Equipping WAP with WEAPONS to Detect Vulnerabilities

Practical Experience Report

Ibéria Medeiros
LaSIGE, Faculdade de Ciências
Universidade de Lisboa – Portugal
ibemed@gmail.com

Nuno Neves
LaSIGE, Faculdade de Ciências
Universidade de Lisboa – Portugal
nuno@di.fc.ul.pt

Miguel Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa – Portugal
miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—Although security starts to be taken into account during software development, the tendency for source code to contain vulnerabilities persists. Open source static analysis tools provide a sensible approach to mitigate this problem. However, these tools are programmed to detect a specific set of vulnerabilities and they are often difficult to extend to detect new ones. WAP is a recent popular open source tool that detects vulnerabilities in the source code of web applications written in PHP. The paper addresses the difficulty of extending these tools by proposing a modular and extensible version of the WAP tool, equipping it with "weapons" to detect (and correct) new vulnerability classes. The new version of the tool was evaluated with seven new vulnerability classes using web applications and plugins of the widely-adopted WordPress content management system. The experimental results show that this extensibility allows WAP to find many new (zero-day) vulnerabilities.**

Keywords*—*Web applications; software security; input validation vulnerabilities; source code analysis; false positives; automatic protection; data mining; modularity; security.

I. INTRODUCTION

Web applications are part of our daily life. Although security starts to be taken into account during web application development, the tendency for source code to contain vulnerabilities persists. The input validation vulnerability category is arguably the most relevant. This is comproved by the fact that code injection vulnerabilities such as SQL injection (SQLI) and cross-site scripting (XSS) remain at the center of this tendency, as reported by OWASP in its top 10 [25] and confirmed by the recent Ashley Madison fiasco [4]. Nevertheless, new technologies are becoming widely deployed as part of web applications. An example are NoSQL databases, particularly convenient to store "big data". With new technologies come also new attack vectors, with 600 TB of data recently stolen from the most used [7] NoSQL database, MongoDB [23].

Static analysis is a technique often used by companies to mitigate the problem of software vulnerabilities [24]. Static analysis tools search for vulnerabilities in source code, helping programmers to fix the code. However, these tools are programmed to detect specific sets of flaws, often SQLI and XSS [10], [15], occasionally a few other [11], [6], and are typically hard to extend to search for new classes of vulnerabilities.

WAP [2] is a recent open source static analysis tool that detects vulnerabilities of eight classes in PHP code, the language most used in web applications [9]. It is apparently popular, as *sourceforge* shows more than 6700 downloads. The tool uses data mining to predict false positives (false vulnerability alarms) and adds fixes (small pieces of PHP code) to remove vulnerabilities. Nevertheless, by analysing its source code, it is possible to understand that WAP is hard to extend for new classes of vulnerabilities.

The paper addresses the difficulty of extending these tools by proposing a modular and extensible version of the WAP tool, equipping it with *weapons* (WAP extensions) to detect and correct new vulnerability classes. This involves restructuring the tool in: (1) modules for the vulnerability classes that it already detects; and, more importantly, (2) a new module to be configured by the user to detect and correct new vulnerability classes without additional programming. This latter module takes as input data about the new vulnerability class: *entry points* (input sources), *sensitive sinks* (functions exploited by the attack), and *sanitization functions* (functions that neutralize malicious input). Then it automatically generates a *weapon* composed of: a *detector* to search for vulnerabilities, *symptoms* to predict false positives, and a *fix* to correct vulnerable code. We used this scheme to enhance the new version of WAP with the ability to detect 7 new classes of vulnerabilities: session fixation, header injection (or HTTP response splitting), email injection, comment spamming injection, LDAP injection, XPath injection, and NoSQL injection.

The paper also demonstrates that this modularity and extensibility can be used to create weapons that deal with non-native entry points, sanitization functions, and sensitive sinks. We demonstrate this point by creating a weapon to detect SQLI vulnerabilities in WordPress [27], the most popular content management system (CMS) [9].

