Security Testing of RESTful APIs with Test Case Mutation

Sébastien Salva and Jarod Sue
LIMOS - UMR CNRS 6158, Clermont Auvergne University, UCA, France

Abstract: The focus of this paper is on automating the security testing of RESTful APIs. The testing stage of this specific kind of components is often performed manually, and this is yet considered as a long and difficult activity. This paper proposes an automated approach to help developers generate test cases for experimenting with each service in isolation. This approach is based upon the notion of test case mutation, which automatically generates new test cases from an original test case set. Test case mutation operators perform slight test case modifications to mimic possible failures or to test the component under test with new interactions. In this paper, we examine test case mutation operators for RESTful APIs and define 18 operators specialised in security testing. Then, we present our test case mutation algorithm. We evaluate its effectiveness and performance on four web service compositions.

1 INTRODUCTION

One of the key motivations for software security is the prevention of attackers exploiting software flaws, which can lead to compromising application security or revealing user data. Despite the continuous growth of the security testing market, there is still an inadequate emphasis on this activity, exposing organisations and end users to unforeseen risks when using vulnerable systems or software. One aspect that may account for this observation, is that selecting security solutions and crafting specific security test cases are two tasks of the software life cycle that demand time, expertise, and experience. Developers often lack the guidance, resources, or skills on how to design, implement secure applications, and test them.

A way to help developers in security testing is the use of test automation, which addresses challenges related to time constraints, complexity, and coverage. In this scope, model based testing (Li et al., 2018) offers the advantage of automating the test case generation. But models are often manually written, and this task is considered as long, difficult and error-prone, even for experts. Numerous others approaches have been proposed to generate test cases without specification, for example by using random testing (Arcuri et al., 2011), graphical user interfaces exploration (Salva and Zafimiharisoa, 2014; Ferreira and Paiva, 2019) or automated penetration testing (Abu-Dabaseh and Alshammari, 2018). Despite their significant benefits, a recurring limitation observed in employing these approaches is the insufficient understanding of the application business logic and context. As a result, they may fail to identify certain security vulnerabilities that require a deeper understanding of how the application behaves.

Focusing on this background, we propose an intermediate solution based upon the notion of test case mutation. Unlike mutation testing that aims at evaluating the effectiveness of an existing test case set by introducing intentional errors into the original source code of an application under test (Papadakis et al., 2019), test case mutation automatically generates new test cases from an original test case set. As the original test cases should encode some knowledge about the application under test, the mutated test cases should deeper cover the application behaviours and features and hence should detect further defects. A test case mutation operator performs slight test case modifications to mimic possible failures or to experiment the system under test with new interactions. Some test case mutation based approaches have been proposed for detecting bugs or crashes (Xuan et al., 2015; Xu et al., 2010; Arcuri, 2018; Arcuri, 2019; Köröglu and Sen, 2018; Paiva et al., 2020). None of them deals with security testing.

In this paper, we propose a new approach, specifically designed for testing the security of RESTful APIs in isolation. This firstly implies that we propose new specific mutation operators devoted to de-
detecting vulnerabilities. This also means that our approach generates new executable test cases but also mock components. We recall that a mock component aims at simulating an existing component, while behaving in a predefined and controlled way to make testing more effective and efficient. Mocks are often used by developers to make test development easier or to increase test coverage. They may indeed be used to simplify the dependencies that make testing difficult (e.g., infrastructure or environment related dependencies). Besides, mocks are used to increase test efficiency by replacing slow-to-access components. In summary, the main contributions of this paper include:

1. a study on mutation operators specialised in the security testing of RESTful APIs, including the definition of 18 operators,
2. an algorithm for the generation of mutated test cases along with test scripts and mock components,
3. the implementation of the approach, along with 4 RESTful API compositions and Log files publicly available in (Sue and Salva, 2024).

The paper is organised as follows: We study and propose test case mutation operators for RESTful APIs in Section 2. Our test case mutation algorithm is presented in Section 3. Section 4 summarises our contributions and draws some perspectives for future work.

