RUHR-UNIVERSITÄT BOCHUM

RUHR-UNIVERSITÄT BOCHUM

# Developing an Online Tool for Attacking JSON Web Tokens

Niklas Conrad

Bachelor's Thesis – July 24, 2025
Chair for Network and Data Security.

Supervisor: Dr.-Ing. Vladislav Mladenov
Advisor: M. Sc. Louis Jannett

hgi Lehrstuhl für Netz- und Datensicherheit

## Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr University Bochum or any other institution or university.

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 part 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

## 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 selbstständig 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.

| Date | Niklas Conrad |
| --- | --- |

## Erklärung

Ich erkläre mich damit einverstanden, dass meine Abschlussarbeit am Lehrstuhl Netz- und Datensicherheit dauerhaft in elektronischer und gedruckter Form aufbewahrt wird und dass die Ergebnisse aus dieser Arbeit unter Einhaltung guter wissenschaftlicher Praxis in der Forschung weiter verwendet werden dürfen. Weiterhin erkläre ich mich damit einverstanden, dass die Abschlussarbeit auf der Webseite des Lehrstuhls bzw. des Betreuers veröffentlicht werden darf.

| Date | Niklas Conrad |
|---|---|

# Contents

# 1 Introduction

This chapter introduces the topic of this thesis, defines the central problem, and provides an overview of the motivation, objectives, related work, and contributions. It is designed to familiarize readers with both the practical and scientific aspects of the subject and to give a clear sense of how the thesis is structured.

## 1.1 Motivation

JSON Web Tokens (JWT) [10] have become a widely adopted standard for securely transmitting identity and authorization information between systems and web applications [7]. JWTs were initially developed to improve performance compared to classical token systems, which rely on database queries. However, this broad use has also exposed numerous security vulnerabilities in real-world implementations. Studies, such as the one by Detering et al. [3], have revealed critical flaws in popular JWT libraries that allow attackers to manipulate tokens and bypass authentication mechanisms. Although many tools can be used to generate JWT attacks, they all have certain limitations. Some require initial installation and configuration, while others offer a less intuitive approach to handling JWTs and attacks. Some depend on a specific operating system, while others rely on certain software, such as Burp Extensions. This thesis addresses these issues by developing an intuitive, web-based application that requires no installation. By implementing all the well-known attacks, it provides security researchers and penetration testers with a practical and user-friendly solution. While executing the actual attacks must be done separately, the precise token generation significantly simplifies and improves the efficiency of security assessments for JWT-based applications.

## 1.2 Objectives

This section defines the main objective of this Bachelor's thesis and places it in the context of the previously outlined motivation. After demonstrating the relevance of JWT security and existing challenges, this work describes the concrete contribution it aims to provide, bridging the gap between the problem statement and its practical implementation, ensuring its real-world applicability.

The primary goal of this thesis is to develop a web-based tool that enables the automated generation of specifically crafted JWTs for various known attack scenarios. A key aspect of the tool is that it does not execute attacks itself; instead, it produces the required tokens, which can then be used manually or in automated tests. This approach enables security researchers and developers to evaluate JWT-based applications for any well known vulnerabilities.

Specifically, the following objectives are pursued:

- Development of an intuitive browser-based *Single Page Application (SPA)* without any backend component, ensuring that no sensitive data is transmitted or stored.
- Support for all relevant JWS [9] and JWE [11] algorithms as defined in the *JSON Web Algorithms (JWA)* [12] standard.
- Automated generation of malicious tokens covering state-of-the-art attacks.
- Ability to manually create malformed tokens, increasing the flexibility during security testing.
- Validation of the tool by testing it on platforms such as the chair's *eHacking Platform* [30] to demonstrate its practical effectiveness.
- Comparative analysis of the developed tool with existing solutions, especially regarding functionality and usability.

By achieving these objectives, this thesis aims to develop a tool that facilitates practical JWT security testing and contributes to the scientific analysis of JWT vulnerabilities in a systematic manner.

## 1.3 Contribution

This Bachelor's thesis presents a novel web-based tool for the automated generation of customized JWTs tailored to various known attack scenarios. The main contributions of this thesis are:

- Development of a web-based tool that, for the first time, enables automated generation of customized *JWTs* covering all common attack scenarios.
- Comprehensive support for all relevant algorithms defined in the *JWA* standard, including the ability to process encrypted JWTs.
- Implementation of a user-friendly *SPA* that performs the entire token generation process directly in the browser without involving any privacy-critical backend components.

- Validation of the tool using realistic testing environments (eHacking Platform [30]), demonstrating that the generated tokens can effectively identify vulnerabilities in JWT-based applications.
- Comparative analysis with existing tools, highlighting the extended functionality and improved usability of the developed solution.

These contributions provide both practical and scientific value, significantly advancing the systematic security analysis of JWT-based applications.

# 2 Related Work

The effectiveness of attacks on JWTs depends on two key factors. First, standard functions such as encoding, decoding, signing, verifying, decrypting, encrypting, and key generation are required. Next, to test more efficiently, automatically generated tokens for the specific attacks are also needed. Despite the existence of tools that facilitate fundamental JWT management, a comprehensive solution for generating JWTs to address all prevalent attacks and, with that, systematically detect JWT vulnerabilities remains unavailable. Many tools focus only on parsing, analyzing, and editing JWTs:

- https://jwt.io/ [16]
- https://fusionauth.io/ [5]
- https://token.dev/ [31]
- https://jwt.ms/ [17]
- https://irrte.ch/ [6]
- https://dinochiesa.github.io/jwt/ [1]

These tools facilitate the standard handling of JWTs and offer a good UI. While some of these tools can assist in creating manual attacks, they do not offer automation for generating attack-specific tokens.

Other tools implement only a small set of attacks:

- JOSEPH [13]
- jwt_tool [18]
- JWT Editor [14]
- JWT HACK [15]

The interest in evaluating the security of JWT libraries remains high, so evaluations of individual frameworks are always part of various research projects. [19] [2]

Please refer to Table A.1 for a detailed comparison of criteria and the tools that meet each requirement. The table is divided into two sections: basic functionalities and supported attacks. When evaluating attack coverage, it is important to examine whether the tool automatically generates payloads, rather than considering the theoretical possibility of manual creation. Standard signature attacks such as signature exclusion, `none`-algorithm, custom keys, and key confusion are considered. As well as header parameter attacks involving `jku`, `x5u`, and `kid` [4] [29]. A more thorough exposition of the selection of attacks and their descriptions can be found in Section 3.2. As shown, none of the existing tools combines all necessary features.

# 3 Background

This chapter provides the essential background needed to understand the problem investigated in this thesis and the motivation for developing the tool. It introduces key concepts and standards related to JWTs and explains common attack scenarios that exploit weaknesses in JWT processing. By presenting JWTs, their signature and encryption mechanisms, and known attack vectors, it establishes the context necessary to highlight the importance of automatically generating prepared tokens for security assessments.

## 3.1 JSON Web Token

*JSON Web Tokens (JWTs)* [10] are a format for the compact and URL-safe transmission of information between two parties in the form of a JSON object. A JWT encodes data representing a set of claims, which may include information about a user's identity, their permissions, or other application-specific data. Thus JWTs enable systems to exchange authentic information. Each JWT consists of three main parts: the header, the payload, and the signature. The header defines the type of token and the algorithm used to secure it, either through signing or encryption. The central part of the token, known as the body or payload, contains the actual information, such as user details or access rights. Finally, the signature protects the token from alteration and helps verify that it originates from a trusted source. JWTs are commonly used in modern web applications for authentication and authorization, for instance, in Single Sign-On (SSO) protocols like OpenID Connect, as the *id_token* [22], or in REST APIs [21]. These technologies are used by thousands of websites, as shown by SSO-Monitor [7] [33] [8].

### 3.1.1 JSON Web Signature

*JSON Web Signature (JWS)* [9] defines how digital signatures can be applied to JSON data to ensure both the integrity and authenticity of transmitted information. A JWS protects against tampering by enabling verification of a token's content using its attached signature. The standard supports various cryptographic algorithms, including symmetric algorithms like HS256 and asymmetric algorithms such as RS256, ES256, and PS256, which rely on public and private key pairs. The algorithm to use is specified in the JWT's header with the `alg` parameter. When a token is created, the signing algorithm processes the base64url-encoded header and payload, producing a cryptographic signature that forms the third component of the token. This mechanism enables receivers to verify that the token has not been tampered with and that it originates from a trusted source.

### 3.1.2 JSON Web Encryption

*JSON Web Encryption (JWE)* [11] defines how JSON objects can be encrypted to guarantee the confidentiality of transmitted data. Unlike JWS, which only ensures integrity and authenticity, JWE also protects the token's content from being read by unauthorized parties. This makes JWE particularly important when sensitive information, such as personal data or confidential session details, must be transmitted over potentially insecure channels. A JWE consists of a header, an encrypted key, an initialization vector, an encrypted body, and a signature. The encrypted body can be decrypted once the encrypted key is decrypted with the initialization vector and the secret key. In contrast to JWS, JWEs also ensure confidentiality.

## 3.2 Attacks on JWTs

To fully understand the need for automated token generation, it is important to grasp the common attack scenarios targeting the verification mechanism of JWTs. These attacks exploit weaknesses in the signature verification or improperly processed header parameters. The attack scenarios addressed in this thesis were intentionally selected to cover all currently known, relevant attack vectors against JWTs described in the literature and observed in practice. Therefore, attacks targeting signature validation weaknesses (e.g., *Signature Exclusion*, *None Algorithm*, *Psychic Signature*) and insecurely processed header parameters (e.g., SSRF or manipulation of the `kid` header) are included. In the following the implemented attacks are briefly explained.

**Signature Exclusion** In a *Signature Exclusion* attack, a JWT is created without a signature by omitting the third segment of the token. Some flawed implementations fail to properly check the token's structure and accept such incomplete tokens as valid. Allowing attackers to bypass authentication or authorization mechanisms completely. CVE-2020-28042 describes this type of implementation flaw. An exemplary illustration is provided in Figure 3.1.

