PURITY: a Planning-based secURITY testing tool

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

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

***Abstract*—Despite sophisticated defense mechanisms security testing still plays an important role in software engineering. Because of their latency, security flaws in web applications always bear the risk of being exploited sometimes in the future. In order to avoid potential damage, appropriate prevention measures should be incorporated in time and in the best case already during the software development cycle. In this paper, we contribute to this this goal and present the PURITY tool for testing web applications. PURITY executes test cases against a given website. It detects whether the website is vulnerable against some of the most common vulnerabilities, i.e., SQL injections and cross-site scripting. The goal is to resemble a malicious activity by following typical sequences of actions potentially leading to a vulnerable state. The test execution proceeds automatically. In contrast to other penetration testing tools, PURITY relies on planning. Concrete test cases are obtained from a plan, which in turn is generated from specific initial values and given actions. The latter are intended to mimic actions usually performed by an attacker. In addition, PURITY also allows a tester to configure input parameters and also tests a website in a manual manner.**

***Keywords*-Testing tool, Model-based testing, security testing, planning problem.**

I. Introduction

The increasing number of web applications available today and accessible for more and more people require to reconsider improving reliability and also security of these applications. Hence, the need for ensuring secure access to services and programs becomes greater. In fact, this demand will become even more important in the future. With increasing complexity of programs but also because of the fact that hackers are always a step ahead of software developers, the sophistication of malicious attacks grows as well. Hence, there is a strong need of bringing security testing into regular testing practice. However, the question is whether testers become more aware of security issues with the increasing number of web applications? If not, then the consequence is that the number of insecure programs becomes greater as well. In addition, and because of the increasing number of service users once a vulnerability is exploited, the consequences to security and privacy issues are becoming worst.

According to [1], the most common vulnerabilities include SQL injections (SQLI), cross-site scripting (XSS), among others. The fact that these known exploits still represent a major threat for web applications confirm the additional demand for security counter measures. Because of the described dangers, there is a strong need for testing in order to prevent or at least to minimize potential damage. However, in order to effectively test an application the tester needs sufficient experience and know-how, which is not always ensured especially in case of web applications. An additional burden is that the testing process is often a manual and lengthy process with a great need for precision. In order to support testers in this process, a number of tools have been developed. Currently several scanners and manual testing tools are available for XSS detection [2], [3], [4], [5], and also for testing against SQLI [6] attacks. Nevertheless, these manual testing tools still bear the burden of demanding a high amount of time for the application to be tested.

Hence, the main challenge relies in the automation of the testing process with a minimum number of necessary user interaction but still providing enough precision and effectiveness. Until now much work in this direction has been focused on fuzzing [7], which deals about testing with (semi-)random values. Other tools rely on evolutionary algorithms [8]. It should also be mentioned that some manual tools can be (manually) automated as well, for example by implementing plug-ins for the corresponding applications [9]. However, again additional effort is needed for carrying out the test automation.

In order to make security testing of web application easier, we propose the penetration testing tool *Planning-based secURITY testing tool* (PURITY). It has been developed for testing of websites to detect potential SQLI as well as reflected and stored types of XSS security issues in an either manually or automated fashion (or something in-between). The tester is asked for a minimum amount of informations. PURITY also offers the possibility to define all test parameters if desired. The tool is partly built upon previously work already presented. It encompasses parts of a model-based approach that relies on attack patterns

* The author is funded in part by the project 3CCar under grant 662192 (EU ECSEL-RIA) and the Austrian Research Promotion Agency (FFG).
** Authors are listed in alphabetical order.

[10], [11], and also on a technique that defines testing as a planning problem [12]. In fact, PURITY improves test case generation using a planner and makes use of the communication implementation and test oracles from the previously works. The tool presents the obtained test results in detail after the test execution terminates.

PURITY is a security testing tool that is easy to use but also provides high configurability and offers extendibility. We will discuss the last two issues in greater detail further in this paper. PURITY is a research prototype written in Java currently available via the authors. In the close future we plan to release the program as open-source tool.

The paper is structured in the following way: Section II provides a list of works about model-based testing and gives an overview about the authors' previously works. Section III discusses PURITY in great detail. The subsections encompass a broad description of the individual parts of the tool. Section IV demonstrates the functionality of the tool on an example. Finally, Section V concludes the work and provides further discussion.

II. Related Work

The work that preceded PURITY is based either on model-based based testing or planning. In general, for the first case models of the SUT are used in order to generate test cases. In turn, the test results are compared to expected values. Works that deal with general issues of model-based testing include [13] and [14].

As mentioned before, our previously works (among others [10], [11]) dealt with the notion of attack patterns, i.e. graphical representations of an attack that represent an abstract test case. The way the program communicates, i.e. executes tests against the SUT, and the vulnerability detection mechanisms were implemented into our tool. We also applied some methods for data manipulation as well.

However, PURITY puts a greater focus on the technique proposed in [12]. Although planning is often used in robotics, we applied this method on security testing. A planner generates a sequence of abstract actions. On the contrary, for every action there is a corresponding method with concrete values. In such way, concrete test cases are executed accordingly to the abstract plan. The detailed description is given below in the next section.

Planning has been applied to testing in other works as well [15], [16], [17]. But in contrast to these works, we automate the plan generation process for SQLI and XSS in web applications and offer a manual interface as well.

