Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing

Andrey Petukhov, Dmitry Kozlov

Computing Systems Lab, Department of Computer Science, Moscow State University
petand@lvk.cs.msu.su, ddk@cs.msu.su

Abstract. The number of reported web application vulnerabilities is increasing dramatically. The most of vulnerabilities result from improper input validation. This paper presents extensions to the Tainted Mode model which allows inter-module vulnerabilities detection. Besides, this paper presents a new approach to vulnerability analysis which incorporates advantages of penetration testing and dynamic analysis. This approach effectively utilizes the extended Tainted Mode model.

Keywords: Web Applications, Second-order injection, Vulnerability Analysis, Penetration Testing, Dynamic Analysis, Taint Analysis.

1 Introduction

Security vulnerabilities in web applications may result in stealing of confidential data, breaking of data integrity or affect web application availability. Thus the task of securing web applications is one of the most urgent for now: according to Acunetix survey [1] 60% of found vulnerabilities affect web applications. The most common way of securing web applications is searching and eliminating vulnerabilities therein. Examples of another ways of securing web application include safe development [38-40], implementing intrusion detection and/or protection systems [32, 35-36], and web application firewalls [37].

According to OWASP [8], the most efficient way of finding security vulnerabilities in web applications is manual code review. This technique is very time-consuming, requires expert skills, and is prone to overlooked errors. Therefore, security society actively develops automated approaches to finding security vulnerabilities. These approaches can be divided into two wide categories: black-box and white-box testing.

The first approach is based on web application analysis from the user side, assuming that source code of an application is not available [2]. The idea is to submit various malicious patterns (implementing for example SQL injection or cross-site scripting attacks) into web application forms and to analyze its output thereafter. If any application errors are observed an assumption of possible vulnerability is made. This approach does not guarantee neither accuracy nor completeness of the obtained results.

The second approach is based on web application analysis from the server side, with assumption that source code of the application is available. In this case dynamic or static analysis techniques can be applied. A comprehensive survey of these techniques was made [34] by Vigna et al. According to this survey several statements could be made:

- The most common model of input validation vulnerabilities is the Tainted Mode model [3]. This model was implemented both by means of static [7, 9, 15] or dynamic analysis [3-6].
- Another approach to model input validation vulnerabilities is to model syntactic structure for sensitive operations arguments. The idea behind this is that the web application is susceptible to an injection attack, if syntactic structure for sensitive operation arguments depends on the user input. This approach was implemented by means of string analysis in static [16, 17] and it was applied to detect SQLI [19] and XSS [18] vulnerabilities in PHP. After all, this approach was implemented to detect injection attacks at runtime [32, 36].
- One of the main drawbacks of static analysis in general is its susceptibility to false positives [20] caused by inevitable analysis imprecisions. This is made worse by dynamic nature of scripting languages. However, static analysis techniques normally perform conservative analysis that considers every possible control path.
- One of the contrary, one of the main drawbacks of dynamic analysis is that it is performed on executed paths and does not give any guarantee about paths not covered during a given execution. However, dynamic analysis having access to internals of web application execution process has the potential of being more precise.

In this paper we focus on the most common model of input validation vulnerabilities. First, we identify several drawbacks of this model. For instance, we analyze why this model can not be used to detect inter-module vulnerabilities, which make second order injection attacks possible. Then, we propose solution to the stated drawbacks.

The contributions of this paper are the following:

- We improve classical Tainted Mode model so that inter-module data flows could be checked.
- We introduce a new approach to automatic penetration testing by leveraging it with information obtained from dynamic analysis. Thus:
    - More accuracy and precision is achieved due to widened scope of web application view (database scheme and contents, intra-module data flows);
    - Input validation routines can be tested for correctness.

2 Case Study

This section describes disadvantages of classical Tainted Mode model. Description of each disadvantage is followed by an example that demonstrates how accuracy and/or completeness of analysis are affected by this particular disadvantage.

2.1 Tainted Mode Model

Dynamic and static analysis uses Tainted Mode model for finding security vulnerabilities that cause improper input validation. This model was implemented by means of static analysis for PHP [7, 15] and Java [9] technologies and by means of dynamic analysis for Perl [3], Ruby [4], PHP [5], and Java [6]. In our previous work [21, 22] we considered implementation of Tainted Mode approach for Python technology as a dynamic analysis module.

According to [26], following assumptions were made within Tainted Mode model:

1. All data received from the client via HTTP-requests is untrustworthy (or tainted).
2. All data being local to the web application is trustworthy (or untainted).
3. Any untrustworthy data can be made trustworthy by special kinds of processing, named sanitization.

With these assumptions made, security vulnerability is defined as a violation of any of the following rules:

1. Untrustworthy (tainted) data should not be used in construction of HTTP responses. This prevents cross site scripting attacks.
2. Untrustworthy (tainted) data should not be saved to local storages. This prevents possible construction of HTTP responses from these sources in future.
3. Untrustworthy (tainted) data should not be used in system calls and in construction of commands to external services such as database, mail, LDAP, etc. This prevents most of injection attacks.
4. Untrustworthy (tainted) data should not be used in construction of commands that would be passed as input to interpreter. This prevents script code injection attacks.

