ADALOGICS

Keycloak Fuzzing Audit

In collaboration with the Keycloak maintainers and the Cloud Native Computing Foundation (CNCF)

"Arthur" Sheung Chi Chan, Adam Korczynski, David Korczynski

8th November 2024

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Contents

About Ada Logics **1**

Executive summary **3**

Engagement Process and Methodology **4**

Grouping of Keycloak Codebase 5

Group 1 5

Group 2 6

Group 3 7

Group 4 8

Keycloak fuzzers **11**

Fuzzing architecture 11

Keycloak fuzzers 13

Use of Seeds for Certain Fuzzers 16

Fuzzers using SAML-type seed 16

Fuzzers using JSON-type seed 16

Issues found **17**

Possible NullPointerException in core/JWKParser 17

Advice following engagement **19**

Conclusion **20**

Executive summary

This report details the engagement whereby Ada Logics integrated continuous fuzzing into the Keycloak project. At the time we started this work, Keycloak was doing no fuzz testing. The engagement, therefore, included both setting up a fuzzing infrastructure and writing fuzz test harnesses. We did this by integrating Keycloak into OSS-Fuzz - an open-source and free service run and offered by Google for critical open-source projects. OSS-Fuzz runs the fuzzers of its integrated projects continuously and reports found crashes to the maintainers. After we integrated Keycloak into OSS-Fuzz, we started writing the fuzz tests. We added the fuzzers to Keycloaks OSS-Fuzz integration as we wrote the fuzzers so we could get feedback from OSS-Fuzz and improve the fuzzers over time so they were in a state to run continuously. We monitored the coverage reports on OSS-Fuzz and used them to make iterative improvements to the fuzz coverage.

In summary, during the engagement, we:

- Integrated Keycloak into OSS-Fuzz
- Wrote 24 new fuzzers for the Keycloak project
- Found and reported 1 issue and submitted a patch for the issue found
- Monitored the coverage changes and fixed build problems when Keycloak made upstream API changes

Engagement Process and Methodology

The engagement had two goals: First, we wanted to build a continuous fuzzing integration around OSS-Fuzz, and second, we wanted to achieve as much coverage gain as we could within the allocated time. In this section, we discuss how we approached this and the challenges involved in fuzzing Keycloak.

The primary objective was to integrate continuous fuzzing into Keycloak using OSS-Fuzz, an open-source service that automates fuzzing executions, bug triaging, and notifications. We implemented fuzzers using the Jazzer engine, ensuring they run continuously against the latest Keycloak main branch.

Fuzzing Keycloak comes with some inherent challenges. For example, many parts of Keycloak interact with remote services such as identity providers, databases and LDAP servers. To test these parts by way of fuzzing, Keycloak requires mocking environments for all calls that the fuzzer must define at runtime. This required extensive engineering efforts in some cases, but the benefits from such fuzzers can be high if successfully implemented. In most cases, when testing these parts of Keycloak, the fuzzer will trigger code that:

1. sends a request to a remote service
2. receives a response from the remote service
3. processes the response from the remote service

As such, there is data from the remote service flowing into Keycloak that Keycloak then processes. To test this, the fuzzer mocks the response from the remote service and uses the fuzzers test case to generate the data Keycloak receives from the remote service.

While fuzzing the Keycloak project has challenges, there are also code paths that are highly suited for fuzz testing. For example, Keycloak implements a set of parsers, some of which parse data that can come from untrusted sources. In addition, some of these parse document types that are prone to security issues in Java applications. For example, XML parsers are highly prone stackoverflow bugs that can be the root cause of denial-of-service vulnerabilities.

During the engagement, we targeted both hard-to-fuzz and the well-suited APIs in Keycloak for several reasons. First and foremost, we wanted to explore the difficulty of going beyond the well-suited targets to document how the Keycloak community could expand the fuzzing coverage once our engagement finished. Second, which is an extension of the first reason, we wanted to explore the gains from targeting code parts that require extensive mocking or other significant engineering efforts. We wanted to explore this in terms of both found crashes and coverage gains. Third, we wanted to end up with a practical implementation that demonstrates the work and outcome of targeting hard-to-fuzz code paths. This aims to provide the Keycloak community with a reference on how we would target hard-to-fuzz code parts - even if the gains are lower than significant.

