On The (In-)Security Of JavaScript Object Signing And Encryption

Dennis Detering
CSPi GmbH
dennis.detering@cspi.com

Juraj Somorovsky
Horst Görtz Institute for IT Security,
Ruhr-University Bochum
juraj.somorovsky@rub.de

Christian Mainka
Horst Görtz Institute for IT Security,
Ruhr-University Bochum
christian.mainka@rub.de

Vladislav Mladenov
Horst Görtz Institute for IT Security,
Ruhr-University Bochum
vladislav.mladenov@rub.de

Jörg Schwenk
Horst Görtz Institute for IT Security,
Ruhr-University Bochum
joerg.schwenk@rub.de

ABSTRACT

JavaScript Object Notation (JSON) has evolved to the de-facto standard file format in the web used for application configuration, cross- and same-origin data exchange, as well as in Single Sign-On (SSO) protocols such as OpenID Connect. To protect integrity, authenticity, and confidentiality of sensitive data, JavaScript Object Signing and Encryption (JOSE) was created to apply cryptographic mechanisms directly in JSON messages.

We investigate the security of JOSE and present different applicable attacks on several popular libraries. We introduce JOSEPH (JavaScript Object Signing and Encryption Pentesting Helper) – our newly developed Burp Suite extension, which automatically performs security analysis on targeted applications. JOSEPH's automatic vulnerability detection ranges from executing simple signature exclusion or signature faking techniques, which neglect JSON message integrity, up to highly complex cryptographic Bleichenbacher attacks, breaking the confidentiality of encrypted JSON messages. We found severe vulnerabilities in six popular JOSE libraries. We responsibly disclosed all weaknesses to the developers and helped them to provide fixes.

KEYWORDS

JOSE, JSON Web Encryption, JSON Web Signature, Key Confusion, Bleichenbacher Attack, Burp Suite

1 INTRODUCTION

Many applications available on the World Wide Web rely on secure communication channels on lower layers, such as Internet Protocol Security (IPSec) [14] or Transport Layer Security (TLS) [8]. These mechanisms provide end-to-end encryption in point-to-point scenarios, where the complete data is securely transported between two communication partners. However, IPSec and TLS become insufficient in complex scenarios where communication is redirected over untrusted third-party intermediates (proxies), or only specific message parts have to be protected. These cases demand additional security technologies on the application layer which provide the fundamental security concepts integrity, authenticity, and confidentiality of arbitrary elements directly on the message level.

In practice, the Extensible Markup Language (XML)-based secure object formats *XML Signature* [11] and *XML Encryption* [9] enjoy great popularity. They have been adopted in many widely deployed protocols and systems using, for example, Security Assertion Markup Language (SAML)-based Single Sign-On [49]. One of the main disadvantages of XML [55] is its high complexity; XML parsers need to support XML namespaces, Document Type Definitions (DTD), different node types, canonicalization, or processing of cryptographic algorithms. XML complexity and its manifold features have led to various errors and vulnerabilities, allowing attackers to perform simple Denial-of-Service attacks [39] or to decrypt encrypted XML contents [22].

An alternative to XML is the JSON [6]. JSON is a platform-independent data format which operates with only four primitives and two structured types [6]. Its simplicity and small set of formatting rules facilitate a developer's effort by implementing JSON-based data structures and reducing network load. With the increased usage of JSON in REST services and protocols like OAuth and OpenID Connect security provided by encryption, digital signatures, and Message Authentication Code (MAC) algorithms is a desired goal. This demand has been addressed by the JOSE working group, which proposed five new Request for Comments (RFC) specifications [24–28].

In this paper we investigate the security of JavaScript Object Signing and Encryption (JOSE). We analyze several vulnerabilities and their practical exploitability based on real world library implementations. *Signature Exclusion* and *Key Confusion* are addressed by JSON Web Signature [26] and are 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 [28]. This attack exploits specific system behavior to recover the encrypted plaintext message. We found vulnerabilities in the latest versions of major JOSE libraries. All in all, six Common

ROOTS, November 16–17, 2017, Vienna, Austria

ACM ISBN 978-1-4503-5321-2/17/11…$15.00
https://doi.org/10.1145/3150376.3150379

Vulnerabilities and Exposures (CVE) identifiers have been assigned and all issues were fixed in cooperation with the maintainers.

To support library developers and security researchers by evaluating JOSE libraries we developed a Burp Suite extension to test implementations for their resistance of the examined attacks. Furthermore, the extension provides several features to assist in manual testing and is easily extensible to add further discovered attacks and checks.

Contributions.

- We analyzed both simple and complex state-of-the-art attacks on JavaScript Object Signing and Encryption.
- We discovered six vulnerabilities in widely used software libraries and responsibly disclosed them to the developers.
- We provide our open source Burp Suite extension called JavaScript Object Signing and Encryption Pentesting Helper (JOSEPH) [7].

2 JAVASCRIPT OBJECT SIGNING AND ENCRYPTION

The JavaScript Object Notation is a "lightweight, text-based, language-independent data interchange format [...] derived from the ECMAScript Programming Language Standard" [6]. In May 2015, the JavaScript Object Signing and Encryption (JOSE) working group [1] standardized two security standards: JSON Web Signature (JWS) [26] and JSON Web Encryption (JWE) [28]. Along with the these standards JSON Web Key (JWK) [25], JSON Web Algorithm (JWA) [24], and JSON Web Token (JWT) [27].[1] These standards have already been integrated into several major protocols, frameworks, and applications. This include SSO protocols like OpenID Connect [43], the Automatic Certificate Management Environment (ACME) protocol [3] used by *Let's Encrypt* [18], the IBM DataPower Gateway solution [16], and Apache's CXF Webservice framework [2].