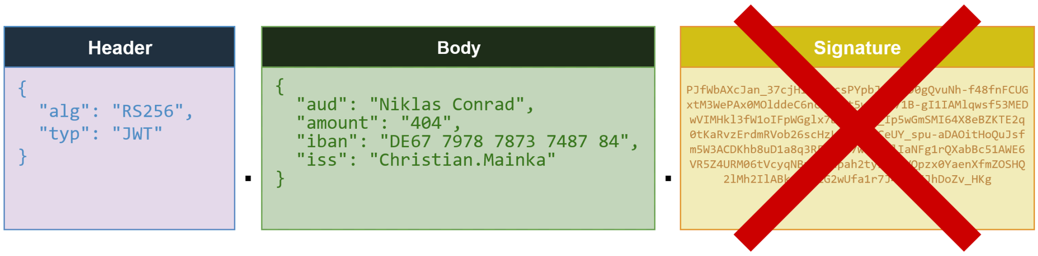


Figure 3.1: Illustration of *Signature Exclusion*. In the *Signature Exclusion* attack, the signature of the token is removed.

**None Algorithm** The *None Algorithm* attack exploits the fact that the JWT standard allows the `alg` field to be set to `none`, indicating an unsigned token. Insecure implementations mistakenly accept tokens with `alg=none` as valid, even when a signature is required. Attackers can then freely modify the token's content and generate tokens that appear legitimate. The vulnerability in the wild was first described by Tim McLean (CVE-2015-9235) [20]. Another example of such a vulnerability is described in CVE-2022-23540. Figure 3.2 presents a schematic depiction of a token with the `none` algorithm.

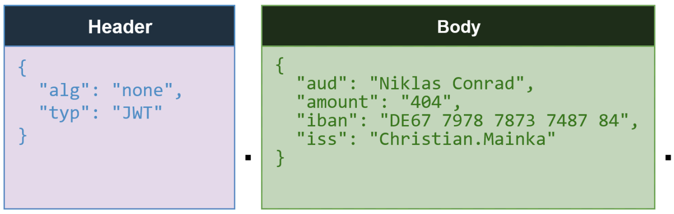


Figure 3.2: Illustration of *None Algorithm*. In the *None Algorithm* attack, the algorithm is set to `none` and no signature is generated.

**Psychic Signature** The *Psychic Signature* attack exploits a flaw in the signature verification process with ECDSA signatures in specific Java versions (Oracle Java 15-18 and OpenJDK 15, 17, and 18). The signature `MAYCAQACAQA` represents the encoding of the parameters `r` and `s` with a value of zero. From a mathematical perspective, this solution is considered trivial in terms of verification computation. Consequently, successful verification is ensured irrespective of the signed content. According to the standard, the signature must not contain the parameter `r` with a value of zero [24]. However, the defective verification mechanism has not implemented this particular case correctly, and therefore, verifies every JWT successfully.

This vulnerability is detailed in CVE-2022-21449. Figure 3.3 illustrates a token with the signature `MAYCAQACAQA` .

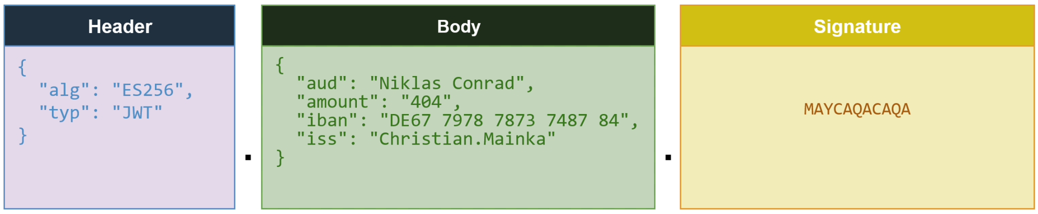


Figure 3.3: Illustration of *Psychic Signature*. The signature `MAYCAQACAQA` is always a valid signature for the *Psychic Signature* vulnerability.

**Custom Key** In a *Custom Key* attack, attackers generate their own key and include it along with the token. Since the standard permits this behavior, it may be implemented in this way. By adding manipulated key material through header fields such as `jwk`, they attempt to convince the application to accept the forged key as legitimate. An example of this vulnerability is described in CVE-2018-0114. Figure 3.4 demonstrates this attack vector in a simplified schematic.

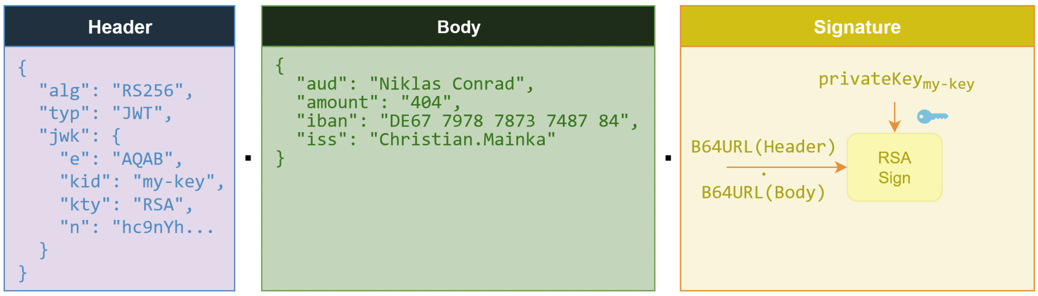


Figure 3.4: Illustration of *Custom Key*. In the *Custom Key* attack, the attacker adds a custom key to the header of the token.

**Key Confusion** The *Key Confusion* attack exploits confusion between asymmetric and symmetric keys. If an implementation does not properly verify the key type, an attacker can, for example, use the public part of an asymmetric key as a symmetric key, resulting in tokens that appear valid even without access to the private key. This attack vector is demonstrated in CVE-2015-9235, where an attacker can use a public key as a symmetric key to sign tokens. Tim McLean also discovered this vulnerability in JWT libraries [20]. A graphical representation of the described method is shown in Figure 3.5.

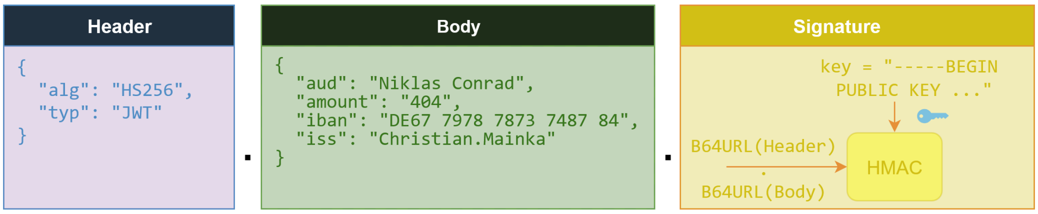


Figure 3.5: Illustration of *Key Confusion*. In the *Key Confusion* attack, the public key string is used as a symmetric key to sign the token.

**Weak HMAC Key** A *Weak HMAC Key* attack targets implementations that accept weak default keys such as "secret." Attackers can sign tokens with these weak keys, and the server may still accept them as valid. Figure 3.6 provides a simplified depiction of the described vulnerability.

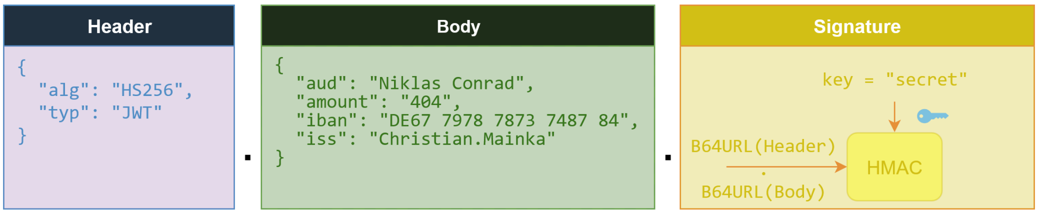


Figure 3.6: Illustration of *Weak HMAC Key*. In the *Weak HMAC Key* attack, the attacker exploits the weak key used by the server.

**Empty Key** In an *Empty Key* attack, a token is signed using an empty HMAC key. Some insecure implementations (CVE-2019-20933) accept such tokens as valid, allowing attackers to forge arbitrary tokens without knowing any key. Figure 3.7 illustrates the core mechanism of the attack in an abstract form.

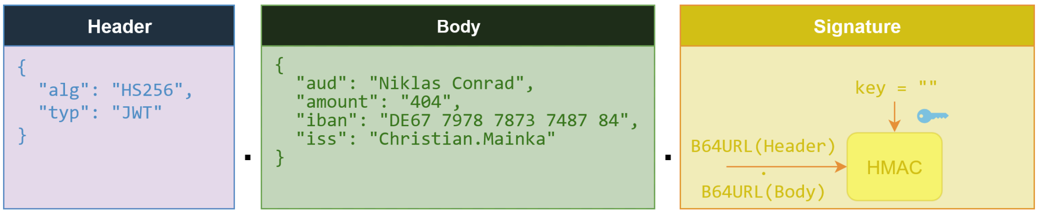


Figure 3.7: Illustration of an *Empty Key* attack. In the *Empty Key* attack, the token is signed with an empty key.

**SSRF via header parameters** In a *Server-Side Request Forgery (SSRF)* attack, attackers abuse header fields like `jku` or `x5u` to force the server to fetch external or internal resources during token verification. This can lead to information disclosure or enable attacks targeting the internal network. An illustrative abstraction of the token structure is given in Figure 3.8.

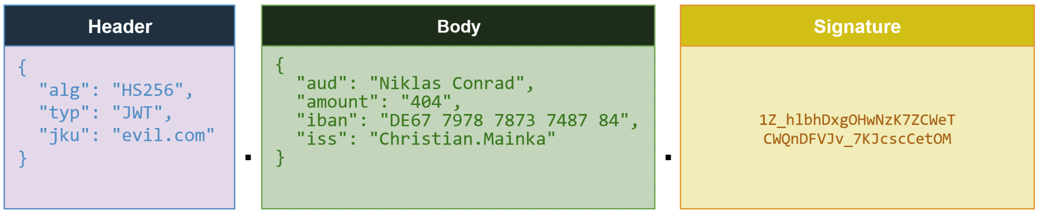


Figure 3.8: Illustration of *SSRF*. The server is tricked into fetching a resource from an external URL specified in the `jku` or `x5u` header.

**Attacks on the `kid` parameter** Attacks on the `kid` parameter exploit the fact that this field specifies the location of the key. Attackers manipulate its value to perform path traversal or even achieve Remote Code Execution (RCE), for example by tricking the application into loading local files or executing injected commands. Using path traversal, an attacker could also generate a signature with known bytes. A generalized visualization of this flaw is depicted in Figure 3.9.