We categorized the Keycloak codebase into four groups based on functionality to identify the most suitable targets for fuzzing which we enumerate below. Our primary focus was on Group 4, which includes the core logic and APIs that handle complex and untrusted data, making them the best candidates for discovering vulnerabilities by way of fuzz testing. We worked on the fuzzers in public in the CNCF-Fuzzing repository from which OSS-Fuzz fetches the fuzzers at compile time in a continuous manner.

Monitoring OSS-Fuzz's performance was essential to ensure comprehensive testing and adapt to changes. We actively addressed issues such as build errors and false positives, submitting fixes upstream to improve Keycloak. This ongoing process maintains the effectiveness of the fuzzing integration when APIs change in the upstream Keycloak repository.

Grouping of Keycloak Codebase

Keycloak's codebase, consisting of over 500,000 lines, includes core logic, API handling, and client and server management, integrating with third-party authentication services through standard protocols like OAuth2 and OpenID Connect. Given its complexity, we divided the code into four groups to identify the best fuzzing targets.

Our efforts concentrated on Group 4, which contains critical components such as core logic and APIs. These handle complex and dynamic data, making them prime targets for fuzzing. Groups 1, 2, and 3 were excluded as they handle minimal processing of untrusted inputs, rely heavily on external libraries, or are constrained by strict protocols, limiting fuzzing effectiveness. Below, we provide an overview of each group and highlight examples of methods relevant to fuzzing or explain why they were excluded.

Group 1

Group 1 includes extensions, adapters, or wrappers for third-party libraries, such as adapter classes that integrate Keycloak with application platforms like WildFly, Tomcat, and others. These classes often contain minimal processing logic, as the main functionality comes from the underlying libraries, platforms, or protocols. Because of their limited internal logic and reliance on external components, we excluded these modules from the Keycloak fuzzing project and did not prioritise them for fuzzing.

Adapters This module contains libraries and components that integrate Keycloak with various application frameworks and platforms. They handle authentication and authorization for applications by interacting with the Keycloak server, supporting technologies like Java EE, Spring Boot, and others.

Dependencies This module manages external dependencies required by Keycloak. This includes third-party libraries and modules that Keycloak relies on for its functionality, ensuring that the correct versions of these dependencies are used.

Util This is a utility module that provides common helper functions and classes used throughout Keycloak on external LDAP services.

Rest This module contains the REST API endpoints that allow external systems to interact with Keycloak. This module is crucial for enabling programmatic access to Keycloak's features, such as user management and token services.

Quarkus This module provides support for running Keycloak on the Quarkus framework, which is a Kubernetes-native Java stack. This module helps optimize Keycloak for cloud environments, improving startup times and resource usage.

Below we exemplify why this group of modules, specifically the `Adapters` module, is not well-suited for fuzzing.

The `readSecureDeployment(XMLExtendedStreamReader, List<ModelNode>)` method in the `KeycloakSubsystemParser` class reads an element from an XML stream and creates a corresponding `ModelNode` configuration. This method relies heavily on the `XMLExtendedStreamReader` library to parse the XML input and on the `ModelNode` utility from the JBoss WildFly core libraries to create configuration nodes. The core operations, such as reading XML content, validation, error handling, and extracting attributes and elements, are largely managed by these third-party libraries. The method itself contains minimal internal logic beyond orchestrating calls to these libraries, making it a poor candidate for fuzzing. Fuzzing is more effective for methods with complex input processing and transformations, which is not the case here. The code can be found in this GitHub link.