The authors from [18] propose the Pattern-driven and Model-based Vulnerability Testing approach (PMVT) for detecting of XSS. The method generates tests for this purpose by using generic vulnerability test patterns and behavioral models of a specific web application. On the contrary, our work does not rely on such models and test purposes but uses different test case generation techniques.

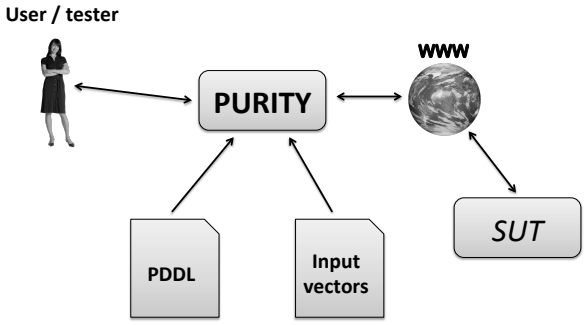

Figure 1. PURITY in context with its environment

Works that apply threat representations are [19] and [20]. In the first paper the authors adapt Petri nets in order to depict formal threat models. These are represented as Predicate/Transition nets and depict a way how to breach a security leak inside a SUT. Here test cases are created automatically from these models by creating code by relying on the Model-Implementation specification. The second work deals with test case generation from Threat Trees. In this approach, valid as well as invalid inputs are created from threat modeling. Although both approaches share similarities with the generated attack steps in our tool, we do not rely on a graphical representation but use PDDL for test case generation. Also, mapping to concrete test cases is done separately from the above process.

A detailed overview about SQLI and XSS can be found in [21] and [22], respectively.

III. PURITY - A Planning Based Security Testing Tool

PURITY is intended to be used for testing web applications. In Figure 1 we depict PURITY in context with its surrounding environment. The tool takes input from the user like the www address of the application, planning definition files (PDDL), which define the initial state and potential attack actions, and potential concrete attack vectors used when testing the application. PURITY generates plans from which concrete test cases to be submitted to the system under test (SUT). PURITY analyses the received feedback from the SUT in order to detect a vulnerable behavior.

PURITY encompasses several elements that interact with each other as well as with the user, which we later describe in more detail. It offers additional possibilities for the tester to define test parameters like the type of attack, the used attack actions, the test data etc. The tests can be carried

out both manually and automatically. Accordingly to the implemented test oracle, the program gives a verdict whether the vector succeeded in triggering a vulnerability. Also, the corresponding tested element is shown to the tester so he or she gets a visual expression of the output.

In fact, the tool offers a great deal of configurability with regards to the implemented technology. The tester can interact with the program on a minimum scale, i.e. by setting only the initial configuration like URL address. On the contrary, a test can be carried out completely manually by assigning specific values to selected parts of the website.

In the following we briefly describe the underlying techniques and the internal architecture of PURITY.

A. Background Techniques

*1) **Attack Pattern-Based Testing**:* The first approach is discussed in detail in the works mentioned in the related research. Until now it has been enriched by applying a test case generation technique from the field of combinatorial testing [11]. Basically it's a test case execution technique that is found upon patterns of attacks like SQLI and XSS. As is the case with PURITY, the approach is highly configurable by the user and can be connected to other techniques as well.

The approach executes accordingly to the specified UML statechart of the attack against the SUT. Such a path through the model serves as an abstract test case, whereas concrete methods and variables are implemented in Java and can be called from the model during execution. It relies on HttpClient[1] for testing and reading of HTTP messages that carry the malicious vector. After targeting an element of the website, the attack is executed and the response is parsed by jsoup[2]. Finally, the program gives a diagnosis about the test and resumes the testing process.

*2) **Planning as a Testing Problem**:* Although planning has already been considered for testing, we introduced an algorithm and technique for its adaptation in security testing of websites. It should be mentioned that this approach plays an important part in the PURITY as well.

In fact, testing can be viewed as a sequence of actions that starts in an initial state and ends after the test verdict. First, the domain and problem files have to be specified in the standard Planning Domain Definition Language (PDDL). The problem is defined by application specific variables or values while the domain encompasses problem independent action definitions. Every action is specified by its set of applied variables and pre- and postconditions. In case those certain preconditions of an action are satisfied, this action is picked from the list and its effects are triggered during execution. Now, the new values may serve as a new precondition for other actions and so on.

It is the task of the planner to automatize this process and, if possible, to generate a plan. In fact, a plan resembles an attack from the abstract point of view. For that case we rely on the planning system Metric-FF [23]. PURITY reads the plan. It has corresponding concrete methods and values for every action from the domain. Then the execution proceeds accordingly to the plan by dynamically calculating new effects from the actions' preconditions. If the concrete tests follow the abstract ones during execution then we get a positive test verdict in the end. On the concrete level this means that an expected output was obtained from the SUT after submitting some attack vector. Otherwise a new plan is generated and the process starts again. The whole process follows our PLAN4SEC algorithm [12], which extended version is also a part of PURITY.