Having introduced the definition of input validation vulnerability we can now describe several drawbacks.

Disadvantage №1. The first drawback is that any sanitization routine removes tainted flag from the string value regardless of the critical statement that uses this value thereafter. For example, function that encodes HTML special characters into entities (e.g. '<' and '>' to '<' and '>') and is suitable for sanitizing data before HTTP response construction, will not sanitize data used in SQL queries construction. This may result in overlooked SQL injection vulnerability. This drawback decreases the completeness of analysis (the vulnerability exists but is overlooked).

Disadvantage №2. The second drawback is inability to handle input validation that is organized as conditional branching like in the example below (pseudo-code is used):

```
input = get_HTTP_param('username')
filter = re.compile('some_correct_input_filter')
if  (re.match(input, filter)) {
          // return error page
} else {
          // do things
}
```

No escaping of bad symbols is performed. Instead, if bad input patterns are detected, a special (that is commented "*return error page*") program branch is executed. Application of a classical tainted mode approach to this code example will lead

to detection of insufficient input sanitization and erroneous vulnerability report will be generated. Thus accuracy of analysis is affected (the detected vulnerability does not exist). The main problem in this case is to detect which of the branches is error handling path. It should be mentioned that this decision cannot be based on the statement and its return value solely. For instance, if regular expression in our example was negated by "!" symbol, "*return error page*" branch would be the second one, which is currently defined by "`else`" statement. In order to handle this issue in static, a string analysis should be applied.

Disadvantage №3. The third drawback is trust to input validation routines. Let's imagine that some web application is being developed within particular framework and developers try to apply Tainted Mode approach to detect input validation vulnerabilities. They fill in the list where sanitization routines from framework API are specified. But should any sanitization routine have an error, vulnerability of web application will not be reported due to total trust to configuration list. This drawback decreases the completeness of analysis (the vulnerability exists but is overlooked).

In [41] authors present a novel approach to the analysis of the sanitization process. They combine static and dynamic analysis techniques to identify faulty sanitization procedures that can be bypassed by an attacker. The approach first utilizes string analysis to build grammar representing a set of values used in a particular statement. Then dynamic analysis is applied to verify the results obtained by the first step. The goal of the second step is to reduce false positives.

Disadvantage №4. The last but not the least drawback lies in the assumption that "all data being local to the web application is trustworthy", which implies the rule of "untrustworthy (tainted) data should not be saved to local storages". The inability to check data taintedness through database is the main disadvantage of the model, even if the first three cases were fixed. Indeed, what sanitization routines should be applied to the data which is inserted into the database besides SQLI escaping? Stored data could be used almost everywhere by other web application modules, resulting in XSS, SQLI, Shell code or script code injection vulnerabilities. Attacks that exploit such vulnerabilities are known as second order injection attacks [23, 24]. Figure 1 gives an example of vulnerability that allows second order XSS. According to the classic model, application on Figure 1 does not contain vulnerabilities.

Recently a model of inter-module input validation vulnerabilities was proposed [42] by Vigna et al. The authors introduce the notion of state entity which models any persistent storage (e.g. cookie, database) used by the web application to save data into. The set of state entities represents the web application extended state. Then, the notion of Module View is introduced. Each View represents all the state-equivalent execution paths in a single module. Every Module View is defined by a pre-condition, which consists of a predicate on the values of the state entities, post-conditions, which describes the changes made to the state entities by the View, and sensitive sinks, contained within the View. In addition to the set of the entity values, the extended state also keeps track of the current sanitization state of each entity. The approach assumes that sanitization routines are correct. The vulnerability exists, if a Module View has a sensitive sink containing unsanitized state entity. The approach was implemented by means of static analysis and has exponential complexity: in a module, the number of extracted views is exponential in the number of state-related conditional statements.

```
                    //shownews.cgi
1:  connection = MySQLdb.connect("localhost", "user", "pwd", "mydb")
2:  cursor = connection.cursor()
3:
4:  cursor.execute("SELECT title, text FROM news")
5:  while (row = cursor.fetchone()) do {
6:      print "<b>" + row["title"] + "</b><br>"
7:      print row["text"]
8:      row = cursor.fetchone()
9:  }
10: //cleanup omitted

                    //postnews.cgi
1:  title = Request.GET["title"]
2:  text = Request.GET["text"]
3:  title = escapeSQL(title)
4:  text = escapeSQL(title)
5:
6:  connection = MySQLdb.connect("localhost", "user", "pwd", "mydb")
7:  cursor = connection.cursor()
8:  cursor.execute("INSERT INTO news (title, text) \
9:        VALUES ('" + title + "', '" + text + "')")
10: //cleanup omitted
```

Fig. 1. Example of vulnerability that allows second order XSS. Modules are written in pseudo code. Module postnews.cgi stores input data into the database and therefore escapes SQL special characters. Module shownews.cgi displays new items entered into database. It assumes the data is trustworthy, so no checks are performed.