2 TEST CASE MUTATION OPERATORS FOR RESTful APIs

This paper focuses on test case mutation operators designed to detect vulnerabilities in RESTful APIs. This testing context introduces specific requirements and a testing architecture, both of which are subsequently presented. From this architecture, we present how test cases are modelled with Input Output Transition Systems (IOTSs) and provide an illustrative example. Then, we study the mutation operators that can be defined within this scope.

2.1 Assumptions

We consider the test architecture depicted in Figure 1, whose attributes are expressed with the following realistic assumptions:

• **Black Box Testing:** we employ a black box perspective, enabling to interact with a RESTful API,

![Figure 1: Black-box test architecture for experimenting RESTful API in isolation.](image)

denoted SUT, only with HTTP requests or responses. We call them (communication) events;

• **Event Content:** observers are able to get all the events related to a RESTful API under test along with their contents (no encryption). In particular, events include parameter assignments allowing to identify the source and the destination of each event. Besides, an event can be identified either as a request or a response;

• **Test in Isolation:** we consider conducting tests in an isolated environment. If the RESTful API is dependent to other services, the later shall be replaced by mock components. We do not assume that those mock components exist, our approach builds them.

2.2 IOTS Test Case Definition

Given the test architecture of Figure 1, we consider that events have the form \( e(\alpha) \) with \( e \) some label, e.g., a path or a status; "*" is a special notation representing any label. \( \alpha \) is an assignment of parameters in \( P \) to a value in the set of values \( V \). These parameters allow the encoding of some specific web service characteristics e.g., if an event is a request, the receiver and sender of this request, etc. We write \( x := * \) the assignment of the parameter \( x \) with an arbitrary element of \( V \), which is not of interest. \( E \) denotes the event set. We also use these additional notations on an event \( e(\alpha) \) to make our algorithm more readable: from\( (e(\alpha)) \) (resp. to\( (e(\alpha)) \)) denotes the source (resp. the destination) of the event. isreq\( (e(\alpha)) \), isresp\( (e(\alpha)) \) are boolean expressions expressing the nature of the event. body\( (e(\alpha)) \), header\( (e(\alpha)) \), status\( (e(\alpha)) \) are expressions returning values in \( \alpha \).

We model a test case with a deterministic IOTS having a tree form and whose terminal states express test verdicts, e.g., pass or inc, which stands for inconclusive. A test step corresponds to an IOTS transition \( q \xrightarrow{e(\alpha),l} q' \) with \( e(\alpha) \) some event and \( l \) a label set, which may be empty. Furthermore, we use the notation \( \emptyset \) labelled on transitions to represent the absence of reaction from a service under test (Phillips, 1987). Classically, we call a sequence of test steps...
a test sequence. The label set allows to easily express some knowledge about the event. For instance, "crash" is used when the HTTP status 500 is received. The special label "mock" identifies events performed by some other dependee services. Since we assume testing SUT in isolation, the dependee services will have to be replaced with mock components.

An IOTS test case has to met a few restrictions to avoid indeterministic behaviours while testing. To this end, a test case must allow at most one input event at any state. In reference to (Tretmans, 2008), this last restriction, we say that a test case is input restricted. Additionally, still in the context of isolation testing and to keep control of the testing process, a mock component should be deterministic and return at most one response after being invoked with the same event. We say that a test case has to be mock response restricted. This is formulated with:

Definition 1: A test case \( tc \) is a deterministic IOTS \( \langle Q, q_0, \Sigma \cup \{ \emptyset \}, L, \rightarrow \rangle \) where:

- \( Q \) is a finite set of states; \( q_0 \) is the initial state;
- \( \Sigma \subseteq \mathcal{E} \) is the finite set of events. \( \Sigma_I \subseteq \Sigma \) is the finite set of input events beginning with "?". \( \Sigma_O \subseteq \Sigma \) is the finite set of output events beginning with "!". with \( \Sigma_O \cap \Sigma_I = \emptyset \);
- \( L \) is a set of labels;

\[
\rightarrow \subseteq Q \times \Sigma \cup \{ \emptyset \} \times L^* \times \Sigma \quad \text{is a finite set of transitions. A transition} \quad (q, e(\alpha), l, q') \quad \text{is also denoted} \quad q \xrightarrow{e(\alpha), l} q';
\]

- \( Q_f = \{ \text{pass, fail, inc} \} \subseteq Q \) is the set of verdict states; if \( q \xrightarrow{e(\alpha), l} q_f \) with \( q_f \in Q_f \), then \( e(\alpha) \in \Sigma_O \cup \emptyset \);
- \( tc \) has no cycles except those in states of \( Q_f \);
- \( tc \) is input restricted i.e. \( \forall q \in Q : \text{event}(q) = \Sigma_O \cup \{ e(\alpha) \} \) for some \( e(\alpha) \in \Sigma_I \) or \( \text{event}(q) = \Sigma_O \cup \{ \emptyset \} \) with \( \text{event}(q) = \{ e(\alpha) \} \); \( e(\alpha) \in \Sigma_I \); \( \exists q_f' \in Q : q_f' \xrightarrow{e(\alpha)} q' \);
- \( tc \) is mock response restricted i.e. \( \forall q \in Q : |\{ e(\alpha) \xrightarrow{l} q' \mid \text{isResp}(e(\alpha)) \wedge \text{mock} \in l\} | \leq 1 \).

![Figure 2: IOTS Test Case example.](image)

An IOTS test case example is illustrated in Figure 2. It checks whether a RESTful API AccMan can be called with "checkAccountRisk". This service is dependent on another service called CheckRisk. The events related to CheckRisk are labelled by "mock" to express that a mock component has to be built to test AccMan in isolation.

IOTS test cases can be written manually, but this activity may be long and error-prone, especially for un-experimented developers. To solve this problem, we proposed in (Salva and Sue, 2023) an approach and tool for generating IOTS test cases from Log files. The approach also allows to recognise some specific behaviours (authentication, token generation, crash, ) and adds on test steps the following labels "login", "token", "token generation", "crash".

### 2.3 Mutation Operators for Security Testing

The literature does not cover the use of test case mutation operators specialised in assessing the security of web services. Consequently, we initially conducted a literature review to collect relevant data about the security testing of RESTful APIs. We searched for papers indexed in online sources (Scopus, Science Direct, IEEE Xplore, ACM Digital Library, Google Scholar). We identified relevant papers via keyword search by using the terms "web services security vulnerabilities attacks" and then, terms "microservice security vulnerabilities attacks". We found 42 and 35 works between 2006-2023. We isolated 24 papers and 3 surveys by using their abstracts and titles. We then crossed these results with the databases CAPEC (CAPEC, 2024) and CWE (CWE, 2024) of the MITRE organisation in order to classify attacks and avoid duplicates. With regard to our black box test architecture, we kept the attacks related to these domains:

- CAPEC-21: Exploitation of Trusted Identifiers
- CAPEC-22: Exploiting Trust in Client
- CAPEC-63: Cross-Site Scripting (XSS)
- CAPEC-151: Identity Spoofing
- CAPEC-153: Input Data Manipulation
- CAPEC-115: Authentication Bypass
- CAPEC-125: Flooding
- CAPEC-278: Service Protocol Manipulation
- CAPEC-594: Traffic Injection

At this step, we collected a total number of 36 attacks. We finally augmented this compilation, by incorporating 7 recommendations provided in the ENISA good practice guide (Skouloudi et al., 2018). Then, we studied these 43 elements to extract mutation operators. During this process, we applied the following criteria:
• C1: in accordance with our test architecture, we build mutation operators applicable to unencrypted events;
• C2: a mutation operator performs small changes, it is here used to build an attack executed with one test case only. Hence, complex attack scenarios cannot be considered;
• C3: knowledge typically plays a crucial role in performing security attacks. We consider having labels in test steps allowing to recognise authentication processes, token generation and errors. Additional labels allow to recognise the existence of variables acting as tokens or session identifiers;
• C4: an operator can derive new test cases and new mock components.

