to contact the developer. The `Decoder` panel simply encodes and decodes `base64url` strings and offers the choice to show it in ASCII or hex, just as described in the previous *User Interface* section 4.2. Within the `Preferences` panel, several options to customize the extension are provided. The state of the settings can be

[9] PortSwigger Ltd., *Interface IHttpListener*, URL: `https://portswigger.net/burp/extender/api/burp/IHttpListener.html`
[10] PortSwigger Ltd., *Interface ITab*, URL: `https://portswigger.net/burp/extender/api/burp/ITab.html`
[11] PortSwigger Ltd., *Interface IContextMenuFactory*, URL: `https://portswigger.net/burp/extender/api/burp/IContextMenuFactory.html`

persistently saved into a JSON-file which will be loaded on startup of the extension and which is located at `$HOME/.joseph/config.json`. The `AttackerResultWindow` extends the `JFrame` class to create an independent, new and non-blocking window to display the attack results.

The `gui` package contains a sub-package `table` with the `Table`, `TableModel` and `TableEntry` classes. These classes are used by standard attacks to list the performed attack requests and show their results, and are displayed within the `AttackerResultWindow`. Standard attack in this case refers to an attack for which it is sufficient to display the columns *Payload type*, *Payload*, *Status*, *Length*, *Time* and *Comment* as result, alongside with the performed request and obtained response. This is the case for the implemented *Signature Exclusion* and *Key Confusion* attacks. The *Bleichenbacher* attack is more complex, thus includes its own `gui` package with customized components.

### Editor Package

The `editor` package contains the `JwsEditor` and `JweEditor`, which both implement Burp Suite's `IMessageEditorTabFactory`[12] interface and are registered as additional message editor tabs. They create new text editor instances for every JOSE component by using Burp Suite's `ITextEditor`[13] interface and provided callback function[14].

### Attacks Package

The `attacks` package is one of the central parts of JOSEPH and contains all available attacks, organized in own sub-packages. Figure 4.11 shows the contents of the `attacks` package and a sample standard attack, together with their relation between each other. An attack package must consist of at least the three `[AttackName]`, `[AttackName]Info` and `[AttackName]AttackRequest` classes.

- The `AttackLoader` class is used as single point of management to register and retrieve all available attacks.
- The `IAttack` interface defines necessary methods which *must* be implemented by any `[AttackName]` attack class. Attack classes contain the main attack logic and operations to actually perform the attack by sending the previously prepared payload requests. All attack classes contain an inner class `AttackExecutor`, which extends from the abstract `SwingWorker`[15] class and is used to process time-consuming GUI-interaction tasks in a non-blocking background thread.

[12]PortSwigger Ltd., *Interface IMessageEditorTabFactory*, `URL: https://portswigger.net/burp/extender/api/burp/IMessageEditorTabFactory.html`
[13]PortSwigger Ltd., *Interface ITextEditor*, `URL: https://portswigger.net/burp/extender/api/burp/ITextEditor.html`
[14]PortSwigger Ltd., *Function createTextEditor()*, `URL: https://portswigger.net/burp/extender/api/burp/IBurpExtenderCallbacks.html#createTextEditor()`
[15]Oracle, *Class SwingWorker<T, V>*, `URL: https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html`

- The `IAttackInfo` interface defines necessary methods which *must* be implemented by any `[AttackName]Info` attack info class. Apart from meta data, such as the attack name and a short description, this class defines additional UI elements to retrieve necessary, supplementary information. Furthermore, it contains the logic to prepare the attack requests by applying several payloads.

- `AttackRequest` is an abstract class which *must* be extended by any `[AttackName]AttackRequest` attack request class. This class contains data about a single payload request and might be expanded to hold additional information for attack specific use.

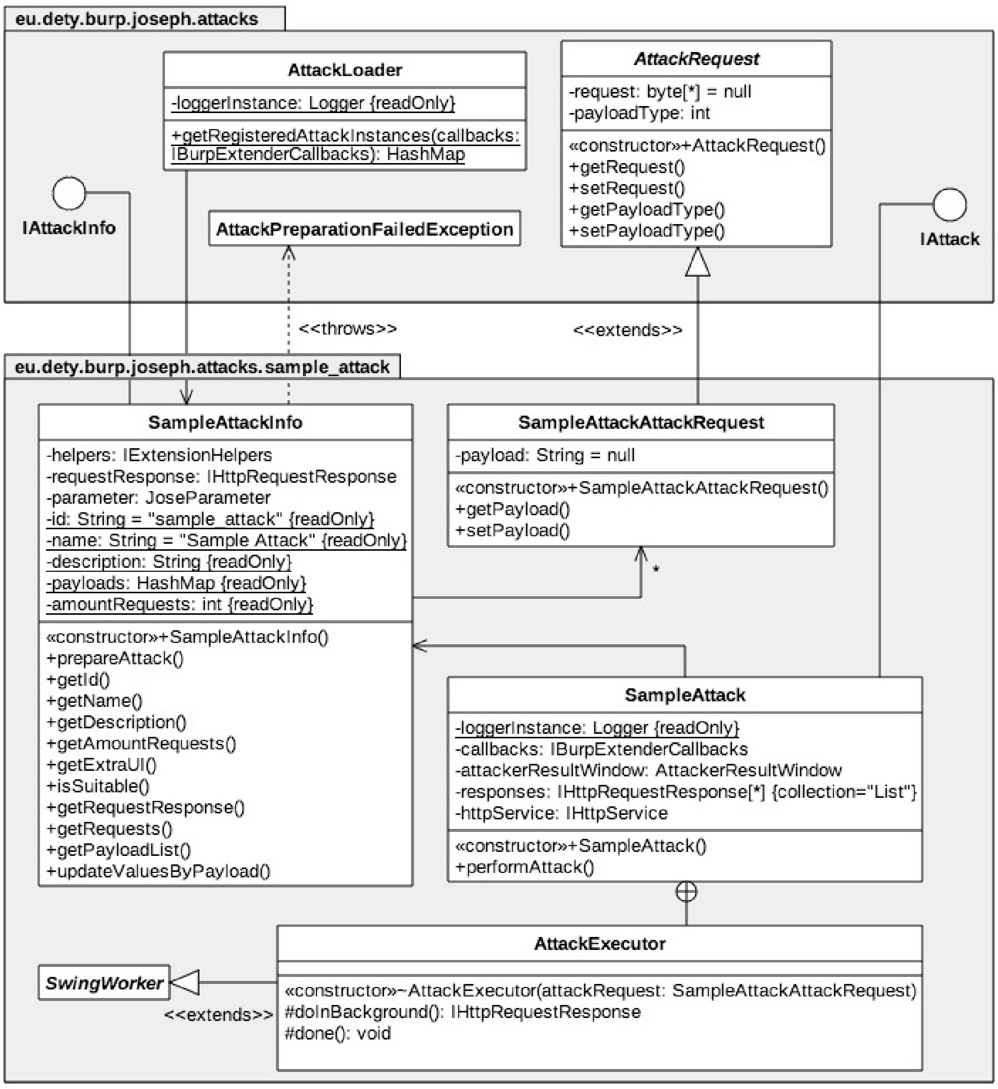

eu.dety.burp.joseph.attacks
AttackLoader
-loggerInstance: Logger {readOnly}
+getRegisteredAttackInstances(callbacks: IBurpExtenderCallbacks): HashMap
AttackRequest
-request: byte[*] = null
-payloadType: int
«constructor»+AttackRequest()
+getRequest()
+setRequest()
+getPayloadType()
+setPayloadType()
IAttackInfo
AttackPreparationFailedException
IAttack
<<throws>>
<<extends>>
eu.dety.burp.joseph.attacks.sample_attack
SampleAttackInfo
-helpers: IExtensionHelpers
-requestResponse: IHttpRequestResponse
-parameter: JoseParameter
-id: String = "sample_attack" {readOnly}
-name: String = "Sample Attack" {readOnly}
-description: String {readOnly}
-payloads: HashMap {readOnly}
-amountRequests: int {readOnly}
«constructor»+SampleAttackInfo()
+prepareAttack()
+getId()
+getName()
+getDescription()
+getAmountRequests()
+getExtraUI()
+isSuitable()
+getRequestResponse()
+getRequests()
+getPayloadList()
+updateValuesByPayload()
SampleAttackAttackRequest
-payload: String = null
«constructor»+SampleAttackAttackRequest()
+getPayload()
+setPayload()
*
SampleAttack
-loggerInstance: Logger {readOnly}
-callbacks: IBurpExtenderCallbacks
-attackerResultWindow: AttackerResultWindow
-responses: IHttpRequestResponse[*] {collection="List"}
-httpService: IHttpService
«constructor»+SampleAttack()
+performAttack()
AttackExecutor
«constructor»~AttackExecutor(attackRequest: SampleAttackAttackRequest)
#doInBackground(): IHttpRequestResponse
#done(): void
SwingWorker
<<extends>>

Figure 4.11.: General attack structure

Detailed instructions on how to develop and integrate a new custom attack are given in the next Section 4.4 *Extensibility*.