The JSON Web Algorithm enumerates cryptographic algorithms and identifiers represented in JSON-based data structures [24]. It describes the semantics and operations used with the JSON Web Signature, JSON Web Encryption, and JSON Web Key specifications. JSON Web Key represents a cryptographic key in a JSON data structure as used in the JSON Web Signature and JWE specifications [25]. The specification lists and describes several parameters registered in the IANA registry for use with JWKs. JSON Web Token is used to transfer claims[2] in a compact, URL-safe representation using the *JWS/JWE Compact Serialization* between two parties [27]. "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" [27, Abstract].

In the following, we describe JSON Web Signature and JSON Web Encryption standards due to the relevancy to our work.

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

[2]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*" [27, Section 2]

2.1 JSON Web Signature

JSON Web Signature specifies methods and algorithms to protect integrity and authenticity of JSON-based data [26]. The available algorithms include, for example, HMAC, RSA-PKCS#1 v1.5 or ECDSA with SHA-256, SHA-384, or SHA-512.

JWS Serialization. The JWS specification defines two types of serialization methods to represent a JWS. 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" [26, Section 2]. Listing 1 presents an example using the *general* JWS JSON Serialization syntax and "demonstrates the capability for conveying multiple digital signatures and/or MACs for the same payload" [26, Appendix A.6]. The first digital signature has been generated with the RSA algorithm and the second one by using ECDSA. The `header` element contains the ID of the public key used for the signature computation. It can include further information such as algorithms or issuer information.

```
{
"payload": "eyJpc3MiOiJqb2UiLA0 ... 19yb290Ijp0cnVlfQ",
"signatures":[{
"protected": "eyJhbGciOiJSUzI1NiJ9",
"header": {"kid":"2010-12-29"},
"signature": "cC4hiUPoj9E ... etdgtv3hF80EGrhuGe77Rw"
},{
"protected": "eyJhbGciOiJFUzI1NiJ9",
"header": {"kid":"e9bc097a-ce51-4036-9562-d2ade882db0d"},
"signature": "DtEhljbEg88VWAKAM ... mWQxfKTUJqPP3-Kg6NU1Q"
}]
}
```

Listing 1: JSON Web Signature in its General JWS JSON Serialization representation [26, Appendix A.6.4]

The *JWS Compact Serialization* is a compact and URL-safe string representation. An example is depicted in Listing 2, showing the three `base64url`-encoded and concatenated resulting strings [30]. All samples used in this paper are shown in its *JWS Compact Serialization* representation.

```
eyJhbGciOiJSUzI1NiJ9              # Header
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9le
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ    # Payload
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZja6h
AAuHIm4Bh-0Qc_lF5YK ... I8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw            # Signature
```

Listing 2: JSON Web Signature in its JWS Compact Serialization representation [26, Appendix A.2.1]

Signature Computation. To create a compact JWS, the following steps have to be performed (see also the graphical illustration in Figure 1):

(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 `BASE64URL(JWS Payload)` *(green)*

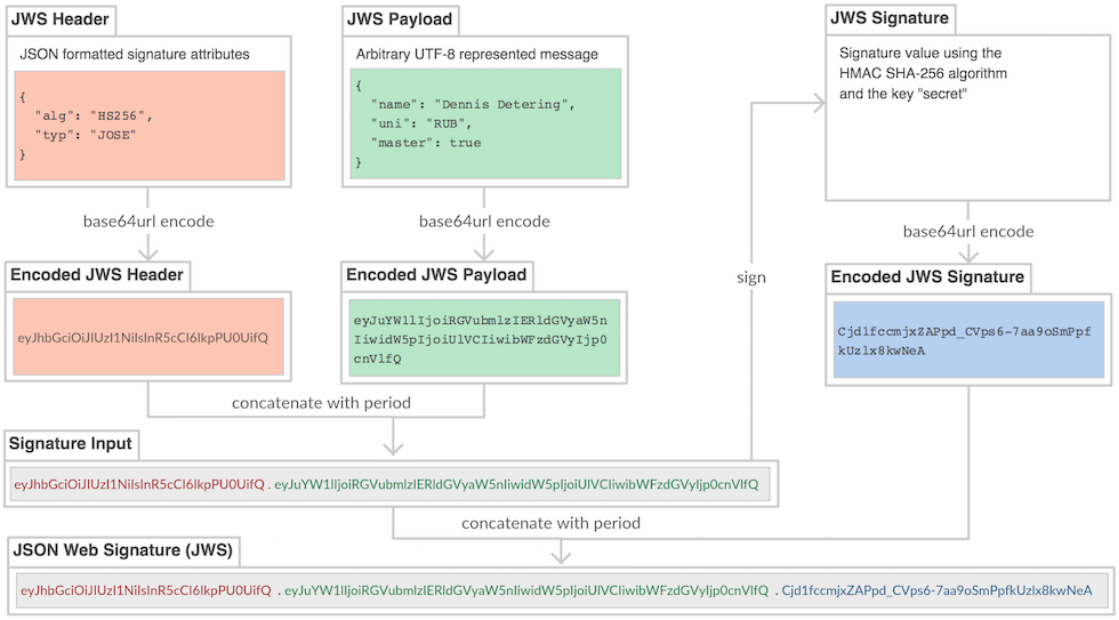

Figure 1: Process of generating a JSON Web Signature

(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 `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)`

Note that some steps may vary slightly if the JWS JSON Serialization is used.

2.2 JSON Web Encryption

JSON Web Encryption provides authenticated encryption to ensure confidentiality, authenticity, and integrity of an arbitrary sequence of octets using JSON-based data structures [28]. The available algorithms are AES-GCM, AES-KW, or AES-CBC with HMAC (with different key sizes).