Similarly, the `writeSps(XMLExtendedStreamWriter, ModelNode)` method in the same class is designed to write an elements from a `ModelNode` to an XML stream using the `XMLExtendedStreamWriter` library. It takes a `ModelNode` containing the configuration data and uses `XMLExtendedStreamWriter` to serialize it to XML. Like the first example, most of the work in this method is done by the third-party XML processing library. The method mainly involves simple iteration over the properties of the ModelNode and delegates the actual serialization to XMLExtendedStreamWriter. Due to its lack of complex data handling or transformation logic, fuzzing this method would not be effective for discovering vulnerabilities, as it relies on well-established third-party libraries for its core functionality. The code can be found in this GitHub link.

Group 2

Group 2 consists of classes that handle remote services, such as processing HTTP requests and responses. While these might seem like good candidates for fuzzing, their effectiveness is limited due to constraints in current fuzzing environments like OSS-Fuzz, which often block remote execution and request/response handling. Even using mock server responses does not yield meaningful results

for fuzzing these methods. Additionally, since Keycloak focuses on identity and access management, the code in this group is less critical and has low fuzzing effectiveness, making it a lower priority for fuzzing.

Authz This module deals with the authorization services provided by Keycloak, including policy enforcement and fine-grained access control. It supports OAuth2-based authorization for resources and scopes, allowing the definition of complex access policies.

Federation This module supports the integration of Keycloak with external user storage systems like LDAP, Active Directory, or custom databases. This module allows Keycloak to federate users from these external sources and manage them within Keycloak.

Integration This module supports the integration of Keycloak with various external systems and services. This can include integration with external identity providers, social login services, or other third-party systems.

Here are some examples to illustrate why this group of modules is not ideal for fuzzing.

The `authorize(AuthorizationRequest)` method from the `AuthorizationResource` class sends an HTTP POST request to Keycloak's token endpoint to request permissions based on the provided AuthorizationRequest. Since the actual authorization is handled externally by the Keycloak server, the method lacks complex internal logic, focusing mainly on managing network requests and responses. The code can be found in this GitHub link.

Similarly, the `introspectRequestingPartyToken(String)` method from the `ProtectionResource` class sends an HTTP POST request to Keycloak's token introspection endpoint to validate a provided token. The validation is done externally on the server, meaning the method itself involves minimal internal processing and does not handle complex, random input. The code can be found in this GitHub link.

These methods are not suitable for fuzzing because they depend on network interactions with external servers, which are restricted by OSS-Fuzz, even though they manage some sensitive authentication data.

Group 3

Group 3 comprises modules not written in Java or unrelated to Keycloak's core functionalities. These modules include deployment scripts, documentation, or code in other languages, such as JavaScript and modules related to themes, styles, and other non-essential components. These modules do not fall within the scope of our Java-based fuzzing efforts, which focus on the primary Java logic that drives Keycloak's core operations. Furthermore, most modules in this group do not process random input in a

way that would make them effective fuzzing targets, making them unsuitable for testing with random data.

BOMs This module provides a consistent set of dependencies that are version-aligned for Keycloak modules and projects. BOM files are used in Maven to manage dependency versions in a structured manner across different modules.

Docs This module contains the documentation for Keycloak, including user guides, developer documentation, and API references. This module is essential for helping users and developers understand how to use and extend Keycloak.

Themes This module customises Keycloak's user interfaces, including login pages, account management, and email templates. This module allows developers to create and apply custom themes to modify the look and feel of Keycloak's frontend.

Misc This is a catch-all module that includes miscellaneous components and utilities that do not fit neatly into other specific modules. This module may contain experimental features or less commonly used utilities.

JS This module contains the JavaScript adapters and libraries that integrate Keycloak with JavaScript-based applications, particularly single-page applications (SPAs). This module is essential for enabling secure authentication in frontend apps.

Testsuite This module contains a comprehensive suite of tests that ensures the reliability, stability, and performance of Keycloak's features. This module includes unit tests, integration tests, and performance benchmarks.

Test-POC This module is used to test proof-of-concept implementations within Keycloak. It is typically used to experiment with new features or approaches before they are integrated into the main codebase.

Distribution This module Handles the packaging and distribution of Keycloak, including creating distributions that can be deployed on various platforms. This module is responsible for generating the final deployable artifacts.