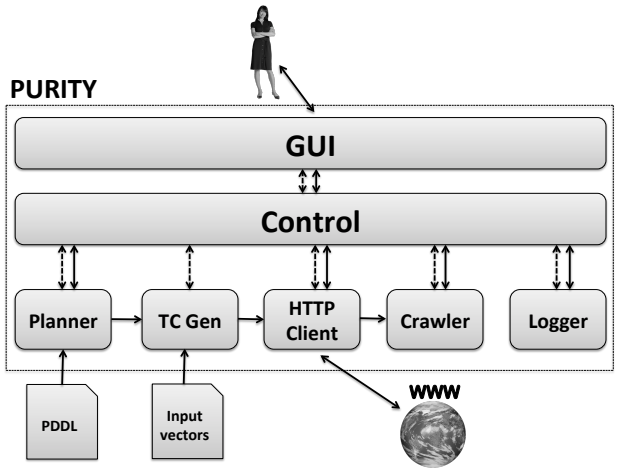

Figure 2. PURITY's internal software architecture

B. Key Components

PURITY encompasses a variety of different components. These have been implemented using different Java libraries as well as other external programs. The most important ones are the following:

- **Metric-FF**: A planning system that is based on a forward chaining planner. It handles PDDL files, thus producing a plan.
- **JavaFF** ([24]): A Java implementation of FF [25] that parses PDDL files and incorporates a planner. However, PURITY makes only use of the parser in order to extract components like objects, initial values, actions etc. from both the problem and domain definitions.
- **Crawler4j**[3]: An open source Web crawler that offers the possibility to define the crawl depth, number of pages to fetch etc.
- **HttpClient**: Implements the client side of the HTTP standard. Used by the program in order to communicate

[1]http://hc.apache.org/httpcomponents-client-ga/

[2]http://jsoup.org/

[3]https://github.com/yasserg/crawler4j

over HTTP with the SUT. Attack vectors are submitted as parts of the request after which the response is read from the tested system.

- **jsoup**: A Java based HTML parser. It is used in order to parse the HTTP response in search for critical data after the attack. A test verdict is given according to that information.
- **Test oracles**: They are implemented inside PURITY. Detection mechanisms for both SQLI and XSS are discussed in detail in the authors' related works. However, it is important to understand how SQLI works in order to know why an expected value is asked for in PURITY's GUI. The general fact is that after a malicious vectors have been submitted, the outcome is always hard to predict. In this case the tester is asked to specify a unique value which will be searched for in the HTTP response after the attack occurs. For example, the tester might already know (e.g. by social engineering) the username of a victim. In that case, this could be specified as an indicator value. On the other hand, the detection of both types of XSS is handled automatically without necessary feedback specification.

Figure 2 depicts the internal software architecture of our tool. All interaction between tester and PURITY proceeds over the GUI. This represents the front end from where the entire functionality can be accessed. However, below that layer the implementation is responsible for the data flow between the individual components and the user.

A web application is accessed either over the World Wide Web or locally, wherever it might be deployed. The URL acts as the starting point once the testing process is started. The communication between PURITY and SUT is handled dynamically by HttpClient. It creates and sends HTTP requests for opening the communication channel in the first place. However, it also injects attack vectors into parameters so it acts as the attacker's very back end. Although it is a separate entity, HttpClient is used in combination with jsoup in order to parse HTML data from the response. The most critical data are user input elements from the website and incoming messages.

The Web crawler browses the SUT and identifies hyperlinks in websites that are connected to the initial URL. It takes the submitted URL as a starting seed and eventually returns all ongoing addresses. It should be mentioned that the tester can restrict the crawl depth and define a maximum number of pages to fetch. During test execution, all incoming data from the crawler is submitted directly to HttpClient.

However, concrete inputs are needed for a test case. PURITY encompasses two initial test sets, one with SQL injections and the other containing XSS vectors. New ones can be obtained externally by attaching them to our tool. During execution, these TXT files are read line by line and sent to HttpClient, which puts them inside HTTP requests as well. If the tester wants to create new input files, he should take care of the data structure for both SQL and JavaScript. Otherwise the data will be sent to the SUT anyway; however no meaningful results would be obtained.

It is very important to note that until now all of the described components from Figure 2 work as parts of the Java implementation. This means that concrete test cases are built automatically from the current URL address, web component data and attack vectors by the test case generator.

On the other hand however, abstract test cases are created by the planner. These inputs contain data with almost no concrete values like the ones mentioned above. Generally speaking, automated planning and scheduling is based on propositional and first-order logic. As will be explained below, the planning language PDDL is used in order to specify objects, predicates etc. in order to construct entities with preconditions and effects. These are called actions and are saved with other data in two PDDL files. Given that initial values and an initial state are specified in that files, as well as a goal description, the planner searches for actions that lead from the initial state to the goal. The resulting sequence is called a plan.

However, this plan cannot be used for testing purposes unless there are concrete values that somehow correspond to abstract values from the planning domain. Therefore we have implemented action definitions (in fact Java methods) in PURITY that fulfill this purpose. The test case generator reads the abstract actions and searches for their concrete counterpart in the implementation. Once found, it is executed. One advantage of this approach is the fact that for one abstract object from an action we can apply a dozen of concrete attack vectors. For example, the implementation picks one attack vector from the input files and applies it to a variable in the implementation. After the plan is executed on the concrete level, PURITY reads the next vector and repeats the plan execution with the new value.

The implementation calls the planner by submitting the two PDDL files. However, PURITY also generates new files of this kind. Since every plan is constructed according to a specific data configuration, a different configuration would also result in a different plan. Exactly that is what PURITY focuses on: It creates new problem definitions with somehow different initial values. Now Metric-FF delivers a new plan that is parsed by PURITY, which carries out the concrete execution. The way PURITY creates new PDDL files will be elaborated further in the paper.