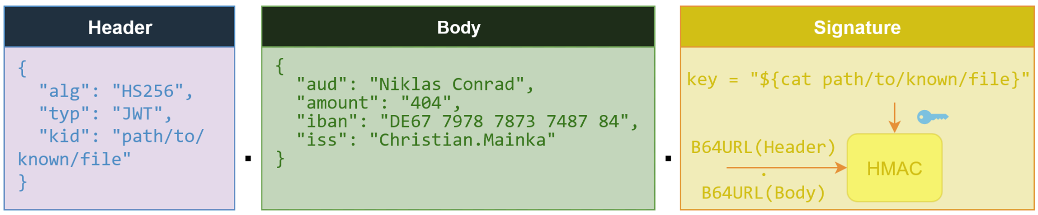


Figure 3.9: Illustration of an attack on the *kid* parameter. The `kid` parameter is manipulated to perform path traversal and verify a signature with known bytes.

# 4 Implementation

This chapter describes the technical realization of the developed tool. It explains the architecture, key design decisions, and the implementation of the basic functionalities and attack scenarios. Thus, it bridges the concepts presented in Chapter 3 with the later validation of the tool. The detailed description clarifies how the tool generates prepared JWTs and outlines the technical measures taken to create a flexible and user-friendly application. To better understand the individual components, a typical workflow using the tool is illustrated schematically in Figure 4.1.

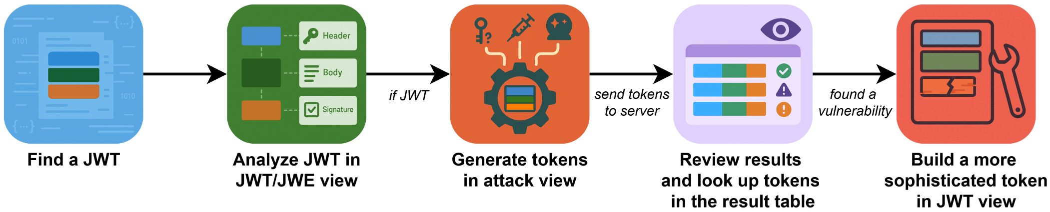


Figure 4.1: Schematic representation of the attack workflow.

**Summary of the Advantages** The developed tool demonstrates several notable improvements compared to existing solutions, integrating functionalities that are otherwise distributed across multiple separate tools. These enhancements relate both to the tool's functional depth and its breadth of application—especially in terms of how comprehensively it addresses known attack scenarios. The key advantages can be summarized as follows:

- **Full client side implementation:** The tool operates entirely within the browser as a single page application (SPA), eschewing backend components. Consequently, all processed data remains exclusively on the user's end device, thereby completely eliminating data protection risks.
- **Comprehensive attack coverage:** All well known attacks are implemented and combined in one tool with a greater depth than other tools.
- **Integrated key management:** The tool supports the generation and management of keys in both PEM and JWK formats, facilitating a seamless workflow for users.

- **Automated attack generation:** The exportation of all generated tokens can be executed collectively, with subsequent integration into external tools such as Burp Intruder. The structured presentation of the tokens in the tool also enables efficient tracking and analysis.
- **Invalid JSON:** The tool facilitates the generation of tokens with duplicate JSON keys. It is also possible to manually create JWTs with invalid JSON, such as malformed or non-standard JSON, and insert custom byte values through special escape sequences. These edge cases are particularly useful in further testing the validation logic of JWT implementations.

The implemented tool is characterized by a unique combination of functional depth, technical flexibility, and user-friendliness. It is designed to address the requirements of security researchers who aim not only to analyze JWTs but also to identify specific vulnerabilities in implementations. A direct comparison of the tools' capabilities in Table 5.5 reveals that the tool in question is distinctly superior to competing tools in terms of vulnerability identification.

## 4.1 Application Architecture

The tool is designed as a browser-based *Single Page Application (SPA)* built with Vanilla JavaScript without frameworks, which allows it to run entirely in the browser without any installation or backend components. This approach ensures that no sensitive data ever leaves the user's device, effectively eliminating data protection concerns. The architecture is intentionally kept simple and fully frontend-based, with the core logic contained in a single JavaScript file to make the code easier to maintain and extend. The only external dependency is the *Web Crypto API*, which is supported by all modern browsers and provides the necessary cryptographic operations. In future versions, the *Web Crypto API* will be bundled using a tool like Webpack to enhance compatibility and maintainability across different environments. The user interface has been developed completely from scratch using HTML, CSS, and JavaScript, avoiding external frameworks or UI libraries to keep full control over the design and ensure maximum flexibility. The tool is released as an open source project on GitHub[1]. The repository will be made available to the public shortly after the thesis is published.

## 4.2 Basic Functionalities

The tool supports all relevant algorithms for JWS and JWE, enabling comprehensive handling of JWTs. Users can encode, decode, and manipulate all parts of the

[1] `https://github.com/Hackmanit/JWTF`

JWT. The application allows importing or generating keys in PEM or JWK format directly within the tool to ensure a smooth workflow. These basic functions form the basis for working with JWTs. Even if the primary focus of this work is on automatically generated attacks, manual processing of the JWTs is indispensable in order to be able to carry out even more targeted and coordinated attacks. To clearly structure the functionality, the application offers four distinct views: JWT, JWE, Attacks, and Key Management. The core features are described in the following:

**JWT Handling** In the JWT view, the application enables the creation and editing of JWTs based on the JWS standard. Users can freely modify the header and payload, including intentionally crafting malformed or non-standard JSON, such as tokens with duplicate keys. The tool supports encoding and decoding JWTs as well as signing them with the implemented JWS algorithms. Additionally, it allows inserting custom byte values through escape sequences to generate unusual token structures. Furthermore, this view is used for step 2 in Figure 4.1 to analyze the token and provide a better understanding of which attacks are possible. The view is also helpful to go back to specific vectors and build more sophisticated tokens after analyzing the automatically generated attacks (step 5).

**JWE Handling** The JWE view is dedicated to handling encrypted JWTs. Its goal is to fully support the creation of JWE tokens, enabling tests on implementations that accept encrypted tokens. The selection of supported JWE algorithms is already implemented, taking into account limitations imposed by browser APIs. The algorithms ECDH-ES, ECDH-ES+AKW, and PBES are currently not implemented due to repeated problems encountered during their implementation. However, the integration of the algorithms is planned for future versions.

**Key Management View** The Key Management view provides a central interface for handling keys. Keys can be generated for all algorithms specified in JWA and stored in local storage if needed. Furthermore, keys can be converted between PEM and JWK formats directly within the tool. On the one hand, this is a user-friendly feature, but this is mainly a function that is highly relevant for key confusion 3.2, since both formats are used as the basis for symmetric keys.

## 4.3 Attack View

The attack view forms the core component of the tool for security analysis of JWTs. In this view, the tool generates prepared tokens crafted specifically to exploit known vulnerabilities by executing various attack scenarios. Users select an attack from the interface, and the tool automatically generates a corresponding token (step 3 in Figure 4.1). Each attack implements its own logic, adjusting the header and inserting suitable signatures or keys as needed. After geneation, tokens can be downloaded as a file with each token separated by newlines, enabling easy integration with tools like Burp Intruder for automated testing. A table displays the generated tokens with their respective attack types, which can be used to look up tokens quickly (step 4) and allow users to copy tokens directly to the clipboard or open them in the JWT view for detailed inspection and further evaluation (step 5).

**Implementation of Attacks** The attack view enables automated generation of various attack scenarios to specifically test JWT implementations for known vulnerabilities. All attacks are based on the ability to create JWTs with customized headers and payloads. The tool also supports inserting special escape sequences to produce duplicate header keys. These prepared tokens serve as the basis for the attacks, allowing tests with intentionally malformed or non-standard token structures. The following sections describe each attack and its specific implementation.

**Signature Exclusion** For the Signature Exclusion attack (see Paragraph 3.2), the tool generates a JWT with a header and payload but omits the signature segment. It is important to note that certain tools are not generating this token, given its fundamental nature. This attack does not require any user interaction.

**None Algorithm** For the None Algorithm attack (see Paragraph 3.2), the tool generates all 16 different tokens, each with a different capitalization of the word ”none” (e.g., ”none”, ”NONE”, ”NoNe”). The other tools only generate less than five variants, which is not sufficient to cover all possible cases. No user interaction is needed.

**Psychic Signature** In the Psychic Signature attack (see Paragraph 3.2), the tool creates a token using the ES256 algorithm and replaces the signature with the value `MAYCAQACAQA`. This function does not differ from the other tools that implement this attack. No user interaction is required.

**Custom Key** The Custom Key attack (see Paragraph 3.2) offers extensive customization options: users can enter their own key or use predefined standard keys for all supported algorithms. They can choose whether the tool generates a token for a specific algorithm or one for each algorithm type. Additionally, users can embed the custom key via `jku` or `x5u` by providing a URL. Injecting the key as a `jwk` does not differ from the tools which implement this attack. But not all tools support the `jku` and `x5u` vectors.

**Key Confusion** For the Key Confusion attack (see Paragraph 3.2), users supply a public key of any supported algorithm type. It is important to note that, due to the reliance of this attack on various implementation-specific behaviors, multiple methods exist for deriving a public key from a symmetric key. After thorough analysis, the following cases were implemented, which can be divided into keys derived from the public key in PEM format and keys derived from the public key in JWK format.

**PEM format:**

1. PEM-encoded key (base64) with header, footer, newlines, and no newline at the end
2. PEM-encoded key (base64) with header, footer, newlines, and a newline at the end
3. PEM-encoded key (base64) with header and footer but without newlines
4. PEM-encoded key (base64) without header, or footer, with newlines
5. PEM-encoded key (base64) without header, footer, or newlines
6. PEM-encoded key (base64) without header, footer, or newlines, but with a newline at the end

**JWK format:**

7. use `e` as key (RS/PS)
8. use `n` as key (RS/PS)
9. use `x` as key (ES)
10. use `y` as key (ES)

In addition to the above cases, there are variations of cases 5 and 7–10 where the key is interpreted as a base64 string. Since a key is either RSA or EC, the tool will generate 11 tokens for each HS algorithm, resulting in 33 tokens total. A key strength of this implementation lies in its thorough handling of a wide range of potential use cases, an aspect that clearly differentiates it from existing tools in the field. In addition, the tool includes built-in support for converting JWK keys to PEM format and vice versa. This feature plays a crucial role in ensuring comprehensive compatibility and full coverage of relevant scenarios.

**Weak HMAC Key** The tool implements a Weak HMAC Key attack using a list of approximately 100,000 known weak keys referenced by PortSwigger [32]. Alternative tools offer supplementary functionality in the form of custom wordlists and an expanded range of options for brute-forcing. However, due to limitations in browser performance, it is not possible to utilize larger brute-force lists; such evaluations should be conducted outside the browser. No user interaction is required.