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 Content Encryption Key (CEK), while the `enc` parameter holds the identifier of the content encryption algorithm used to perform authenticated encryption on the plaintext [28].

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*. Figure 2 depicts one example of a JWE in its *JWE Compact Serialization* representation. In this example, 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.

Message Encryption. The process of generating a JWE strongly depends on the used algorithms and contains up to 19 steps, described in detail in [28, Section 5.1]. One example of how a JWE might be generated is graphically illustrated in Figure 2:

(1) Create the JSON objects building the header(s) and encode them using `base64url` *(red)*.
(2) Generate Content Encryption Key. Depending on the key encryption algorithm specified in the `alg` header parameter, the CEK can contain a randomly generated secret which is encrypted with the recipient's key, or a public key share used for the elliptic curve key agreement [28] *(purple)*. These are two exemples of the existing five Key Mangement Modes which describe the CEK generation process [28, Section 2].
(3) Encode the CEK value using `base64url`. Generate an Initialization Vector (IV) and encode it with `base64url` *(blue)*. If compression is enabled, the plaintext must be compressed.
(4) Use the selected encryption algorithm to process the plaintext *(green)* using the CEK and IV values from the previous steps. Compute the authentication tag over the ciphertext and Additional Authenticated Data (AAD) (if present) *(white)*. The result is the ciphertext *(yellow)* and the authentication tag *(orange)*. These values are all encoded using `base64url`.

3 BURP SUITE

Burp Suite is "an integrated platform for performing security testing of web applications" [44] developed by PortSwigger Ltd. and

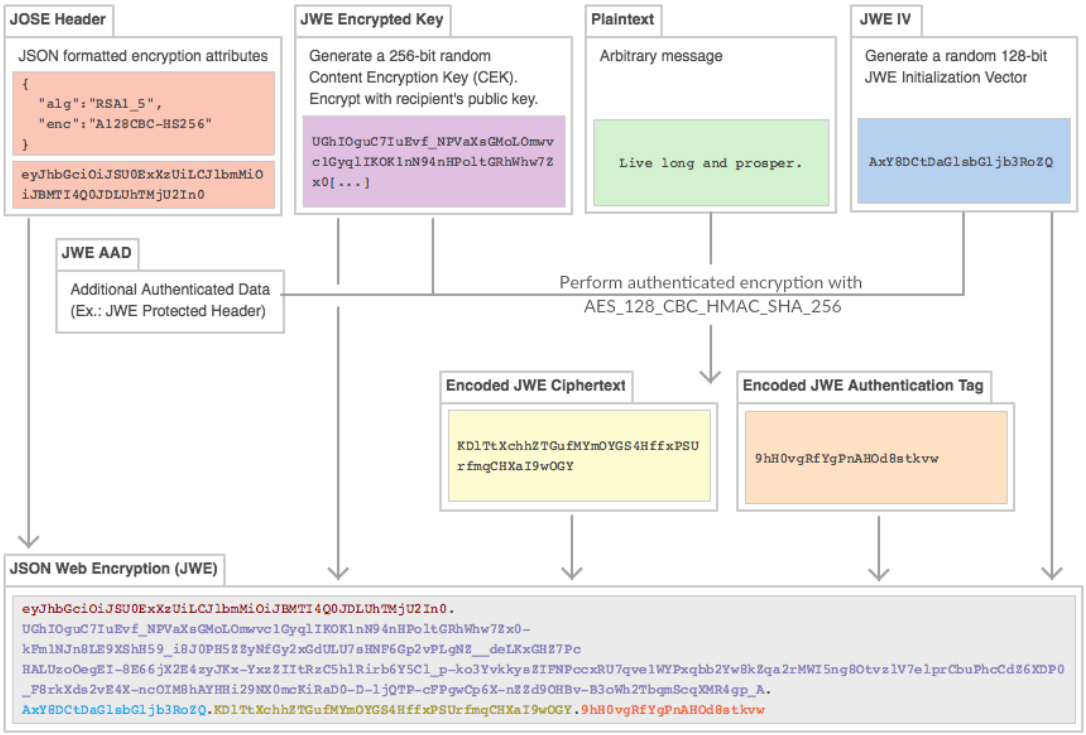

Figure 2: Process of generating a JSON Web Encryption

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" [44] 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" [44]. 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" [44]. The vulnerability scanner is only available in the *professional* version of the Burp Suite and can both actively and passively scan the target for a large list of known security issues [45].

Intruder. "An intruder tool for performing powerful customized attacks to find and exploit unusual vulnerabilities" [44]. Its main characteristic is to inspect specific entry points, such as parameters or headers, by performing, for instance, brute force, fuzzing,[3] or enumeration checks.

Repeater. "A repeater tool for manipulating and resending individual requests" [44]. 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" [44]. By collecting a list of samples for a session, Cross-Site Request Forgery (CSRF), or other security-relevant tokens, this tool estimates the degree of randomness and analyzes its quality – including the standard FIPS[4] tests.

Extender. An extension Application Programming Interface (API) "allowing [one] to easily [develop custom] plugins, to perform complex and highly customized tasks within Burp" [44]. Burp Suite in general is closed source and only exposes certain interfaces and functions for public access [47]. Its functionality can be extended with plugins developed in Java, Python by using JPython, or Ruby by using JRuby. Developers have the opportunity to publish their bundled extensions to the *BApp Store* [46], Burp Suite's own application store containing extensions written by its community.

[3] "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 [53]."
[4] Federal Information Processing Standards (FIPS) are standards and guidelines developed by the National Institute of Standards and Technology (NIST) [41].