Operator This module provides the Keycloak Operator, which facilitates the deployment and management of Keycloak on Kubernetes and OpenShift clusters. This module automates provisioning, scaling, and upgrading Keycloak instances.

Group 4

Group 4 covers the modules that manage the core logic and APIs of the Keycloak framework. These modules are the most promising fuzzing targets because they are central to Keycloak's primary functionalities. This group includes APIs for Keycloak services and components that process data from

third-party or external sources, which are often untrusted. Some of these modules handle parsing and processing of untrusted data, potentially exposing them to exploitation. The security and reliability of Keycloak rely heavily on the robustness of these modules, making them the most suitable candidates for fuzzing. Thus, we identified Group 4 as the primary focus for fuzzing to ensure the highest level of security and reliability for the Keycloak project.

Common This module contains shared utilities and components across different Keycloak modules. This includes basic classes, constants, and utility methods that are reused throughout the Keycloak project.

Core This is the foundational module of Keycloak, which includes core functionalities such as realm management, user management, and role-based access control. It is central to Keycloak's identity and access management capabilities.

Crypto This module handles cryptographic operations within Keycloak, including encryption, decryption, digital signatures, and key management. This module ensures the security of tokens, credentials, and other sensitive data, making it one of the most critical modules in the Keycloak project.

Server-SPI / Server-SPI-Private This module contains the Keycloak server SPI and allows developers to extend Keycloak's functionalities by implementing custom providers. The Server-SPI module is public and intended for general extensions, while Server-SPI-Private contains internal APIs that are not meant for public use but are essential for the server's internal operations.

SAML-Core-API / SAML-Core This module manages the support for SAML in Keycloak, including the core API and functionality for handling SAML requests and responses. This module is used to integrate Keycloak with SAML-based identity providers and to handle responses from providers.

Services This module provides the backend services that support Keycloak's core features, such as authentication, authorization, session management, and event logging. It is a critical module that handles the actual logic and operations of Keycloak.

Model This module manages the data model used by Keycloak, including the persistence of realms, users, roles, and other entities. It is responsible for the interaction with the underlying database or storage systems.

Keycloak's large codebase has only the Group 4 modules suitable for fuzzing, representing just 5-7% of the total. Despite this small percentage, these modules are among the most critical, involving token parsing, data serialization/deserialization, and cryptographic functions essential for system security, making thorough fuzzing crucial.