Generally speaking, the implementation generates concrete data accordingly to abstract ones. On the other hand, it also creates abstract data that is meant to be processed by the planner. The planner produces in turn new abstract data for the implementation. This is a cyclic process that continues as long as plans are generated and attack vectors are available.

Finally, the logger collects all relevant data produced during the execution. The tester has the choice whether he

wants to log all events during the execution or just critical messages like exceptions.

C. PLAN4SEC 2.0

Here we discuss the extended algorithm behind the automated execution in PURITY. PLAN4SEC was already introduced in [12]. In this paper we extend the approach in order to cover additional functionality. The improved PLAN4SEC 2.0 is depicted in Algorithm 1.

As was the case with the initial version of the algorithm, it relies on data from the PDDL files and concrete values. It also makes use of the domain specification, attack type information, HTML method variables and a function that maps actions from the plan to corresponding Java methods.

The final output is a table with all attack vectors and SUT parameter values that lead to a vulnerability breach. The corresponding function res reports $FAIL$ whenever a test triggers a vulnerability, whereas $PASS$ is thrown otherwise.

The main improvement of PLAN4SEC 2.0 is a dynamic PDDL generation, crawler consideration and processing of new outputs during the execution. The last point represents information that cannot be foreseen before the testing starts. However, it is applied dynamically into the testing process.

The idea behind this algorithm is the following one. For every URL address the program parses user input elements from the website as well as the current initial values from the problem's PDDL. Additionally it initializes the crawler, which in time returns all hyperlinks from the website in form of URLs. Now the program checks whether HTML elements have been encountered during the parsing of the website in step 4 (E). As mentioned before, these are the input fields where the user is supposed to interact with the SUT. The goal is to test every of these elements separately before continuing the execution. Since these values cannot be known at the beginning, the program has to identify them for every incoming URL. Now the planner returns the first sequence of actions from the domain and problem files (step 6). Afterwards the first attack vector is picked from the input files.

The function Φ takes as arguments the abstract action (a) from the plan and maps it to its concrete counterpart in Java (c).

During plan execution, the test case generator assigns the attack vector (x') to one of the HTML inputs (e') from the website. Afterwards, when the plan execution terminates, the program still remains in the loop of that vector but assigns it now to another HTML element in E and repeats the plan execution again from the beginning (steps 10-20). Now, generated abstract actions are read one by one from the saved plan. PURITY traverses through all concrete Java methods in order to find the corresponding action implementation (step 12). When encountered, it is executed and eventually generates new values.

Algorithm 1 PLAN4SEC 2.0 – Improved plan generation and execution algorithm

Input: Domain D, set of problem files $P = \{\mathtt{p}_0, \ldots, \mathtt{p}_n\}$, address URL, set of initial values $U = \{(\mathtt{t}, \mathtt{m}) | \mathtt{t} \in T, \mathtt{m} \in M\}$ with a set of attack types $T = \{\mathtt{t}_0, \ldots, \mathtt{t}_n\}$ and set of HTTP methods $M = \{\mathtt{m0}, \ldots, \mathtt{m}_n\}$, set of attack vectors $X = \{\mathtt{x}_0, \ldots, \mathtt{x}_n\}$, set of concrete actions $C = \{\mathtt{c}_0, \ldots, \mathtt{c}_n\}$ and a function $\Phi = a \mapsto c$ that maps abstract actions to concrete ones.
Output: Set of plans $PL = \{\mathtt{A}_0, \ldots, \mathtt{A}_n\}$ where each $A_i = \{\mathtt{a}_0, \ldots, \mathtt{a}_n\}$, set of HTML elements $E = \{\mathtt{e}_0, \ldots, \mathtt{e}_n\}$ and a table with positive test verdicts V.

1: $PL = \emptyset$
2: **for** $SELECT\ URL$, X, C, U, $p \in P$, D **do**
3: **while** $URL.hasNext()$ **do**
4: $E = \mathbf{parse}(URL)$ ▷ Identify user input fields
5: **while** $U \neq \emptyset$ **do**
6: $A = \mathbf{makePlan}(p, D)$
7: $PL = PL \cup \{A\}$
8: $res(A) = FAIL$
9: **for** $x' \in X$ **do**
10: **for** $e' \in E$ **do**
11: **for** $a \in A$ **do** ▷ Execute plan
12: $a' = \mathbf{ConcreteAct}(a, \Phi, x', e')$
13: **if** $\mathbf{Exec}(a')$ fails **then**
14: $res(A) = PASS$
15: **else**
16: $res(A) = FAIL$
17: $V = V \cup res(A)$
18: **end if**
19: **end for**
20: **end for**
21: **end for**
22: $p = \mathbf{makePDDL}(U, p, D)$ ▷ New problem
23: $P = P \cup p$
24: **end while**
25: $URL = \mathbf{crawler.next}()$ ▷ Pick next URL
26: **end while**
27: **end for**
28: Return (V) as result

In such a way a plan is re-run for a certain address and a certain attack vector several times. Only after testing of all user input elements, the execution proceeds further. In case that no input elements are available, the program switches immediately to the next part. After the tests have been executed for all input elements, eventually a positive test verdict is saved into the table (step 17). A concrete example for this table is given in the case study section.