The paper presents a second improvement to WAP. It improves the false positive prediction to make it more precise and accurate. We propose to increase the granularity of the analysis, adding more symptoms to the original set and a new, larger, data set. A re-evaluation of machine learning classifiers was performed to select the new top 3 classifiers.

The version of WAP presented in the paper is the first static analysis tool configurable to detect and correct new classes of vulnerabilities without programming. To the best of our knowledge, it is also the first static analysis tool that detects NoSQL injection and comment spamming injection. The latter is currently the most exploited vulnerability in applications based on WordPress [9].

We evaluated the tool experimentally with 54 web application packages and 115 WordPress plugins, adding up to more than 8,000 files and 2 million lines of code. The tool discovered respectively 366 and 153 zero-day vulnerabilities, i.e., 519 previously-unknown vulnerabilities, presently being

reported to their developers. In our experiments, our modular and extensible tool has shown a much higher ability to detect zero-day vulnerabilities than the original version.

The contributions of the paper are: (1) a modular and extensible static analysis tool that allows creating *weapons* to detect and correct new vulnerability classes, without requiring programming; (2) a new version of the WAP tool able to detect vulnerabilities of 15 classes, instead of the original 8 classes.

II. The original WAP tool

This section presents briefly the original WAP tool [2], [12]. WAP detects input validation vulnerabilities in PHP web applications. The tool combines source code static analysis (taint analysis) to detect candidate vulnerabilities and data mining to predict if the flagged candidate vulnerabilities are false positives. It is also able to correct source code by inserting *fixes*. The version currently available (v2.1) handles eight vulnerability classes: SQLI, XSS (reflected and stored), remote file inclusion (RFI), local file inclusion (LFI), directory or path traversal (DT/PT), OS command injection (OSCI), source code disclosure (SCD), and PHP command injection (PHPCI). WAP is now a OWASP project [1].

The tool, developed in Java, is composed of 3 modules (Fig. 1): (1) *Code analyzer:* parses the source code, generates an abstract syntax tree (AST), does taint analysis, and generates trees describing candidate vulnerable data-flow paths (from an entry point to a sensitive sink). The code analyzer may return false positives as it may not recognize that certain code structures effectively sanitize data flows. (2) *False positive predictor:* obtains *symptoms* (source code features) from the candidate vulnerable data-flow paths and uses a combination of 3 classifiers to make the prediction (Logistic Regression, Random Tree, Support Vector Machine). (3) *Code corrector:* identifies the fixes to add and the places where they have to be inserted; then modifies the source code with the fixes.

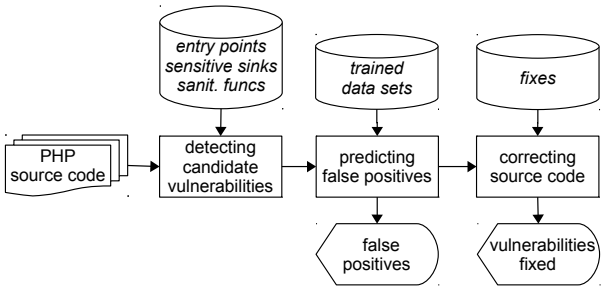


Fig. 1: Overview of the WAP tool modules and data flow.

III. Restructuring WAP

We propose to extend the WAP tool to be configurable to handle new classes of input validation vulnerabilities, so we restructure the tool making it modular. We explain this process considering WAP's three original modules: code analyzer, false positive predictor, and code corrector.

A. Code analyzer

The code that does taint analysis uses three pieces of data about each class of vulnerabilities: entry points, sensitive sinks, and sanitization functions. Data coming from entry points is considered tainted (i.e., non-trustworthy). This component tracks how this data flows through variables and functions, verifying if it reaches a sensitive sink. Sanitization functions block the flow of tainted data. Therefore, the taint analyzer is coded to recognize the set of functions for each vulnerability class and specific characteristics on this class (if they exist).

Restructuring the code analyzer implies, on the one hand, to reorganize the taint analyzer in sub-modules and, on the other hand, to create a generic detection sub-module configurable by the user for new vulnerability classes. The AST has to be left unmodified as it is input to all the sub-modules.