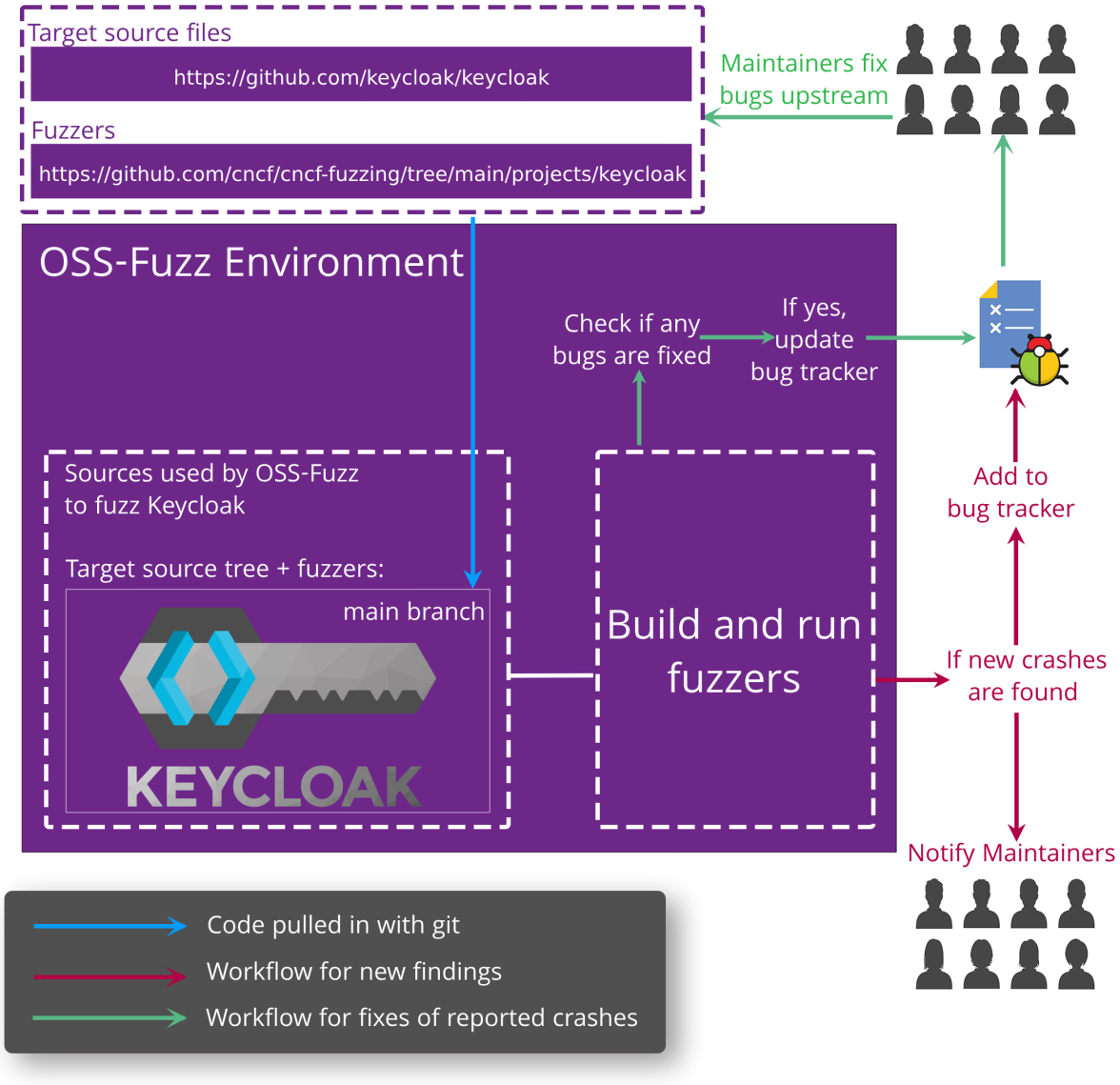
Keycloak is divided into modules handling identity and access management, from core functions like authentication and authorization to integration with external systems and customization. Supporting modules enhance robustness, documentation, and deployment, while others provide shared utilities. We focus on fuzzing Group 4 because these modules are fundamental to Keycloak's security and

reliability. Modules integrating with external systems, such as Adapters, are less suitable for OSS-Fuzz due to blocked external requests.

Keycloak fuzzers

Fuzzing architecture

In this section we present details on the Keycloak fuzzing set up. We first present a high-level view of the overall fuzzing architecture and how it supports running the fuzzers continuously. We then enumerate the fuzzers we wrote during the audit, and finally we go into detail with the crashes that the fuzzers found.



A central component in the Keycloaks fuzzing infrastructure is its integration into OSS-Fuzz. The Keycloak source code and Keycloaks fuzz tests are the two core elements that OSS-Fuzz uses to fuzz Keycloak. The following diagram gives an overview of how OSS-Fuzz uses these two elements and

what happens when an issue is found/fixed.

The current OSS-Fuzz set up builds the fuzzers by cloning the upstream Keycloak GitHub repository to get the latest Keycloak source code and its fuzz tests. OSS-Fuzz then builds the fuzzers against the cloned Keycloak source code which ensures that the fuzzers always run against the latest Keycloak commit.

This build cycle happens daily and OSS-Fuzz will verify if any existing bugs have been fixed. If OSS-fuzz finds that any bugs have been fixed OSS-Fuzz marks the crashes as fixed in the OSS-Fuzz bug tracker and notifies maintainers.

