Planning-based Security Testing of Web Applications

Josip Bozic
Graz University of Technology
Institute of Software Technology
A-8010 Graz, Austria
jbozic@ist.tugraz.at

Franz Wotawa
Graz University of Technology
Institute of Software Technology
A-8010 Graz, Austria
jbozic@ist.tugraz.at

ABSTRACT

Web applications are deployed on machines around the globe and offer almost universal accessibility. The systems ensure functional interconnectivity between different components on a 24/7 basis. One of the most important requirements represents data confidentiality and secure authentication. However, implementation flaws and unfulfilled requirements can result in security leaks that can be eventually exploited by a malicious user. Here different testing methods are applied in order to detect software defects and prevent unauthorized access in advance.

Automated planning and scheduling provides the possibility to specify a specific problem and to generate plans, which in turn guide the execution of a program. In this paper, a planning-based approach is introduced for modeling and testing of web applications. The specification offers a high degree of extendibility and configurability but overcomes the limits of traditional graphical representations as well. In this way, new testing possibilities emerge that eventually lead to better vulnerability detection, thereby ensuring more secure services.

CCS CONCEPTS

• **Security and privacy → Penetration testing**; **Web application security**; • **Computing methodologies → Planning and scheduling**;

KEYWORDS

Planning, security testing, model-based testing, web applications

ACM Reference Format:
Josip Bozic and Franz Wotawa. 2018. Planning-based Security Testing of Web Applications. In *Proceedings of International Workshop on Automation of Software Test (AST'18).* ACM, Gothenburg, SWE, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The testing for vulnerabilities in software represents an important task. Over the last few years the same vulnerabilities remain on the top 10 list of most common security leaks in web applications [7]. The reasons therefore may be human implementation errors or insufficient security mechanisms on side of the application.

SQL injections (SQLI) and cross-site scripting (XSS) are still among the most common attacks despite increased tester awareness of their potential damage. Here malicious user inputs eventually get bypassed by the applications' input filters. This usually results in theft of confidential data or causes some other harm to the user or the system. The consequent costs for the repair of trust and damage can be critical for service providers so security measures have to be implemented.

For this reason, a huge number of approaches and ideas have contributed to this permanent problem.

Artificial intelligence (AI) was initially used in robotics and intelligent agents [27] but is applied, although to a smaller degree, to testing as well [9, 15]. Here sequences of actions are generated in order to test certain type of systems for intrinsic leaks. In general, security testing of a system is done either by checking whether a running system can be tested or hacked by malformed inputs or whether it can be broken down completely.

Other techniques include combinatorial testing (CT). One of the main motivations behind CT is test set reduction [21, 26]. Here a security leak is eventually detected by a reduced number of tests. This implies a reduction of testing time but still ensures the ability to identify causes of vulnerabilities.

Some techniques rely on models, either of the application [31, 32] or the attacks itself [28, 29]. Here a system under test (SUT) is checked whether it behaves in line with its specification (white-box) or tests it independently of the its inner structure (black-box). In both cases the testing procedure is guided by a graphical representation that correlates to the SUT.

The approach in this paper contributes to the application of AI for security testing of web applications. The work presents the adaptation of automated planning and scheduling to testing of common XSS and SQLI vulnerabilities. In addition to that, a test execution framework encompasses a crawler, which helps in exhaustive testing of the SUT. The idea behind the approach is to automatically test with unexpected user sequences and values in order to detect these vulnerabilities. The approach as well as the first empirical results are elaborated on a concrete example.

The paper is organized as follows. Section 2 introduces planning to security testing. The basic concepts of automated planning and scheduling are discussed as well as its adaptation to web applications. Then, Section 3 discusses the functionality of the automated test execution framework. Section 4 depicts an example of the combined approach in action. Section 5 enumerates related works and Section 6 concludes the paper.