4 SELECTED ATTACKS ON JOSE

The following sections describe selected attacks against JWT, JWS, and JWE. For each attack, the basic idea and underlying problem is explained.

4.1 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.

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 [33]. 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.

An example of a vulnerable library is given in Listing 3. In order to check whether a given algorithm is supported and to load its related class, the PHP JOSE [40] library relies on the mandatory `alg` header value and checks the existence of a class with this name on a certain location using the namespace definition. Listing 3 shows the implementation of the `getSigner()` function, which performs this check (l. 4) and returns a new instance if the class exists (l. 5). If invalid, an `InvalidArgumentException` is thrown, stating that the given algorithm is not supported (l. 7). In case of the "None" signer, the `verify()` function returns `true` if the signature is empty (Listing 4).

```
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: PHP JOSE `getSigner()` function (version 2.1.3).[5]

```
1 public function verify($key, $signature, $input) {
2     return $signature === '';
3 }
```

Listing 4: PHP JOSE `verify()` function of the None signer.

With version 2.1.3, an `allowUnsecure` flag has been introduced and set to `false` by default (Listing 5, 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 whether `allowUnsecure` is permitted (Listing 5, ll. 4-6). As a result, the creation of a `Signer` instance will be mitigated and the verification process is stopped with an exception.

```
1 public static function load($jwsTokenString,
2     $allowUnsecure = false)
3 [...]
4  if ($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 5: A vulnerable version of the PHP JOSE library.[6]

The problem with this amendment is how the algorithm value has been checked. An attacker might have used different capitalization to bypass this check, since the class name is matched in a case-insensitive manner.

The 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 6 shows the GIT diff call of the related file.

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

Listing 6: Commit diff excerpt of the PHP JOSE library showing the changes to fix the case-sensitivity of the algorithm value.[7]

4.2 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 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

[5] See: https://github.com/namshi/jose/blob/master/src/Namshi/JOSE/JWS.php for full file.

[6] See: https://github.com/namshi/jose/commit/127b4415e66d89b1fcfb5a07933db0b5ff5cd636 for full commit.

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

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) together is considered realistic.

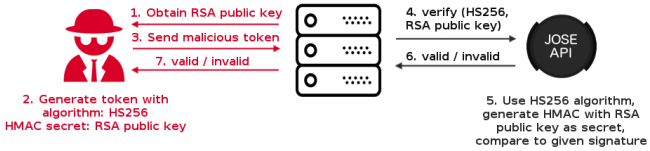

Figure 3: Attack scenario for the Key Confusion Attack

Tim McLean has discovered that many libraries rely solely on the user-controlled algorithm header parameter `alg` to distinguish which algorithm is used for verification [33]. He has observed that the JOSE 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 public key (e.g., RSA):

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

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

The vulnerability occurs 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 a 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. Therefore, it generates a new HMAC with the given *public key* and compares it to the provided signature. An exemplary attack workflow is illustrated in Figure 3.

Tim McLean has put much effort in informing the public and especially the maintainers of many JOSE libraries via blog post [33], Twitter [36], proposal to the JOSE working group [35], and direct mail [34]. As a result, the corresponding security considerations have been added to the JWS specification in RFC 7515 [26].

4.3 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 [5]. Bleichenbacher exemplarily applied his attack to the SSL v3.0 protocol with experimental results of recovering an encrypted message from 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)*.

In 2002, the W3C consortium published the XML Encryption standard [10]. Up until today, the RSA with PKCS#1 v1.5 padding algorithm is one of the two mandatory key transport mechanisms to be implemented for compliance. In 2012, Jager et al. described several attacks against the PKCS#1 v1.5 key transport mechanism, based on the known Bleichenbacher attack [20]. They were able to "recover the secret key used to encrypt [the] transmitted payload data [by exploiting] differences in error messages and in the timing behavior" [20].

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. Jager et al. have shown that their attacks are applicable on the early JWE implementations as well [20].

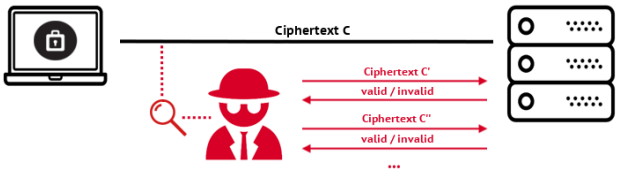

Figure 4: Attack scenario for the Bleichenbacher Million Message Attack

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 can retrieve 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 [38]. "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" [38]. The only prerequisite 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 4.

PKCS#1 v1.5 Encryption Padding. The PKCS#1 encryption padding version 1.5 is specified in RFC 2313 [31] 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 an integer, such that $0 < m < N$:

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

The leading zero byte `0x00` "ensures that the encryption block, converted to an integer, is less than the modulus" [31, Section 8.1] and the second byte `0x02` specifies the *block type* as a public-key encryption operation. 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 [5]:

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

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 $2B \leq ms \bmod N < 3B$, where $B = 2^{8(l-2)}$ [5]. 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. We refer to [5] for more details.

5 JOSEPH

JOSEPH is the name of our developed Burp Suite extension and stands for *JavaScript Object Signing and Encryption Pentesting Helper*. The following sections aims to give an overview of its structure and features.

5.1 Design, Structure & Extensibility

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 utilize 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. In addition to the highlighting, 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.