In each fuzzing iteration, OSS-Fuzz uses its corpus accumulated from previous fuzz runs. If OSS-Fuzz detects any crashes when running the fuzzers, OSS-Fuzz performs the following actions:

1. A detailed crash report is created.
2. An issue in the OSS-Fuzz bug tracker is created.
3. An email is sent to maintainers with links to the report and relevant entry in the bug tracker. OSS-Fuzz has a 90 day disclosure policy, meaning that a bug becomes public in the bug tracker if it has not been fixed. The detailed report is never made public to non-maintainers. As such, the world will know that there was an issue, but it will not have access to data such as stacktrace and reproducer.
4. The Keycloak maintainers will fix issues upstream, and OSS-Fuzz will pull the latest Keycloak `main` branch the next time it performs a fuzz run and verify that a given issue has been fixed.

OSS-Fuzz projects have the option to mark found crashes false positives with the “WontFix” status label.

Keycloak fuzzers

In this part we enumerate the fuzzers we wrote during the engagement. All fuzzers live in the CNCF-Fuzzing repository.

Fuzzer	Description
AuthenticatorFuzzer	At a high level, this fuzzer does three things. First, it chooses an authenticator to test. Second, it creates a context to authenticate from the data that the engine provides to the fuzzer. As such, the context is pseudo-randomized. Third and finally, it invokes the authenticators `authenticate` method with the pseudo-randomized context. As such, we pass the fuzzers pseudo-random data to different authenticators' `authenticate` methods to test for crashes. Authenticators are exposed to untrusted input and are therefore important fuzz targets where they can be fuzz tested. For example, authenticators receive data like usernames, passwords and one-time-passwords before they are authenticated. As such, if input can compromise authenticators, the fuzzer is likely to find security-related issues.
AuthzClientFuzzer	This fuzzer sets up a mock server in such a way that the fuzzer controls the response payload from the server. It then creates an `AuthzClient` that communicates with the mock server. The fuzzer chooses which of the `AuthzClient`'s methods it should in invoke in each iteration. As such, the fuzzer tests if a client can be compromised by way of the servers response.
CertUtilsFuzzer	This fuzzer tests different utility helpers related to certificate creation and processing. It tests the `BCCertificateUtilsProvider`, `ElytronCertificateUtilsProvider` and `BCFIPSCertificateUtilsProvider` providers.
ClientSigVerifierFuzzer	This fuzzer chooses a client signature verifier provider and verifies pseudo-random bytes. It creates the verifier by way of the corresponding `Factory` with a mock session and client that are set up in such a way that the fuzzer can control the response that the client receives.

Fuzzer	Description
CommonCryptoUtilsFuzzer	This fuzzer tests a series of utility methods related to key and certificate processing. In each iteration, the fuzzer chooses which method to test.
CommonUtilsFuzzer	Tests a list utility methods mostly related to encoding. The fuzzer chooses which method to test in each iteration and invokes it with the fuzzers testcase.
CredentialValidatorFuzzer	This fuzzer creates pseudo-random credentials and validates them by way of either `OTPCredentialProvider`, `PasswordCredentialProvider`, `RecoveryAuthnCodesCredentialProvider` or `WebAuthnCredentialProvider`. The fuzzer chooses a single provider in each iteration and invokes that providers `isValid()` method with pseudo-randomized realm model, user model and credential input.
DefaultAuthFlowsFuzzer	Pseudo-randomly invokes different methods of `DefaultAuthenticationFlows` and passes a pseudo-randomized `RealmModel`.
JoseParserFuzzer	Invokes Keycloaks `JOSEParser` with a raw string provided by the fuzzer
JweAlgorithmProviderFuzzer	This fuzzer chooses one of Keycloaks algorithm providers, an encryption algorithm to create an `JWEHeader`. It then either creates a `JWEKeyStorage` and invokes `encodeCek` or invokes `decodeCek`.
JweFuzzer	This fuzzer chooses an algorithm for an JWE object, an encryption algorithm and then creates an entirely pseudo-randomized `JWE` object or a `JWE` object with a pseudo-randomized header. The fuzzer then tests different encoding, decoding, serialization and deserialization methods of the `AesCbcHmacShaJWEEncryptionProvider` and `AesGcmJWEEncryptionProvider`.
JwkParserFuzzer	Tests Keycloaks `JwkParserFuzzer` with a pseudo-random string.