### The Bleichenbacher PKCS1 Attack Package

The `bleichenbacher_pkcs1` package is the currently most complex attack available in JOSEPH. Apart from the three necessary attack classes, the Bleichenbacher PKCS#1 attack includes the `BleichenbacherPkcs1DecryptionAttackExecutor` class which also extends the abstract `SwingWorker` class and is used to perform the second phase of the attack: the actual ciphertext decryption. Additionally, this class makes use of the custom `Interval` class, representing a single interval used by the Bleichenbacher algorithm, and the `BleichenbacherPkcs1Oracle` class, holding all oracle responses and their validity according to PKCS#1 v1.5. Furthermore, the `BleichenbacherPkcs1Oracle` class contains the logic to compare the validity of new responses based on Dice similarity metrics, as implemented by the used *SimMetrics* Java library of similarity and distance metrics[16].

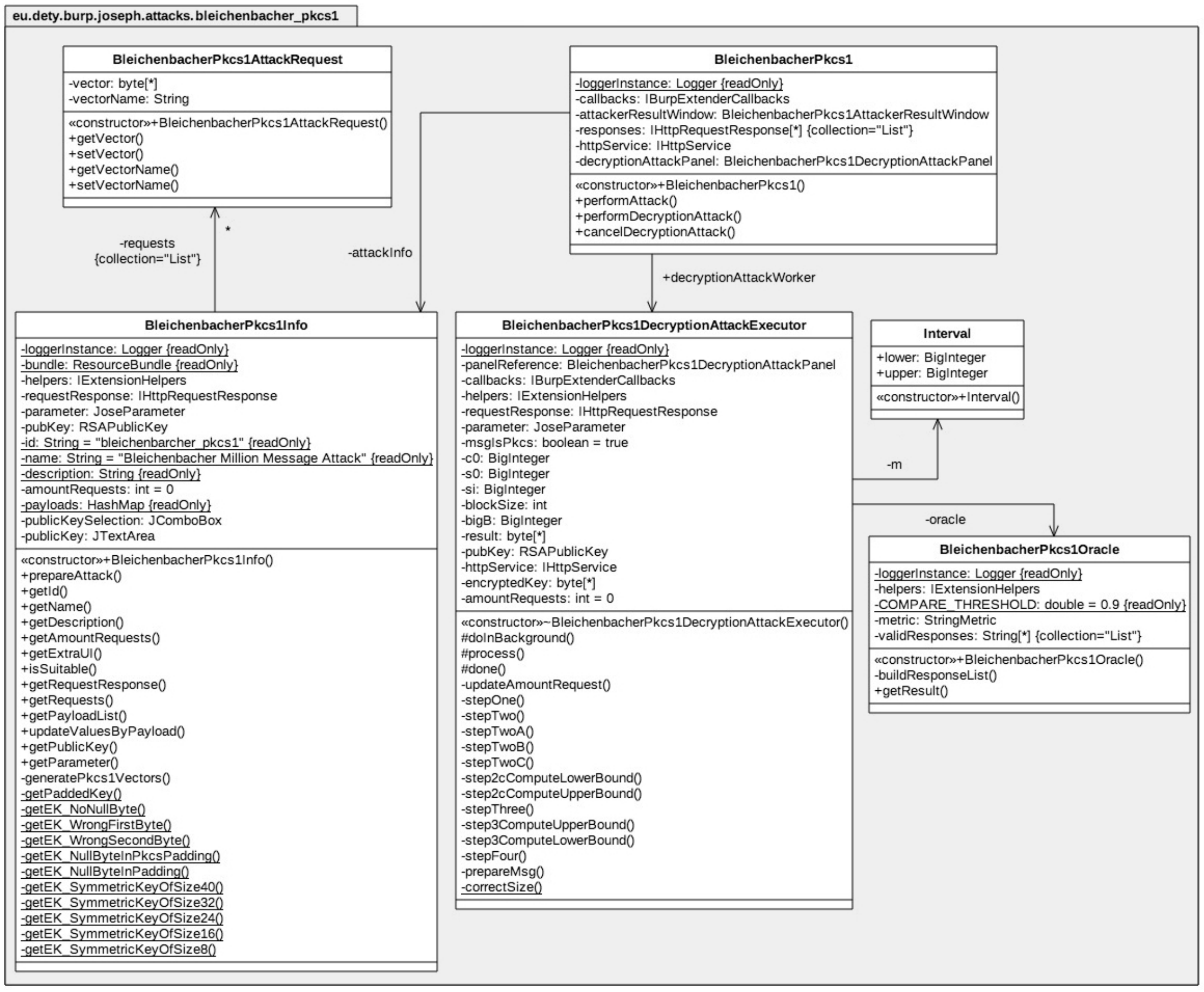

eu.dety.burp.joseph.attacks.bleichenbacher_pkcs1
BleichenbacherPkcs1AttackRequest
-vector: byte[*]
-vectorName: String
«constructor»+BleichenbacherPkcs1AttackRequest()
+getVector()
+setVector()
+getVectorName()
+setVectorName()
BleichenbacherPkcs1
-loggerInstance: Logger {readOnly}
-callbacks: IBurpExtenderCallbacks
-attackerResultWindow: BleichenbacherPkcs1AttackerResultWindow
-responses: IHttpRequestResponse[*] {collection="List"}
-httpService: IHttpService
-decryptionAttackPanel: BleichenbacherPkcs1DecryptionAttackPanel
«constructor»+BleichenbacherPkcs1()
+performAttack()
+performDecryptionAttack()
+cancelDecryptionAttack()
-requests {collection="List"}
*
-attackInfo
+decryptionAttackWorker
BleichenbacherPkcs1Info
-loggerInstance: Logger {readOnly}
-bundle: ResourceBundle {readOnly}
-helpers: IExtensionHelpers
-requestResponse: IHttpRequestResponse
-parameter: JoseParameter
-pubKey: RSAPublicKey
-id: String = "bleichenbarcher_pkcs1" {readOnly}
-name: String = "Bleichenbacher Million Message Attack" {readOnly}
-description: String {readOnly}
-amountRequests: int = 0
-payloads: HashMap {readOnly}
-publicKeySelection: JComboBox
-publicKey: JTextArea
«constructor»+BleichenbacherPkcs1Info()
+prepareAttack()
+getId()
+getName()
+getDescription()
+getAmountRequests()
+getExtraUI()
+isSuitable()
+getRequestResponse()
+getRequests()
+getPayloadList()
+updateValuesByPayload()
+getPublicKey()
+getParameter()
-generatePkcs1Vectors()
-getPaddedKey()
-getEK_NoNullByte()
-getEK_WrongFirstByte()
-getEK_WrongSecondByte()
-getEK_NullByteInPkcsPadding()
-getEK_NullByteInPadding()
-getEK_SymmetricKeyOfSize40()
-getEK_SymmetricKeyOfSize32()
-getEK_SymmetricKeyOfSize24()
-getEK_SymmetricKeyOfSize16()
-getEK_SymmetricKeyOfSize8()
BleichenbacherPkcs1DecryptionAttackExecutor
-loggerInstance: Logger {readOnly}
-panelReference: BleichenbacherPkcs1DecryptionAttackPanel
-callbacks: IBurpExtenderCallbacks
-helpers: IExtensionHelpers
-requestResponse: IHttpRequestResponse
-parameter: JoseParameter
-msgIsPkcs: boolean = true
-c0: BigInteger
-s0: BigInteger
-si: BigInteger
-blockSize: int
-bigB: BigInteger
-result: byte[*]
-pubKey: RSAPublicKey
-httpService: IHttpService
-encryptedKey: byte[*]
-amountRequests: int = 0
«constructor»~BleichenbacherPkcs1DecryptionAttackExecutor()
#doInBackground()
#process()
#done()
-updateAmountRequest()
-stepOne()
-stepTwo()
-stepTwoA()
-stepTwoB()
-stepTwoC()
-step2cComputeLowerBound()
-step2cComputeUpperBound()
-stepThree()
-step3ComputeUpperBound()
-step3ComputeLowerBound()
-stepFour()
-prepareMsg()
-correctSize()
Interval
+lower: BigInteger
+upper: BigInteger
«constructor»+Interval()
-m
-oracle
BleichenbacherPkcs1Oracle
-loggerInstance: Logger {readOnly}
-helpers: IExtensionHelpers
-COMPARE_THRESHOLD: double = 0.9 {readOnly}
-metric: StringMetric
-validResponses: String[*] {collection="List"}
«constructor»+BleichenbacherPkcs1Oracle()
-buildResponseList()
+getResult()

Figure 4.12.: Class diagram of the Bleichenbacher PKCS1 attack package

[16] M.P. Korstanje, *SimMetrics*, URL: `https://github.com/Simmetrics/simmetrics`

Figure 4.12 depicts the class diagram of the Bleichenbacher attack package without displaying any dependencies to classes and interfaces outside the package. Due to the complexity of this attack it defines its own GUI elements with a custom `AttackerResultWindow` class and amended `Table`, `TableModel` and `TableEntry` classes within a `gui` sub-package. Further, the new class `BleichenbacherPkcs1DecryptionAttackPanel` is defined and added as tab to the `AttackerResultWindow`, displaying the status and results of the second decryption phase of the attack. The class diagram of the `gui` sub-package is illustrated in Figure 4.13.