AST'18, May 28-29, 2018, Gothenburg, SWE

ACM ISBN 978-1-4503-5743-2/18/05.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 PLANNING FOR TESTING

Automated planning and scheduling, or planning, has been already used in testing. An adaptation for security testing of web applications was introduced by the authors of this paper in [11] and has been applied further for testing of the TLS protocol in [12]. However, the current work presents an improved approach in terms of test case generation (number and variety of plans) and execution. Until now the program that generates plans, the planner, returned only one (optimal) solution for a certain testing problem. However, currently we use a different planning system where the plan generation process can be guided by the tester. It takes as inputs a planning problem, thereby eventually returning a solution in form of a plan. The plan itself consists of multiple actions that lead the execution from an initial to a final state. The initial state is usually an idle state where the execution starts. On the other hand, the final state is a state when the execution terminates, returning either a passing or failing test verdict.

The underlying definitions for planning were initially given in [16]. They were adapted to our previous work in [11] and look as follows:

Definition 2.1. The planning problem is given with the quadruple (P, I, G, A). P defines the set of predicates, I and G the initial and goal states, whereas A denotes the set of actions. A predicate $p \in P$ represents a first-order logic formula and is used by $a \in A$. Every state is specified by predicates that are true in this state.

In case that the preconditions of an action a are true in a certain state S, then it will be added to the plan. In this case, a will act as a transition to the new state S', thus $S \xrightarrow{a} S'$. The predicates from its postconditions will be valid in S' and act as the new state's precondition.

Definition 2.2. A solution for a planning problem (P, I, G, A) is a plan that comprises a set of actions so that $S_1 \xrightarrow{a_1} S_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} S_n$.

The definition of predicates and their corresponding conditions will guide the planning process. Sometimes more solutions do exist for a planning problem. During its search, the planner decides what path to choose according to the underlying planning algorithm. On the other hand, if no solution can be produced for a problem, then the planner terminates further search.

2.1 Planning Model

In order to specify a planning problem, we create a planning model. In contrast to other graphical representations, e.g. UML state machines, we describe our model in the Planning Domain Definition Language, PDDL [22]. PDDL is used in AI in order to specify planning problems, e.g. for intelligent systems [23].

The reasons for this decision are the potential extendibility and configurability of such planning models. For example, actions can be defined without preconditions, which makes the overall model more comprehensive. Thus, model explosion can be avoided. New specifications can be added easily or several models can be combined. Consequently, every change in the model will result in the generation of diverse plans. Here we are able to obtain plans with greater variety that eventually improve fault detection. In addition to that, a planning model might cause execution traces that are usually not defined in a model of the specification of a SUT.

Regarding PDDL, two descriptions have to be specified, namely:

- Domain: Comprehends the definitions that are present in every problem.
- Problem: Comprehends the definitions that are valid only in a specific problem.

In general, PDDL uses a type-object hierarchy and the definition of first-order logic predicates. The objects are used in both specifications and are put as parts of predicates. Additionally, actions are specified by parameters, pre- and postconditions. The conditions are defined by predicates and describe when an action is selected and what effects it will cause when executed.

The problem specification usually defines application specific parameters. However, our approach keeps both specifications as minimal as possible. The problem defines objects and the initial and final states. The corresponding problem specification in PDDL looks as follows.

```
(define (problem mbtp)
 (:domain mbtd)
 (:objects x)
 (:init (inInitial x))
 (:goal (and (inFinished x))))
```

Problem description in PDDL

The initial state represents the starting point of the test execution. This is the point before any action has been carried out. On the other hand, the final state is reached when a test case, i.e. a plan, has been carried out. In reality, the initial state is the point before the execution framework even accesses a specific URL address of a SUT. The final state is eventually reached after an attack and all its aforegoing actions have been carried out and a test verdict is given. Our specification relies on only one object, x, that takes the role inside a specific state during execution.

In addition to that, PDDL's domain encompasses, among others, the definitions of predicates and actions. Our definition for testing of web applications is given below.

```
(define (domain mbtd)
 (:requirements
    :strips :typing)
 (:predicates
    (ininitial ?x)
    (inGotSite ?x)
    (inAttackedSQLI ?x)
    (inAttackedXSS ?x)
    (inFinished ?x) )

 (:action GetSite
    :parameters(?x)
    :precondition (and )
    :effect (and (inGotSite ?x)))

 (:action AttackXSSGet
    :parameters(?x)
    :precondition (and
```

```
      (inGotSite ?x)
      (inAttackedSQL ?x)))
   :effect (and
      (inAttackedXSS ?x)
      (inFinished ?x)))

(:action AttackXSSPost
   :parameters(?x)
   :precondition (and
      (inGotSite ?x)
      (inAttackedSQL ?x))
   :effect (and
      (inAttackedXSS ?x)
      (inFinished ?x)))

(:action AttackSQLGet
   :parameters(?x)
   :precondition (and
      (inGotSite ?x))
   :effect (and
      (inAttackedSQL ?x)
      (inFinished ?x)))

(:action AttackSQLPost
   :parameters(?x)
   :precondition (and
      (inGotSite ?x)
      (inAttackedXSS ?x))
   :effect (and
      (inAttackedSQL ?x)
      (inFinished ?x)))
```