Fuzzer	Description
JwksUtilsFuzzer	Tests Keycloaks `JWK`'s `setOtherClaims` method by calling it with a string provided by the fuzzer and then subsequently adding it to a `JSONWebKeySet` and retrieving the key from the keyset.
KeycloakModelUtilsFuzzer	Tests the static methods in the `KeycloakModelUtils` class.
KeycloakUriBuilderFuzzer	Creates a `KeycloakUriBuilder` from a string provided by the fuzzer, configures it in pseudo-random manner with a call to either `schemeSpecificPart`, `userInfo`, `host`, `replaceMatrix`, `replaceQuery` or `fragment` and finally builds it.
PemUtilsProviderFuzzer	Tests utility methods of `BCPemUtilsProvider`, `ElytronPEMUtilsProvider` and `BCFIPSPemUtilsProvider` related to decoding encoding.
SamlParserFuzzer	Keycloak implements various parsing for parsing SAML objects. This fuzzer chooses one of these parsing routines in each iteration and tests it.
SamlProcessingUtilFuzzer	Tests various utility methods of the `JAXBUtil` class.
SamlValidationUtilFuzzer	Tests the `validate` and `checkSchemaValidation` methods of the `JAXPValidationUtil` class and the `validateRedirectBindingSignature` method of the `RedirectBindingSignatureUtil` class.
SamlXmlUtilFuzzer	This fuzzer targets methods in `XMLEncryptionUtil` and `XmlSignatureUtil` classes of the `org.keycloak.saml.processing.core.util` package. It passes random data to fuzz all their static utility methods.
ServicesJwsFuzzer	Creates `DefaultTokenManager` and tests one of its methods. The fuzzer requires substantial mocks which the fuzzer controls and returns data based generated by way of the fuzzers testcase.
ServicesUtilsFuzzer	This fuzzer is a collection of miscellaneous utility methods across different utility classes.
TokenUtilFuzzer	Tests various utility methods for token processing.
TokenVerifierFuzzer	Tests `org.keycloak.TokenVerifier.verify()`.

Use of Seeds for Certain Fuzzers

A seed is an initial set of files that the fuzzer should use as a starting to mutate over and enhances the fuzzing process. In complex systems like Keycloak, which involve intricate data structures and specific API requirements, seeds help fuzzers produce inputs that closely mimic real-world scenarios. This approach improves fuzzing accuracy, uncovers otherwise hidden vulnerabilities, and ensures consistent and reproducible results, ultimately strengthening Keycloak's security and reliability.

We integrated seeds into specific fuzzers within the Keycloak OSS-Fuzz setup to generate two types of inputs: SAML format, requiring a specific XML structure, and JSON format, including JWK, which follows a JSON structure. Precise syntax in these formats helps avoid errors that could lead to false positives. The following lists outline the fuzzers that use SAML-type and JSON-type seeds, enabling them to generate effective inputs, minimize syntax errors, and explore more logical pathways.

Fuzzers using SAML-type seed

1. SamlParserFuzzer
2. SamlProcessingUtilFuzzer
3. SamlValidationUtilFuzzer
4. SamlXmlUtilFuzzer

Fuzzers using JSON-type seed

1. JwkParserFuzzer
2. JoseParserFuzzer

Issues found

In this part of the report, we present the single low-severity issue discovered through the fuzzing process, which the Ada Logics team has already addressed by pushing a fix upstream.

#	ID	Title	Severity	Fixed
1	ADA-KEYCLOAK-2024-1	Possible NullPointerException in core/JWKParser	Low	Yes

Possible NullPointerException in core/JWKParser

Severity	Low
Status	Fixed
id	ADA-KEYCLOAK-2024-1
Component	JWKParser

In `JWKParser`, the logic depends on the x and y field in the otherclaim map stored within the JWK object to generate 2 big integers for base EC points.