Using these criteria, we finally wrote 18 mutation operators tailored to testing in isolation the security of black box RESTful APIs. These operators are outlined in Table 1, where column 2 gives short descriptions, columns 3 and 4 give the expected behaviours that should be observed after the execution of mutated test steps and conditions on the application of the operators.

3 TEST CASE MUTATION

![Figure 3: Approach Overview.](image)

As illustrated in Figure 3, we propose a test case mutation approach and a tool for RESTful APIs, consisting of three main stages:

1. our approach takes either existing IOTS test cases, or Log files that are used to generate IOTS test cases. As stated previously, the paper (Salva and Sue, 2023) presents algorithms and a tool for performing this step;
2. mutation operators are applied on IOTS test cases to perform slight modifications that aim to mimic security attacks. These modifications may result in numerous mutated test cases. To address this, we suggest strategies to restrict their generation. Mutation operators are then applied on test cases: we check whether the test steps meet some mutation conditions to restrict the transformations on the relevant steps only; we modify the original test cases and complete them with tests steps and verdicts to get new IOTS mutants;
3. the mutants are finally converted into test scripts and mock components, which will be used to check whether SUT is vulnerable.

We formalise those steps in the remainder of this section.

3.1 Test Mutation Operator Definition

A mutation operator \( M \) of an IOTS test case \( tc \) is made up of three elements. The first is the function \( Condition \), which aims at restricting the application of the operator to some events of \( tc \). The next function \( Change \) applies the mutation on \( tc \) and produces an initial mutant \( tc_m \). Finally, Expected is a function that completes \( tc_m \) with test sequences finished by verdict states to express the expected observations after the execution of a mutated event.

**Definition 2 (Mutation Operator):** A mutation operator is the tuple \((Condition, Change, Expected)\) such that:

- **Condition**: \( Q \times \Sigma \times L^* \times Q \rightarrow \{true, false\} \) is a function that expresses restrictions on test steps,
- **Change**: \( IOTS \rightarrow IOTS \) is a mutant derivation function,
- **Expected**: \( IOTS \rightarrow IOTS \) is a mutant completion function, such that for any test sequence \( q' \leftarrow (e_1(a_1)f_1)\ldots(e_l(a_l)f_l) \rightarrow q \) of the IOTS, \( q \in Q_f \) is a final state.

The function \( Condition(q \leftarrow e(a,l)\rightarrow q') \) of a mutation operator \( M \) may be used on the label \( e \), on the assignments \( a_\ell \), or on the label list \( l \). This function can be used to define a generic operator, for example with a condition of the form \( e == * \) supplemented with some conditions on \( a \). But, a more specific operator can also be defined with a condition on precise events and parameters. The last column of Table 1 provides several condition examples.

\( Change(tc) \) applies the mutation operator on a test case \( tc \) and returns a mutant. We here assume having some transitions marked with the special label “mutation”, which targets the transitions to transform. Given under the form of a procedure, \( Change \) could have the following form:

Reach a transition \( t : \leftarrow q \xrightarrow{e(a,l)\rightarrow} q_1 \);
Modify \( t \);
(possibly) Keep the next outgoing transitions from \( q_1 \) to \( q_r \) such that \( v(q_1) \xrightarrow{e(a,l)\rightarrow} q_r \) is SUT;
Prune the useless transitions from \( q_r \) to a terminal state.
**Expected** \((tc_m)\) completes a mutant returned by Change with new test steps such that the last test steps end by a verdict state. Column 3 of Table 1 summarises the test steps that are added for every mutation operator.