Domain description in PDDL

The specification defines all possible states that are encountered during test execution as predicates. These are used for the definition of actions, together with the already mentioned object x as a parameter. The action's pre- and postconditions can be specified by zero, one or more predicates that guide the plan generation. Actions can be picked multiple times as part of one plan if the conditions are met. Actually, they are defined in a way that ensures that many plans can be generated by the planner, for example by omitting or using multiple conditions.

Both specifications are saved in form of PDDL files and submitted to a planning system for constructing a plan. Section 2.3 explains the connection between both definitions with regard to testing of web applications in more detail. Then, plans are generated by the planner according to a planning algorithm (see Section 2.2). An example of a generated plan with seven actions is given below.

```
[GetSite(x), AttackSQLGet(x), GetSite(x),
 AttackSQLPost(x), AttackXSSPost(x),
 GetSite(x), AttackXSSGet(x)]
```

As can be seen, the plan comprehends actions as specified in the domain definition.

Table 1: Generated plans for XSS and SQLI

1	GetSite(x), AttackSQLGet(x), AttackXSSGet(x), AttackSQLPost(x)
2	GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x), AttackSQLPost(x)
3	GetSite(x), GetSite(x), AttackSQLGet(x), AttackXSSGet(x), AttackSQLPost(x), GetSite(x), AttackXSSPost(x)
4	GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x), GetSite(x), AttackSQLPost(x)
5	GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSGet(x),GetSite(x), GetSite(x), AttackSQLPost(x), AttackXSSGet(x)
6	GetSite(x), GetSite(x), AttackSQLGet(x), AttackXSSGet(x), GetSite(x), GetSite(x), AttackSQLPost(x), AttackXSSPost(x)
7	GetSite(x), AttackSQLGet(x), AttackXSSPost(x), GetSite(x), AttackSQLPost(x)
8	GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSPost(x), AttackSQLPost(x), GetSite(x), AttackXSSPost(x)
9	GetSite(x), AttackSQLGet(x), GetSite(x), GetSite(x), AttackXSSPost(x), AttackSQLPost(x), GetSite(x), GetSite(x), AttackXSSGet(x), AttackSQLPost(x)
10	GetSite(x), AttackSQLGet(x), GetSite(x), AttackXSSPost(x),GetSite(x), AttackSQLPost(x), GetSite(x), AttackXSSGet(x), GetSite(x), AttackSQLPost(x)

2.2 Planning System

In the previous work, we relied on Metric-FF [5] for plan generation. Unfortunately, that planner usually returns only one plan for a planning problem. However, in the current approach we generate a high number of plans. This number is multiplied with the number of concrete values so a much higher number of test cases is the result. For this purpose we use a Java implementation of the Graphplan algorithm, JavaGP [3].

The algorithm was initially introduced in [10]. Here a planning graph is constructed in the search for a solution. The algorithm starts in an initial state and extends the graph level as long as no plan is generated. First, the planner checks whether the goals of the effects in the last graph level are true in the conditions of the initial level by using backward chaining. If this is not the case, then new actions are selected and the graph is extended. Then the newly obtained goals act as the starting point for solution extraction. In this case, the depth of the planning graph is increased. In JavaGP this parameter can be set manually in order to define or restrict the number of generated plans. The higher the value, the more plans will be generated. With increasing size of the planning graph, the number of necessary steps to obtain a plan increases as well.