2.2 Penetration Testing

Penetration testing approach is based on simulation of attacks against web applications. Currently, penetration testing is implemented as black box testing. Thus the scope of analysis is limited to HTTP responses. Actually black box penetration testing is a four-step process:

1. The first step is to identify all pages being part of the web application. This phase is crucial for black box testing as attacks could be launched only against recognized application Data Entry Points [25]. This task can be fulfilled automatically (using web crawlers), manually (recorded by a proxy) or semi automatically (crawler asks for operator's assistance).
2. The second step is to extract Data Entry Points (DEPs) from pages visited in the fist step. The result is a set of DEPs to be analyzed.
3. The third step is simulation of attacks also known as fuzzing. Every parameter in every DEP is fuzzed with malicious patterns and used within an HTTP request to web application.
4. Finally every received HTTP response is scanned for indications of vulnerability.

The main disadvantage [27] of automated penetration testing tools is poor coverage of web application Data Entry Points. While the quality of web application penetra-

tion testing is in direct proportion to coverage percentage of the first phase in this paper we are disclosing some problems of the latter phases that can severely decrease the overall completeness of analysis.

Disadvantage №1. The main drawback of penetration testing is that its strait-forward implementation does not take into account the state of web application and its database scheme. Let's imagine an application with a user registration interface. Such interface normally consists of html form fields, including login name which should be unique. A generic scenario of handling such a case will include the following steps:

1. Get user input;
2. Validate user input;
3. Query if data specified by the user already exist in the database.
4. If true report an error.
5. If false insert data into the database.

Let's imagine that user registration form is going to be fuzzed with 'Organization' HTTP-parameter. Furthermore, 'Organization' name is output in 'view profile' page, but escaping of possible html tags is performed with an error thus permitting the second order XSS injection.

It's obvious that only first HTTP request would be processed normally. Every following request would generate an error page, thus preventing complete testing of 'Organization' parameter validation routine.

Disadvantage №2. The second disadvantage of penetration testing is its black box testing implementation: the outside user does not know what data flow paths web application data takes and can observe only HTTP responses. A good option may be to suppress all errors generated by common web application language runtime functions and to present generic error page which ensures that the error details are hidden from both intruder and security tester. Besides, such implementation does not permit to detect vulnerabilities that allow second order code injection.

3 The Improved Tainted Mode Model

In this section we shall begin with the formal definition of the classical Tainted Mode model, and then proceed to the formal definition for enhanced model. Finally, some notes on implementation of enhanced model by means of static and dynamic analysis are given.

3.1 Formal Definition of the Classical Tainted Mode Model

A web application is denoted as a map of request and the web application state to response, Data Dependency Graph and the new web application state:

$$W: Req \times State \rightarrow DDG \times Response \times State\ . \quad (1)$$

`Req` is HTTP-request, submitted to the web application. `State` in the left part refers to the web application state, which consists of contents of the web application environment (i.e. database, file system, LDAP, etc.). `Resp` denotes HTTP-response, re-

turned by the web application. DDG = (V, E) is Data Dependency Graph (DDG, also known as the program dependency graph [28]), which represents the path of execution and data flow, taken by the web application to process the given request. Finally, State in the right part is the state that the web application transitions into.

The Classical Tainted Mode model defines the following model of vulnerability:

$F: V \to \{input, filter, critical, common\}$ – a map of DDG vertices to their type that indicates what kind of action is performed: initialization of variables with untrustworthy data, sanitization of data, sensitive operation or anything else.

The vulnerability exists, if:

$$\exists\ path\ v_1 \ldots v_i \ldots v_k \in DDG:\\ F(v_i) = input,\ F(v_k) = critical,\\ \forall j:\ F(v_j) \neq filter\ \ 1 \le i < j < k$$

In other words, there's a path that untrustworthy data takes to get into the sensitive operation without sanitization.

3.2 Formal Definition of the Improved Tainted Mode Model

The first drawback of the classical model is that the critical and the sanitizing statements are treated without regards to class. This problem may be fixed by introduction of extra classes for the statements:

$FC = \{t_1, \ldots, t_k\}$ – classes of a sanitizing statement (e.g. html tags encoding, escaping of shell or sql special character, etc).

$CC = \{c_1, \ldots, c_m\}$ – classes of a critical statement (e.g. html output, SQL query, system call, string-as-code evaluation).

$IC = \{p_1, \ldots, p_l\}$ – classes of statements that mark data as tainted (e.g. getting HTTP parameters, getting unescaped data from database (see below)).

$Match: IC \times FC \times CC \to \{0, 1\}$ – table that matches every critical statement to the particular filter statement, suitable for the data sanitization (e.g. PHP addslashes to mysql_query), given an assigned tainted class.