A major difference to the initial version of the algorithm is the fact that PURITY generates and executes several

problem definitions. In fact, a new PDDL file is generated dynamically after all attack vectors have been executed against one web page. For this case we have to take a look at the initial values of an individual problem definition. These specify the starting conditions for further plan generation. A sample of a few initial values is given below:

```
(:init
                (inInitial x)
                (Logged no)
                (not (statusinit two))
                (Type sqli)
                (= (sent se) 0)
                (not (Empty url))
                (GivenSQL sqli)
                (GivenXSS xssi)
                (Method post)
                (Response resp)
                (not (Found exp resp))
                (not (FoundScript script resp))
                ...
)
```

Initial values description in PDDL

The goal is to generate new problem files with different initial values so that different plans are generated as well. With a new sequence of actions the test execution will also differ from the previously plan. During execution the problem file is parsed in search for an already set initial value, which will be replaced by a new one from the corresponding data set from U. It should be mentioned that the same set of values is specified twice, once in the PDDL files and at the concrete level in Java.

For demonstration purposes we explain the change of two of the initial values, namely `Type` and `Method`. Here `Type` can have three values, namely `sqli`, `rxss` and `sxss` whereas `Method` encompasses only `get` and `post`. Both sets are implemented in the PDDL files as well in Java on the concrete side. If `sqli` was the initial value of `Type` in the first problem file then another will take its place, e.g. `rxss`. The program will keep the implementation of the current problem (p) but will replace its current value (e.g. `sqli`) with a new one (`rxss`) in step 22. Then, a new PDDL file is saved (step 23) with this specification and marked as next in line for procession. The plan generation is invoked again as well as the attacking sequence.

However, one important attribute of PURITY is that a method for the generation of initial values is directly invoked from another method of the same kind. This means that for every value of `Type` we obtain several files with different initials. After all of them have been executed, we generate new files for a new value of `Method`. For example, for three values of `Type` and two different values of `Method` we generate a total of six PDDL files, which results in six plans and attack executions. Theoretically, by taking the values of one more initial predicate, e.g. `(inInitial x)` with 22 possible values for `x`, a sum of $(3 \times 2 \times 22 =)$ 132 problem files can be generated and so on. It is important to note that for every abstract value the corresponding concrete values have to be set as well, e.g. if the method from the plan is `post` then the website will be tested only with that method despite the fact that `get` might be its default value. But since there has to be at least one file where `(Method get)` is specified, this will be executed as well.

Actually, in this way we hope to trigger unwanted behavior of the SUT and test whether this might lead to a security breach as well. The more initial values are specified and manipulated, the more different tests are carried out for this sake. This PDDL generation process will continue as long as all combinations of objects from U are executed.

In fact, this principle can be applied to every initial value so a huge number of test cases are generated. Of course, the last method would call no other because all value combinations would have been already executed. However, it remains the task of the programmer to implement new generation methods.

The entire testing process will last as long as the crawler returns new hyperlinks. Afterwards, the execution terminates permanently.

D. Structure of Inputs in PURITY

As mentioned before in the paper, one of the primary motivations for this tool was to ease the effort for the tester to effectively test a program. For this case, the amount of interaction is kept as small as possible. For instance, it is completely sufficient just to give the URL address of the SUT and click a button in order to start the testing process. The rest will be handled by the program automatically. This fact increases the usability of the tool while keeping the execution time relatively low. On the other hand the tester might want to know the functionality behind PURITY and interacts with the system. If this is the case, the tool offers several possibilities to realize that. However, first we have to describe what types of user inputs are used in PURITY:

Type of attack: Can be either SQLI or XSS. According to this choice, the program expects different attack vectors but also applies different test oracles. Since previously works with attack patterns have proven that reflected and stored XSS can be detected the same way, we don't make any distinction. However if the tool is run completely automatically, the SUT will be tested for both vulnerabilities.

Attack vectors: A text file is attached to the PURITY and read line by line. Every row should contain one vector that resembles either one SQL query or JavaScript code. Since our tool is meant for testing purposes, a harmful vector for SQLI can have a structure like:

```
a' UNION ALL SELECT 1, @@version;#'
```

On the other hand, an input for XSS could reflect if it looks like:

```
<iframe src="http://www.orf.at"></iframe>
```

In both cases the vector is assigned to a variable and sent via request over HTTP to the SUT. The tester can attach TXT files with attack vectors to PURITY before the testing process starts. Otherwise the tool will make use of two files that are already included.

Domain.pddl: This file is already attached to the tool and it encompasses predicate and function definitions as well as actions. Once set, the domain file will remain unchanged during automated test execution, i.e. all further plans will be constructed according to the same definition.

Problem.pddl: The problem file is part of the tool and specifies objects and initial values. During execution these values will be replaced with new ones, thus continuously creating new problem files. Every generated plan will be derived from a different problem specification.

In the paper we will use the symbolic names `domain.pddl` and `problem.pddl` for both specifications regardless of the files' name.

E. Modes of Use

After starting PURITY, the tester has the choice between four different test execution modes. Figure 3 depicts the GUI of the tool.

The minimum requirement for every one of them is to specify the URL address of the SUT. The initial values for the crawler are initially set to -1 for both the crawl depth and number of pages to fetch. These values decide about how deep the crawler will go into the application by starting from the original URL and how many pages will be fetched during that search (-1 stands for unlimited.). In the following we briefly describe each mode of use PURITY offers.

1) ***Completely Automatic****:* This mode is the most extensive one because it performs the execution in a completely automated manner. It will be picked per default if the checkbox `auto-generate plans` is selected.