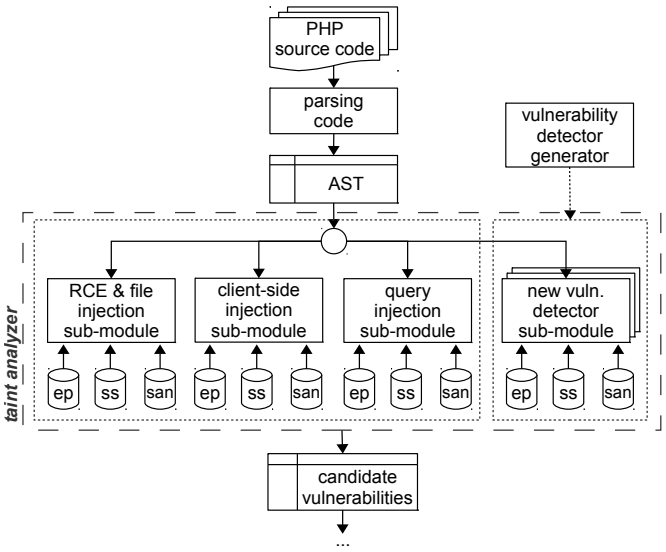


Fig. 2: Reorganization of WAP's code analyzis module.

Fig. 2 shows the restructured code analyzer. At the top the figure shows that PHP code is converted to an AST that is common input to all sub-modules. The sub-modules are: (1) *RCE & file injection*, dealing with vulnerabilities involving file system, files, and URLs leading to remote code execution (RCE). These vulnerabilities are OSCI, PHPCI, RFI, LFI, DT, and SCD. (2) *client-side injection*, handling vulnerabilities related with injection of client-side code (e.g., JavaScript code), namely reflected and stored XSS. (3) *query injection*, for vulnerabilities associated to queries, i.e., SQLI. (4) *vulnerability detector generator*, the generic detector configurable by the user for new vulnerabilities. (5) *new vulnerability detector sub-module*, the detectors generated by (4), one for each new vulnerability class. Each sub-module is fed with entry points (*ep*), sensitive sinks (*ss*), and sanitization (*san*) functions. These sets of data are now stored in external files, allowing the inclusion of new items without recompiling the tool.

WAP's parser was implemented using ANTLR [19]. This framework provides tree walkers to navigate through ASTs. The new vulnerability detector sub-module (sub-module (5)) leverages a tree walker to track data flow and understand if tainted data reaches a sensitive sink.

WAP (original)		WAP (new version)
attribute	**symptom**	**symptom**
validation		
Type checking	is_string is_int is_float is_numeric ctype_digit	is_double is_integer is_long is_real is_scalar ctype_alpha ctype_alnum intval
Entry point is set	isset	is_null empty
Pattern control	preg_match	preg_match_all ereg eregi strnatcmp strcmp strncmp strncasecmp strcasecmp
White list Black list	*user functions*[1] *user functions*[2]	
Error and exit	error exit	
string manipulation		
Extract substring	substr	preg_split str_split explode split spliti
String concatenation	concatenation operator	implode join
Add char	addchar	str_pad
Replace string	substr_replace str_replace preg_replace	preg_filter ereg_replace eregi_replace str_ireplace str_shuffle chunk_split
Remove whitespaces	trim	rrim ltrim
SQL query manipulation		
Complex query Numeric entry point FROM clause Aggregated function	ComplexSQL IsNum FROM AVG COUNT SUM MAX MIN	
classification		
Class	false positive (FP) real vulnerability (RV)	

[1] user functions containing white lists to validate user inputs
[2] user functions containing black lists to block user inputs

TABLE I: Attributes and symptoms of the original WAP and the new ones. In the new WAP all symptoms are also attributes.