The JOSEPH tab. When enabling the extension, an extra *JOSEPH* tab appears on the main navigation of Burp Suite. This tab contains different sections for the features of the extension, namely the *Attacker*, a *Manual* tab, *Preferences*, and the *Decoder*. The *Decoder* is a simple helper utility to encode/decode `base64url` strings and display them in an ASCII or hexadecimal format. The Burp Suite itself

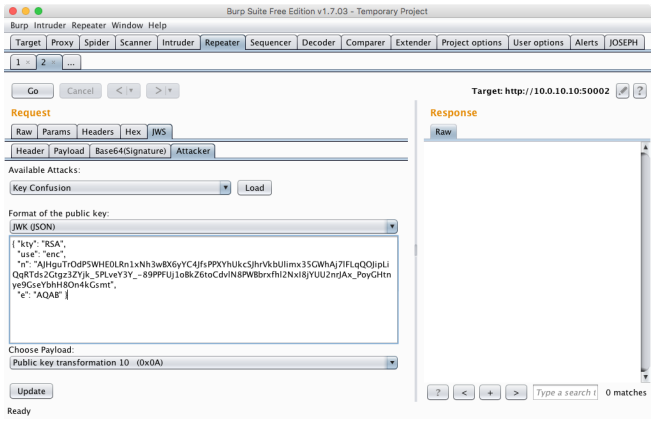

Figure 5: Screenshot of the attacker tab within the Repeater

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. The *Preferences* 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*. A second option allows to enable/disable the highlighting feature of messages containing JOSE values in the *HTTP history*. The third option 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 and HTTP headers. The preferences can be persistently saved into a configuration file on the hard disk to survive a restart or crash of the Burp Suite. The *Manual* tab is used for special cases where non-standard JOSE implementations need to be tested and JOSEPH is not able to automatically recognize and handle it. In this case, the user is still able to apply the attacks to it.

One of the main features of JOSEPH is the *attack engine*, enabling the user to apply the investigated attacks and test implementations for their vulnerability. If a message is detected to contain a JOSE value, this message can be sent to the JOSEPH extension by right clicking on the message and selecting *Send to JOSEPH* from the context menu. Within the JOSEPH *Attacker*, a new tab will be added, showing some very basic information about the token and a list of available attacks to choose from. If a specific attack is loaded, a short description of the selected attack will be displayed and, if necessary as defined by the attack itself, additional form elements will appear requesting additional information needed to perform the attack. 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 in case the attack takes a relatively long time. This new window is structured similarly to Burp Suite's native *Intruder* tool and shows several details, such as the response status, length, time, an attack related *payload type*, and short information about the payload itself.

5.2 Test Cases

Apart from supporting manual testing, the Burp Suite extension comprises three (semi-)automatic checks to test for Signature Exclusion, Key Confusion, and the Bleichenbacher MMA vulnerabilities.

Signature Exclusion. 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 [40], the following four spelling variations have been chosen: `None`, `none`, `NONE`, `nOnE`.

Key Confusion. The Key Confusion attack needs the target's public key as input parameter. 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 a JWK data set. Both formats are supported by the developed *JOSEPH* Burp Suite extension. Various string operations are performed on the given input in order to increase the probability of finding the correct string representation used by the server. Basically, several different variables 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 encompass even more variations.

Bleichenbacher MMA. Testing an implementation for the Million Message Attack (MMA) 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 conformity of a given ciphertext. This is performed by using various test vectors to generate encrypted messages with the provided public key, such as using differed sizes of the symmetric key, removing or adding `0x00` bytes at specific positions and using wrong first and second bytes.
The responses need 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 significantly, depending on the used implementation (and underlying programming language). This, therefore, renders 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 [5]. In our implementation we partially reused the code from WS-Attacker [32].

6 EVALUATION

6.1 Library Selection

Overall, nine libraries have been analyzed as part of this research, which were mainly taken from the overview at https://jwt.io (Table 1). The focus has been set on the PHP and Python platforms. With our help, maintainers of vulnerable libraries investigated related JOSE libraries they have developed in Ruby and C. Based on their investigation further vulnerabilities have been found. We list them in the table as well.

The Ruby and C libraries have been passively investigated, after vulnerabilities in other libraries of the same maintainers have been discovered.

Table 1: List of Evaluated Libraries

Name	Platform	Link
jose-PHP	PHP	https://github.com/nov/jose-php
json-jwt	PHP	https://github.com/emarref/jwt
jose	PHP	https://github.com/namshi/jose
jwt	PHP	https://github.com/lcobucci/jwt
php-jwt	PHP	https://github.com/firebase/php-jwt
jwcrypto	Python	https://github.com/latchset/jwcrypto
python-jose	Python	https://github.com/mpdavis/python-jose
pyjwkest	Python	https://github.com/rohe/pyjwkest
pyjwt	Python	https://github.com/jpadilla/pyjwt
json-jwt	*Ruby*	*https://github.com/nov/json-jwt*
jose	*C*	*https://github.com/latchset/jose*

All tests of the attacks against implementations of the JOSE specifications explained have been performed on the library level. Thus, additional operations on received messages or specific countermeasures in web applications using those libraries might possibly prevent practical exploitability or extend the attack surface, but were not taken into consideration in this research. Apart from auditing the publicly available source codes, a minimal testing environment has been set up as an HTTP wrapper to process incoming requests, which calls 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*[8] were used, whereas for PHP libraries, a combination of the web server *nginx* and the *PHP-FPM*[9] FastCGI implementation were used.

6.2 Vulnerable Libraries

Overall, this research revealed critical vulnerabilities in six JOSE libraries (see Table 2). All tested libraries were resistant against our Signature Exclusion attempts, which is presumably due to McLean's previous research and his effort in informing as many library maintainers as possible [33]. In our analysis, we additionally focused on the evaluation of HMAC timing attacks, which exploit measurable timing differences in comparing two strings using native functions. Based on those differences, which form a so-called covert timing channel, an attacker is able to gradually identify the correct byte

[8] URL: http://flask.pocoo.org/
[9] URL: https://php-fpm.org/

at a specific position of the unknown original signature and thus determine a valid signature to an arbitrary message. Since timing attacks are difficult to perform over a network, we have not included an implementation of the HMAC timing attack into the developed Burp Suite extension.