Thus, every node in DDG will be mapped as follows:

$$G: V \to FC \cup CC \cup IC = \begin{cases} t_i \in FC, \text{if } F(V) = filter, \\ c_j \in CC, \text{if } F(V) = critical, \\ p_l \in IC, \text{if } F(V) = input, \\ \varnothing \ otherwise \end{cases} \quad (2)$$

Using this class division, vulnerability may be defined as follows:

$$\exists\ path\ v_1 \ldots v_i \ldots v_k \in DDG:\\ F(v_i) = input,\ F(v_k) = critical,\\ \forall j:\ F(v_j) \neq filter\ or\\ \forall j:\ F(v_j) = filter\ \ 1 \le i < j < k \Rightarrow Match(G(v_i), G(v_j), G(v_k)) = 0 \quad (3)$$

In other words, there's a path that untrustworthy data takes to get into the critical statement without corresponding sanitization.

The next step is to make the model capable of incorporating data flows through the web application database. In order to do so, the web application representation (1) is expanded to the following:

$$W: (Scheme, Req \times State \rightarrow DDG \times Resp \times \{query\} \times State) \quad (4)$$

In addition to Req, State, DDG and Resp, defined in (1), new entities Scheme and {query} are introduced. Scheme is a set of relational data schemes [10], which represents the web application database:

$$Scheme = \{R_1(A_{11}, \ldots, A_{1k_1}), \ldots, R_n(A_{n1}, \ldots, A_{nk_n})\} \quad . \quad (5)$$

{query} ∈ SQL is a set of database queries (that is subset of SQL language [11]), generated by the web application when processing certain request.

To make this model aware of inter-module data flows, we propose to interconnect the DDGs accessing the same fields in the web application database. First, we define the database access matrix based on the current database scheme (5):

$$M = \{1, \ldots, n\} \times \{1, \ldots, \max(k_i)\} \rightarrow \{0, r, w\} \quad . \quad (6)$$

The main purpose of this matrix is to identify columns and relations that might be affected by a particular query, and how. For example, if query accesses the second relation by selecting the second and the third attributes from it, the corresponding matrix elements would be M[2, 2] = r and M[2, 3] = r, while other elements of matrix would be zeroes.

The second step is to build a database access matrix for every query generated by the web application. It may be achieved by mapping elements from SQL language (i.e. queries) and the web application data scheme to access matrices:
$Q: SQL \times Scheme \rightarrow M: m_{ij} \neq 0 \Leftrightarrow$ SQL query might access column A_{ij} of the R_i relation. The most conservative way to build this map is to ignore WHERE clause as well as the actual result of a query.

The final step is to interconnect statements from distinct DDG using constructed map Q. Every variable in a particular DDG that accesses database column as M[i, j] = r should be linked to variables in other DDGs that access the same database column as M[i, j] = w. The resulting set of interconnected DDG allows analyzing data flows through database.

Let's refer to Figure 1. Statement row["title"] at shownews.cgi:6 gets its value from database field "news.title". This is a part of the shownews.cgi DDG. According to database access matrix (Figure 2), this field is written at postnews.cgi:9, variable title. A DDG for this variable depends upon "title" HTTP parameter at postnews:1. The two DDGs are interconnected. Thus, statement row["title"] at shownes.cgi:6 becomes dependent upon "title" HTTP parameter at postnews:1.

This is the place where some words about data re-tainting should be said. Let us consider the following example [24]. Imagine that there is a web application that inserts input data into database. The SQL special characters are correctly escaped. An-

other web application module gets this data from the database and uses it for construction of another SQL query. Obviously that data from the database is returned unescaped and should be marked as tainted again. This is modeled by `IC` class assignment to the corresponding DDG nodes. This is important caveat regarding second order SQL injections. It is not clear, if the model based on the extended application states [42] can handle the described issue.

Concerning implementation of this model some reasoning should be given. Static analysis can provide all possible DDGs. The task of building precise DDG for script languages, however, is a difficult task [13, 29]. Thus, conservative analysis of imprecise DDGs results in false positives. On the other hand, DDG can be automatically built by dynamic analysis module for the current execution path. The most important point of dynamic analysis is to create test cases ensuring acceptable code coverage. Otherwise false negatives (overlooked vulnerabilities) might appear in disregarded control paths. The task of obtaining a set of database queries for the given DDG is trivial for dynamic analysis implementation. For the static analysis implementation special techniques such as string analysis [19] should be applied in order to construct possible database queries.

```
Scheme = {users (id, login, passwd, email),
          news (id, title, text)}

shownews.cgi
"SELECT title, text FROM news"
```

users	0	0	0	0
news	0	r	r	0

```
postnews.cgi
"INSERT INTO news (title, text) VALUES ('A', 'B')
```

users	0	0	0	0
news	w	w	w	0

Fig. 2. Database access matrices for queries, generated by code sample from Figure 1.

The improved version of the model addresses only the first and the fourth drawback stated in section 2.1. The second and the third drawbacks are addressed in the next section. It is important to note, that the improved model has some limitations. First of all, it cannot handle stored procedures. Second, the model does not incorporate other types of storage: object oriented databases [12], xml [14], and file system. Generalization of this model in order to ensure support of generic data storages is yet to be made.