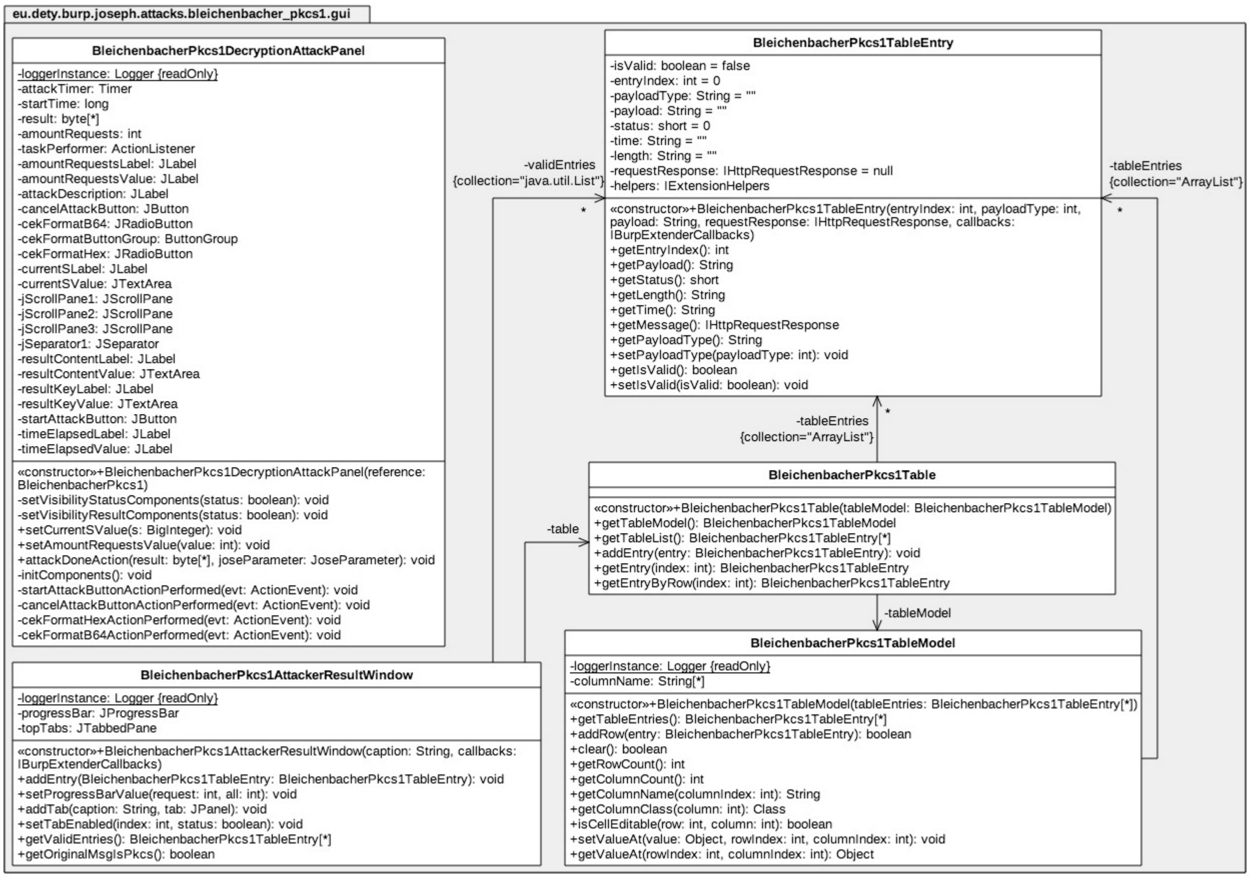

eu.dety.burp.joseph.attacks.bleichenbacher_pkcs1.gui
BleichenbacherPkcs1DecryptionAttackPanel
-loggerInstance: Logger {readOnly}
-attackTimer: Timer
-startTime: long
-result: byte[*]
-amountRequests: int
-taskPerformer: ActionListener
-amountRequestsLabel: JLabel
-amountRequestsValue: JLabel
-attackDescription: JLabel
-cancelAttackButton: JButton
-cekFormatB64: JRadioButton
-cekFormatButtonGroup: ButtonGroup
-cekFormatHex: JRadioButton
-currentSLabel: JLabel
-currentSValue: JTextArea
-jScrollPane1: JScrollPane
-jScrollPane2: JScrollPane
-jScrollPane3: JScrollPane
-jSeparator1: JSeparator
-resultContentLabel: JLabel
-resultContentValue: JTextArea
-resultKeyLabel: JLabel
-resultKeyValue: JTextArea
-startAttackButton: JButton
-timeElapsedLabel: JLabel
-timeElapsedValue: JLabel
«constructor»+BleichenbacherPkcs1DecryptionAttackPanel(reference: BleichenbacherPkcs1)
-setVisibilityStatusComponents(status: boolean): void
-setVisibilityResultComponents(status: boolean): void
+setCurrentSValue(s: BigInteger): void
+setAmountRequestsValue(value: int): void
+attackDoneAction(result: byte[*], joseParameter: JoseParameter): void
-initComponents(): void
-startAttackButtonActionPerformed(evt: ActionEvent): void
-cancelAttackButtonActionPerformed(evt: ActionEvent): void
-cekFormatHexActionPerformed(evt: ActionEvent): void
-cekFormatB64ActionPerformed(evt: ActionEvent): void
BleichenbacherPkcs1AttackerResultWindow
-loggerInstance: Logger {readOnly}
-progressBar: JProgressBar
-topTabs: JTabbedPane
«constructor»+BleichenbacherPkcs1AttackerResultWindow(caption: String, callbacks: IBurpExtenderCallbacks)
+addEntry(BleichenbacherPkcs1TableEntry: BleichenbacherPkcs1TableEntry): void
+setProgressBarValue(request: int, all: int): void
+addTab(caption: String, tab: JPanel): void
+setTabEnabled(index: int, status: boolean): void
+getValidEntries(): BleichenbacherPkcs1TableEntry[*]
+getOriginalMsgIsPkcs(): boolean
-validEntries
{collection="java.util.List"}
*
-table
BleichenbacherPkcs1TableEntry
-isValid: boolean = false
-entryIndex: int = 0
-payloadType: String = ""
-payload: String = ""
-status: short = 0
-time: String = ""
-length: String = ""
-requestResponse: IHttpRequestResponse = null
-helpers: IExtensionHelpers
«constructor»+BleichenbacherPkcs1TableEntry(entryIndex: int, payloadType: int, payload: String, requestResponse: IHttpRequestResponse, callbacks: IBurpExtenderCallbacks)
+getEntryIndex(): int
+getPayload(): String
+getStatus(): short
+getLength(): String
+getTime(): String
+getMessage(): IHttpRequestResponse
+getPayloadType(): String
+setPayloadType(payloadType: int): void
+getIsValid(): boolean
+setIsValid(isValid: boolean): void
-tableEntries
{collection="ArrayList"}
*
-tableEntries
{collection="ArrayList"}
*
BleichenbacherPkcs1Table
«constructor»+BleichenbacherPkcs1Table(tableModel: BleichenbacherPkcs1TableModel)
+getTableModel(): BleichenbacherPkcs1TableModel
+getTableList(): BleichenbacherPkcs1TableEntry[*]
+addEntry(entry: BleichenbacherPkcs1TableEntry): void
+getEntry(index: int): BleichenbacherPkcs1TableEntry
+getEntryByRow(index: int): BleichenbacherPkcs1TableEntry
-tableModel
BleichenbacherPkcs1TableModel
-loggerInstance: Logger {readOnly}
-columnName: String[*]
«constructor»+BleichenbacherPkcs1TableModel(tableEntries: BleichenbacherPkcs1TableEntry[*])
+getTableEntries(): BleichenbacherPkcs1TableEntry[*]
+addRow(entry: BleichenbacherPkcs1TableEntry): boolean
+clear(): boolean
+getRowCount(): int
+getColumnCount(): int
+getColumnName(columnIndex: int): String
+getColumnClass(column: int): Class
+isCellEditable(row: int, column: int): boolean
+setValueAt(value: Object, rowIndex: int, columnIndex: int): void
+getValueAt(rowIndex: int, columnIndex: int): Object

Figure 4.13.: Class diagram of the Bleichenbacher PKCS1 attack GUI sub-package

## 4.4. Extensibility

One of JOSEPH's design goals was its simple extensibility, mainly in the sense of easily adding new attacks to it. The sources contain an `__attack_template` package, which is an empty but fully working implementation of a standard attack, intended for use as a basis for a quick start. The following instructions provide a step-by-step guide to add a new attack to JOSEPH. This guide uses an exemplary attack called *Algorithm Check* that aims to check which JWS algorithms are supported on a system.