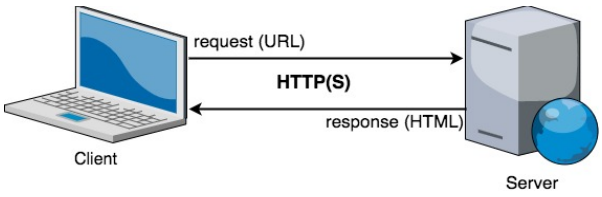

Figure 1: HTTP communication between client and server

Finally, if a path is found that reaches from the initial to the last graph level, then a plan has been found. The individual actions on that path will constitute the atomic parts of the plan.

In our case, the planning system reads the generated PDDL files and returns a set of plans that will be submitted to the test execution framework. A small excerpt of the test set is depicted in Table 1.

The main motivation for generating multiple plans is to cause unintended behaviour, eventually confusing the SUT. Also, the planning algorithm generates plans with repetitive actions, which means that some actions are executed multiple times in one test case. The test suite encompasses plans with a broad diversity. As depicted in the table, the length of the plans differs in number, whereas some bigger plans comprehend smaller ones. Also, we set the initial precondition to cause the action `GetSite` to be the first action in every plan.

2.3 Planning for Web Applications

The basic concept of the communication between a client and an application on a server is depicted in Figure 1. Usually the client accesses a web application via its URL address by sending a HTTP request. The server returns a corresponding HTTP response. Both message types encompass a standardized structure with unique parameters, whereas its values are generated when these messages are created. All message content, including harmful one, between peers is transferred via HTTP messages. Vulnerabilities are triggered either due to security leaks on the side of the client or the sever. A detailed description of both attacks can be found in [17] and [13].

The implementation of the HTTP communication is elaborated in Section 3. Section 2.1 gives an overview of the adaptation of HTTP functionality to planning in our approach. It focuses on client-side messages, so it omits man-in-the-middle attacks where the attacker emulates the server in front of a victim. Hence, server-side processes will not be taken into consideration in the planning specification. However, the reactions from the server to the client's actions will be checked automatically by the framework during execution. Also, the goal of this approach is to test an application, i.e. to prove a vulnerability that could be exploited in the aftermath. HTTP request messages encompass methods that are usually `GET` or `POST`. In the first case a client requests data from a URL address, whereas data is submitted to the destination in the second case. However, both methods can be used to submit and receive data. An example of the request line of HTTP `GET` looks like:

```
GET /site/login_form.php?username=john
&password=!js123 HTTP/1.1
```

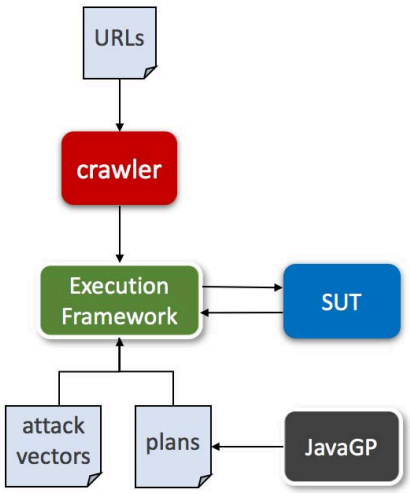

Figure 2: Planning-based testing framework

On the other hand, the counterpart `POST` can have the following form:

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=john&password=!js123
```

Here both depictions describe an example of a login form with two parameters. However, in certain cases it does make sense to send a `GET` message attempt where a `POST` is expected and vice versa. For this reason, we labelled actions to resemble such HTTP requests so that a request is sent with another method instead of the expected one. In such way we are able to trigger unexpected executions by the tester. The corresponding parameters will be instantiated with malicious inputs and sent as part of a real HTTP message to the server.

The establishment of a communication between attacker and SUT, as well as the plan-guided test execution is described in the next section in more detail.

3 TEST EXECUTION FRAMEWORK

The test execution framework is implemented in order to resemble the communication between client and server. It encompasses the functionality of creating, sending, receiving and reading HTTP messages. Additionally, HTTP responses are parsed and critical elements for detection of XSS and SQLI are extracted. This plays a crucial role for the test oracle (see Section 3.1), which is consulted every time an attack was carried out. The framework is programmed in Java. The overall structure of the implemented approach is depicted in Figure 2.

For functionality of HTTP communication we rely on HttpClient [1], which offers the possibility for configuring HTTP messages. A Java library, jsoup [4], serves as the parser for incoming responses from the SUT.

The generated plans serve as abstract test cases. After the planner generates the abstract test set, it is accessed by the execution

framework. The abstract plan guides the execution on the abstract level. On the other side, a concretization phase is needed in order to carry out instantiated test cases. This is done automatically by the testing framework. Here concrete Java methods are implemented that are called according to the current action from the abstract sequence. In fact, every action from PDDL corresponds to one Java method. However, it should be noted that the number of plan does not determine the overall number of concrete test cases. In fact, this is not known in advance but figured out during execution since we don't know how many plans will be generated. If every action from a plan is executed and a passing test verdict is triggered during the testing process, then we consider this plan to be a successful test case. Afterwards, the next plan will be picked from the test set. This process will continue as long as abstract test cases are available or the execution crashes.

We define malicious inputs as attack vectors. These are concrete inputs that are meant for testing purposes and can be either SQL commands or XSS scripts. Both test sets are attached separately to the framework and are retrieved at runtime during execution. For this paper, we use a custom test set for both types of attacks.

Another important part of the framework is the crawler. The task of the crawler is to ensure that a SUT is tested completely. It accepts a manually specified list of URL addresses that are meant for testing and searches for their corresponding hyperlinks. All encountered links are added to the list of seeds. After the list of addresses has been finalized, the testing process can start. In such way, we want to automatize the testing process for every web application.

During execution, web sites are parsed and various user input HTML elements, e.g. input fields and textareas, are extracted by the parser. This information will be submitted for further test execution, which will assign the attack vectors to these input elements. Every element will be tested against every of the attack vectors. However, in case that multiple input elements are found on a website, every of these will be tested individually. For example, a HTTP `GET` attack against a website with two user inputs may look as follows:

```
GET /site/login_form.php?username=
<script>alert(0)</script>&password=
```

Parameter values have to be URL-encoded for submission (e.g. the XSS script `<script>alert(0)</script>` will be submitted as `%3Cscript%3Ealert%280%29%3C%2Fscript%3E`) but for demonstration purposes we denote the original form of the attack vectors.

In the example the parameter `username` is tested while `password` remains blank. However, afterwards the same attack will be repeated but `password` will be tested so `username` remains empty.

```
GET /site/login_form.php?username=&password=
<script>alert(0)</script>
```

The same principle, when adapted to HTTP `POST`, will have the following form:

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=<script>alert(0)</script>
&password=
```

Figure 3: Filtered XSS input

```
POST /site/login_form.php HTTP/1.1
Host: w3company.com
username=&password=<script>alert(0)</script>
```

The same principle is used when testing for SQLI, only the attack vector will differ in that case.

This demonstrates how only one action from the plan, one attack vector for every type of attack and two input elements will affect the testing procedure. The importance of planning is the guidance of the execution. A plan with a different action sequence may lead to different testing results.

3.1 Test Oracle

After every attack the resulting response from the server is parsed and checked according to a criterion. Then, a verdict is given whether the attack was successful with the submitted attack vector. In order to get an insight about the functionality of the test oracle, Figure 3 depicts the behaviour of an application that has been tested with a XSS script.

When the user sends data to a website, this can be reflected back to the sender inside the body of the HTTP response. However, this responded information can serve as an indication whether the test was successful or not.

Application-intern security measures include the filtering of user inputs. Whereas some inputs are allowed, others are on the black list. Characters like < and >, among others, are usually on this list. Since XSS relies on scripts being activated and doing some harm, it is important that they do not get detected by input filters. Critical elements are usually neutralized by an application with mechanisms like HTML escape encodings. In such way, critical symbols are rewritten and cannot cause any harm on the victim's size. Here < is escaped with `<` while > is encoded with `>`.

Critical XSS elements encompass a variety of different elements like images (`<img>`), frames (`<frame>`) etc. In fact, all our XSS attack vectors contain such elements with a code attached to them. The execution framework decides that a test was successful when the submitted script was not detected and is reflected back to the sender in its original form. Thus, the communication would look like the one in Figure 4.