If he desires, the tester can load his own input files into the tool before pushing the start button. Since this mode tests automatically for both SQLI and XSS, he can use test sets for both vulnerabilities. The user can edit the initial domain specification and delete actions in a simple editor if desired. In such way fewer actions are taken into consideration by Metric-FF so the plans will get simpler as well. From now on the reduced domain file will remain unchanged during the entire testing process.

When the execution starts, PURITY submits the initial PDDL files to Metric-FF that in turn generates the first abstract test case. If no plan could be generated, the user will be notified. From now on the procedure follows PLAN4SEC 2.0 as long as attack vectors are available and plans are generated. An example of this type of execution is demonstrated in Section IV.

Of all modes, this one covers most of the functionality of PURITY and demonstrates the adaptation of planning in testing at its best.

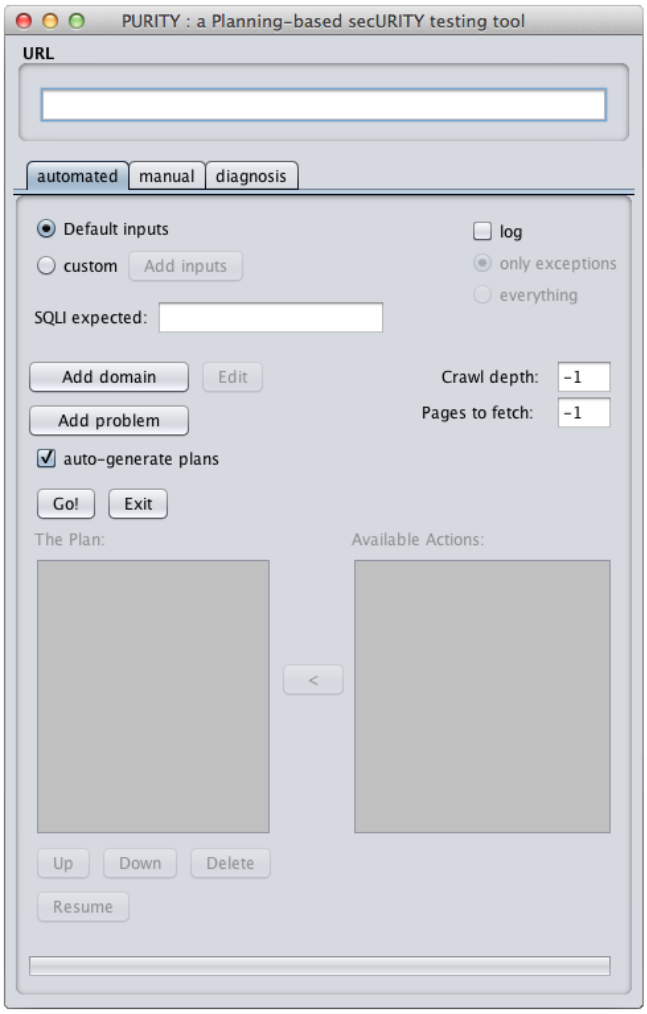

Figure 3. PURITY'S GUI

2) ***Partly Automatic****:* This selection relies on testing as well but to a much lesser extent. In this case the tester generates just one plan, which actions are displayed in a separate window. Additionally, all available actions are parsed from `domain.pddl` and displayed as well. Now the tester can make experiments by deleting and adding actions or changing the order of their appearance.

The new list will be sent to the test case generator and carried out automatically as would be the usual case. However, the difference to the completely automated approach lies in the fact that this time a plan is executed only once. To be precise, only one execution is carried out per attack vector. Figure 4 depicts the section that contains the generated plan. As can be seen, plan actions can be either removed or added from the menu.

Actually this mode is meant for the tester to manipulate planning related data and to check the corresponding effects.