Table 2: List of Vulnerable Libraries

CVE No.	Vulnerability	Library	Version
CVE-2016-5429	HMAC Timing Attack	jose-php (PHP)	< 2.2.1
CVE-2016-5430	Bleichenbacher MMA	jose-php (PHP)	< 2.2.1
CVE-2016-5431	Key Confusion Attack	jose-php (PHP)	≤ 2.2.1
CVE-2016-7037	HMAC Timing Attack	JWT (PHP)	< 1.0.3
CVE-2016-6298	Bleichenbacher MMA	jwcrypto (Python)	< 0.3.2
CVE-2016-7036	HMAC Timing Attack	python-jose (Python)	< 1.3.2
–	Bleichenbacher MMA	json-jwt (Ruby)	< 1.6.5
–	Bleichenbacher MMA	jose (C)	< v4

In the following, the vulnerabilities discovered by JOSEPH are outlined in more detail.

Key Confusion. Almost all analyzed libraries in place were found to have protection mechanism against Key Confusion attacks. However, we identified that the jose-php library, implementing one of the recommended countermeasures of passing an additional parameter with a list of allowed algorithms to the verification function (see Listing 7), sets the expected algorithms parameter to *null* by default. This leads to the problem that developers are not forced to specify expected algorithms and that old vulnerable code is still working when upgrading to the new version, without any warning or notification of the existing issue.

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

Listing 7: Signature of the verify function of jose-php

Bleichenbacher MMA. Four of the analyzed libraries were found to be vulnerable to the Bleichenbacher Million Message Attack, by either applying an error-based or a timing-based padding oracle. 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.[10] 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 8 shows the three relevant parts and their occurrence in the code.

```
throw new JOSE_Exception_DecryptionFailed('Master key
    decryption failed');

throw new JOSE_Exception_DecryptionFailed('Encryption/Mac
     key derivation failed');

throw new JOSE_Exception_DecryptionFailed('Payload
    decryption failed');
```

Listing 8: 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 [37] – 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[11] does not strictly validate the PKCS#1 v1.5 format as defined in the specification [29].

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,[12] 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 [37], 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 [37]. Such modifications could also be used for the jose-php library but has been set out of scope for this paper.

The JWCrypto library had the same issue of exposing information of specific failing steps during the decryption process. The most relevant `Exceptions` are depicted in Listing 9.

```
raise InvalidJWEKeyLength(keylen, len(cek))

raise InvalidJWEData('Decryption Failed')

raise InvalidJWEData('Failed to verify MAC')
```

Listing 9: 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 bits, and all specified encryption algorithms, the *Content Encryption Key* could be recovered within less than 100,000 requests to the server and used to decrypt the hidden message. Figure 6 depicts

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

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

[12] See: https://github.com/phpseclib/phpseclib/blob/2.0.4/phpseclib/Crypt/RSA.php#L2496

an exemplary recovering of the CEK and displays the decrypted hidden message.

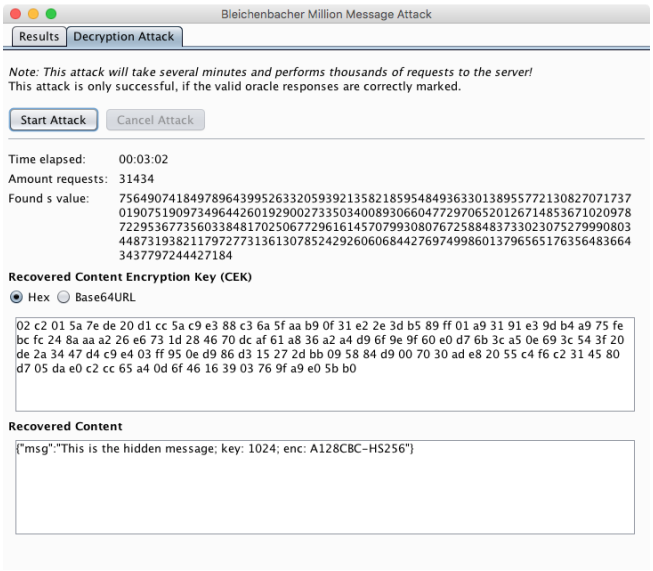

Figure 6: Screenshot showing the decrypted CEK and hidden message of a successful Million Message Attack.

The C library (jose) and the Ruby library (json-jwt) both suffered from leaking information about the correct padding format due to timing differences in the decryption process of the CEK, thus being vulnerable to the Bleichenbacher MMA. Both libraries are maintained by previously informed developers (regarding the issues in the jose-php and jwcrypto libraries), thus the mitigation was performed in the same disclosure process. Unfortunately, the developers did not request CVE identifiers for these libraries.

7 RELATED WORK

Security Research on JOSE. Tim McLean focused on a security analysis of JSON Web tokens and discovered the *Signature Exclusion* and *Key Confusion* vulnerabilities [33]. Many JOSE implementations were fixed after his disclosure. Apart from the theoretical threat description, no proof-of-concept or testing tool has been published.

In 2017 Nguyen et al. analyzed the application of various cryptographic attacks on JOSE libraries [42]. For example, they found that an invalid curve attack is applicable on the go-jose library. Sanso extended the analysis of practical invalid curve attacks. He found two additional libraries to be vulnerable: node-jose and jose2go [48].