1. Copy the `__attack_template` package and rename it to the attack name: `algorithm_check`. This package contains the three required classes of a standard attack. Rename each three of them, following the given naming convention. This should result in having the three classes `AlgorithmCheck`, `AlgorithmCheckInfo` and `AlgorithmCheckAttackRequest`. All good Integrated Development Environments (IDEs) provide refactoring features, which should aid in this step being only a single click per class.

2. Register the new attack in the `AttackLoader` class. This can easily be done by adding the following (adjusted) snippet to the `getRegisteredAttackInstances()` function.

```
/* Algorithm Check Attack */
AlgorithmCheckInfo algorithmCheckInfo = new AlgorithmCheckInfo(callbacks);
registeredAttackInstances.put(algorithmCheckInfo.getName(),
    algorithmCheckInfo);
loggerInstance.log(AttackLoader.class, "Attack registered: Algorithm Check",
   Logger.LogLevel.INFO);
```

3. Within the `AlgorithmCheckInfo` class, change the meta data of the `id`, `name` and `description` variables to fit to the new attack.

```
// Unique identifier for the attack class
private static final String id = "algorithm_check";

// Full name of the attack
private static final String name = "Algorithm Check";

// Attack description
private static final String description = "<html>The <em>Algorithm Check</em>
   attack checks the supported algorithms of the implementing JWS system
   against a list of available algorithms from the specifications.</html>";
```

This is basically enough to add a new attack to JOSEPH. On recompilation, the *Algorithm Check* should appear in the list of available attacks. Next, the attack logic needs to be implemented.

4. For this specific attack, add a new variable `algorithms` which contains the list of algorithms that should be tested as array. Here, only three algorithms are included. This array can be easily extended with no additional effort. Additionally, dynamically calculate the amount of necessary requests by using the length of the `algorithms` array.

```
// Array of algorithms to test
private static final String[] algorithms = {"HS256", "RS256", "ES256"};

// Amount of requests needed
private static final int amountRequests = algorithms.length;
```

5. Each attack needs to list its different payload types as `enum`. In this case, it makes sense to name them equally to the tested algorithm.

```
// Types of payload variation
enum PayloadType {
   HS256,
   RS256,
   ES256
}
```

6. The `payloads` hashmap contains all available payloads together with a verbose name. To minimize the effort of extending the list of algorithms to be tested, this hashmap is dynamically created by using the `PayloadType` enumeration. This hashmap is used for the *Editor Attacker* tab and *Repeater* to select a single payload.

```
// Hashmap of available payloads with a verbose name
private static final HashMap<String, PayloadType> payloads = new HashMap<>();
static {
    for (PayloadType payload : PayloadType.values()) {
        put(String.format("Algorithm: %s", payload.name()), payload);
    }
}
```

7. Based on this list, the `updateValuesByPayload()` function can be implemented, which applies a single payload to the given JWS values. For the *Algorithm Check* attack, the `alg` algorithm header value is replaced with the payload's algorithm.

```
@Override
public HashMap<String, String> updateValuesByPayload(Enum payloadTypeId,
    String header, String payload, String signature) {
        HashMap<String, String> result = new HashMap<>();

        result.put("header", header.replaceFirst(
            "\"alg\":\"(.+?)\"",
            "\"alg\":\"" + payloadTypeId.name() + "\"")
        );
        result.put("payload", payload);
        result.put("signature", signature);

        return result;
}
```

8. If further user input is needed, amending the `getExtraUI()` function allows for adding additional UI elements. The `JPanel` and related constraints for the used `GridBagLayout`[17] passed to this function, enable a developer to easily add and position any attack specific widgets. As no extra UI is needed for this exemplary attack, returning `false` is enough.

```
@Override
public boolean getExtraUI(JPanel extraPanel, GridBagConstraints constraints) {
    return false;
}
```

However, to give a short example, adding an extra label and textarea might look like the following. It is important to change the return value to `true`, if any additional widgets are added.

```
@Override
public boolean getExtraUI(JPanel extraPanel, GridBagConstraints constraints) {
    JLabel extraBoxLabel = new JLabel("Extra UI Example");
    extraBox = new JTextArea(10, 50);
    extraBox.setLineWrap(true);

    constraints.gridy = 0;
    extraPanel.add(extraBoxLabel, constraints);

    constraints.gridy = 1;
    extraPanel.add(extraBox, constraints);

    return true;
}
```

9. Each attack needs to implement a check to determine the suitability of applying its payloads, based on the passed JOSE type and algorithm. This is done in the `isSuitable()` function. For the exemplary attack, it is sufficient to check whether the provided JOSE value is a JWS.

```
@Override
public boolean isSuitable(JoseParameter.JoseType type, String algorithm) {
    return (type == JoseParameter.JoseType.JWS);
}
```

[17]Oracle, *Class GridBagLayout*, URL: https://docs.oracle.com/javase/7/docs/api/java/awt/GridBagLayout.html

10. The `prepareAttack()` function prepares all requests that will be processed by launching the complete attack. Amend this function to add the specific `AlgorithmCheckAttackRequest` objects to the `requests` list.

```java
@Override
public AlgorithmCheck prepareAttack(IBurpExtenderCallbacks callbacks,
    IHttpRequestResponse requestResponse, IRequestInfo requestInfo,
    JoseParameter parameter) throws AttackPreparationFailedException {

  this.requestResponse = requestResponse;
  this.parameter = parameter;
  this.requests.clear();

  for (PayloadType payload : PayloadType.values()) {
    try {
      String[] tmpComponents =
        Decoder.getComponents(this.parameter.getJoseValue());
      String tmpDecodedHeader = Decoder.getDecoded(tmpComponents[0]);
      String tmpReplaced = tmpDecodedHeader.replaceFirst(
        "\"alg\":\"(.+?)\"",
        "\"alg\":\"" + payload.name() + "\""
      );
      String tmpReplacedEncoded = Decoder.getEncoded(tmpReplaced);
      String[] tmpNewComponents = {
        tmpReplacedEncoded,
        tmpComponents[1],
        tmpComponents[2]
      };
      String tmpParameterValue = Decoder.concatComponents(tmpNewComponents);

      byte[] tmpRequest = JoseParameter.updateRequest(
        this.requestResponse.getRequest(),
        this.parameter, helpers,
        tmpParameterValue
      );

      requests.add(new AlgorithmCheckAttackRequest(tmpRequest,
        payload.ordinal(), payload.name()));

    } catch (Exception e) {
      throw new AttackPreparationFailedException("Attack preparation failed.
         Message: " + e.getMessage());
    }
  }
  return new AlgorithmCheck(callbacks, this);
}
```

Those straightforward steps suffice to add a new simple attack. For showing the results, the standard `AttackerResultWindow` is used automatically. Use the `bleichenbacher_pkcs1` attack package as reference for more complicated attacks.

# 5. Conclusions and Future Work

This thesis featured critical attacks on JSON Web Signature and JSON Web Encryption, which still practically affect current applications. There exist several important areas in need of prospective research.

## Summary

The JavaScript Object Signing and Encryption working group put much effort into developing means to apply cryptographic mechanisms to JSON messages. All five specifications include a *Security Considerations* section to address well known vulnerabilities and common attacks, and provide detailed examples on how to work with the described methods.

Nevertheless, this thesis proved that a textual security considerations section within a specification is not sufficient to ensure secure implementations. The Bleichenbacher Million Message Attack is known for almost 20 years and the dangers of using standard methods for cryptographic operations have often been practically shown. During the research of this thesis we discovered several libraries being vulnerable to those attacks. Previous research of Tim McLean, with a special focus on JOSE implementations, revealed *signature exclusion* and *signature faking* issues, after which numerous libraries needed to be fixed. Thus, it is necessary to provide security mechanisms on the lowest possible level, which includes deprecating insecure cryptographic algorithms and parameters already within the specification.

JOSEPH is the first attempt to offer a tool to aid in security analyses of JOSE implementations and to test the susceptibility to known attacks. By providing the features of automatic recognition and visualization, semi-automatic testing of known attacks, manual manipulation and easy, dynamic extensibility, JOSEPH is hopefully a helpful contribution to the security community and a good starting point for further research and more secure implementations.

## Future Work

The JavaScript Object Signing and Encryption is a set of young specifications, which require a lot more research and evaluation. Apart from modifications to the specifications itself, to e.g. deprecate insecure cryptographic algorithms, it is necessary to further analyze real world implementations and applications. Some future research might include the following topics:

- *Replay attacks.* From the specification perspective, only JWTs provide possible means to protect against replay attacks, namely the *expiration time* and *JWT ID* claims. An analysis of the resistance of libraries and implementing applications against replay attacks might be of public interest.