### 3.2 Test Case Generation

The test architecture of Figure 1 emphasises the control and observation logics. The controller parts have the capability to send events to SUT. These events are those that can be modified by mutation operators to send unexpected requests or attacks. The observer parts will be used to collect responses, which are interpreted to decide whether SUT is vulnerable or not.

In this context, we say that a test step \(q \xrightarrow{e(\alpha)} q'\) of a test case is mutable if the recipient of the event is SUT itself and if the operator \(M\) may be applied on this test step. Likewise, we use the notation \(mutable(M)\) in \(tc\) to get the set of test steps on which the mutation operator \(M\) can be applied. It is worth noting that this set may be empty. This is captured by the following definition:

**Definition 3 (Mutable Test Step)**: Let \(M\) be a mutation operator, tc be an IOTS test case for the service SUT, and \(q \xrightarrow{e(\alpha)} q'\) be a test step.

- \(q \xrightarrow{e(\alpha)} q'\) is mutable \(M\) iff \(to(e(\alpha)) = SUT \land M.Condition(q \xrightarrow{e(\alpha)} q') \land ((e(\alpha) \in \sigma_l \lor \text{"mock"} \in l))\).

- \(mutable(M)\) in \(tc = \{ q \xrightarrow{e(\alpha)} q' \in tc \mid q \xrightarrow{e(\alpha)} q'\} \)

Furthermore, we define the IOTS operator mark, which simply adds a label “mutation” on the mutable test steps.

**Definition 4 (IOTS Operator mark)**: Let \(t = q \xrightarrow{e(\alpha)} q'\) be a test step of a test case tc = \(\langle Q, q_0, \Sigma \cup \{\emptyset\}, L, \rightarrow \rangle\).

Mark \(t\) in \(tc = \langle Q_2, q_0, \Sigma_2 \cup \{\emptyset\}, L_2, \rightarrow_{2} \rangle\) is the IOTS test case derived from the test case tc where \(Q_2, \Sigma_2, L_2, \rightarrow_{2}\) are defined by the following rules:

\[
\begin{align*}
&\text{mark } t = q \xrightarrow{e(\alpha)} q' \text{ such that } q \xrightarrow{e(\alpha)} q' \in tc, q \xrightarrow{e(\alpha)} q' \text{ is mutable } M; \\
&\text{mark } t = q \xrightarrow{e(\alpha)} q' \text{ such that } t_2 = q \xrightarrow{e(\alpha)} q' \text{ and } q_2 \neq t_2.
\end{align*}
\]

We are now ready to present our test case mutation algorithm given in Algorithm 1: it takes a mutation operator \(M\) along with a test case set \(TC\). It produces a new test case set, denoted \(TC_M\). It covers every mutable test step of a test sequence \(ts\) (line 3) starting from the initial state of the test case such that \(ts\) is finished by the state \(pass\). We choose to only mutate test sequences finished by \(pass\) to avoid bringing confusion in the test result analysis. Indeed, if we mutate a test sequence finished by \(fail\) and if we obtain a fail verdict while testing, it is very difficult to deduce whether SUT is faulty on account of the mutation. As the set of mutants may become large, Algorithm 1 calls the function \(selection(TC, TC_M)\), which returns a boolean value. This function expresses a mutation generation strategy, e.g., “applies \(M\) on every test case only once”, which stops the mutation of the test cases once some conditions are met. In this case, the function returns false. Algorithm 1 marks the chosen test step with “mutable” to help the mutation operator target the test step to change. A new test case \(tc_2\) is built by applying the function \(M.Change\) and by completing its branches not finished by a verdict state with \(M.Expected\) in order to express the expected behaviour after the execution of the mutated test step. Additionally, the mutant \(tc_2\) is completed once more (line 7) with the operator \(compl : IOTS \rightarrow IOTS\) to add transitions that express all the behaviours that might be observed and the related test verdicts. The resulting mutant \(tc_2\) is stored in \(TC_M\).