4 Dynamic Analysis and Penetration Testing Integration

In previous section we have described the improved model for input validation vulnerabilities. This section describes our implementation of this model and justifies it.
As it was mentioned, the model could be implemented by means of both static and dynamic analysis. Traditionally, static analysis tools are used during the software de-

velopment phase to ensure the product quality. However, many authors [42-44] note that web applications are often developed under time-to-market pressure by developers with insufficient security skills. Furthermore, high skills are required to tune and apply static analysis tools. So, static analysis is usually used by security assessment teams: they have skilled analysts and completeness of assessment can be guaranteed (thanks to conservativeness of static analysis).

On the other hand, dynamic analysis is usually used during the deployment phase, ensuring web application runtime protection [32, 35, 36, 45]. This approach has two issues when used in production environments: runtime overhead and updates problem. To the best of our knowledge, runtime protection based on taint propagation is natively supported only in Perl and Ruby. Thus, every update should be manually applied to the instrumented environment.

We believe that the easier the security assessment tool is, the more likely it will be used by developers to assess their applications (one of the reasons why web application scanners are so popular). Our aim is to give developers a tool that can be used as easily as web application scanners and that is capable to produce more accurate results. Thus, we implement our model by means of dynamic analysis. Quality of dynamic analysis is in direct proportion with input test cases quality [31]. Input data is taken from web application functional testing process. Thus, our approach does not address any code coverage issues and relies solely on the quality [30] of test cases prepared by the software testing stuff.

Our proposal is to combine dynamic analysis approach with penetration testing. Arguments lying behind it are as follows:

- During penetration testing real attacking patterns are submitted to an application. By combining penetration testing with dynamic analysis the scope of the web application view is widened, so error suppression and custom error pages are not an issue.
- By submitting real attacking patterns it is possible to test data validation routines for correctness, not just trust them blindly (see the Tainted Mode disadvantage №3);
- The dynamic analysis implementation knows the web application from inside, so more accurate penetration test cases can be generated. Below two examples illustrating this may be found:
  - Supposing in the certain module HTTP parameter `parA` is involved only in construction of the HTTP response then it is no use fuzzing it with values other than XSS attacks;
  - POST parameters inserted into database as keys (penetration testing disadvantage №1) can be indentified. After this a special fuzzing algorithm could be applied for fuzzing such form.

Figure 3 shows the overall scheme of our approach. The first phase consists of the following actions: test cases playback, generation of execution traces and analysis thereof. The goal is to determine the set of possible vulnerabilities. This is achieved by approximation of the web application with a set of interconnected DDGs. Then the Improved Tainted Mode model is applied to it.

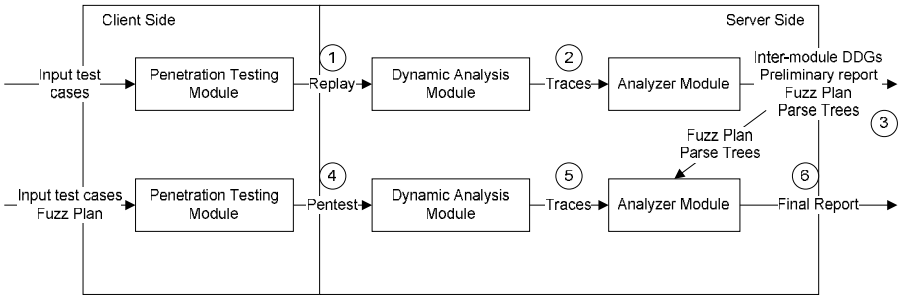

Fig. 3. The overall scheme of the integrated Dynamic Analysis and Penetration Testing approach.

In order to validate the results of the first phase, the second phase is undertaken. Various attacks are submitted to the web application to check, whether particular vulnerability exists or not. We consider that the vulnerability exists, if any malicious patterns were found in any sensitive operation. The result is the list of vulnerabilities that were successfully validated by penetration test and the list of possible vulnerabilities. Possible vulnerability is a reported issue by the Improved Tainted Mode model that was not confirmed during penetration testing. A possible vulnerability may result from two reasons. First of all, it could be the false positive, caused by the drawback 2, section 2.1. Second, this may be the result of incomplete malicious patterns database that was used during the penetration test.

5 Implementation

The prototype implementation of our approach consists of three main components: the dynamic analysis module, which is an extension of the Python interpreter [46] that collects traces of the executed application, the analyzer, which builds DDGs for the collected traces and performs analysis thereof, and the penetration testing module that submits input data (both normal and malicious) to the web application.

5.1 The dynamic analysis module

The purpose of the dynamic analysis module is to collect traces of the executed web application modules. See steps marked as 2 and 5 on the Figure 3. The trace of an execution consists of the sequence of events. Each event represents particular byte code operation with values of its operands. Thus, a trace can be viewed as a linear program with particular values in every statement.

In order to implement the gathering of traces we instrumented Python interpreter. The instrumentation was twofold. First of all, we implemented serialization of arbitrary data structures into string. Then, we added logging for every byte code operation. Please note that our approach is not meant for runtime protection, so performance overhead was not an issue. Traces are saved into XML. Further processing of these traces is carried out by the analyzer module.

5.2 The analyzer module

The second module in our prototype implementation is the analyzer. The analyzer performs the following tasks.