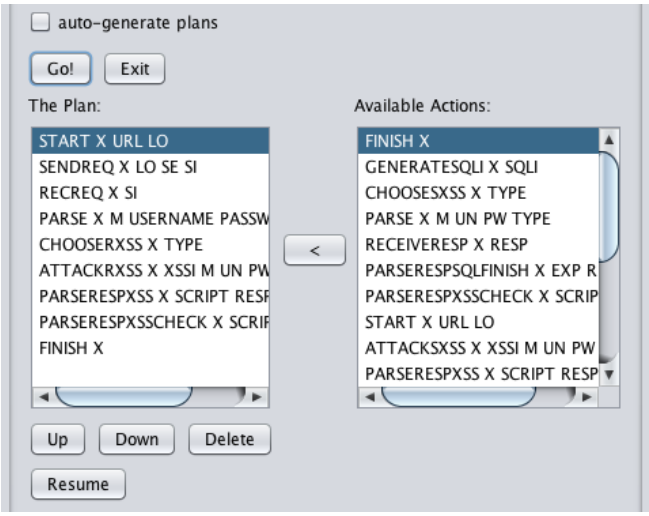

Figure 4. Section for partly automated testing

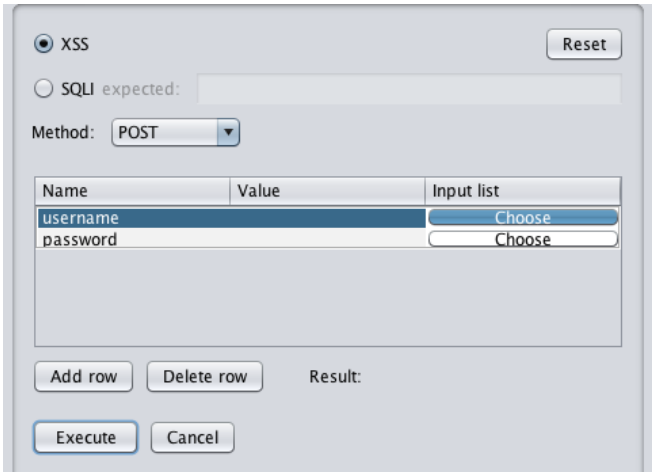

Figure 5. Menu for manual testing

*3) **Completely Manual**:* With this choice the tester can test a single website by manually writing values for all its user input fields. HttpClient parses the page from the specified URL and displays all HTML elements that could be tested. Now the user can add or remove parameters if he wishes. For example, sometimes it is proven to be useful to submit one parameter twice in a request, e.g. by submitting `username=Ben&username=[malicious script]`. This configuration can be defined manually in the table. In order to realize this, the tester adds a row in the elements table and writes the name of the parameter and its value. Afterwards he might initialize the testing process. However, there will be no result table displayed since the user has a clear insight what parts of the SUT are tested with a known attack vector. Since no planning and crawler are used and no test files are attached, only one test case will be executed per attack. Figure 5 shows the manual testing section. All extracted HTML elements are shown in the table and concrete values are added in the cells from the corresponding column.

In fact, this functionality and the following one define PURITY as a manual testing tool as well.

*4) **Partly Manual**:* If user input fields are encountered during parsing of a website in the completely manual mode, the tester is also offered the possibility to test one specific element against a list of vectors in an automated manner. In order to accomplish this, a button is located in the table beside the field that is wished to be tested. This opens a file chooser where one or more vector files can be selected. After the desired vulnerability is checked as well, the testing process can be started. Now the desired website's input element will be tested automatically against all vectors from the input file(s).

IV. Case Study

We demonstrate the functionality of PURITY by testing one of the SUT's from [12] by choosing the automated mode.

As can be seen, the user is asked to specify an URL, the input data sets, PDDL files and the specifications for the crawler. In order to select the completely automated mode, the tester has to select the corresponding checkbox. Otherwise the program chooses the partly automated mode. The tester can edit the domain specification if he wishes or add several test input files.

Once the specification is selected, testing can be started. In this example the URL represents the local address where the SUT has been deployed. First, the planner is called with the selected PDDL's and the corresponding output is read. As explained above, the planning system will return a plan to the implementation which will start immediately the concrete test execution. First the initial values of the current `problem.pddl` are parsed in search for critical objects, like `type` or `method`. The configuration is saved and the execution continues by fetching the first vector from the TXT file as explained in the PLAN4SEC section. Since it may be hard to predict what hyperlinks are connected to the initial URL, we can expect to encounter sites with no input fields at all. In this case the current test run is terminated and the next one starts.

After all tests have been executed for the initial URL, the crawler fetches the next ongoing address. For example, in the above demonstration the new address would be `http://localhost:8080/bodgeit/home.jsp`. The concrete test execution will now continue by parsing a new problem definition after which the vector files will be read from the beginning and so on.

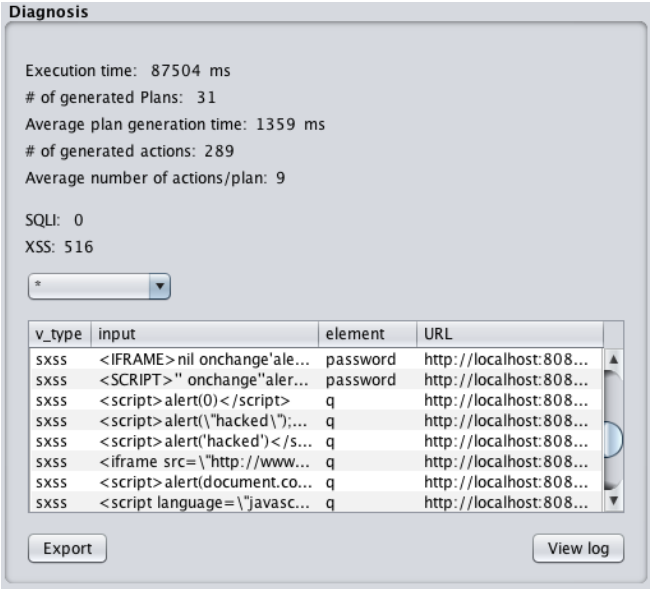

Figure 6. Output table

After execution, all positive test verdicts are displayed in the table. Figure 6 depicts such a table where every row contains the type of triggered vulnerability, the responsive attack vector, name of the vulnerable HTML element and the corresponding URL address. The table can be exported as an Excel sheet. Besides that, the diagnosis window also shows some statistical data like the generation and execution time for all tests, the number of generated plans and actions etc. Also, the total amount of successful tests for both SQLI and XSS is shown. The reasons behind the high number of XSS vectors lies in the fact that HTML input elements were very vulnerable for the tested SUT. This means that a high number of submitted vectors were able to exploit the security leaks. A more detailed explanation for such results when testing for XSS is given in the evaluation in [11].