**Empty Key** The Empty Key attack (see Paragraph 3.2) generates three tokens with an empty key for each HS algorithm. Other tools implement this attack without notable differences. No user interaction is required.

**SSRF via jku/x5u** For SSRF (see Paragraph 3.2) attacks, the tool generates four tokens embedding external URLs in the header: Twice through direct `jku` and `x5u` fields, and twice using these fields within a JWK object. Users must provide their own URL, which the tool inserts into the tokens to simulate Server-Side Request Forgery. The distinguishing quality that sets this function apart is its coverage of all possible cases while other tools only implement a subset of these cases.

**Attacks on kid parameter** For attacks on the `kid` parameter (see Paragraph 3.2). The selection of payloads is based on the standard payloads of the individual attack classes:

- **Path Traversal:** `../../../../../../../../etc/passwd` [23] or `../../../../../../../../dev/null` for the `/dev/null` vector [26], the token is additionally signed with a null byte so that a successful path manipulation results in a valid token.
- **Command Injection:** `;id;`, `& id &` , or `|| id ||` test whether the `kid` value is evaluated in shell commands. These payloads are used since they are the standard PoC for command injection [27].
- **Command Injection Timebased:** `;ping -c 10 127.0.0.1;`, `& ping -c 10 127.0.0.1 &` , or `|| ping -c 10 127.0.0.1 ||` test whether the `kid` value is evaluated in shell commands. These commands offer a side-channel since they are designed to delay the response from the server, allowing detection of successful command injection [25].
- **SQL Injection:** `xxxx' UNION SELECT 'aaa`, with the token signed using `aaa` so that it appears valid if the injection succeeds [28].

It is important to note that, given the nature of this tool, the number of payloads is constrained. In order to test all possible scenarios, a significantly larger number of tokens and payloads would need to be created. Therefore, in order to facilitate more precise and specific testing, it is possible to integrate one's own `kid` payloads. Users can add custom `kid` payloads. Optionally, an expected key can be defined per payload, enabling precise token generation if known paths or predictable content exist in the target environment.

# 5 Execution of the Attacks and Results

This chapter provides a comprehensive overview of how the attack scenarios were carried out and what results were observed when tested against a purpose-built vulnerable application. The test environment included seven verifier configurations, each reflecting a common JWT validation flaw. As summarized in Table 5.1, the developed tool was able to uncover every intended vulnerability across all verifier instances.

| Attack Vector | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Signature Exclusion | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| None Alg | ○ | ● | ● | ○ | ○ | ○ | ○ |
| SSRF | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Psychic Signature | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| Custom Key | ● | ○ | ○ | ○ | ○ | ● | ○ |
| Key Confusion | ● | ○ | ○ | ● | ● | ○ | ○ |
| Empty Key | ● | ○ | ○ | ○ | ○ | ○ | ○ |

Table 5.1: Results of which verifiers were exploitable with the implemented tool. The ● indicator signifies that the generated token has been verified successfully. A notable exception is the vector SSRF, where the ● signifies that a request has been initiated the server. The presence of the ○ indicator signifies that the signature in question has not been verified as valid. For SSRF ○ shows that no request has been send.

## 5.1 Description of the Test Environment - eHacking Platform

The *eHacking Platform* [30], developed by Dr. Vladislav Mladenov and Dr. Christian Mainka, offers a modular environment for exploring topics in IT security. One of the modules focuses specifically on JWT security analysis[1] and includes seven verifier configurations, each intentionally misconfigured to replicate real-world JWT verification errors. These verifiers simulate typical implementation mistakes found

[1] `https://spa-elearning.cloud.nds.rub.de/json-sec/`

in production systems. The vulnerabilities were examined using the developed tool and are detailed in the following:

**Verifier 1** Verifier 1 performs only basic checks. While it ensures that the `alg` field is not set to `none`, it skips signature verification. This means that even unsigned tokens are accepted.

**Verifier 2** Verifier 2 accepts tokens that use the `none` algorithm.

**Verifier 3** Verifier 3 blacklists the `none` algorithm. However, the implementation fails to consider the variant `NoNe`, which still bypasses the filter.

**Verifier 4** This verifier is vulnerable to a key confusion attack (see Paragraph 3.2). It mistakenly interprets the public key parameter `n` in the JWK as a symmetric key, enabling attackers to forge signatures using HS algorithms.

**Verifier 5** Verifier 5 is also vulnerable to key confusion. It treats a stripped PEM key (without header, footer, and newlines) as a Base64-encoded symmetric key.

**Verifier 6** This verifier accepts custom asymmetric keys via the JWK header (see Paragraph 3.2). Asymmetric keys can be injected and verified successfully, while symmetric tokens are rejected.

**Verifier 7** Verifier 7 is susceptible to SSRF (see Paragraph 3.2). If the token includes an `x5u` field within a JWK, the server attempts to fetch the resource—even if signature verification ultimately fails. This confirms the SSRF behavior.

## 5.2 Testing the Tool Against eHacking

The testing process itself was straightforward and efficient. Each attack type can be configured through the tool's interface, and the resulting tokens can easily be exported, for example, into Burp Suite's Intruder. Identifying successful payloads is made simple thanks to the token indexing and attack mapping in the tool's output. The results clearly demonstrate that all vulnerabilities present in the eHacking platform were successfully detected using the tool. Thus, the tool proved capable of identifying a variety of JWT attacks, including Signature Exclusion, None Algorithm, SSRF, Custom Key, and Key Confusion.

The original token used for the eHacking platform is shown in Listing 1 and Listing 2. The token was generated by the eHacking platform and contains a JWK header with an RSA public key. The body includes a pseudo transaction with fields like `aud`, `amount`, `iban`, and `iss`. The goal for each attack was to modify the body of the token and to verify it successfully against the respective verifier. The body of the token used for the attack is shown in Listing 3. The `aud` field was changed to `Niklas Conrad`, the `amount` was set to 404, and the `iban` was modified to a different value. The `iss` field remained unchanged.

```json
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "RS256",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "use": "sig",
        "kid": "ced2a065-9f53-4c22-b872-702da906fd04",
        "n": "iYJqkyG6j8AtavEVg8UDWhal3ICVxchgWWVBgoVdnz_xiftm-YDXR6lDrYb-FJvub1gy2ctpwG-NUU8TzKPJ2PgFL3YG3tanldXm-jLooSgLPSgDG2MsEksWTo_iqjVJJ8oGXVec5HKQU90gcqBQPPMVjZ38BD9fkpbaYN-XC6TOCb5N5Kb04kiZvuf6dtrMEtlrThqtUsFexPNWzOGBSq9ZMEiZ5n8Lc4vUV2tDXzoOnTVAc_GIyGkUWQQ0go9cEUqoEQ693Kq89Fk2Wk_dCgSO8n_9NQnFixcJucs9JND2j5jHJ5jm0fcJv0aGlHlJ0aV9kHchHP77xQt1u_p2Qw"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```

Listing 1: Orginal JWT Header from the eHacking Platform.

```json
{
    "aud": "Vladislav.Mladenov",
    "amount": "99",
    "iban": "DE72 1234 5678 8901 23",
    "iss": "Christian.Mainka"
}
```

Listing 2: Orginal JWT Body from the eHacking Platform

```
{
    "aud": "Niklas Conrad",
    "amount": "404",
    "iban": "DE67 7978 7873 7487 84",
    "iss": "Christian.Mainka"
}
```

Listing 3: JSON body from used for the token generation.

The following paragraphs detail the results of the attacks against each verifier configuration, including the specific tokens generated and their verification outcomes.

**Verifier 1** It seems that the only check implemented in Verifier 1 is the algorithm check. It must not be `none`, but the signature is not verified at all. Even Signature Exclusion (see Paragraph 3.2) is possible. The token generated by our tool is shown in Listing 10. The verification was successfully completed. Since Signature Exclusion requires no changes to the header, the header remains unchanged and is not shown here.

**Verifier 2** In Verifier 2, the `none` algorithm (see Paragraph 3.2) is allowed. The tool generates 16 tokens (see Paragraph 4.3) with the `none` algorithm, which are all valid. One of the generated tokens is shown in Listing 11. The verification was successfully completed. The header of the token is shown in Listing 4. The header contains the `none` algorithm.

```
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg" : "none",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "use": "sig",
        "kid": "ced2a065-9f53-4c22-b872-702da906fd04",
        "n": "iYJqkyG6j8AtavEVg8UDWhal3ICVxchgWWVBgoVdnz_xiftm-YDXR6lDrY
          b-FJvub1gy2ctpwG-NUU8TzKPJ2PgFL3YG3tanldXm-jLooSgLPSgDG2MsEk
          sWTo_iqjVJJ8oGXVec5HKQU90gcqBQPPMVjZ38BD9fkpbaYN-XC6TOCb5N5K
          b04kiZvuf6dtrMEtlrThqtUsFexPNWzOGBSq9ZMEiZ5n8Lc4vUV2tDXzoOnT
          VAc_GIyGkUWQQ0go9cEUqoEQ693Kq89Fk2Wk_dCgSO8n_9NQnFixcJucs9JN
          D2j5jHJ5jm0fcJv0aGlHlJ0aV9kHchHP77xQt1u_p2Qw"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```

Listing 4: Header of the token generated for Verifier 2 - None Algorithm.

**Verifier 3** In Verifier 3, the `none` algorithm (see Paragraph 3.2) is allowed, but the implementation uses a blacklist to prevent the use of the `none` algorithm in different capitalizations. The tool generates 16 tokens (see Paragraph 4.3) with the `none` algorithm, but only the token with the `NoNe` algorithm is valid. The verification was successfully completed. The forged token is shown in Listing 12. The header is shown in Listing 5.


```json
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "NoNe",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "use": "sig",
        "kid": "ced2a065-9f53-4c22-b872-702da906fd04",
        "n": "iYJqkyG6j8AtavEVg8UDWhal3ICVxchgWWVBgoVdnz_xiftm-YDXR6lDrY
        ↪  b-FJvub1gy2ctpwG-NUU8TzKPJ2PgFL3YG3tanldXm-jLooSgLPSgDG2MsEk
        ↪  sWTo_iqjVJJ8oGXVec5HKQU90gcqBQPPMVjZ38BD9fkpbaYN-XC6TOCb5N5K
        ↪  b04kiZvuf6dtrMEtlrThqtUsFexPNWzOGBSq9ZMEiZ5n8Lc4vUV2tDXzoOnT
        ↪  VAc_GIyGkUWQQ0go9cEUqoEQ693Kq89Fk2Wk_dCgSO8n_9NQnFixcJucs9JN
        ↪  D2j5jHJ5jm0fcJv0aGlHlJ0aV9kHchHP77xQt1u_p2Qw"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```


Listing 5: Header of the token generated for Verifier 3 - NoNe Algorithm.

**Verifier 4** In Verifier 4, the key confusion attack (see Paragraph 3.2) is possible. The eighth payload (see Paragraph 4.3) of the attack signs with the modulo n interpreted as an ASCII string. When generating tokens for HS256, HS384, and HS512, the tool forges three valid tokens, one of which is shown in Listing 13. The verification was successfully completed. The header of the token is shown in Listing 6. The header contains the algorithm HS256.