XML Security. JOSE is closely related to the XML security specifications XML Signature and XML Encryption. These specifications suffered from several vulnerabilities and attacks in the past [50]. It has been shown how to break XML Encryption with adaptive chosen-ciphertext attacks [19, 23, 32] or how to break XML Signature implementations with XML Signature Wrapping and Exclusion attacks [51, 52].

Further attacks on XML Signatures were researched by James Forshaw in his whitepaper *Exploiting XML Digital Signature Implementations* [13].

Burp Suite Extensions. 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. The *SAML Raider* [4] is a Burp Suite extension for testing SAML infrastructures with the two core functionalities: manipulating SAML messages and managing X.509 certificates [4]. 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 [15], focuses on Single Sign-On protocols and is the first plugin supporting the recognition and manipulation of messages containing JSON Web Tokens. 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.

8 CONCLUSIONS AND FUTURE WORK

We showed how known cryptographic attacks can be successfully adapted on JOSE, bypassing the security of digital signatures or encrypted payload. We discovered that although JavaScript Object Signing and Encryption is a set of young specifications, the library developers repeat the same mistakes leading to security gaps. According to our evaluation, covering libraries written in PHP, Python, Ruby, and C, six of them were found to be vulnerable.

To close the gap between a specification and an implementation, we implemented JOSEPH – a Burp Suite extension able to recognize, visualize, manipulate JSON messages, as well as to semi-automatically carry out different attacks.

There exist more known attacks against cryptographic systems and possible pitfalls. Such attacks are *adaptive chosen-ciphertext attacks on CBC mode* [50, 54], *Bleichenbacher attack on RSA signatures* [17], or *invalid curve attacks* [21]. With respect to the future work, the analysis of such attacks on JOSE is considered essential. Furthermore, the usage of JOSE in complex systems like JSON-based web services and protocols like OpenID Connect should be in the scope of further researches. Similar to the security analysis of XML-based services [12] an in-depth evaluation could lead to the discovering of new attacks.

ACKNOWLEDGMENTS

This work was partially supported by the European Commission through the FutureTrust project (grant 700542-Future-Trust-H2020-DS-2015-1).

REFERENCES

[1] [n. d.]. IETF jose Working Group. Javascript Object Signing and Encryption (jose). ([n. d.]). http://datatracker.ietf.org/wg/jose/
[2] Apache Software Foundation. [n. d.]. JAX-RS JOSE. ([n. d.]). http://cxf.apache.org/docs/jax-rs-jose.html
[3] Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. 2016. *Automatic Certificate Management Environment (ACME).* Internet-Draft draft-ietf-acme-acme-04. Internet Engineering Task Force. https://tools.ietf.org/html/draft-ietf-acme-acme-04 Work in Progress.
[4] R. Bischofberger and E. Duss. [n. d.]. SAML Raider - SAML2 Burp Extension. ([n. d.]). https://github.com/SAMLRaider/SAMLRaider
[5] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '98).* Springer-Verlag, London, UK, UK, 12. http://dl.acm.org/citation.cfm?id=646763.706320
[6] T. Bray. 2014. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard). (March 2014). http://www.ietf.org/rfc/rfc7159.txt

[7] Dennis Detering. [n. d.]. JOSEPH – JavaScript Object Signing and Encryption Pentesting Helper. ([n. d.]). https://github.com/RUB-NDS/JOSEPH

[8] T. Dierks and E. Rescorla. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard). (Aug. 2008). http://www.ietf.org/rfc/rfc5246.txt Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685.

[9] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. 2002. XML Encryption Syntax and Processing. *W3C Recommendation* (2002).

[10] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. 2002. XML Encryption Syntax and Processing. W3C Recommendation. (December 2002). https://www.w3.org/TR/xmlenc-core/

[11] Donald Eastlake, Joseph Reagle, David Solo, Frederick Hirsch, and Thomas Roessler. 2008. XML Signature Syntax and Processing (Second Edition). *W3C Recommendation* (2008).

[12] Andreas Falkenberg, Meiko Jensen, Prof. Dr-Ing Jörg Schwenk, and Juraj Somorovsky. [n. d.]. WS-Attacks. ([n. d.]). http://www.ws-attacks.org/

[13] James Forshaw. 2013. Exploiting XML Digital Signature Implementations. In *Hack In The Box - Kuala Lumpur 2013*.

[14] S. Frankel and S. Krishnan. 2011. IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. RFC 6071 (Informational). (Feb. 2011). http://www.ietf.org/rfc/rfc6071.txt

[15] T. Guenther. [n. d.]. Extension for Processing and Recognition of Single Sign-On Protocols (EsPReSSO). ([n. d.]). https://github.com/RUB-NDS/BurpSSOExtension

[16] IBM Deutschland GmbH. [n. d.]. IBM DataPower Gateway. ([n. d.]). http://www-03.ibm.com/software/products/de/datapower-gateway

[17] Intel Security. [n. d.]. BERserk Vulnerability. ([n. d.]). http://www.intelsecurity.com/resources/wp-berserk-analysis-part-1.pdf

[18] Internet Security Research Group (ISRG). [n. d.]. Let's Encrypt. ([n. d.]). https://letsencrypt.org/

[19] T. Jager, S. Schinzel, and J Smorovksy. 2012. 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)*.

[20] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. 2012. Bleichenbacher's Attack Strikes Again: Breaking PKCS#1 v1.5 in XML Encryption. In *ESORICS (Lecture Notes in Computer Science)*, Sara Foresti, Moti Yung, and Fabio Martinelli (Eds.), Vol. 7459. Springer. http://dblp.uni-trier.de/db/conf/esorics/esorics2012.html#JagerSS12

[21] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. 2015. *Practical Invalid Curve Attacks on TLS-ECDH*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-24174-6_21

[22] Tibor Jager and Juraj Somorovsky. 2011. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*.