B. False positive predictor

The 3 classifiers of the original WAP use 15 attributes to classify the vulnerabilities found by the taint analyzer as true or false. These attributes represent 24 symptoms that may be present in source code, divided in three categories: validation, string manipulation and SQL query manipulation. Table I shows these attributes and symptoms in the two left-hand columns. The symptoms are PHP functions that manipulate entry points or variables. The attributes represent symptoms of the same kind, e.g., the *type checking* attribute represent the symptoms that check the data type of variables. Therefore, an attribute represents several symptoms. A special attribute is used to indicate the class of each instance (the 16th, last row).

We propose to improve this component in two directions: (1) by adding more symptoms to the original set used in WAP (*static symptoms*), and (2) allowing the user to define new symptoms (*dynamic symptoms*).

1) Static symptoms: By investigating the symptoms associated with false positives we have understood that there were several relevant symptoms not considered originally in WAP. These symptoms are listed in the right-hand column of Table I. Moreover, we increased the granularity of the analysis by specifying that *all symptoms are attributes* (both old and new, 2nd and 3rd columns). Therefore, instead of 16 attributes we now have 61.

Modifying the attributes requires training again the classifiers and, as the number of attributes is much higher, we need also a much larger number of instances (samples of code annotated as false positive or not). The original WAP was trained with a data set of 76 instances: 32 annotated as false positives and 44 as real vulnerabilities. Each instance had 16 attributes set to 1 or 0, indicating the presence or not of symptoms for the attributes, and an attribute saying if the instance is a false positive or not. We increased the number of instances to 256, each one with 61 attributes. The instances are evenly divided in false positives and vulnerable (balanced data set). To create the data set we used WAP configured to output the candidate vulnerabilities, and we ran it with 29 open source PHP web applications. Then, each candidate vulnerability was processed manually to collect the attributes and to classify it as being a false positive or not. Finally, noise was eliminated from the data set by removing duplicated and ambiguous instances.

To perform the data mining process we used the WEKA tool [26] with the original classifiers and induction rules. We also want a top 3 of classifiers, as originally. Our goals are that classifiers: (1) predict as many false positives correctly as possible; (2) have a fallout as low as possible (wrong classifications of vulnerabilities as false positives), avoiding to miss vulnerabilities found by the taint analyzer.

Table II depicts the evaluation of the 3 best classifiers (we omit the rest for lack of space). The first 7 metrics were adopted from [12]; the last 2 are new. The last column shows the formulas to calculate each metric, based in values extracted from the confusion matrix (Table III, last 2 columns).

Metrics (%)	SVM	Logistic Regression	Random Forest	Formula
tpp	94.5%	93.0%	90.6%	*tpp = recall = tp / (tp + fn)*
pfp	4.7%	4.7%	2.3%	*pfp = fallout = fp / (tn + fp)*
prfp	95.3%	95.2%	97.5%	*prfp = pr positive = tp / (tp + fp)*
pd	95.3%	95.3%	97.7%	*pd = specificity = tn / (tn + fp)*
ppd	94.6%	93.1%	91.2%	*ppd = inverse pr = tn / (tn + fn)*
acc	94.9%	94.1%	94.1%	*accuracy = (tp + tn) / N*
pr	94.9%	94.2%	94.4%	*precision = (prfp + ppd) / 2*
inform	89.8%	88.3%	88.3%	*informedness = tpp + pd -1 = tpp - pfp*
jacc	90.3%	88.8%	88.5%	*jaccard = tp / (tp + fn + fp)*

TABLE II: Evaluation of the machine learning models applied to the data set.

Classifiers are usually selected based on accuracy and precision, but in this case the three classifiers have very similar values in both metrics: between 94% and 95%. Moreover, the compliance to goal (1) is measured by *tpp*. In terms of this metric, Support Vector Machine (SVM) had the best results

	Observed							
	SVM		Logistic Regression		Random Forest		Classifier	
Predicted	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes (FP)	No (not FP)	Yes	No
Yes (FP)	121	6	119	6	116	3	*tp*	*fp*
No (not FP)	7	122	9	122	12	125	*fn*	*tn*

TABLE III: Confusion matrix of the top 3 classifiers and confusion matrix notation (last two columns).

and Logistic Regression (LR) the second best. In terms of goal (2), Random Forest (RF) had the best fallout rate (*pfp*). The *inform* metric expresses how the classifications made by the classifier are close to the correct (real) classifications, whereas *jacc* measures the classifications in the false positive class, taking into account false positives and negatives [20]. For *inform*, we combine the best values of *tpp* and *pfp*, i.e., the *tpp* from SVM and the *pfp* from RF, resulting in 92%, while for *jacc* we use the correct and misclassifications of all classifiers, resulting in 92%. These measures confirm our choice of the top 3 classifiers. These classifiers are the same as those used in the original WAP, except RF that substitutes Random Tree.