- *Adaptive chosen-ciphertext attacks on CBC mode.* Between draft 05 and draft 06 of the JWA specification [24], the supported AES encryption algorithms using the Cipher Block Chaining (CBC) mode of operation for use with JWEs changed to include message authentication. The use of authenticated encryption prevents known attacks abusing the malleability of the CBC mode, presented in 2002 by Serge Vaudenay [63] and practically applicable to, for instance, XML Encryption [10]. The correctness of implementations and sufficiency of protection might be an interesting area of further investigation.
- *BERserk attack.* “In 2006, Daniel Bleichenbacher described a signature forgery attack against implementations of RSA signature verification which do not completely validate PKCS#1 v1.5 padding” [64] [65]. The exploitability of the PKCS#1 v1.5 weakness was successfully proven within this thesis, thus investigating the *BERserk* attack could be considered in future evaluations.
- *Invalid Curve Attacks.* Invalid curve attacks address Elliptic Curve Cryptography (ECC) algorithms and abuse missing checks of the membership of the cyclic group for the provided points. The paper *Practical Invalid Curve Attacks on TLS-ECDH* [66] gives an example of the practical exploitability in TLS implementations. In November 2016, Cedric Staub informed the security community about an invalid curve attack vulnerability for the ECDH-ES algorithm within the *go-jose* [67] JOSE library[1][2]. The applicability of invalid curve attacks on a variety of other libraries could be analyzed.

There exist many more known attacks against cryptographic systems and possible pitfalls between theoretically secure and practically implemented systems. Apart from research, future work on the JOSEPH Burp Suite extension will help penetration testers, library maintainers, developers and other users to easily test their implementations. Feedback on desired enhancements and occurring issues are likely to arise after publicizing JOSEPH as open source project. Addressing these requests will be part of meaningful future work.

[1]Email to the Openwall security mailinglist, `URL: http://www.openwall.com/lists/oss-security/2016/11/03/1`
[2]Release information of the security fixes, `URL: https://github.com/square/go-jose/releases/tag/v1.0.4`

# A. Appendix

## A.1. Base64 vs. Base64url

The following Listing A.1 gives a short example of the difference between `base64` and `base64url` encoding.

```
# Input
<jose>Any difference?</jose>

# Base64
PGpvc2U+QW55IGRpZmZlcmVuY2U/PC9qb3NlPg==

# Base64url
PGpvc2U-QW55IGRpZmZlcmVuY2U_PC9qb3NlPg
```

Listing A.1: Example showing the difference between `base64` and `base64url` encoding

## A.2. Registered Header Parameter

The following three tables list the registered header parameters in the IANA registry, as described in the JWK, JWS and JWE specifications.

Table A.1.: List of registered parameter available for use with JWE

| Header Param. | Name | Description | Required |
|---|---|---|---|
| `alg` | Algorithm | Identifies the cryptographic algorithm used to encrypt or determine the value of the CEK. | YES |
| `enc` | Encryption Algorithm | Identifies the content encryption algorithm used to perform authenticated encryption on the plaintext. | YES |
| `zip` | Compression Algorithm | The compression algorithm applied to the plaintext before being encrypted. The only value defined by the specification is `DEF`, representing the `DEFLATE` algorithm. | NO |
| `jku` | JWK Set URL | URI that refers to a resource for a set of JSON-encoded public keys. | NO |
| `jwk` | JSON Web Key | Contains the public key. | NO |
| `kid` | Key ID | Hint indicating which key was used. | NO |
| `x5u` | `X.509` URL | URI that refers to a resource for the `X.509` public key certificate or certificate chain. | NO |
| `x5c` | `X.509` Certificate Chain | Contains the `X.509` public key certificate or certificate chain. | NO |
| `x5t` | `X.509` Certificate `SHA-1` Thumbprint | `base64url`-encoded `SHA-1` digest of the `DER` encoding of the `X.509` certificate. | NO |
| `x5t#S256` | `X.509` Certificate `SHA-256` Thumbprint | `base64url`-encoded `SHA-256` digest of the `DER` encoding of the `X.509` certificate. | NO |
| `typ` | Type | Used by JWE applications to declare the media type of this complete JWE. | NO |
| `cty` | Content Type | Used by JWE applications to declare the media type of the secured content (payload). | NO |
| `crit` | Critical | Indicates that extensions (to JWE and/or JWA specifications) are being used that MUST be understood and processed. | NO |

Table A.2.: List of registered header parameter available for the JWS header

| Header Param. | Name | Description | Required |
|---|---|---|---|
| `alg` | Algorithm | Identifies the cryptographic algorithm used to secure the JWS | YES |
| `jku` | JWK Set URL | URI that refers to a resource for a set of JSON-encoded public keys. | NO |
| `jwk` | JSON Web Key | Contains the public key. | NO |
| `kid` | Key ID | Hint indicating which key was used. | NO |
| `x5u` | `X.509` URL | URI that refers to a resource for the `X.509` public key certificate or certificate chain. | NO |
| `x5c` | `X.509` Certificate Chain | Contains the `X.509` public key certificate or certificate chain. | NO |
| `x5t` | `X.509` Certificate `SHA-1` Thumbprint | `base64url`-encoded `SHA-1` digest of the `DER` encoding of the `X.509` certificate. | NO |
| `x5t#S256` | `X.509` Certificate `SHA-256` Thumbprint | `base64url`-encoded `SHA-256` digest of the `DER` encoding of the `X.509` certificate. | NO |
| `typ` | Type | Used by JWS applications to declare the media type of this complete JWS. | NO |
| `cty` | Content Type | Used by JWS applications to declare the media type of the secured content (payload). | NO |
| `crit` | Critical | Indicates that extensions (to JWS and/or JWA specifications) are being used that MUST be understood and processed. | NO |

Table A.3.: List of registered parameter available for use with JWK

| Header Param. | Name | Description | Required |
|---|---|---|---|
| `kty` | Key Type | Identifies the cryptographic algorithm family used with the key, such as `RSA` or `EC`. | YES |
| `use` | Public Key Use | Identifies the intended use of the public key and is employed to indicate whether a public key is used for encrypting data or verifying the signature on data. Values defined by this specification are `sig` (signatures) and `enc` (encryption). | NO |
| `key_ops` | Key Operations | Identifies the operation(s) for which the key is intended to be used, e.g. `sign`, `verify`, `encrypt`. | NO |
| `alg` | Algorithm | Identifies the algorithm intended for use with the key. | NO |
| `kid` | Key ID | Hint indicating which key was used. | NO |
| `x5u` | `X.509` URL | URI that refers to a resource for the `X.509` public key certificate or certificate chain. | NO |
| `x5c` | `X.509` Certificate Chain | Contains the `X.509` public key certificate or certificate chain. | NO |
| `x5t` | `X.509` Certificate `SHA-1` Thumbprint | `base64url`-encoded `SHA-1` digest of the `DER` encoding of the `X.509` certificate. | NO |
| `x5t#S256` | `X.509` Certificate `SHA-256` Thumbprint | `base64url`-encoded `SHA-256` digest of the `DER` encoding of the `X.509` certificate. | NO |

## A.3. Timing Attack

The example in Listing 3.12, showing measurable timing differences when comparing two strings by using native string comparison techniques, has been generated with the following sample python script:

```python
import time
from random import randint

ORIG = "73702ca3cf26f97b69d6891bff8f93a06d0bcc68"

results = {
  0:  {"hex": "0", "t": []},
  1:  {"hex": "1", "t": []},
  2:  {"hex": "2", "t": []},
  3:  {"hex": "3", "t": []},
  4:  {"hex": "4", "t": []},
  5:  {"hex": "5", "t": []},
  6:  {"hex": "6", "t": []},
  7:  {"hex": "7", "t": []},
  8:  {"hex": "8", "t": []},
  9:  {"hex": "9", "t": []},
  10: {"hex": "a", "t": []},
  11: {"hex": "b", "t": []},
  12: {"hex": "c", "t": []},
  13: {"hex": "d", "t": []},
  14: {"hex": "e", "t": []},
  15: {"hex": "f", "t": []},
}

for a in range(10000000):
  # Get random number between 0-15
  rand = randint(0,15)
  comp = results[rand]["hex"] + ("0" * 39)

  # Get start time
  start = time.clock()
  # Native string comparison
  comp == ORIG
  # Get end time
  end = time.clock()
  # Add time delta to result dictionary
  results[rand]["t"].append((end-start))

print ("\t\t%s(Original)" % ORIG)

# Print results
for idx, val in results.iteritems():
  print "%.13f %s" % (
    (reduce(lambda x, y: x + y, val["t"]) / float(len(val["t"]))),
    str(val["hex"]) + ("0" * 39)
  )
```

Listing A.2: Python script to illustrate the possibility to measure timing differences of the native string comparison.