[23] Tibor Jager and Juraj Somorovsky. 2011. How To Break XML Encryption. In *The 18th ACM Conference on Computer and Communications Security (CCS)*.

[24] M. Jones. 2015. JSON Web Algorithms (JWA). RFC 7518 (Proposed Standard). (May 2015). http://www.ietf.org/rfc/rfc7518.txt

[25] M. Jones. 2015. JSON Web Key (JWK). RFC 7517 (Proposed Standard). (May 2015). http://www.ietf.org/rfc/rfc7517.txt

[26] M. Jones, J. Bradley, and N. Sakimura. 2015. JSON Web Signature (JWS). RFC 7515 (Proposed Standard). (May 2015). http://www.ietf.org/rfc/rfc7515.txt

[27] M. Jones, J. Bradley, and N. Sakimura. 2015. JSON Web Token (JWT). RFC 7519 (Proposed Standard). (May 2015). http://www.ietf.org/rfc/rfc7519.txt

[28] M. Jones and J. Hildebrand. 2015. JSON Web Encryption (JWE). RFC 7516 (Proposed Standard). (May 2015). http://www.ietf.org/rfc/rfc7516.txt

[29] J. Jonsson and B. Kaliski. 2003. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447 (Informational). (Feb. 2003). http://www.ietf.org/rfc/rfc3447.txt

[30] S. Josefsson. 2006. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard). (Oct. 2006). http://www.ietf.org/rfc/rfc4648.txt

[31] B. Kaliski. 1998. PKCS #1: RSA Encryption Version 1.5. RFC 2313 (Informational). (March 1998). http://www.ietf.org/rfc/rfc2313.txt Obsoleted by RFC 2437.

[32] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. 2015. How to Break XML Encryption – Automatically. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C. https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser

[33] Tim McLean. 2015. Blog post: Critical vulnerabilities in JSON Web Token libraries. (March 2015). https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html

[34] Tim McLean. 2015. Direct Email: Critical vulnerabilities in JSON Web Token libraries. (March 2015). https://bitbucket.org/b_c/jose4j/wiki/04-01-15-Transparency

[35] Tim McLean. 2015. JOSE mailing list: Critical vulnerabilities in JSON Web Token libraries. (March 2015). http://www.ietf.org/mail-archive/web/jose/current/msg05036.html

[36] Tim McLean. 2015. Twitter: Critical vulnerabilities in JSON Web Token libraries. (March 2015). https://twitter.com/McLean0/status/578281292237815808

[37] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. 2014. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/meyer

[38] MITRE. [n. d.]. CAPEC-463: Padding Oracle Crypto Attack. ([n. d.]). http://capec.mitre.org/data/definitions/463.html

[39] Timothy D. Morgan and Omar Al Ibrahim. 2014. XML Schema, DTD, and Entity Attacks: A Compendium of Known Techniques. (May 2014). URL: http://www.vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf.

[40] Alessandro Nadalin. [n. d.]. JSON Object Signing and Encryption library for PHP. ([n. d.]). https://github.com/namshi/jose

[41] National Institute of Standards and Technology. [n. d.]. FIPS General Information. ([n. d.]). https://www.nist.gov/information-technology-laboratory/fips-general-information

[42] Quan Nguyen. 2017. Practical Cryptanalysis of Json Web Token and Galois Counter Mode's Implementations. In *Real World Crypto Conference 2017*. https://rwc.iacr.org/2017/Slides/nguyen.quan.pdf http://www.realworldcrypto.com/rwc2017.

[43] OpenID Foundation. [n. d.]. What is OpenID Connect? ([n. d.]). http://openid.net/connect/

[44] PortSwigger Ltd. [n. d.]. Burp Suite. ([n. d.]). https://portswigger.net/burp/

[45] PortSwigger Ltd. 2017. Burp Scanner – Issue Definitions. (2017). https://portswigger.net/KnowledgeBase/Issues/

[46] PortSwigger Ltd. 2017. Burp Suite – BApp Store. (2017). https://portswigger.net/bappstore/

[47] PortSwigger Ltd. 2017. Burp Suite – Extender API. (2017). https://portswigger.net/burp/extender/api/index.html

[48] Antonio Sanso. 2017. Critical vulnerability in JSON Web Encryption (JWE) – RFC 7516. (2017). http://blog.intothesymmetry.com/2017/03/critical-vulnerability-in-json-web.html.

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

[50] Juraj Somorovsky. 2013. *On the Insecurity of XML Security*. Ph.D. Dissertation. Ruhr-Universität Bochum.

[51] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. 2011. All Your Clouds Are Belong to Us – Security Analysis of Cloud Management Interfaces. In *The ACM Cloud Computing Security Workshop (CCSW)*.

[52] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. 2012. On Breaking SAML: Be Whoever You Want to Be. In *21st USENIX Security Symposium*. Bellevue, WA.

[53] The Open Web Application Security Project. [n. d.]. Fuzzing. ([n. d.]). https://www.owasp.org/index.php/Fuzzing

[54] Serge Vaudenay. 2002. 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 (EUROCRYPT '02)*. Springer-Verlag, London, UK, UK, 13. http://dl.acm.org/citation.cfm?id=647087.715705

[55] World Wide Web Consortium (W3C). [n. d.]. Extensible Markup Language (XML) 1.0 (Fifth Edition). ([n. d.]). https://www.w3.org/TR/REC-xml/