Before the attack vector is submitted to a website, all of its critical HTML elements are parsed and their occurrence is noted. After the test has been carried out, the implementation reads the response from the server and re-counts the critical HTML elements again. Here the number of these elements should be increased by one, since the sent script should have been added to the response body as

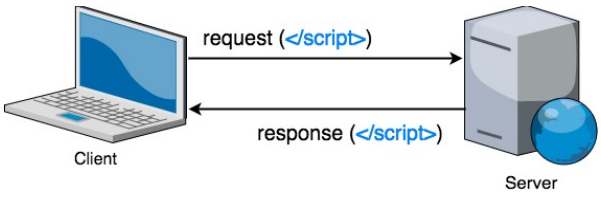

Figure 4: Unfiltered XSS input

well. For example, if the attack vector contains a `<script>`, then the occurrence of this (unfiltered) element will be higher than before. In this case, our program returns a *PASS*. However, if the number remains the same, then the framework indicates a *FAIL*. In this case, the attempt was probably filtered out or something else happened. The same mechanism is used to test both reflected and persistent XSS, since both rely on a surplus of HTML elements.

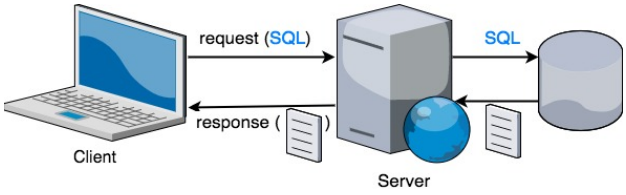

Figure 5: Procedure of SQL injection

SQL injections work similar with regard to input filtering. However, in this case a SQL statement is submitted in order to access data from the SUT's database. In order for the attack to succeed, information is retrieved from the database and unintentionally sent to the user. Figure 5 shows the procedure. However, since the behaviour of an application is impossible to predict, we specify one expected value that the tester excepts to retrieve from the database. For example, if a company's database is tested then the attacker could figure out information like email addresses via social engineering. This key value is searched automatically inside the server's response body after the attack. If found, the framework returns a *PASS*. Otherwise, the submitted SQL injection will be escaped, similar to XSS, and remains ineffective.

4 CASE STUDY

We demonstrate the applicability of our approach to testing of a known web application for testing purposes [6]. Before the execution starts, we let the planner generate a set of plans according to the planning model. It should be noted that a minor change in the planning specification will result in different plans. The tester can manipulate the specification if the necessity emerges or in order to increase the variety of plans. Since the attack vectors for SQLI and XSS are already attached to the implementation, we need only to specify the URL address of the SUT. Since the crawler is in use, the tester can submit the address of the homepage in order to test the application in depth or add the address for one single website. For this demonstration, we add the root URL of the locally employed SUT: `http://localhost/mutillidae/`. Then, the crawler returns, among many others, the following links:

```
/index.php?page=dns-lookup.php
/index.php?page=user-info.php
/index.php?page=login.php
/set-up-database.php
```

Now the executioner reads the first plan that resembles the one shown in Section 2.1. The first action `GetSite` in the plan indicates to read the first URL from the list and extracts important data like user inputs and other HTML elements. For the first URL address, the parser returns two fields, namely `username` and `password`. Afterwards, a SQLI attack is carried out against both input elements with the method `GET`. We specify an expected value that resembles an existing entry in the database, e.g. `john@smiths.com`. The first attack vector is `' OR 1 = 1 -- .` However, the server's response will be negative for both tested elements since it does not contain the expected email address. Then, another action will instruct to carry out the same attack but with a `POST` method. In contrast to the last attempt, the attack here is successful for both input elements.

The framework grabs the next action and reloads the current URL address again, thereby grabbing its elements. Then, the execution continues to XSS and takes the first input from the list of attack vectors. It submits the input against both elements and checks for the existence of an additional HTML element. In this case, the submitted attack vector is a `<script>alert(0)</script>`. Since it was successful with the `POST` method, the parser encountered a surplus of one `<script>` in the response, thus returning a *PASS*. In fact, a XSS vulnerability was detected in both input elements.

The execution continues further and tries additional XSS and SQLI attempts but with different methods. In both cases the framework returns a *FAIL*. After the last action from the plan is carried out, the execution repeats the execution but uses other attack vectors and checks whether different values lead to different test verdicts.