```
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "HS256",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "use": "sig",
        "kid": "ced2a065-9f53-4c22-b872-702da906fd04",
        "n": "iYJqkyG6j8AtavEVg8UDWhal3ICVxchgWWVBgoVdnz_xiftm-YDXR6lDrYb-FJvub1gy2ctpwG-NUU8TzKPJ2PgFL3YG3tanldXm-jLooSgLPSgDG2MsEksWTo_iqjVJJ8oGXVec5HKQU90gcqBQPPMVjZ38BD9fkpbaYN-XC6TOCb5N5Kb04kiZvuf6dtrMEtlrThqtUsFexPNWzOGBSq9ZMEiZ5n8Lc4vUV2tDXzoOnTVAc_GIyGkUWQQ0go9cEUqoEQ693Kq89Fk2Wk_dCgSO8n_9NQnFixcJucs9JND2j5jHJ5jm0fcJv0aGlHlJ0aV9kHchHP77xQt1u_p2Qw"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```

Listing 6: Header of the token generated for Verifier 4 - Key Confusion - symmetric key is `n`.

**Verifier 5** In Verifier 5, the key confusion attack (see Paragraph 3.2) is possible. The fifth payload (see Paragraph 4.3) of the attack signs with the PEM string (without header, footer, and newlines) interpreted as a BASE64 string. When generating tokens for HS256, HS384, and HS512, the tool forges three valid tokens, one of which is shown in Listing 14. The verification was successfully completed. The header of the token is shown in Listing 7. The header contains the algorithm HS256.


```
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "HS256",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "use": "sig",
        "kid": "ced2a065-9f53-4c22-b872-702da906fd04",
        "n": "iYJqkyG6j8AtavEVg8UDWhal3ICVxchgWWVBgoVdnz_xiftm-YDXR6lDrY
        ↪  b-FJvub1gy2ctpwG-NUU8TzKPJ2PgFL3YG3tanldXm-jLooSgLPSgDG2MsEk
        ↪  sWTo_iqjVJJ8oGXVec5HKQU90gcqBQPPMVjZ38BD9fkpbaYN-XC6TOCb5N5K
        ↪  b04kiZvuf6dtrMEtlrThqtUsFexPNWzOGBSq9ZMEiZ5n8Lc4vUV2tDXzoOnT
        ↪  VAc_GIyGkUWQQ0go9cEUqoEQ693Kq89Fk2Wk_dCgSO8n_9NQnFixcJucs9JN
        ↪  D2j5jHJ5jm0fcJv0aGlHlJ0aV9kHchHP77xQt1u_p2Qw"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```


Listing 7: Header of the token generated for Verifier 5 - Key Confusion - symmetric key is the PEM string (without header, footer, and newlines) interpreted as BASE64.

**Verifier 6** In Verifier 6, the custom key attack (see Paragraph 3.2) is possible. The custom key function generated four Tokens, one for each algorithm (see Paragraph 4.3). The token with the symmetric custom key was not verified successfully, but the asymmetric keys were verified successfully. The forged token for RS256 is shown in Listing 15. The verification was successfully completed. The header of the token is shown in Listing 8. The header contains the algorithm RS256 and the custom JWK.

```
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "RS256",
    "jwk": {
        "kty": "RSA",
        "alg": "RS256",
        "kid": "945b6ee8-9547-45c4-b0b9-992906d79f83",
        "n": "3IklRLCQsE5Gg0N6DptiPola9R2le_b0nyMpGGbm6OBqldrZkP-gwFgeBN
        ↪  oLS44xuGMR_hLPLLbr1uUizhRcrakA-L1hmiwdz6Kmfn9iRD6Dbkz7upL_h6
        ↪  Bv6-8aoEFvCikm4BNGiEv5NtFTyLdD22LNJA3LTQaaZUFHiUxjtYJ_CJP4Xm
        ↪  DVIleG1cVwE5DEEUzxiczfWLo45lzOCvnPDTu8Syt-nbRmEexUckoe9hkCSo
        ↪  iPSJcqOM6uWwQ9GOujXB_0p4GKsW8HeQRVEbpYZgXbCVsC29jlz2h8fPqw2g
        ↪  TYznjFSWh8mr6_Z3SpI1jdZMkb2wVldHpiOn555JVQYw",
        "e": "AQAB"
    },
    "kid": "945b6ee8-9547-45c4-b0b9-992906d79f83"
}
```

Listing 8: Header of the token generated for Verifier 6 - Custom Key - RS256.

**Verifier 7** In Verifier 7, the SSRF attack (see Paragraph 3.2) is possible. The third payload generated by the tool (see Paragraph 4.3) contains a x5u parameter in the JWK. The server sends a request to the URL specified in the x5u parameter. The forged token is shown in Listing 16. The verification was not successful, but since the server sends a request, SSRF is possible. The header of the token is shown in Listing 9. The header contains the x5u parameter in a JWK. The SSRF attack was tested with Burp Suite; thus, the URL is:
`http://5w2bti8ddaaw0a5uah8o4yovmmsdgf44.oastify.com`