The confusion matrix of these classifiers is presented in Table III. SVM and LR classified incorrectly a few instances, and RF classified 3 real vulnerabilities as being false positives. Notice that this misclassification is represented as *fp* in the confusion matrix, representing the instances belonging to class *No* that were classified in class *Yes*. However, in the context of vulnerability detection this represents false negatives, i.e., vulnerabilities that were not detected.

2) Dynamic symptoms: We use the term dynamic symptoms to designate symptoms defined by the user that configures the tool for new vulnerabilities, whereas static symptoms are those that come with the tool. For every dynamic symptom the user has to provide a category and a type. For example, if the user develops a function *val_int* to validate integer inputs (instead of *is_int*) he has to provide the information that the function belongs to the validation category and that it has an effect similar to the static symptom (function) *is_int*. Based on this information, the tool understands how to handle function *val_int* when predicting false positives.

Fig. 3 presents the reorganization of the false positive predictor. When a candidate vulnerability is processed by this module: first the static and dynamic symptoms are collected from the source code; then a vector of 61 attributes is created using the map from static symptoms to attributes (stored into the tool) and the map of dynamic symptoms to attributes (created dynamically); then the vector is classified using machine learning classifiers; finally, in case of a real vulnerability, it is sent to the code corrector module to be fixed.

C. Code corrector

When a vulnerability is found, the code corrector inserts a fix that does sanitization or validation of the data flow. To make WAP modular we created two sub-modules: (1) *code fixing* sub-module, which receives the vulnerability class and the code to be fixed and inserts the fix; (2) *fix creation* that uses information and constraints provided by the user to generate a new fix for a new class of vulnerabilities. The first does essentially what the original version of WAP already did so we focus on (2).

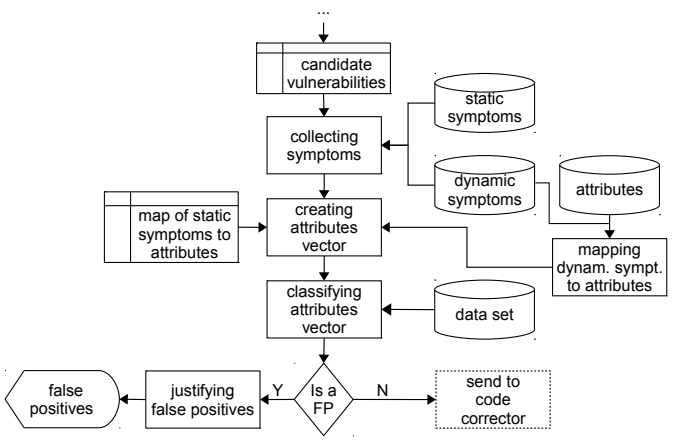


Fig. 3: Reorganization of the false positives predictor module.

We propose three *fix templates* to generate automatically fixes: *PHP sanitization function*, *user sanitization*, and *user validation*. The one that is used depends on the information provided by the user. The *PHP sanitization function* template is applied when the user specifies the PHP sanitization function used to sanitize data and the sensitive sink associated to this functions, for a given vulnerability. The sanitization function is used as fix. The *user sanitization* template is chosen if the user indicates the malicious characters that may be used to exploit the vulnerability and a character that can be used to neutralize them (e.g., the backslash). The *user validation* template is used if the user only specifies the set of malicious characters used to exploit the vulnerability. In that case the fix checks the presence of these characters, issuing a message in case there is a match. Fixes are inserted in the line of the sensitive sink, as in the original WAP [12].