**Algorithm 1: IOTS Test Case Mutation.**

```
input : Test case set TC, Mutation Operator M
output: Test case set TC_M
1 TC_M := β;
2 foreach tc ∈ TC do
3    foreach q ∈ mutable(M) in ts such that
4       ts = q_0 \rightarrow_{1} \rightarrow_{l} \rightarrow_{1} q_1 \rightarrow_{\text{pass}} \rightarrow_{inc} \rightarrow\rightarrow\rightarrow\rightarrow q_1
5       selection(TC, TC_M) do
6         mark q in tc such that q \xrightarrow{e(\alpha)} q' ∈ mutable(M) in ts arbitrarily chosen;
7         \(tc_2 := M.Change(tc, q \xrightarrow{e(\alpha)} q')\);
8         \(tc_2 := M.Expected(tc_2)\);
9         compl tc_2;
10        TC_M := TC_M ∪ \{tc_2\};
```

586

ENASE 2024 - 19th International Conference on Evaluation of Novel Approaches to Software Engineering
The inference rule $r_1$ takes all the transitions of an IOTS to build a new test case. $r_2$ completes the test case with a ... allows to generate concrete test scripts and mock components by means of these operators. A significant contribu-

Security Testing of RESTful APIs with Test Case Mutation
587

Figure 4: IOTS Mutated Test Case example.

Figure 4 illustrates an example of mutant obtained from the test case of Figure 2 by applying the operator "Token removal" on the second mutable test step (*/ok*). The verdict is pass if a response is observed with a status 403 or a status 401, which encodes that the request has been rejected on account of insufficient permissions.

### 3.3 Generation of Concrete Test Cases

Finally, executable test scripts are generated from IOTS test cases. We have chosen to generate test cases using the frameworks Citrus and Mockserver. Given an IOTS test case $tc \in TC_M$, some parameters may still be assigned to "*". For example, this happens for parameters used to identify sessions. In short, we assign these parameters with stored values collected from Log files or with random values. To generate a test script, the transitions of $tc$ labelled by "mock" are used to derive rules of the form $request(...).respond(...)$, which mimic the behaviour of a dependee service. Then, the method "verificationMock" is written according to these rules. Figure 6 shows a rule example written with the language provided by the framework MockServer.

![Mock component piece of code](Image)

Figure 5: Example of test script for the service AccMan.

Figure 6: Mock component piece of code, which implements the events !/EvaluateRisk and !/ok of the test case of Figure 4.

### 4 CONCLUSION

In this paper, we present an original solution to generate mutated test cases for testing the security of RESTful APIs. We propose a list of 18 mutation operators specialised in the generation of security test cases. Subsequently, we introduced an algorithm allowing to generate concrete test scripts and mock components by means of these operators. A significant contribu-
tion of this algorithm is its ability to generate mock components to test a RESTful API in isolation. We provide a preliminary evaluation of our algorithm to study the effectiveness of the mutated test cases and how its efficiency. This part is detailed in (Salva and Sue, 2024). Our results demonstrate its capability to construct hundreds of test cases and mock components within minutes, and show good scalability. Besides, the mutants enable the detection of weaknesses in REST APIs and enhance code coverage.

At the moment, our mutation operators allow to infer mutants that mimic attacks performed by one test step. As part of future work, we aim to define more sophisticated operators that could support the mutation of several steps at a time, thus constructing more complex attack scenarios.

ACKNOWLEDGMENT

Research supported by the industrial chair on Digital Confidence https://www.uca-fondation.fr/chaires/confiance-numerique/

REFERENCES


## APPENDIX

Table 1: Test case mutation operators for the security testing of RESTful APIs.