```
{
    "typ": "JWT",
    "gid": "4906acf7-a79a-45eb-9919-f79158eac72a",
    "alg": "RS256",
    "jwk": {
        "kty": "RSA",
        "e": "AQAB",
        "n": "vRDkI8bC7--s3rE6Tp-xwo7ACC_7RT7Ps2Q7YJUhTF4XmcNXDbBjsbXMaC
        ↪  MiE2e_UQGqQQQ_PLRVmwVp_k9bwQ",
        "kid": "1234",
        "x5u": "http://569b3iidnakwaafukhioeyyvwm2dqee3.oastify.com"
    },
    "kid": "ced2a065-9f53-4c22-b872-702da906fd04"
}
```

Listing 9: Header of the token generated for Verifier 7 - SSRF - embedded `x5u`.

**Results from the tool against eHacking** As already summarized above and shown in Table 5.1, it was possible to solve all verifiers or find all intended vulnerabilities with the tool. This indicates that the tool will also uncover errors in the verification mechanism in real frameworks. Another aspect is the workflow used for testing. Despite the author's involvement in the implementation, the workflow appears intuitive and efficient. The individual attacks are easy to configure; the token list can then be easily exported and, in this case, reused in Burp Intruder. If a token stands out in the intruder, you can use the index in the generated table in the tool to see which attack vector worked.

## 5.3 Testing Other Tools Against eHacking

In the following, the other tools are tested against eHacking to get an overview of which attacks are already implemented by these tools. This is later used to compare the selection of attacks.

### 5.3.1 JOSEPH

JOSEPH [13] is a Burp Suite extension developed during a master's thesis by Dennis Detering at Ruhr-University Bochum in collaboration with Spike Reply GmbH.

JOSEPH supports the following attack types:

- None Algorithm (referred to as Signature Exclusion in JOSEPH)
- Key Confusion

**Verifier 1** JOSEPH does not produce a valid token using the "Signature Exclusion" option here, since the verifier rejects `none`. However, JOSEPH successfully generates valid tokens via the "Key Confusion" attack.

**Verifier 2** Here, JOSEPH generates four variants of `none` tokens, all of which pass. Verification is successful.

**Verifier 3** JOSEPH again creates four tokens with `none` variants, but does not include the bypass variant `NoNe`. Verification fails as a result.

**Verifier 4** It is not possible to import a JWK into JOSEPH. Regardless of the format, it always fails with an error message. The attack also generated no valid tokens with the correct PEM key.

**Verifier 5** JOSEPH does not generate a working payload for this verifier. Verification fails.

**Verifier 6** JOSEPH doesn't support custom key attacks at all.

**Verifier 7** SSRF payloads are not part of JOSEPH's current functionality.

**Results From Joseph Against eHacking** JOSEPH is only able to successfully exploit Verifier 1 and Verifier 2. Verifier 4 may be exploitable using JOSEPH; however, due to a lack of proficiency in its utilization, the key could not be imported. The results of the attacks are summarized in Table 5.2.

| Attack Vector | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Signature Exclusion | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| Key Confusion | ● | ○ | ○ | ○ | ○ | ○ | ○ |

Table 5.2: Results of which verifiers were exploitable with JOSEPH. The ● indicator signifies that the generated token has been verified successfully. The presence of the ○ indicator signifies that the signature in question has not been verified as valid.

### 5.3.2 jwt_tool

"jwt_tool" [18] is a Python tool developed by Ticarpi. It is a command-line tool that can be used to generate and verify JWTs. It supports various algorithms, and among others, the following attack types:

- None Algorithm
- Key Confusion
- Custom Key (referred to as "key injection" in "jwt_tool")
- Empty Key (referred to as "blank password" in "jwt_tool")
- Signature Exclusion (referred to as "null signature" in "jwt_tool")
- Psychic Signature
- "Spoof JWKS": Generates one token with the `jku` header set to a specified URL.

**Verifier 1** "jwt_tool" generates a valid token with a signature exclusion. The token is verified successfully.

**Verifier 2** "jwt_tool" generates four valid tokens with the `none` algorithm. All of them verified successfully.

**Verifier 3** Once more "jwt_tool" generates four tokens with different capitalizations of `none`. None of them are verified successfully, since "jwt_tool" does not implement the `NoNe` variant.

**Verifier 4** One token is generated with the key confusion attack, but it is not verified successfully.

**Verifier 5** One token is generated with the key confusion attack, but it is not verified successfully.

**Verifier 6** "jwt_tool" generates a valid token with the custom key attack. The verification is successful.

**Verifier 7** Since "jwt_tool" does not directly support SSRF attacks, the "spoof JWKS" attack is used. No request is sent to the URL; therefore, this verifier is not exploitable with "jwt_tool".

**Results From jwt_tool Against eHacking** "jwt_tool" is able to successfully exploit Verifier 1, Verifier 2, and Verifier 6. The other verifiers are not exploitable with "jwt_tool". The used attacks are summarized in Table 5.3.

| Attack Vector | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Signature Exclusion | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| Key Confusion | ● | ○ | ○ | ○ | ○ | ● | ○ |
| Custom Key | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| None Algorithm | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| Spoof JWKS | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

Table 5.3: Results of which verifiers were exploitable with jwt_tool. The ● indicator signifies that the generated token has been verified successfully. The presence of the ○ indicator signifies that the signature in question has not been verified as valid. For "spoof JWKS" the ○ indicator signifies that no request was sent to the specified URL.

### 5.3.3 JWT Editor

*JWT Editor* [14] is a Burp Suite extension designed to serve as a comprehensive toolset for the analysis and manipulation of JSON Web Tokens (JWTs) within the Burp Suite environment. It automatically detects JWTs in HTTP, offering functionality for editing, signing, verifying, encrypting, and decrypting tokens. In addition, it supports the execution of several well-known attacks targeting flawed or insecure JWT implementations.

*JWT Editor* supports the following attack types:

- None Algorithm
- Key Confusion (referred to as "HMAC Key Confusion" in *JWT Editor*)

- Custom Key (referred to as “Embedded JWK” in *JWT Editor*)
- Empty Key
- Psychic Signature
- SSRF

**Verifier 1** *JWT Editor* generates a valid token. The verification is successful.

**Verifier 2** *JWT Editor* generates a valid token with the `none` algorithm. The verification is successful.

**Verifier 3** There are four different variants of `none`, but the `NoNe` variant is not implemented. The verification fails.

**Verifier 4** Once the JWK is imported, there are two possible tokens generated. The first one with trailing new line characters and the second one without. None of them is verified successfully.

**Verifier 5** Again, *JWT Editor* can generate two tokens, one with trailing new line characters and one without. None of them is verified successfully.

**Verifier 6** *JWT Editor* generates a valid token with the custom key attack. The verification is successful.

**Verifier 7** Two tokens are generated, one with the `jku` header and one with the `x5u` header. No request is sent to the URL; therefore, this verifier is not exploitable with *JWT Editor*.

**Results From JWT Editor Against eHacking** *JWT Editor* is able to successfully exploit Verifier 1, Verifier 2, and Verifier 6. The other verifiers are not exploitable with *JWT Editor*. The used attacks are summarized in Table 5.4.

| Attack Vector | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Custom Key | ● | ○ | ○ | ○ | ○ | ● | ○ |
| Key Confusion | ● | ○ | ○ | ○ | ○ | ○ | ○ |
| None Algorithm | ○ | ● | ○ | ○ | ○ | ○ | ○ |
| SSRF | ○ | ○ | ○ | ○ | ○ | ○ | ○ |

Table 5.4: Results of which verifiers were exploitable with JWT Editor. The ● indicator signifies that the generated token has been verified successfully. The presence of the ○ indicator signifies that the signature in question has not been verified as valid. For SSRF the ○ indicator signifies that no request was sent to the specified URL.

## 5.4 Summary of the Results

To assess the performance and coverage of the developed tool, it was compared against three well-known alternatives: `jwt_tool` and the Burp Suite extensions JWT Editor and JOSEPH. All tools were evaluated using the same intentionally insecure test environment, which included seven different verifier implementations. Each one simulated a known flaw in JWT verification logic. The developed tool successfully detected vulnerabilities across all seven configurations (see Table 5.1). It supports a broad range of attack vectors, including signature exclusion, exploitation of the none algorithm, several types of key confusion, SSRF, and a custom key injection method. In contrast, the JOSEPH extension was only able to bypass two verifiers (see Table 5.2). Its attack scope is limited to non-algorithm exploitation and a basic version of the key confusion technique. Additionally, it does not account for all variations in JWT headers, such as case sensitivity in algorithm names or key confusion variations, which limits its effectiveness. The `jwt_tool` command-line utility was successful in three cases (see Table 5.3). It managed the standard none algorithm attack and the custom key attack, but it also lacks support for variations of the generated payloads. The JWT Editor extension also succeeded in bypassing three verifiers (see Table 5.4). Its graphical interface allows for convenient editing and signing of tokens. However, like `jwt_tool`, it lacks a broader payload set required for more extensive testing. This comparison, shown in Table 5.5, clearly demonstrates the superior coverage of the developed tool in terms of token generation, attack vector bandwidth, and variety of the respective payloads. This, combined with the intuitive workflow and a user-friendly interface, makes the developed tool a valuable asset for security researchers and penetration testers. An even more detailed overview of the implemented features in comparison to the other tools can be found in Table A.1. This once again demonstrates the limited range of implemented attacks in the other tools. However, it also illustrates that the developed tool incorporates additional advantageous features, including a key generator, JWE handling, and a user interface. A distinctive characteristic that is not essential

| Attack Vector | Signature Exclusion - Verifier 1 | None Algorithm - Verifier 2 | None Algorithm - Verifier 3 | Key Confusion - Verifier 4 | Key Confusion - Verifier 5 | Custom Key - Verifier 6 | SSRF - Verifier 7 |
|---|---|---|---|---|---|---|---|
| Developed Tool | ● | ● | ● | ● | ● | ● | ● |
| JOSEPH | ● | ● | ○ | ○ | ○ | ○ | ○ |
| jwt_tool | ● | ● | ○ | ○ | ○ | ● | ○ |
| JWT Editor | ● | ● | ○ | ○ | ○ | ● | ○ |

Table 5.5: Results of which verifiers were exploitable with all tools. The ● indicator signifies that the generated token has been verified successfully. It is shown that the developed tool is able to exploit all verifiers. The presence of the ○ indicator signifies that the signature in question has not been verified as valid. For Verifier 7 the ● indicator signifies that the server sent a request to the specified URL. ○ signifies that no request was sent to the specified URL.

for the execution of the tests is the capacity to generate invalid JSON. This feature enables the generation of duplicate keys and the insertion of byte values. Although invalid JSON and byte values are more commonly used for fuzzing, this remains a valuable feature for evaluating the validation process and identifying unexpected behavior, which could be exploited. The presence of duplicate keys can facilitate a more sophisticated attack, exploiting the potential for divergent logics and parsing mechanisms. Furthermore, this tool can function in any environment without the necessity of a preliminary setup.

# 6 Conclusion

This thesis has successfully developed a web-based tool for generating JSON Web Tokens tailored to all well-known attack scenarios to help security researchers and penetration testers assess the security of JWT-based applications. Unlike other existing tools, the developed solution does not require installation or configuration. It supports all JWS and JWE algorithms defined in the JWA standard, while only depending on the browser's JavaScript engine and the `Web Crypto API`. Through systematic testing, the tool has proven its effectiveness in generating tokens that can be used to exploit vulnerabilities in JWT implementations. Beyond the functionality, the tool also offers an intuitive user interface that simplifies the process of creating and testing JWTs. The tool has demonstrated a streamlined workflow while testing. The developed solution not only introduces a practical tool for JWT security testing but also fills a clear gap in the existing landscape of JWT tools. While other tools focus on parsing, analyzing, editing JWTs, or implementing a limited set of attacks, this tool provides a comprehensive solution that covers all relevant attack scenarios.

**Future Work** could focus on extending the tool's capabilities to include more attack scenarios, such as very specific attacks targeting specific versions of JWT libraries or implementations (e.g., CVE-2022-23529). Another feature could be the calculation of the public RSA key from the signatures of two tokens. This would allow the tool to execute a key confusion attack without knowing the public key in advance. It could also explore the integration of attacks on JWEs. To further enhance the automation of security testing, one could consider sending the tokens directly to a target application and implementing a feedback mechanism to evaluate the success of the attacks. Since the tool depends on the `Web Crypto API`, it could be helpful to bundle `Web Crypto API` with the tool to ensure compatibility even if the API is changing. In order to further guarantee both robustness and functionality, it would be useful to implement a series of tests. The implementation of these tests would ensure that the tool continues to function as expected. Future updates will include the implementation of the previously omitted encryption algorithms. There are certainly many more improvements and features that could be added. The plan is to continue developing and improving the tool by implementing these improvements and new features. Also, this tool paves the way for further research by greatly facilitating testing and enabling a deeper and more comprehensive investigation into the security of JWT frameworks.

# A Appendix

## A.1 List of AI-Tools

In the preparation of this thesis, several AI tools were utilized to enhance the research and writing process. I hereby declare that I have carefully checked all the suggestions generated by AI for correctness. The following tools were particularly instrumental:

**ChatGPT** Used for brainstorming research questions and to rephrase texts to improve readability. It was not used for crafting text bodies. It was also used to generate illustrations.

**DeepL** Applied for translating key passages from German sources into English.

**Grammarly** Used for spellchecking and identifying grammatical errors.

## A.2 List of Generated Tokens for the Verifiers

### Verifier 1


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
↪  WM3MmEiLCJhbGciOiJSUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJlIjoiQVFBQi
↪  IsInVzZSI6InNpZyIsImtpZCI6ImNlZDJhMDY1LTlmNTMtNGMyMi1iODcyLTcwMmR
↪  hOTA2ZmQwNCIsIm4iOiJpWUpxa3lHNmo4QXRhdkVWZzhVRFdoYWwzSUNWeGNoZ1dX
↪  VkJnb1ZkbnpfeGlmdG0tWURYUjZsRHJZYi1GSnZ1YjFneTJjdHB3Ry1OVVU4VHpLU
↪  EoyUGdGTDNZRzN0YW5sZFhtLWpMb29TZ0xQU2dERzJNc0Vrc1dUb19pcWpWSko4b0
↪  dYVmVjNUhLUVU5MGdjcUJRUFBNVmpaMzhCRDlma3BiYVl0LVhDNlRPQ2I1TjVLYjA
↪  0a2ladnVmNmR0ck1FdGxyVGhxdFVzRmV4UE5Xek9HQlNxOVpNRWlaNW44TGM0dlVW
↪  MnREWHpvT25UVkFjX0dJeUdrVVdRUTBnbzljRVVxb0VRNjkzS3E4OUZrMldrX2RDZ
↪  1NPOG5fOU5RbkZpeGNKdWNzOUpORDJqNWpISjVqbTBmY0p2MGFHbEhsSjBhVjlrSG
↪  NoSFA3N3hRdDF1X3AyUXcifSwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLWI4NzI
↪  tNzAyZGE5MDZmZDA0In0.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50Ijoi
↪  NDA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaHJpc
↪  3RpYW4uTWFpbmthIn0.
```


Listing 10: Token generated for Verifier 1 - Signature Exclusion.