D. Weapons

A *weapon* is a WAP extension composed by a detector, a fix and, optionally, a set of dynamic symptoms. To generate weapons we developed a *weapon generator*, external to WAP. The data needed to create a weapon is: (1) for the *detector*, the sanitization and sensitive sinks functions, plus additional entry points if they exist; (2) for the *fix*, data for the fix templates (Section III-C); (3) *dynamic symptoms*, in case the user has white/black lists of functions, or functions that do not belong to the static symptoms list (in this case, the correspondence between dynamic and static symptoms is required).

To generate a weapon, the weapon generator uses the *vulnerability detector generator* (see Section III-A) that it configures with (1), generating a new detector with the *ss*, *san* and *ep* files containing the data provided by the user. Next, it configures the selected fix template with (2), generating a new fix. Then, it creates a file with (3). The last step is to put together the three parts, linking them to WAP. Detection is activated using a command line flag also provided by the user (e.g., *-nosqli*).

When the weapon is activated, WAP parses the code, generating an AST, next the detector navigates under the AST using the data stored in its files. The candidates vulnerabilities found by the detector are processed by the false positives

predictor using the symptoms defined in WAP and contained in the weapon, and the real vulnerabilities are fixed using the *code fixing* module (see Section III-C) with the fix of the weapon.

E. Effort to modify WAP

Modifying WAP involved an effort with three facets: (1) making the AST independent of the navigation made by the detectors (tree walkers); (2) restructuring the code to create the three sub-modules for the vulnerabilities originally considered (Section III-A), to integrate the dynamic symptoms (Section III-B), and to make the code corrector able to receive new fixes (Section III-C); (3) coding the *weapon generator* module (Section III-D). From the three facets, (3) was the one that required more effort. We had to build a new java package to create weapons (*new vulnerability detector sub-module*), a frontend for the user to configure the weapon generator, templates to create automatically fixes, and to integrate the weapon in WAP. When the weapon generator is executed it creates a new java package and compiles it, building a *jar* to be integrated with the WAP tool.

IV. Extending WAP with new vulnerabilities

This section presents the seven new vulnerability classes with which we extended WAP, as well as how this extension was done. The section also presents the extension to detect SQLI in WordPress plugins that uses WordPress functions as entry points, sanitization functions, and sensitive sinks. To demonstrate how we can take advantage of the modularity we created in WAP, we opted by extending it in two different ways: reusing the sub-modules presented in Section III-A (Section IV-B) and with weapons (Section IV-C). However, a normal user would probably use the second form.

A. New vulnerabilities

We equiped the tool to detect the following seven vulnerabilities: LDAP injection (LDAPI), XPath injection (XPathI), NoSQL injection (NoSQLI), comment spamming (CS), header injection or HTTP response splitting (HI), email injection (EI), and session fixation (SF). With the exception of SF, all of them are input validation vulnerabilities, meaning that they are created by lack of sanitization or validation of user inputs before they reach a sensitive sink.

The first three vulnerabilities are associated to the construction of queries or filters that are executed by some kind of engine, e.g., a database management system. They behave similarly to SQLI, i.e., if a query is built with unsanitized user inputs containing malicious characters, the query executed is a modification of the original one [21], [17].

CS has the goal of manipulating the ranking of spammers' web sites, making them appear at the top of search engines' results. Web applications that allow the users to submit contents with hyperlinks are the potential victims of the attack. Attackers inject comments, for example, containing links to their own web site [8], [9]. To avoid CS, applications have to check if the content of posts contains hyperlinks (URLs).

Header injection or HTTP response splitting (HI) allows an attacker to manipulate the HTTP response, breaking the normal response using the $\backslash n$ and $\backslash r$ characters. This allows the attacker to inject malicious code (e.g., JavaScript) in a new header line or even a new HTTP response. The vulnerability can be avoided by sanitizing these characters (e.g., substituting them by a space) [21].

Email injection (EI) is similar to HI, allowing an attacker to inject the line termination character, in clear or encoded (*%0a* and *%0d*), with the aim of manipulating the email components (e.g., sender, destination, message). The vulnerability can also be avoided by sanitizing these characters [21].