<table>
<thead>
<tr>
<th>Mutation</th>
<th>Description</th>
<th>Expected Behaviour</th>
<th>Mutation Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Duplication</td>
<td>duplicate a request event to SUT</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } &quot;\text{crash}&quot; \notin l \land \text{from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
<tr>
<td>HTTP Verb Change</td>
<td>changing the HTTP verb of a request</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{status(\alpha) := 400} \land \text{from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>isReq(e(\alpha)) = true</td>
</tr>
<tr>
<td>XSS attack</td>
<td>XSS attack</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;.,body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>body(\alpha) \notin \text{\textquoteleft\textquoteleft}header(\alpha)\textquoteright\textquoteright) = \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Cryptographic failures</td>
<td>Replay an event using untrusted connection</td>
<td>( q \xrightarrow{\Delta t} \text{pass with contains(&quot;ERR_CERT_AUTHORITY_INVALID&quot;, body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
<tr>
<td>Token Removal</td>
<td>Delete a token in event</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) \geq 401 \lor status(\alpha) \leq 403 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>&quot;token&quot; \in l</td>
</tr>
<tr>
<td>Token Removal on the creation</td>
<td>Delete a token in event</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) \geq 401 \lor status(\alpha) \leq 403 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>&quot;token creation&quot; \in l</td>
</tr>
<tr>
<td>Token Alteration</td>
<td>Replacing a token by another one; three types: expired authentication token, token existing but not for this session, and token not existing</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) \geq 401 \lor status(\alpha) \leq 403 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>&quot;token&quot; \in l</td>
</tr>
<tr>
<td>Stress Testing</td>
<td>replay events to the tested service a lot of times in a small window</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;.,body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
<tr>
<td>SSRF Enforce \text{&quot;deny by default&quot;}</td>
<td>request or response from an unknown service</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;.,body(\alpha)) \lor status(\alpha) := 404 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
<tr>
<td>Body data manipulation</td>
<td>replay request using unauthorized data</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) \geq 400 \lor status(\alpha) \leq 422 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>body(\alpha) \notin \text{\textquoteleft\textquoteleft}header(\alpha)\textquoteright\textquoteright) = \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Cookie manipulation</td>
<td>change a cookie to inject an attack</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{status(\alpha) := 400 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>cookie(\alpha) \notin \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Failed Login Attempt</td>
<td>duplicating login event with wrong credentials</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;.,body(\alpha)) \land \text{from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>isReq(\text{\textquoteleft\textquoteleft}logon\textquoteright\textquoteright) \notin \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Duplication</td>
<td>Path manipulation</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) := 404 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>isReq(\text{\textquoteleft\textquoteleft}logon\textquoteright\textquoteright) \notin \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>SQL injection</td>
<td>manipulate input data to inject SQL code</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{\text{status(\alpha) := 400 \lor contains(&quot;error&quot;.,body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>body(\alpha) \notin \text{\textquoteleft\textquoteleft}header(\alpha)\textquoteright\textquoteright) = \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Session management</td>
<td>add a (long) delay during which no reaction should be observed before the next event</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) := 401 \land contains(&quot;error&quot;: sessionterminated(body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
<tr>
<td>Information leakage</td>
<td>modify a request to get access to sensitive information</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;:connexioninterruption&quot;,body(\alpha)) \lor status(\alpha) := 408 \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>isReq(\text{\textquoteleft\textquoteleft}logon\textquoteright\textquoteright) \notin \text{\textquoteleft\textquoteleft} )</td>
</tr>
<tr>
<td>Depended service shutdown</td>
<td>shutdown a mock component after requesting it</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;crash&quot;} \notin l \land \text{contains(&quot;error&quot;: connexioninterruption&quot;,body(\alpha)) \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( q \xrightarrow{\Delta t} \text{pass with } \text{&quot;mock&quot;} \notin l )</td>
</tr>
<tr>
<td>Buffer overflow</td>
<td>overflow input data for trying to crash a server</td>
<td>( q \xrightarrow{\Delta t} \text{pass with status(\alpha) := 400 \land \text{&quot;crash&quot;} \notin l \land from(q} \xrightarrow{\Delta t} \text{pass}) = \text{SUT} )</td>
<td>( e = \star )</td>
</tr>
</tbody>
</table>