Building of DDGs from traces. This is the third step on the Figure 3. As was mentioned earlier, a trace is viewed as a linear program. First, a DDG representing a single trace is build. Then single DDGs are interconnected into inter-module DDGs as was described in section 3.2.

Analysis of DDGs for possible vulnerabilities. This is the third step on the Figure 3. The Improved Model defines four types of nodes: input, filter, critical and other. Every input, filter and critical node has its own classes (see map `G` in formula 2, section 3.2). In order to perform vulnerability analysis the map `G` should be defined.

In order to define a map `G` a survey of Python libraries (both built-in and third party) and Apache web server integration frameworks was undertaken. The goal of this survey was to find out which routines in Python could used to get data from HTTP requests, to sanitize the data or to produce HTTP responses, to make a database query, a system call, etc. The following frameworks developed for integration of Python with Apache web server were covered: CGI (built-in support), FastCGI [48], Mod_python [49], SCGI [50], WSGI [51]. The following popular publishing frameworks and libraries were also covered in the survey: Django [52], Pylons [53], CherryPy [54], Spyce [55].

The analyzer module takes the map `G` from the configuration files. This files contain input, filter, and critical statements and classes thereof, which we call signatures. These files could be easily extended to incorporate signatures for other frameworks or libraries.

Generation of fuzz plans for penetration testing module. This is the third step on the Figure 3. Analysis of the built DDGs results in the list of possible vulnerabilities. In order to check, if vulnerabilities really exists the penetration test is performed. During blind penetration test various malicious patterns are placed in every HTTP field (GET and POST parameters, headers) and submitted to web application. Analysis of DDGs can improve this. For every input HTTP parameter the analyzer module calculates the set of dependent sensitive operations arguments. These arguments determine the classes of corresponding sensitive operations, which in turn determine the classes of attacks that are rational to use in penetration testing. The directed penetration test is more suitable than the blind penetration test, which utilizes complete enumeration of HTTP-parameters and attacks.

The directed penetration test is based on the fuzz plan, which is generated by the analyzer module. The fuzz plan defines what attacks should be used to test particular HTTP parameter.

Final report generation. This is the sixth step on the Figure 3. The validation of possible vulnerabilities is performed after the penetration test has completed. We consider that the vulnerability exists, if any malicious patterns were found in any sensitive operation. For instance, in order to detect whether particular query to database, Shell interpreter, string evaluation routine (such as `eval`), etc. was exposed to code injection a comparison of parse trees is performed [32, 33]. Template parse trees for appropriate critical statement calls are built at the first phase of the approach (see sec-

tion 4). If parse trees generated at the second phase differ from template ones then vulnerability exists.

The result is the list of vulnerabilities that were successfully validated by penetration test and the list of possible vulnerabilities. Possible vulnerability is a reported issue by the Improved Tainted Mode model that was not confirmed during penetration testing.

5.3 The penetration testing module

The purpose of the penetration testing module is to deliver input data to web application both normal and malicious. Currently, this module is built on the top of the OWASP WebScarab tool [47]. The penetration testing module performs the following tasks.

User session replay. This is the first step on the Figure 3. As was pointed out in section 4, the input data for the vulnerability analysis is taken solely from the committed functional tests. Currently, we use the proxy plug-in of WebScarab tool to record HTTP requests and responses during the functional testing phase of web application. The recorded data we call user sessions. The penetration testing module may act as a simple HTTP client in order to replay the recorded sessions, thus delivering input data to the dynamic analysis module. However, sometimes the straightforward session replay cannot be performed due to the following reasons:

First, pattern recognition tasks (e.g. noisy images of letters or digits) are sometimes included into the web applications to restrict automated access to some interfaces. We ask for operator's assistance in such cases.

Second, most web applications use auto increment columns of their database relations to construct pages with user interface. This point may be illustrated with a simple example. Suppose, that web application has "post message" and "delete message" web interfaces. HTTP request from the "post message" interface leads to adding the following tuple into the database: (19, "A test message"), where 19 is the value of the auto increment column. The "delete message" interface could be organized as hidden field (or a static link) with id of the message. So HTTP request would be something like:

```
POST HTTP://mydomain/deletepost.html HTTP/1.0
Content-Length: 5
Content-Type: application/x-www-form-urlencoded
id=19
```

Let us replay this simple session. Due to the auto increment column the HTTP request will result in inserting the (20, "A test message") tuple into database. It is obvious that HTTP request to delete the posted message will fail. The stated issues were taken into account during implementation of this step.

Penetration tests run. This is the fourth step on the Figure 3. As was identified in section 4, the penetration test is aimed at validating the existence of vulnerabilities by issuing different attacks.

The input data include the recorded user sessions, the fuzz plan and the fuzz library. The penetration testing module is built on the top of the fuzzer plug-in of the

WebScarab tool. The malicious patterns for attacking were taken from fuzz libraries of freely available scanners [47, 57] and from [56].

6 Evaluation

We evaluated our prototype implementation on the application specially developed for testing purposes and on several real-world, publicly available Python applications, which are summarized in Table 1.

We manually navigated the applications simulating functional testing phase. We should admit that for the current experiment we did not measure the code coverage, so the results are not so representative. The submitted requests were recorded by the WebScarab proxy plug-in and saved as input data for our prototype.