For unadulterated results, this script has been tested on a freshly booted mini-PC running Ubuntu 12.04.5 in headless[1] mode, with 2 CPUs and 4 GB memory. The script itself has been run using Python 2.7.

For comparability, the script has also been tested on:

- An iMac with OSX 10.11.6, Intel Core i5 and 12 GB memory, not in headless mode.

```
     (Original) 73702ca3cf26f97b69d6891bff8f93a06d0bcc68
0.0000005887734 0000000000000000000000000000000000000000
0.0000005894727 1000000000000000000000000000000000000000
0.0000005877479 2000000000000000000000000000000000000000
0.0000005863251 3000000000000000000000000000000000000000
0.0000005869497 4000000000000000000000000000000000000000
0.0000005883077 5000000000000000000000000000000000000000
0.0000005873364 6000000000000000000000000000000000000000
0.0000006251094 7000000000000000000000000000000000000000
0.0000005892808 8000000000000000000000000000000000000000
0.0000005907609 9000000000000000000000000000000000000000
0.0000005881786 a000000000000000000000000000000000000000
0.0000005892698 b000000000000000000000000000000000000000
0.0000005876023 c000000000000000000000000000000000000000
0.0000005878498 d000000000000000000000000000000000000000
0.0000005914318 e000000000000000000000000000000000000000
0.0000005917146 f000000000000000000000000000000000000000
```

- A newly created Ubuntu 16.04.1 x64 server VM instance, with 2 CPUs and 2 GB Memory, in headless mode.

```
     (Original) 73702ca3cf26f97b69d6891bff8f93a06d0bcc68
0.0000006659684 0000000000000000000000000000000000000000
0.0000006634875 1000000000000000000000000000000000000000
0.0000006703510 2000000000000000000000000000000000000000
0.0000006721953 3000000000000000000000000000000000000000
0.0000006838214 4000000000000000000000000000000000000000
0.0000006565073 5000000000000000000000000000000000000000
0.0000006731525 6000000000000000000000000000000000000000
0.0000007237154 7000000000000000000000000000000000000000
0.0000006675485 8000000000000000000000000000000000000000
0.0000006669291 9000000000000000000000000000000000000000
0.0000006732856 a000000000000000000000000000000000000000
0.0000006700631 b000000000000000000000000000000000000000
0.0000006667463 c000000000000000000000000000000000000000
0.0000006707665 d000000000000000000000000000000000000000
0.0000006903125 e000000000000000000000000000000000000000
0.0000006622921 f000000000000000000000000000000000000000
```

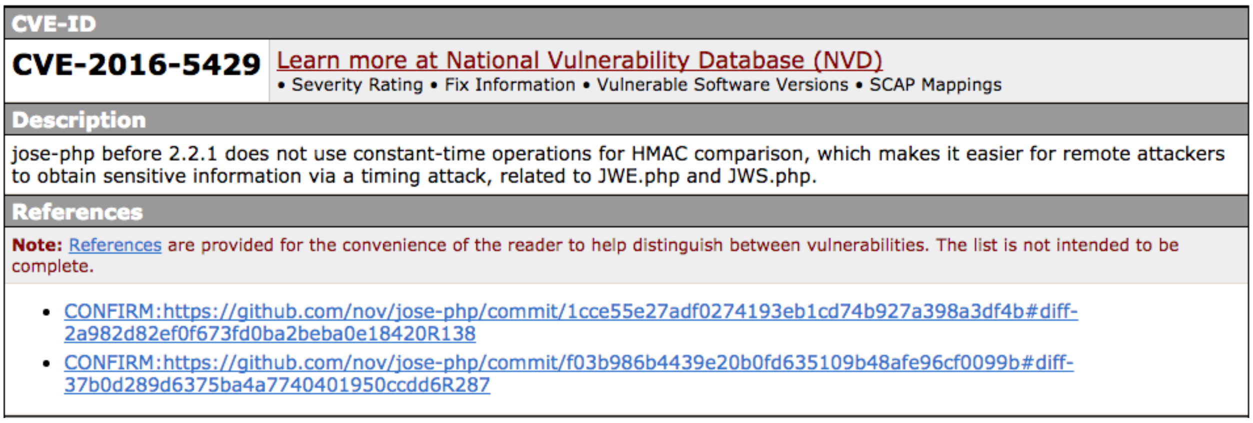
[1]"A headless system is a computer that operates without a monitor, graphical user interface (GUI) or peripheral devices, such as keyboard and mouse" [68].

## A.4. CVE Overview

The following Table A.4 gives an overview of the received CVE identifiers, along with short information about the vulnerability, affected library and its current status of publication at time of writing. Screenshots of the published CVE entries are depicted in the following three figures.

Table A.4.: List of received CVE identifiers

| CVE No. | Vulnerability | Library | Published |
|---|---|---|---|
| `CVE-2016-5429` | Timing-attack on HMAC comparison | jose-php by Nov Matake & Gree Inc. | ✓ |
| `CVE-2016-5430` | Bleichenbacher MMA | jose-php by Nov Matake & Gree Inc. | ✓ |
| `CVE-2016-5431` | Key Confusion attack mitigation deactivated by default | jose-php by Nov Matake & Gree Inc. | ✗ |
| `CVE-2016-6298` | Bleichenbacher MMA | jwcrypto by Simo Source | ✓ |
| `CVE-2016-7037` | Timing-attack on HMAC comparison | JWT by Malcolm Fell | ✗ |
| `CVE-2016-7036` | Timing-attack on HMAC comparison | python-jose by Michael Davis | ✗ |



**CVE-ID**

**CVE-2016-5429** Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

**Description**

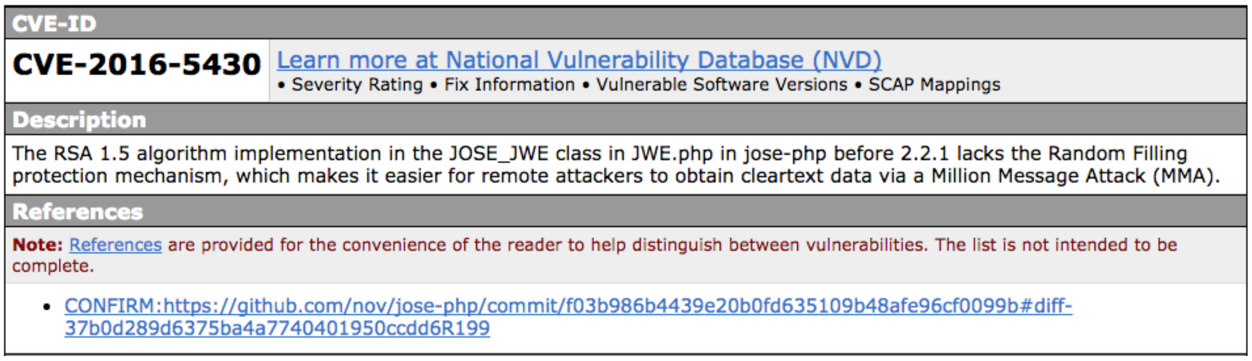
jose-php before 2.2.1 does not use constant-time operations for HMAC comparison, which makes it easier for remote attackers to obtain sensitive information via a timing attack, related to JWE.php and JWS.php.

**References**

**Note:** References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- CONFIRM:https://github.com/nov/jose-php/commit/1cce55e27adf0274193eb1cd74b927a398a3df4b#diff-2a982d82ef0f673fd0ba2beba0e18420R138
- CONFIRM:https://github.com/nov/jose-php/commit/f03b986b4439e20b0fd635109b48afe96cf0099b#diff-37b0d289d6375ba4a7740401950ccdd6R287

Figure A.1.: Screenshot of the published `CVE-2016-5429` entry[2]



**CVE-ID**

**CVE-2016-5430** Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

**Description**

The RSA 1.5 algorithm implementation in the JOSE_JWE class in JWE.php in jose-php before 2.2.1 lacks the Random Filling protection mechanism, which makes it easier for remote attackers to obtain cleartext data via a Million Message Attack (MMA).