On the other side, not a single SQLI leak was detected. The reason for this is either that the expected value hasn't been the right choice or that software intern filtering mechanisms were efficient enough to escape the malicious code in the first place. Another reason for a failure is usually the fact that a website does not use a database or no user input fields are available.

In [12] we have evaluated our initial planning based testing approach against some web applications. The obtained results have proven that the new concept could be built on further. In this work we added several new features and made a unique planning based testing framework. In the future research section we discuss further possibilities that could be realized in the future.

V. Conclusion and Future Work

In this paper we presented our planning-based security testing tool PURITY. Its main purpose is to test web applications for SQL injections and cross-site scripting. The tester can use it on black- or white-box basis. PURITY encompasses a novel test case generation technique that is based on automated plan generation. On the other hand, the tester is also offered the possibility to execute the tool in a manual manner. He can set test parameters by himself and see how the SUT reacts on different inputs. The test results are represented in form of a table that provides a visual impression of where the vulnerability has occurred.

Although a research prototype, PURITY succeeded in testing of several web applications. It offers a high degree of compatibility, which is demonstrated in the tool description above. However, we still want to improve it further by including additional features, e.g. adding more actions into the domain specification and increasing the configurability by allowing more manual intervention.

PURITY can be experimented with so even new test scenarios can be adapted. Although it does not represent a demand, the tester can add his own attack vectors or manipulate existing ones. As was demonstrated in [12], the execution time is relatively low when applying the plan generation testing technique. In the future we plan to include some of the mentioned possibilities into PURITY and improve the already existing functionality.

Acknowledgement

The research presented in the paper has been funded in part by the Austrian Research Promotion Agency (FFG) under grant 832185 (MOdel-Based SEcurity Testing In Practice).

References

[1] "Owasp top ten project," http://owasp.com/index.php/Category:OWASP_Top_Ten_Project, accessed: 2015-02-10.

[2] "Webscarab," https://www.owasp.org/index.php/Webscarab, accessed: 2015-02-10.

[3] "Burp suite," http://portswigger.net/burp/, accessed: 2014-01-28.

[4] "Zed attack proxy (zap)," https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project, accessed: 2015-02-10.

[5] "Xsser," http://xsser.sourceforge.net/, accessed: 2015-02-10.

[6] "sqlmap," http://sqlmap.org/, accessed: 2015-02-10.

[7] "Defensics," http://www.codenomicon.com/products/defensics/, accessed: 2015-02-10.

[8] F. Duchene, S. Rawat, J.-L. Richier, and R. Groz, "Kameleon-Fuzz: Evolutionary Fuzzing for Black-Box XSS Detection," in *CODASPY*. ACM, 2014, pp. 37–48.

[9] B. Garn, I. Kapsalis, D. E. Simos, and S. Winkler, "On the applicability of combinatorial testing to web application security testing: A case study," in *Proceedings of the 2nd International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation (JAMAICA'14)*. ACM, 2014.

[10] J. Bozic and F. Wotawa, "Security testing based on attack patterns," in *Proceedings of the 5th International Workshop on Security Testing (SECTEST'14)*, 2014.

[11] J. Bozic, D. E. Simos, and F. Wotawa, "Attack pattern-based combinatorial testing," in *Proceedings of the 9th International Workshop on Automation of Software Test (AST'14)*, 2014.

[12] J. Bozic and F. Wotawa, "Plan it! automated security testing based on planning," in *Proceedings of the 26th IFIP WG 6.1 International Conference (ICTSS'14)*, September 2014, pp. 48–62.

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

[14] I. Schieferdecker, J. Grossmann, and M. Schneider, "Model-based security testing," in *Proceedings of the Model-Based Testing Workshop at ETAPS 2012. EPTCS*, 2012, pp. 1–12.

[15] A. Leitner and R. Bloem, "Automatic testing through planning," Technische Universität Graz, Institute for Software Technology, Tech. Rep., 2005.

[16] S. J. Galler, C. Zehentner, and F. Wotawa, "Aiana: An ai planning system for test data generation," in *1st Workshop on Testing Object-Oriented Software Systems*, 2010, pp. 30–37.

[17] M. Schnelte and B. Gldali, "Test case generation for visual contracts using ai planning," in *INFORMATIK 2010, Beitr.ge der 40. Jahrestagung der Gesellschaft fr Informatik e.V. (GI)*, 2010, pp. 369–374.

[18] A. Vernotte, F. Dadeau, F. Lebeau, B. Legeard, F. Peureux, and F. Piat, "Efficient detection of multi-step cross-site scripting vulnerabilities," in *Proceedings of the 10th International Conference on Information System Security (ICISS'14)*, 2014, pp. 358–377.

[19] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska, and W. Xu, "Automated security test generation with formal threat models," in *IEEE Transactions on Dependable and Secure Computing 9 (4)*, 2012, pp. 526–540.

[20] A. Marback, H. Do, K. He, S. Kondamarri, and D. Xu, "Security test generation using threat trees," in *Proceedings of the ICSE Workshop on Automation of Software Test (AST'09)*, 2009, pp. 62–69.

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

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

[23] "Metric-ff," http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html, accessed: 2015-02-10.

[24] "Javaff," http://www.inf.kcl.ac.uk/staff/andrew/JavaFF/, accessed: 2014-01-28.

[25] J. Hoffmann and B. Nebel, "The ff planning system: Fast plan generation through heuristic search," in *Journal of Artificial Intelligence Research 14*, 2001, pp. 253–302.