After both sets of attack vectors are exhausted, the next URL address will be picked for testing. However, when testing the second entry from the list, we could detect only XSS vulnerabilities since the website doesn't comprehend a database entry at all.

The execution continues as long as all URLs have been tested and returns a finalized test result overview.

If a plan triggers a vulnerability, then the action sequence and their values that lead to the detection can be traced back. The obtained information can give an insight about the security leak of the application.

5 RELATED WORK

Two major fields represent the bulk of the work in the presented paper. Namely planning and security testing of web applications. Several works address either one or both areas and provide some insight into proposed methods. Planning was adapted on different case studies in different ways. On the other hand, methods like model-based testing focus more on (security) testing issues by usually relying on a graphical representation of the SUT.

A detailed introduction about planning is given in [18].

One of the first usages of planning for test case generation was described in [20]. The authors guide the plan generation by manipulating the initial and goal states in the specification.

A more advanced approach that combines planning and testing includes [15]. The authors rely on attack graphs by taking into consideration probability and cost measures for choosing attack actions. They introduce an algorithm for selection of an optimal attack strategy and use an attack policy. A policy has a cost so an attacker can determine the probability of attack by choosing different (sub)policies. A Markov Decision Process (MDP) is used to produce an optimal attack, i.e. an attack that has an overall minimized cost, in form of an attack graph. The algorithm does an exhaustive search of every action with regard to its cost at any possible decision point during the search. Although a generated plan from our approach resembles an attack path as well, we do not rely on probabilistic calculations for the attack construction but attack with a huge variety of plans instead.

Other works that deal with planning techniques for testing include [24], [19] and [25], whereas model-based approaches can be found in [14] and [30]. Tools that target the tested vulnerabilities in this paper include [2] and [8].

6 CONCLUSION AND FUTURE WORK

In this work, automated planning and scheduling has been adapted to security testing of web applications. A planning model has been specified in PDDL and serves as an input for the planner. The output is a set of plans that represents abstract test cases that are read by a test execution framework. Every action from a plan guides the execution against the SUT. In addition to that, a concretization phase is implemented that encompasses HTTP functionality. A website is parsed and HTML user input elements are extracted for testing. The crawler supports the execution by adding hyperlinks to the list of initial seeds.

Two sets of attack vectors act as user inputs in order to test a SUT against XSS and SQLI. Initial test results have proven that planning-based security testing is able to detect both types of vulnerabilities in an automated manner.

The use of planning models can help in keeping the representation relative small but maintain wide test case generation possibilities. Adjustments can be added easily to the model, whereas the plan generation can be controlled by the tester.

In the future, we plan to extend the planning model specification by taking more real-world activities into consideration. Also, other attack types can be modelled as well. In fact, since a hacking procedure represents a sequence of actions, other malicious activities can be included into the model. Another possibility is the combination of different attack types for more exhausting security testing. A comparison with other state-of-the-art techniques from both planning and testing will help to measure performance metrics of our approach as well.

ACKNOWLEDGEMENT

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 865248 (Securing Web Technologies with Combinatorial Interaction Testing - SecWIT).

REFERENCES

[1] [n. d.]. Apache HttpComponents - HttpClient. ([n. d.]). Retrieved February 2, 2018 from https://hc.apache.org/httpcomponents-client-ga/

[2] [n. d.]. IBM Security AppScan. ([n. d.]). Retrieved March 14, 2018 from https://www.ibm.com/security/application-security/appscan

[3] [n. d.]. JavaGP - Java Implementation of Graphplan. ([n. d.]). Retrieved December 11, 2017 from https://github.com/pucrs-automated-planning/javagp

[4] [n. d.]. jsoup: Java HTML Parser. ([n. d.]). Retrieved February 2, 2018 from https://jsoup.org/

[5] [n. d.]. Metric-FF. ([n. d.]). Retrieved December 12, 2016 from http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html

[6] [n. d.]. OWASP Mutillidae 2 Project. ([n. d.]). Retrieved February 4, 2018 from https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project

[7] [n. d.]. OWASP Top Ten Project. ([n. d.]). Retrieved January 31, 2018 from https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[8] [n. d.]. sqlmap. http://sqlmap.org/. ([n. d.]). Accessed: 2018-03-14.