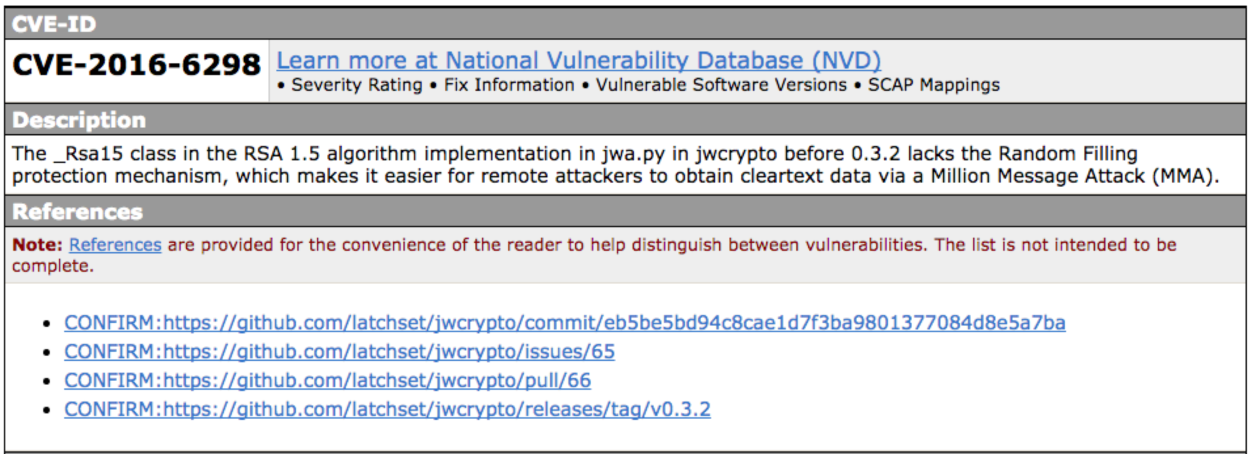
**References**

**Note:** References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- CONFIRM:https://github.com/nov/jose-php/commit/f03b986b4439e20b0fd635109b48afe96cf0099b#diff-37b0d289d6375ba4a7740401950ccdd6R199

Figure A.2.: Screenshot of the published `CVE-2016-5430` entry[3]

[2]URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5429`
[3]URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5430`



CVE-ID

**CVE-2016-6298** Learn more at National Vulnerability Database (NVD)
• Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings

Description

The _Rsa15 class in the RSA 1.5 algorithm implementation in jwa.py in jwcrypto before 0.3.2 lacks the Random Filling protection mechanism, which makes it easier for remote attackers to obtain cleartext data via a Million Message Attack (MMA).

References

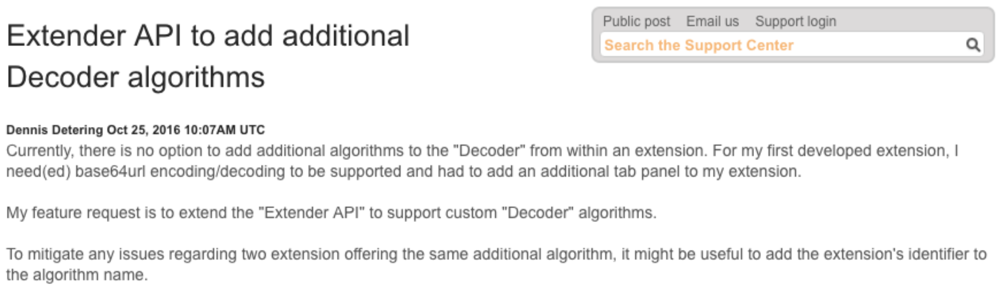
**Note:** References are provided for the convenience of the reader to help distinguish between vulnerabilities. The list is not intended to be complete.

- CONFIRM:https://github.com/latchset/jwcrypto/commit/eb5be5bd94c8cae1d7f3ba9801377084d8e5a7ba
- CONFIRM:https://github.com/latchset/jwcrypto/issues/65
- CONFIRM:https://github.com/latchset/jwcrypto/pull/66
- CONFIRM:https://github.com/latchset/jwcrypto/releases/tag/v0.3.2

Figure A.3.: Screenshot of the published `CVE-2016-6298` entry[4]

## A.5. Burp Suite Feature Requests

The Burp Suite contains a *Decoder* tab, which enables the user to use several algorithms for encoding and decoding. For working with JOSE representations, the `base64url`-encoding is needed. Unfortunately, there exists no API or other method to extend the existing *Decoder* algorithms and add new ones. Therefore, a completely new and standalone *Decoder* tab had to be developed and added to the JOSEPH extension. The related feature request to add such an API is shown in Figure A.4.



Extender API to add additional Decoder algorithms

**Dennis Detering Oct 25, 2016 10:07AM UTC**
Currently, there is no option to add additional algorithms to the "Decoder" from within an extension. For my first developed extension, I need(ed) base64url encoding/decoding to be supported and had to add an additional tab panel to my extension.

My feature request is to extend the "Extender API" to support custom "Decoder" algorithms.

To mitigate any issues regarding two extension offering the same additional algorithm, it might be useful to add the extension's identifier to the algorithm name.

Figure A.4.: Screenshot of the published Burp Suite feature request[5]

Additionally, extracting and working with HTTP headers from within the extension is not that easy and requires custom parsing. For parameters, there exists the `IParameter`[6] interface to operate on a single object and several methods to aid in manipulation of parameters within a request. There already existed a public feature request[7] for better HTTP header support, which has been put back on track with a comment of ours.

[4]URL: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-6298`
[5]URL: `https://support.portswigger.net/customer/en/portal/questions/16719383-extender-api-to-add-additional-decoder-algorithms?new=16719383`
[6]PortSwigger Ltd., *Interface IParameter*, URL: `https://portswigger.net/burp/extender/api/burp/IParameter.html`
[7]URL: `https://support.portswigger.net/customer/portal/questions/15940213-irequestinfo-getheaders`

# List of Figures

# List of Listings

# List of Tables

# List of Acronyms

**AAD** Additional Authenticated Data

**ACME** Automatic Certificate Management Environment

**AEAD** Authenticated Encryption with Associated Data

**AES** Advanced Encryption Standard

**API** Application Programming Interface

**CBC** Cipher Block Chaining Mode

**CEK** Content Encryption Key

**CAPEC** Common Attack Pattern Enumeration and Classification

**CBC** Cipher Block Chaining

**CSRF** Cross-Site Request Forgery

**CVE** Common Vulnerabilities and Exposures

**CWE** Common Weakness Enumeration

**DTD** Document Type Definitions

**ECC** Elliptic Curve Cryptography

**ECDSA** Elliptic Curve Digital Signature Algorithm

**FIPS** Federal Information Processing Standards

**GCM** Galois/Counter Mode

**GUI** Graphical User Interface

**HMAC** Keyed-Hash Message Authentication Code

**HTTP** Hypertext Transfer Protocol

**IANA** Internet Assigned Numbers Authority

**IDE** Integrated Development Environment

**IETF** Internet Engineering Task Force

**IPSec** Internet Protocol Security

**IV** Initialization Vector

| | |
|---|---|
| **JCE** | Java Cryptography Extension |
| **JDK** | Java Development Kit |
| **JOSE** | JavaScript Object Signing and Encryption |
| **JSON** | JavaScript Object Notation |
| **JWA** | JSON Web Algorithm |
| **JWK** | JSON Web Key |
| **JWS** | JSON Web Signature |
| **JWE** | JSON Web Encryption |
| **JWT** | JSON Web Token |
| **MAC** | Message Authentication Code |
| **MIME** | Multipurpose Internet Mail Extensions |
| **MMA** | Million Message Attack |
| **NIST** | National Institute of Standards and Technology |
| **OAEP** | Optimal Asymmetric Encryption Padding |
| **PEM** | Privacy Enhanced Mail |
| **RFC** | Request for Comments |
| **RSA** | Rivest-Shamir-Adleman |
| **SAML** | Security Assertion Markup Language |
| **SSL** | Secure Sockets Layer |
| **SSO** | Single Sign-on |
| **TLS** | Transport Layer Security |
| **UI** | User Interface |
| **URL** | Uniform Resource Locator |
| **XML** | Extensible Markup Language |

# Bibliography

[1] Security Services Technical Committee, "Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0." [Online]. Available: https://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[2] World Wide Web Consortium (W3C), "Extensible Markup Language (XML) 1.0 (Fifth Edition)." [Online]. Available: https://www.w3.org/TR/REC-xml/

[3] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159 (Proposed Standard), Internet Engineering Task Force, Mar. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7159.txt

[4] IETF jose Working Group. Javascript Object Signing and Encryption (jose). [Online]. Available: http://datatracker.ietf.org/wg/jose/

[5] J. Panzer, B. Laurie, and D. Balfanz, "Magic Signatures." [Online]. Available: https://salmon-protocol.googlecode.com/svn/trunk/draft-panzer-magicsig-01.html

[6] J. Bradley and N. Sakimura, "JSON Simple Sign 1.0 draft 01." [Online]. Available: http://jsonenc.info/jss/1.0/

[7] J. Bradley and N. Sakimura, "JSON Simple Encryptiono 1.0 draft 00." [Online]. Available: http://jsonenc.info/enc/1.0/

[8] D. Hardt and Y. Goland, "Simple Web Token." [Online]. Available: https://msdn.microsoft.com/de-de/library/azure/hh781551.aspx

[9] E. Rescorla and J. Hildebrand, "JavaScript Message Security Format," Internet Engineering Task Force, 2011. [Online]. Available: https://tools.ietf.org/html/draft-rescorla-jsms-00

[10] J. Somorovsky, "On the Insecurity of XML Security," Ph.D. dissertation, Ruhr-Universität Bochum, 2013.