The selected applications were not analyzed in order to determine extra input, filter or critical statements, so the default configuration files created during frameworks survey were used.

Application	Vulnerabilities	Reported Issues	False Positives	Known Vulnerabilities
Test application	8	8	0	
Spyce 2.1.3	8	8	0	CVE-2008-0980
Trac 0.9.1	2	2	0	CVE-2005-4065 CVE-2005-4305

Table 1. Python applications used in our experiments. Vulnerabilities are referenced by their Common Vulnerabilities and Exposures ID (CVE) [58].

Our tool was able to find all the known vulnerabilities. It is important to note the absence of the false positives. Hence, there was no overhead for the investigation of the generated reports. A more detailed evaluation of even more Python applications is yet to be made.

7 Conclusions

Number of reported web applications vulnerabilities is increasing dramatically. Most of them result from improper or none input validation by the web application. Most existing approaches are based on the Tainted Mode vulnerability model which cannot handle inter-module vulnerabilities.

This paper present an enhanced Tainted Mode model that incorporates inter-module data flows. We also introduced a new approach to automatic penetration testing by leveraging it with knowledge from dynamic analysis.

Future work will focus on two main directions. First of all, generalization of the model will be developed to support analysis of data flows through other data storage types or implemented by means of stored procedures and triggers. Second, special at-

tention to development of automatic crawling mechanisms will be given. The main problem here is automatic form filling. We plan to elaborate the following idea. First, static analysis can be used to detect data flows from HTTP parameters to the database fields. This results in data dependencies of the database columns upon values of particular form fields. Finally, the database can be queried to get values which will be used in automatic form filling, thus leveraging crawling process.

References

1. Andrews, M.: “The State of Web Security”. IEEE Security & Privacy, vol. 4, no. 4, pp. 14-15 (2006).
2. Auronen, L.: “Tool-Based Approach to Assessing Web Application Security”. Seminar on Network Security (2002).
3. Ragle, D.: “Introduction to Perl's Taint Mode.” http://www.webreference.com/programming/perl/taint
4. Thomas, D., Fowler, Ch., Hunt, A.: “Programming Ruby: The Pragmatic Programmer’s Guide”. Addison Wesley Longman, Inc (2001).
5. Anh, N.-T., Guarnieri, S., Greene, D., Shirley, J., Evans, D.: “Automatically Hardening Web Applications Using Precise Tainting”. IFIP Security Conference (2005).
6. Haldar, V., Chandra, D., Franz, M.: “Dynamic Taint Propagation for Java”. In: Proceedings of the 21st Annual Computer Security Applications Conference (2005).
7. Jovanovic, N., Kruegel, Ch., Kirda, E.: “Pixy: A Static Analysis Tool for. Detecting Web Application Vulnerabilities”.
8. Curphey, M., Wiesman, A., Van der Stock, A., Stirbei, R.: “A Guide to Building Secure Web Applications and Web Services”. OWASP (2005).
9. Livshits, V., Lam, M.: “Finding security errors in Java programs with static analysis”. In: Proceedings of the 14th Usenix Security Symposium (2005).
10. Codd, E. F.: “A relational model of data for large shared data banks”. In: Communications of the ACM, vol.13, no. 6, pp. 377-387 (1970).
11. ”Database Language SQL”. ISO/IEC 9075 (1992).
12. Kim, W.: ”Introduction to Object-Oriented Databases”. The MIT Press (1990).
13. Landi, W.: “Undecidability of static analysis”. In: ACM Letters on Programming Languages and Systems (1992).
14. Staken, K.: “Introduction to Native XML Databases”. http://www.xml.com/pub/a/2001/10/31/nativexmldb.html.
15. Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., Kuo, S.-Y.: “Securing web application code by static analysis and runtime protection”. In: WWW '04: Proceedings of the 13th International Conference on World Wide Web (2004).
16. Christensen, A. S., Mller, A., Schwartzbach, M. I.: "Precise analysis of string expressions". In: Proceedings of International Static Analysis Symposium, SAS '03 (2003).
17. Gould, C., Su, Z., Devanbu, P.: “Static checking of dynamically generated queries in database applications”. In: Proceedings of 26th International Conference on Software Engineering, IEEE Press (2004).
18. Minamide, Y.: “Static approximation of dynamically generated web pages”. In: Proceedings of the 14th International Conference on World Wide Web (2005).
19. Wassermann, G., Su, Z.: “Sound and precise analysis of web applications for injection vulnerabilities”. In: Proceedings of the SIGPLAN conference on Programming language design and implementation (2007).
20. Chess, B., McGraw, G.: “Static analysis for security”. IEEE Security & Privacy, vol. 2, no. 6, pp. 76-79 (2004).
21. Kozlov, D., Petukhov, A.: “Implementation of Tainted Mode Approach to Finding Security Vulnerabilities for Python Technology”. In: Proceedings of the First Spring Young Researchers’ Colloquium on Software Engineering, vol 1, pp. 45-47 (2007).
22. Kozlov, D., Petukhov, A.: SpoC 007 - Python Tainted Mode. http://www.owasp.org/index.php/SpoC_007_-_Python_Tainted_Mode.
23. Ollmann, G.: “Second order code injection attacks”. NTGSSoftware Insight Security Research (2004).
24. Anley, C.: “Advanced SQL Injection In SQL Server Applications”, NTGSSoftware Insight Security Research (2002).