### Verifier 2


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
↪  WM3MmEiLCJhbGciOiJub25lIiwiandrIjp7Imt0eSI6IlJTQSIsImUiOiJBUUFCIi
↪  widXNlIjoic2lnIiwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLWI4NzItNzAyZGE
↪  5MDZmZDA0IiwibiI6ImlZSnFreUc2ajhBdGF2RVZnOFVEV2hhbDNJQ1Z4Y2hnV1dW
↪  QmdvVmRuel94aWZ0bS1ZRFhSNmxEclliLUZKdnViMWd5MmN0cHdHLU5VVThUektQS
↪  jJQZ0ZMM1lHM3RhbmxkWG0takxvb1NnTFBTZ0RHMk1zRWtzV1RvX2lxalZKSjhvR1
↪  hWZWM1SEtRVTkwZ2NxQlFQUE1WalozOEJEOWZrcGJhWU4tWEM2VE9DYjVONUtiMDR
↪  raVp2dWY2ZHRyTUV0bHJUaHF0VXNGZXhQTld6T0dCU3E5Wk1FaVo1bjhMYzR2VVYy
↪  dERYem9PblRWQWNfR0l5R2tVV1FRMGdvOWNFVXFvRVE2OTNLcTg5RmsyV2tfZENnU
↪  084bl85TlFuRml4Y0p1Y3M5Sk5EMmo1akhKNWptMGZjSnYwYUdsSGxKMGFWOWtIY2
↪  hIUDc3eFF0MXVfcDJRdyJ9LCJraWQiOiJjZWQyYTA2NS05ZjUzLTRjMjItYjg3Mi0
↪  3MDJkYTkwNmZkMDQifQ.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50IjoiN
↪  DA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaHJpc3
↪  RpYW4uTWFpbmthIn0.
```


Listing 11: Token generated for Verifier 2 - None Algorithm.

### Verifier 3


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
↪ WM3MmEiLCJhbGciOiJOb05lIiwiandrIjp7Imt0eSI6IlJTQSIsImUiOiJBUUFCIi
↪ widXNlIjoic2lnIiwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLWI4NzItNzAyZGE
↪ 5MDZmZDA0IiwibiI6ImlZSnFreUc2ajhBdGF2RVZnOFVEV2hhbDNJQ1Z4Y2hnV1dW
↪ QmdvVmRuel94aWZ0bS1ZRFhSNmxEclliLUZKdnViMWd5MmN0cHdHLU5VVThUektQS
↪ jJQZ0ZMM1lHM3RhbmxkWG0takxvb1NnTFBTZ0RHMk1zRWtzV1RvX2lxalZKSjhvR1
↪ hWZWM1SEtRVTkwZ2NxQlFQUE1WalozOEJEOWZrcGJhWU4tWEM2VE9DYjVONUtiMDR
↪ raVp2dWY2ZHRyTUV0bHJUaHF0VXNGZXhQTld6T0dCU3E5Wk1FaVo1bjhMYzR2VVYy
↪ dERYem9PblRWQWNfR0l5R2tVV1FRMGdvOWNFVXFvRVE2OTNLcTg5RmsyV2tfZENnU
↪ 084bl85TlFuRml4Y0p1Y3M5Sk5EMmo1akhKNWptMGZjSnYwYUdsSGxKMGFWOWtIY2
↪ hIUDc3eFF0MXVfcDJRdyJ9LCJraWQiOiJjZWQyYTA2NS05ZjUzLTRjMjItYjg3Mi0
↪ 3MDJkYTkwNmZkMDQifQ.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50IjoiN
↪ DA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaHJpc3
↪ RpYW4uTWFpbmthIn0.
```


Listing 12: Token generated for Verifier 3 - NoNe Algorithm.

### Verifier 4


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
↪ WM3MmEiLCJhbGciOiJIUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJlIjoiQVFBQi
↪ IsInVzZSI6InNpZyIsImtpZCI6ImNlZDJhMDY1LTlmNTMtNGMyMi1iODcyLTcwMmR
↪ hOTA2ZmQwNCIsIm4iOiJpWUpxa3lHNmo4QXRhdkVWZzhVRFdoYWwzSUNWeGNoZ1dX
↪ VkJnb1ZkbnpfeGlmdG0tWURYUjZsRHJZYi1GSnZ1YjFneTJjdHB3Ry1OVVU4VHpLU
↪ EoyUGdGTDNZRzN0YW5sZFhtLWpMb29TZ0xQU2dERzJNc0Vrc1dUb19pcWpWSko4b0
↪ dYVmVjNUhLUVU5MGdjcUJRUFBNVmpaMzhCRDlma3BiYVl0LVhDNlRPQ2I1TjVLYjA
↪ 0a2ladnVmNmR0ck1FdGxyVGhxdFVzRmV4UE5Xek9HQlNxOVpNRWlaNW44TGM0dlVW
↪ MnREWHpvT25UVkFjX0dJeUdrVVdRUTBnbzljRVVxb0VRNjkzS3E4OUZrMldrX2RDZ
↪ 1NPOG5fOU5RbkZpeGNKdWNzOUpORDJqNWpISjVqbTBmY0p2MGFHbEhsSjBhVjlrSG
↪ NoSFA3N3hRdDF1X3AyUXcifSwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLWI4NzI
↪ tNzAyZGE5MDZmZDA0In0.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50Ijoi
↪ NDA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaHJpc
↪ 3RpYW4uTWFpbmthIn0.GJxpMKsRFsLzduqj4ONvwUhyCVrQbWgThaohdCgE_Sk
```


Listing 13: Token generated for Verifier 4 - Key Confusion - symmetric key is n.

### Verifier 5


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
WM3MmEiLCJhbGciOiJIUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJlIjoiQVFBQi
IsInVzZSI6InNpZyIsImtpZCI6ImNlZDJhMDY1LTlmNTMtNGMyMi1iODcyLTcwMmR
hOTA2ZmQwNCIsIm4iOiJpWUpxa3lHNmo4QXRhdkVWZzhVRFdoYWwzSUNWeGNoZ1dX
VkJnb1ZkbnpfeGlmdG0tWURYUjZsRHJZYi1GSnZ1YjFneTJjdHB3Ry1OVVU4VHpLU
EoyUGdGTDNZRzN0YW5sZFhtLWpMb29TZ0xQU2dERzJNc0Vrc1dUb19pcWpWSko4b0
dYVmVjNUhLUVU5MGdjcUJRUFBNVmpaMzhCRDlma3BiYVlOLVhDNlRPQ2I1TjVLYjA
0a2ladnVmNmR0ck1FdGxyVGhxdFVzRmV4UE5Xek9HQlNxOVpNRWlaNW44TGM0dlVW
MnREWHpvT25UVkFjX0dJeUdrVVdRUTBnbzljRVVxb0VRNjkzS3E4OUZrMldrX2RDZ
1NPOG5fOU5RbkZpeGNKdWNzOUpORDJqNWpISjVqbTBmY0p2MGFHbEhsSjBhVjlrSG
NoSFA3N3hRdDF1X3AyUXcifSwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLWI4NzI
tNzAyZGE5MDZmZDA0In0.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50Ijoi
NDA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaHJpc
3RpYW4uTWFpbmthIn0.J3o932LCYf4bE21JnAh9vdY2lX6PNBe7uFSbqamaobA
```


Listing 14: Token generated for Verifier 5 - Key Confusion - symmetric key is the PEM string (without header, footer, and newlines) interpreted as BASE64.

### Verifier 6


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY ⌟
↪ WM3MmEiLCJhbGciOiJSUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJhbGciOiJSUz ⌟
↪ I1NiIsImtpZCI6Ijk0NWI2ZWU4LTk1NDctNDVjNC1iMGI5LTk5MjkwNmQ3OWY4MyI ⌟
↪ sIm4iOiIzSWtsUkxDUXNFNUdnME42RHB0aVBvbGE5UjJsZV9iMG55TXBHR2JtNk9C ⌟
↪ cWxkclprUC1nd0ZnZUJOb0xTNDR4dUdNUl9oTFBMTGJyMXVVaXpoUmNyYWtBLUwxa ⌟
↪ G1pd2R6NkttZm45aVJENkRia3o3dXBMX2g2QnY2LThhb0VGdkNpa200Qk5HaUV2NU ⌟
↪ 50RlR5TGREMjJMTkpBM0xUUWFhWlVGSGlVeGp0WUpfQ0pQNFhtRFZJbGVHMWNWd0U ⌟
↪ 1REVFVXp4aWN6ZldMbzQ1bHpPQ3ZuUERUdThTeXQtbmJSbUVleFVja29lOWhrQ1Nv ⌟
↪ aVBTSmNxT002dVd3UTlHT3VqWEJfMHA0R0tzVzhIZVFSVkVicFlaZ1hiQ1ZzQzI5a ⌟
↪ mx6Mmg4ZlBxdzJnVFl6bmpGU1doOG1yNl9aM1NwSTFqZFpNa2Iyd1ZsZEhwaU9uNT ⌟
↪ U1SlZRWXciLCJlIjoiQVFBQiJ9LCJraWQiOiI5NDViNmVlOC05NTQ3LTQ1YzQtYjB ⌟
↪ iOS05OTI5MDZkNzlmODMifQ.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW50I ⌟
↪ joiNDA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJDaH ⌟
↪ Jpc3RpYW4uTWFpbmthIn0.hXG-sYI3d8XFhvNIxGtJf3Xn-197hmGHN8IA0riL_lO ⌟
↪ 0YPH1RsQ60Yu0zl3KgOffJUSVHBG09Guo0jz3SWpD-OTKR-ZIpGGUx81Yjdo6rqeT ⌟
↪ 53F2MrbMnyeXGwYaf2D6-rEdiSjjAX7Z-zlAoZRAjmirsN24kPxF2grgtQfPw0euO ⌟
↪ uFBTZQ_ryJCr43ao9IXkF02Lm1hVpl3ujOxNaC0cDJutevfAySZcX8mMBp3N0RuiS ⌟
↪ ZtkiB7nGtFTmLwtEXqpMX9zv9BQU1HISJsFzgFjvDk5WODA3Paw8XZB2ze8hko4eL ⌟
↪ rSPUjlCvpdl2P_MMpo26NNJ5zQhlA1jZLtd28Pg
```


Listing 15: Token generated for Verifier 6 - Custom Key - RS256.

### Verifier 7