[11] T. Jager and J. Somorovsky, "How To Break XML Encryption," in *The 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[12] D. Kupser, C. Mainka, J. Schwenk, and J. Somorovsky, "How to Break XML Encryption – Automatically," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser

[13] T. Jager, S. Schinzel, and J. Smorovksy, "Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption," in *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS 2012)*, 2012.

[14] T. Jager, K. Paterson, and J. Somorovsky, “One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography,” in *20th Annual Network & Distributed System Security Symposium*, 2013. [Online]. Available: http://nds.rub.de/media/nds/veroeffentlichungen/2013/03/08/BackwardsCompatibilityAttacks.pdf

[15] J. Forshaw, “Exploiting XML Digital Signature Implementations,” in *Hack In The Box - Kuala Lumpur 2013*, 2013.

[16] T. McLean, “Critical vulnerabilities in JSON Web Token libraries.” [Online]. Available: https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries/

[17] R. Bischofberger and E. Duss, “SAML Raider - SAML2 Burp Extension.” [Online]. Available: https://github.com/SAMLRaider/SAMLRaider

[18] T. Guenther, “Extension for Processing and Recognition of Single Sign-On Protocols (EsPReSSO).” [Online]. Available: https://github.com/RUB-NDS/BurpSSOExtension

[19] D. Crockford, “The application/json Media Type for JavaScript Object Notation (JSON),” RFC 4627 (Informational), Internet Engineering Task Force, July 2006, obsoleted by RFC 7159. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[20] The Unicode Consortium, “The Unicode Standard.” [Online]. Available: http://www.unicode.org/versions/latest/

[21] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Signature (JWS),” RFC 7515 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7515.txt

[22] M. Jones and J. Hildebrand, “JSON Web Encryption (JWE),” RFC 7516 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7516.txt

[23] M. Jones, “JSON Web Key (JWK),” RFC 7517 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7517.txt

[24] M. Jones, “JSON Web Algorithms (JWA),” RFC 7518 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7518.txt

[25] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” RFC 7519 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7519.txt

[26] HDKNR.COM, “JSON Web Encryption (JWE).” [Online]. Available: http://hdknr.github.io/docs/identity/jwe.html#introduction

[27] OpenID Foundation, “What is OpenID Connect?” [Online]. Available: http://openid.net/connect/

[28] R. Barnes, J. Hoffman-Andrews, and J. Kasten, “Automatic Certificate Management Environment (ACME),” Internet Engineering Task Force, Internet-Draft draft-ietf-acme-acme-04, Oct. 2016, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-acme-acme-04

[29] Internet Security Research Group (ISRG), “Let’s Encrypt.” [Online]. Available: https://letsencrypt.org/

[30] Atlassian, “What is Atlassian Connect?” [Online]. Available: https://developer.atlassian.com/static/connect/docs/latest/guides/introduction.html

[31] IBM Deutschland GmbH, “IBM DataPower Gateway.” [Online]. Available: http://www-03.ibm.com/software/products/de/datapower-gateway

[32] Apache Software Foundation, “JAX-RS JOSE.” [Online]. Available: http://cxf.apache.org/docs/jax-rs-jose.html

[33] S. Josefsson, “The Base16, Base32, and Base64 Data Encodings,” RFC 4648 (Proposed Standard), Internet Engineering Task Force, Oct. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4648.txt

[34] PortSwigger Ltd, “Burp Suite.” [Online]. Available: https://portswigger.net/burp/

[35] The Open Web Application Security Project, “Fuzzing.” [Online]. Available: https://www.owasp.org/index.php/Fuzzing

[36] National Institute of Standards and Technology, “FIPS General Information.” [Online]. Available: https://www.nist.gov/information-technology-laboratory/fips-general-information

[37] M. Jones, “JSON Web Algorithms (JWA) Draft 15,” RFC 7518 (Proposed Standard), Internet Engineering Task Force, 2013. [Online]. Available: https://tools.ietf.org/html/draft-ietf-jose-json-web-algorithms-15

[38] A. Nadalin, “JSON Object Signing and Encryption library for PHP.” [Online]. Available: https://github.com/namshi/jose

[39] N. Matake and GREE Inc., “PHP JOSE (Javascript Object Signing and Encryption) Implementation.” [Online]. Available: https://github.com/nov/jose-php

[40] P. Félix, “Some thoughts on the recent JWT library vulnerabilities.” [Online]. Available: https://pfelix.wordpress.com/2015/04/11/some-thoughts-on-the-recent-jwt-library-vulnerabilities/

[41] 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*, ser. CRYPTO ’98. London, UK, UK: Springer-Verlag, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=646763.706320

[42] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, “Efficient padding oracle attacks on cryptographic hardware,” in *Proceedings of the 32Nd Annual Cryptology Conference on Advances in Cryptology — CRYPTO 2012 - Volume 7417*. New York, NY, USA: Springer-Verlag New York, Inc., 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32009-5_36

[43] V. Klíma, O. Pokorný, and T. Rosa, *Attacking RSA-Based Sessions in SSL/TLS*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45238-6_33

[44] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, and E. Simon, "XML Encryption Syntax and Processing," W3C Recommendation, December 2002. [Online]. Available: https://www.w3.org/TR/xmlenc-core/

[45] J. Jonsson and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1," RFC 3447 (Informational), Internet Engineering Task Force, Feb. 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3447.txt

[46] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: Breaking pkcs#1 v1.5 in xml encryption," in *ESORICS*, ser. Lecture Notes in Computer Science, S. Foresti, M. Yung, and F. Martinelli, Eds., vol. 7459. Springer, 2012. [Online]. Available: http://dblp.uni-trier.de/db/conf/esorics/esorics2012.html#JagerSS12

[47] MITRE. CAPEC-463: Padding oracle crypto attack. [Online]. Available: http://capec.mitre.org/data/definitions/463.html

[48] B. Kaliski, "PKCS #1: RSA Encryption Version 1.5," RFC 2313 (Informational), Internet Engineering Task Force, Mar. 1998, obsoleted by RFC 2437. [Online]. Available: http://www.ietf.org/rfc/rfc2313.txt

[49] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting ssl/tls implementations: New bleichenbacher side channels and attacks," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer

[50] S. Source, "JWCrypto – Implementation of JOSE Web standards." [Online]. Available: https://github.com/latchset/jwcrypto

[51] N. Matake, "JSON::JWT – JSON Web Token and its family (JSON Web Signature, JSON Web Encryption and JSON Web Key) in Ruby." [Online]. Available: https://github.com/nov/json-jwt

[52] latchset, "C-language implementation of Javascript Object Signing and Encryption." [Online]. Available: https://github.com/latchset/jose

[53] E. Rescorla, "Preventing the Million Message Attack on Cryptographic Message Syntax," RFC 3218 (Informational), Internet Engineering Task Force, Jan. 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3218.txt

[54] B. Kaliski and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437 (Informational), Internet Engineering Task Force, Oct. 1998, obsoleted by RFC 3447. [Online]. Available: http://www.ietf.org/rfc/rfc2437.txt

[55] MITRE. CWE-385: Covert timing channel. [Online]. Available: https://cwe.mitre.org/data/definitions/385.html

[56] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Comput. Netw.*, vol. 48, no. 5, Aug. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2005.01.010

[57] B. B. Brumley and N. Tuveri, *Remote Timing Attacks Are Still Practical.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23822-2_20

[58] S. A. Crosby, D. S. Wallach, and R. H. Riedi, "Opportunities and limits of remote timing attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 3, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1455526.1455530

[59] M. Fell, "JWT – An implementation of the JSON Web Token (JWT) draft in PHP." [Online]. Available: https://github.com/emarref/jwt

[60] M. Davis, "A JOSE implementation in Python." [Online]. Available: https://github.com/mpdavis/python-jose/

[61] NCC Group, "Double hmac verification." [Online]. Available: https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2011/february/double-hmac-verification/

[62] S. Arciszewski, "Preventing timing attacks on string comparison with a double hmac strategy." [Online]. Available: https://paragonie.com/blog/2015/11/preventing-timing-attacks-on-string-comparison-with-double-hmac-strategy

[63] S. Vaudenay, "Security flaws induced by cbc padding - applications to ssl, ipsec, wtls ..." in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, ser. EUROCRYPT '02. London, UK, UK: Springer-Verlag, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=647087.715705

[64] Intel Security, "BERserk Vulnerability." [Online]. Available: http://www.intelsecurity.com/resources/wp-berserk-analysis-part-1.pdf

[65] H. Finney, "Bleichenbacher's RSA signature forgery based on implementation error." [Online]. Available: https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html

[66] T. Jager, J. Schwenk, and J. Somorovsky, *Practical Invalid Curve Attacks on TLS-ECDH*. Cham: Springer International Publishing, 2015. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24174-6_21

[67] Square, Inc., "An implementation of JOSE standards (JWE, JWS, JWT) in Go." [Online]. Available: https://github.com/square/go-jose/

[68] TechTarget, "Definition: Headless System." [Online]. Available: http://internetofthingsagenda.techtarget.com/definition/headless-system