25. Huang, Y.-W., Huang, S.-K., Lin, T.-P., Tsai, Ch.-H.: “Web application security assessment by fault injection and behavior monitoring”. In: Proceedings of the 12th international conference on World Wide Web, May 20-24 (2003).
26. Huang, Y.-W., Lee, D. T.: “Web Application Security - Past, Present, and Future”. Computer Security in the 21st Century, Springer US, pp. 183-227 (2005).
27. Wiegenstein, A., Weidemann, F., Schumacher, M., Schinzel, S.: “Web Application Vulnerability Scanners - a Benchmark”. Virtual Forge GmbH (2006).
28. Ferrante, J., Ottenstein, K. J., Warren, J. D.: “The program dependence graph and its use in optimization”. In: ACM Transactions on Programming Languages and Systems, vol. 9 no. 3, pp. 319-349 (1987).
29. Xu, B., Qian, J., Zhang, Xi., Wu, Zh., Chen, L.: “A brief survey of program slicing”. ACM SIGSOFT Software Engineering Notes, vol. 30, no. 2 (2005).
30. Cornett, S.: “Code Coverage Analysis”, http://www.bullseye.com/coverage.html
31. Bell, Th.: “The concept of dynamic analysis”. In: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 216-234, September 6-10 (1999).
32. Su, Zh., Wassermann, G.: “The essence of command injection attacks in web applications”. In: ACM SIGPLAN Notices, vol. 41, no.1, pp. 372-382 (2006).
33. Buehrer, G.T., Weide, B.W., Sivilotti, P.: “Using parse tree validation to prevent SQL injection attacks”. In: Proceedings of the 5th international workshop on Software engineering and middleware (2005).
34. Cova, M., Felmetsger, V., Vigna, G.: “Testing and Analysis of Web Services”. Springer (2007).
35. Pietraszek, T., Berghe, C. V.: “Defending against Injection Attacks through Context-Sensitive String Evaluation”. In: Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection, (2005).
36. Halfond, W., Orso, A.: “AMNESIA: Analysis and Monitoring for Neutralizing SQL-Injection Attacks”. In: Proceedings of the International Conference on Automated Software Engineering, pp. 174–183, (2005).
37. Ristic, I.: “Web application firewalls primer”. (IN)SECURE, vol. 1, no. 5, pp. 6-10, (2006).
38. Cook, W. R., Rai., S.: “Safe Query Objects: Statically Typed Objects as Remotely Executable Queries”. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), (2005).
39. McClure, R., Kr¨uger, I.: “SQL DOM: Compile Time Checking of Dynamic SQL Statements”. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 05), pp. 88–96, (2005).
40. Meier, J.D., Mackman, A., Wastell, B.: “Threat Modeling Web Applications”, Microsoft Corporation, (2005).
41. Cova, M., Felmetsger, V., Balzarotti, D., Jovanovic, N., Kruegel, Ch., Kirda, E., Vigna, G.: “Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications”. Oakland, (2008).
42. Balzarotti, D., Cova, M., Felmetsger, V., Vigna, G.: “Multi-Module Vulnerability Analysis of Web-based Applications”. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS), Alexandria, October, (2007).
43. "J2EE and Scripting Languages", Version 2.0, February, (2007).
44. Kappel, G., Michlmayr, E., Pröll, B., Reich, S., Retschitzegger, W.: “Web Engineering - Old wine in new bottles?” In: Proceedings of the 4th International Conference on Web Engineering (ICWE2004), pp. 6-12, July, (2004).
45. Cova, M., Balzarotti, D., Felmetsger, V., G. Vigna, G.: “Swaddler: An Approach for the Anomaly-based Detection of State Violations in Web Applications”. In: Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID), September, (2007).
46. Python Programming Language. http://www.python.org/
47. OWASP WebScarab Project. http://www.owasp.org/index.php/OWASP_WebScarab_Project
48. FastCGI Home. http://www.fastcgi.com/
49. Mod_python – Apache/Python Integration. http://modpython.org/
50. SCGI: Simple Common Gateway Interface. http://python.ca/scgi/
51. PEP 333 – Python Web Server Gateway Interface v1.0. http://www.python.org/dev/peps/pep-0333/
52. The Django framework. http://www.djangoproject.com/
53. Pylons Python Web Framework. http://pylonshq.com/
54. CherryPy Framework. http://www.cherrypy.org/
55. Spyce – Python Server Pages (PSP). http://spyce.sourceforge.net/
56. XSS (Cross Site Scripting) Cheat Sheet. http://ha.ckers.org/xss.html
57. OWASP WSFuzzer Project. http://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project
58. CVE - Common Vulnerabilities and Exposures. http://cve.mitre.org/