[9] M. Backes, J. Hoffmann, R. Kunnemann, P. Speicher, and M. Steinmetz. 2017. Simulated Penetration Testing and Mitigation Analysis. In *CoRR abs/1705.05088 (2017)*.

[10] A. Blum and M. Furst. 1995. Fast Planning Through Planning Graph Analysis. In *IJCAI95*. 1636–1642.

[11] J. Bozic and F. Wotawa. 2014. Plan It! Automated Security Testing Based on Planning. In *Proceedings of the 26th IFIP WG 6.1 International Conference (ICTSS'14)*. 48–62.

[12] J. Bozic and F. Wotawa. 2017. Planning the Attack! Or How to use AI in Security Testing?. In *Proceedings of First International Workshop on AI in Security (IWAIse)*.

[13] J. Clarke, K. Fowler, E. Oftedal, R. M. Alvarez, D. Hartley, A. Kornbrust, G. O'Leary-Steele, A. Revelli, S. Siddharth, and M. Slaviero. 2012. *SQL Injection Attacks and Defense, Second Edition*. Syngress.

[14] Fabien Duchene, Sanjay Rawat, Jean-Luc Richier, and Roland Groz. 2014. KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection. In *CODASPY*. ACM, 37–48.

[15] K. Durkota and V. Lisy. 2014. Computing Optimal Policies for Attack Graphs with Action Failures and Costs. In *7th European Starting AI Researcher Symposium (STAIRS'14)*.

[16] R. E. Fikes and N. J. Nilsson. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In *Artificial Intelligence*. 189–208.

[17] S. Fogie, J. Grossman, R. Hansen, A. Rager, and P. D. Petkov. 2007. *XSS Attacks: Cross Site Scripting Exploits and Defense*. Syngress.

[18] M. Ghallab, D. Nau, and P. Traverso. 2004. Automated Planning: Theory and Practice. In *Morgan Kaufmann*.

[19] N. Ghosh and S. K. Ghosh. 2009. An Intelligent Technique for Generating Minimal Attack Graph. In *Proceedings of the 1st Workshop on Intelligent Security (SecArt'09)*.

[20] A. E. Howe, A. von Mayrhauser, and R. T. Mraz. 1997. Test Case Generation as an AI Planning Problem. In *Automated Software Engineering, 4*. 77–106.

[21] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. 2004. Software Fault Interactions and Implications for Software Testing. In *IEEE Transactions on Software Engineering 30 (6)*.

[22] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. 1998. PDDL - The Planning Domain Definition Language. In *The AIPS-98 Planning Competition Comitee*.

[23] S. A. McIlraith, T. C. Son, and H. Zeng. 2001. Semantic Web Services. In *IEEE Intelligent Systems 16(2)*.

[24] A. M. Memon, M. E. Pollack, , and M. L. Soffa. 2000. A Planning-based Approach to GUI Testing. In *Proceedings of the 13th International Software / Internet Quality Week (QW'00)*.

[25] J. Lucangeli Obes, C. Sarraute, and G. Richarte. 2010. Attack Planning in the Real World. In *Proceedings of the 2nd Workshop on Intelligent Security (SecArt'10)*.

[26] M.S. Raunak, D.R. Kuhn, and R. Kacker. 2017. Combinatorial Testing of Full Text Search in Web Applications. In *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*.

[27] S. J. Russell and P. Norvig. 1995. *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[28] A. Shameli-Sendi, M. Dagenais, and L. Wang. 2017. Realtime Intrusion Risk Assessment Model based on Attack and Service Dependency Graphs. In *Computer Communications*.

[29] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. 2002. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[30] O. Nguena Timo, A. Petrenko, and S. Ramesh. 2017. Multiple Mutation Testing from Finite State Machines with Symbolic Inputs. In *Proceedings of the 29th IFIP International Conference on Testing Software and Systems (ICTSS'17)*.

[31] M. Utting and B. Legeard. 2006. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann Publishers Inc.

[32] J. Zander, I. Schieferdecker, and P.J. Mosterman. 2011. Model-Based Testing for Embedded Systems. In *CRC Press*.