```
eyJ0eXAiOiJKV1QiLCJnaWQiOiI0OTA2YWNmNy1hNzlhLTQ1ZWItOTkxOS1mNzkxNThlY
↪  WM3MmEiLCJhbGciOiJSUzI1NiIsImp3ayI6eyJrdHkiOiJSU0EiLCJlIjoiQVFBQi
↪  IsIm4iOiJ2UkRrSThiQzctLXMzckU2VHAteHdvN0FDQ183UlQ3UHMyUTdZSlVoVEY
↪  0WG1jTlhEYkJqc2JYTWFDTWlFMmVfVVFHcVFRUV9QTFJWbXdWcF9rOWJ3USIsImtp
↪  ZCI6IjEyMzQiLCJ4NXUiOiJodHRwOi8vNTY5YjNpaWRuYWt3YWFmdWtoaW9leXl2d
↪  20yZHFlZTMub2FzdGlmeS5jb20ifSwia2lkIjoiY2VkMmEwNjUtOWY1My00YzIyLW
↪  I4NzItNzAyZGE5MDZmZDA0In0.eyJhdWQiOiJ0aWtsYXMgQ29ucmFkIiwiYW1vdW5
↪  0IjoiNDA0IiwiaWJhbiI6IkRFNjcgNzk3OCA3ODczIDc0ODcgODQiLCJpc3MiOiJD
↪  aHJpc3RpYW4uTWFpbmthIn0.A8zvAACijfz3vgo2st0_W3DUbZHaVEg3RbvxyF8It
↪  NVQHZk7ar2zrOhUJbpXSzspdkjNJA2CSBjRbAc32u4wQF5vHICRkUve9Y8kLBPLY0
↪  CBFSfWmdLkKiwJSjZb3vk5o11PDSUsEs3AQ-5vG9rogozpXHrxbTcMu1D-wj3Dw0I
↪  cwUbHYI1-2SHIcpHdofW8_3rxgZ8QmcPljzWPWWjZ-PmUX9G1P2-gpvC79qJuuCjR
↪  Vz8yDAS-TmL52pg49Yb_f6m8LIjwEYvJT-CrcUyjBt36Jd3JG4ulWb8X-3XcRhCYC
↪  0MnpcpSeziL9GC-IVrvPi4_oJvyKBaX3CWE-YGALQ
```


Listing 16: Token generated for Verifier 7 - SSRF - embedded x5u.

## A.3 Comparison of Tools - Table

| Basic Features | jwt.io | fusion auth.io | token.dev | jwt.ms | irrte.ch | dinochiesa.github.io/jwt/ | JOSEPH | jwt_tool | JWT Editor | My Tool |
|---|---|---|---|---|---|---|---|---|---|---|
| UI | ● | ● | ● | ● | ● | ● | ● | ○ | ● | ● |
| Encode Decode | ● | ● | ● | ◐ | ● | ● | ● | ● | ● | ● |
| Encode Duplicate Claims | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Sign Verify | ● | ○ | ● | ○ | ● | ● | ● | ● | ● | ● |
| Supports all Sig. Alg. | ● | ○ | ○ | ○ | ○ | ● | ● | ● | ● | ● |
| Encrypt Decrypt | ○ | ○ | ○ | ○ | ○ | ● | ◐ | ○ | ● | ● |
| Key Generator | ◐ | ○ | ○ | ○ | ◐ | ● | ● | ● | ● | ● |
| **Attacks (Automated Payload Generation)** | | | | | | | | | | |
| Signature Exclusion | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ○ | ● |
| None Algorithm | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ● |
| Custom Key | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● |
| Key Confusion | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ● |
| SSRF | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ◐ | ● | ● |
| kid Attacks | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Psychic Signature | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● |
| Empty Key | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● |
| Weak HMAC | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● |
| Empty Key | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● | ● | ● |
| Invalid JSON | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |

Table A.1: Comparison of tools and their features.

Comparison of tools based on their basic functionality and features for generating attacks. Implemented features are marked with ● and not implemented features are marked with ○. The symbol ◐ indicates that the feature is partially implemented. The table is divided into two sections: basic functionalities and supported attacks. While it is important to evaluate the attack coverage, it is important to note that the first six tools are not developed for generating attack tokens, and therefore, the theoretical possibility of manual creation is not considered.

# List of Figures

# List of Tables

# Bibliography


[1] *Apigee JWT Debugger*. 2025. URL: https://dinochiesa.github.io/jwt/ (visited on 07/13/2025).

[2] Hamed Asemi. *A Study On API Security Pentesting*. Master's Thesis. 2023. URL: https://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=4386&context=theses.

[3] Dennis Detering, Juraj Somorovsky, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. "On The (In-)Security Of JavaScript Object Signing And Encryption". In: *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*. ROOTS. Vienna, Austria: Association for Computing Machinery, 2017. ISBN: 9781450353212. DOI: 10.1145/3150376.3150379. URL: https://doi.org/10.1145/3150376.3150379.

[4] OWASP Foundation. *Testing JSON Web Tokens – OWASP Web Security Testing Guide*. 2025. (Visited on 07/23/2025).

[5] *FusionAuth JWT Decoder*. 2025. URL: https://fusionauth.io/dev-tools/jwt-decoder (visited on 07/13/2025).

[6] *irrte.ch JWT JS Decoder*. 2025. URL: https://irrte.ch/jwt-js-decode/index.html (visited on 07/13/2025).

[7] Louis Jannett, Maximilian Westers, Tobias Wich, Christian Mainka, Andreas Mayer, and Vladislav Mladenov. *SoK: SSO-MONITOR - The Current State and Future Research Directions in Single Sign-on Security Measurements*. 2024. DOI: 10.1109/EuroSP60621.2024.00018.

[8] Louis Jannett, Maximilian Westers, Tobias Wich, Christian Mainka, Andreas Mayer, and Vladislav Mladenov. *SoK: SSO-Monitor - The Current State and Future Research Directions in Single Sign-On Security Measurements*. 2024. DOI: TBD.

[9] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Signature (JWS)*. RFC 7515. http://www.rfc-editor.org/rfc/rfc7515.txt. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7515.txt.

[10] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. http://www.rfc-editor.org/rfc/rfc7519.txt. RFC Editor, May 2015. URL: http://www.rfc-editor.org/rfc/rfc7519.txt.

[11] M. Jones and J. Hildebrand. *JSON Web Encryption (JWE)*. RFC 7516. RFC Editor, May 2015.

[12] Michael B. Jones. *JSON Web Algorithms (JWA)*. RFC 7518. May 2015. DOI: 10.17487/RFC7518. URL: https://www.rfc-editor.org/info/rfc7518.

[13] *JOSEPH - Burp Suite Extension for JWT Security Testing*. 2025. URL: https://github.com/RUB-NDS/JOSEPH (visited on 07/13/2025).

[14] *JWT Editor - Burp Suite Extension by PortSwigger*. 2025. URL: https://github.com/PortSwigger/jwt-editor (visited on 07/13/2025).

[15] *jwt-hack - JWT Security Testing Tool*. 2025. URL: https://github.com/hahwul/jwt-hack (visited on 07/13/2025).

[16] *JWT.IO - JSON Web Tokens*. 2025. URL: https://jwt.io/ (visited on 07/13/2025).

[17] *jwt.ms - Microsoft JWT Decoder*. 2025. URL: https://jwt.ms/ (visited on 07/13/2025).

[18] *jwt_tool - JWT Toolkit by ticarpi*. 2025. URL: https://github.com/ticarpi/jwt_tool (visited on 07/13/2025).

[19] B. Kim and K. Natnael. *Evaluating the Security of RFC 8725: An Analysis of JWT Best Practice in JWS*. Bachelor's Thesis. 2025. URL: https://www.diva-portal.org/smash/get/diva2:1951665/FULLTEXT01.pdf.

[20] Tim McLean. *Critical vulnerabilities in JSON Web Token libraries*. Blog post. Mar. 2015. URL: https://www.chosenplaintext.ca/2015/03/31/jwt-algorithm-confusion.html (visited on 07/23/2025).

[21] OpenAPI Initiative / Linux Foundation. *OpenAPI Specification – formerly Swagger Specification*. https://swagger.io/specification/. Version 3.1.1. 2025. (Visited on 07/23/2025).

[22] OpenID Foundation. *OpenID Connect Core 1.0 - ID Token*. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html#IDToken (visited on 07/23/2025).

[23] OWASP Foundation. *Path Traversal*. 2023. URL: https://owasp.org/www-community/attacks/Path_Traversal (visited on 07/16/2025).

[24] Thomas Pornin. *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*. RFC 6979, https://datatracker.ietf.org/doc/html/rfc6979. Section 3.4. 2013.

[25] PortSwigger. *Detecting blind OS-command injection using time delays*. 2024. URL: https://portswigger.net/web-security/os-command-injection#detecting-blind-os-command-injection-using-time-delays (visited on 07/17/2025).

[26] PortSwigger. *Injecting self-signed JWTs via the `kid` parameter*. 2024. URL: https://portswigger.net/web-security/jwt#injecting-self-signed-jwts-via-the-kid-parameter (visited on 07/16/2025).

[27] PortSwigger. *OS command injection*. 2024. URL: https://portswigger.net/web-security/os-command-injection#ways-of-injecting-os-commands (visited on 07/16/2025).

[28] PortSwigger. *SQL injection UNION attacks*. 2024. URL: https://portswigger.net/web-security/sql-injection/union-attacks (visited on 07/16/2025).

[29] PortSwigger Web Security. *JWT attacks – Web Security Academy*. 2025. (Visited on 07/23/2025).

[30] Ruhr-Universität Bochum / eHacking Collection. *eHacking Collection – Vulnerable SAML Service Provider und weitere Challenges*. 2025. URL: https://sp-elearning.cloud.nds.rub.de/ (visited on 07/17/2025).

[31] *Token Dev JWT Debugger*. 2025. URL: https://token.dev/ (visited on 07/13/2025).

[32] Wallarm, Inc. *Common JWT Secrets List*. https://github.com/wallarm/jwt-secrets/blob/master/jwt.secrets.list. 2025. (Visited on 07/23/2025).

[33] Maximilian Westers, Tobias Wich, Louis Jannett, Vladislav Mladenov, Christian Mainka, and Andreas Mayer. *SSO-Monitor: Fully-Automatic Large-Scale Landscape, Security, and Privacy Analyses of Single Sign-On in the Wild*. 2023. arXiv: 2302.01024 [cs.CR]. URL: https://arxiv.org/abs/2302.01024.