Direct source link:

https://github.com/keycloak/keycloak/blob/a8db79a68cf2ae9d680f6a6bc70e9b408c2e405b/core/src/main/java/org/keycloak/jose/jwk/JWKParser.java#L83-L84

```
81      private PublicKey createECPublicKey() {
82          String crv = (String) jwk.getOtherClaims().get(ECPublicJWK.CRV)
                ;
83          BigInteger x = new BigInteger(1, Base64Url.decode((String) jwk.
                getOtherClaims().get(ECPublicJWK.X)));
84          BigInteger y = new BigInteger(1, Base64Url.decode((String) jwk.
                getOtherClaims().get(ECPublicJWK.Y)));
85
86          String name;
87          switch (crv) {
```

But according to the `ec()` method in `JWKBuilder`, it does not explicitly put any value to the otherclaim map and thus the map will remain the default empty map in some cases. It causes the retrieval of x and y fields to return a null value and subsequently make Base64.decode throws a `NullPointerException` when the null value is parsed.

Direct source link:

https://github.com/keycloak/keycloak/blob/a8db79a68cf2ae9d680f6a6bc70e9b408c2e405b/core/src/main/java/org/keycloak/jose/jwk/JWKBuilder.java#L118

```
114     public JWK ec(Key key) {
115         ECPublicKey ecKey = (ECPublicKey) key;
116 
117         ECPublicJWK k = new ECPublicJWK();
118 
119         String kid = this.kid != null ? this.kid : KeyUtils.createKeyId
                (key);
120         int fieldSize = ecKey.getParams().getCurve().getField().
                getFieldSize();
```

Mitigation

Add conditional checking to ensure the otherclaim map is intialised and configured and is not empty.

Possible effect

NullPointerException is being thrown for empty otherclaim map.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=61287

Upstream fix

https://github.com/keycloak/keycloak/issues/22375

Code behaviour after the fix

No more exceptions are thrown with those invalid inputs.

Advice following engagement

Fuzzing Coverage and Limitations Although we have successfully fuzzed several critical components of Keycloak, some of its extensive codebase remains unfuzzed. Expanding coverage to additional, less critical areas would enhance integration, but we face two main limitations: OSS-Fuzz blocks network requests for Java projects, preventing fuzzing of network-related services, and Keycloak's tendency to wrap exceptions as `RuntimeException` or general `Exception` hampers effective issue detection. To mitigate these issues, we should refine both fuzzers and the Keycloak codebase to avoid generic exception handling, allowing better utilization of Java's exception mechanisms.

Frequent Updates to Project API and Structure Keycloak's frequent API and structural updates often cause fuzzer build failures. Continuous monitoring and prompt adjustments are essential to maintaining the effectiveness of the fuzzing process amid ongoing changes.

Fuzzer Maintenance Strategy Currently, the fuzzers are hosted in the CNCF-Fuzzing repository, and moving them upstream to the Keycloak repository would allow earlier detection of build failures caused by API changes, integrating the fuzzers more closely with the project's development cycle and enhancing overall stability.

Expanding Fuzzing Coverage Many modules remain uncovered due to complex object parameter requirements that are difficult to fuzz. A thorough assessment of methods to generate random, complex objects could significantly improve our fuzzing efforts. Adopting tools like OSS-Fuzz-Gen, which automatically generates new fuzzers, could help expand coverage and reveal additional vulnerabilities.

Conclusion

In this engagement, we at Ada Logics developed an extensive fuzzing suite for Keycloak, focusing on a targeted portion of the codebase that we identified as suitable for fuzzing. We integrated this fuzzing suite into the OSS-Fuzz service, ensuring that all fuzzers run continuously under OSS-Fuzz's management. We developed a total of 34 fuzzers, which led to the discovery and resolution of one bug. We also actively monitor and update the build scripts and fuzzers to keep up with changes in the Keycloak project's structure and APIs.

The Cloud Native Computing Foundation (CNCF) commissioned this work.