Session fixation allows an attacker to force a web client to use a specific ("fixed") session ID, allowing him to access the account of the user. Avoiding this vulnerability is not trivial as there is no sanitization function to apply or set of malicious characters to recognize. A way to defend against SF is to avoid using a session token provided by the user [21], [16].

B. Reusing the sub-modules

The detection of four of the vulnerability classes described in the previous section can be integrated in the sub-modules of Section III-A and the fixes to remove them can be created using a fix template (Section III-C). Table IV shows the classes of vulnerabilities integrated in each sub-module and the sensitive sinks added to detect each vulnerability. These functions were inserted in the *ss* file of each sub-module. No sanitization functions or entry points were added to the *san* and *ep* files.

In relation to LDAPI and XPathI, a fix was created for each one using the *user validation* fix template. For CS we changed WAP's `san_read` and `san_write` fixes. These fixes deal with the sensitive sinks specified above for the CS vulnerability. They validate the user inputs contents against JavaScript code, so we changed them to also check the input contents against URIs/hyperlinks. For SF we created a fix from scratch.

Sub-module	Vuln.	Sensitive sink
RCE & file injection	SF	setcookie, setdrawcookie, session_id
client-side injection	CS	file_put_contents, file_get_contents
query injection	LDAPI XPathI	ldap_add, ldap_delete, ldap_list, ldap_read, ldap_search xpath_eval, xptr_eval, xpath_eval_expression

TABLE IV: Sensitive sinks added to the WAP sub-modules to detect new vulnerability classes.

C. Creating weapons

We used the scheme presented in SectionIII-D to create three weapons, for (1) NoSQLI, (2) HI and EI, and (3) SQLI for WordPress.

1) NoSQLI weapon: NoSQL is a common designation for non-relational databases used in many large-scale web applications. There are various NoSQL database models and many engines that implement them. MongoDB [13] is the most popular engine implementing the document store model [7]. Therefore, we opted for creating a weapon to detect NoSQLI in PHP web applications that connect to MongoDB. We configured the weapon generator with: (1) the `find`, `findOne`, `findAndModify`, `insert`, `remove`, `save`, `execute` sensitive sinks and the `mysql_real_escape_string` sanitization function; (2) the *PHP sanitization* fix template to sanitize the user inputs that

reach that sink with that sanitization function, resulting in the *san_nosqli* fix; and (3) no dynamic symptoms. The weapon is activated by the *-nosqli* flag.

2) HI and EI weapon: We configured the weapon generator with: (1) the `header` and `mail` sensitive sinks and no sanitization functions; (2) the *user sanitization* fix template to check the malicious characters presented in Section IV-A and to replace them by a space, resulting in the *san_hei* fix; and (3) no dynamic symptoms. The weapon is activated with the *-hei* flag of WAP.

3) SQLI for WordPress weapon: WordPress has a set of functions that sanitize and validate different data types, which are used in some add-ons. It has also its own sinks to handle SQL commands (*$wpdb* class). If we want to analyze, for example, WordPress plugins with WAP for SQLI vulnerabilities, we need a weapon that recognizes these functions. Therefore, we configured the weapon generator with: (1) the sensitive sinks and sanitization functions from *$wpdb*; (2) the *PHP sanitization* fix template to sanitize the user inputs that reach those sinks with those sanitization functions, resulting in the *san_wpsqli* fix; and (3) dynamic symptoms, with validation functions from *$wpdb* and their corresponding static symptoms. The weapon is activated by the flag *-wpsqli*.

V. Experimental evaluation

The objective of the experimental evaluation was to answer the following questions: (1) Is the new version of WAP able to detect the new vulnerabilities (Sections V-A and V-B)?; (2) Does it remain able to detect the same vulnerabilities as WAP v2.1 (Section V-A)?; (3) Is it more accurate and precise in predicting false positives than WAP v2.1 (Section V-A)?; (4) Can it be equipped with weapons configured with non-native PHP functions and detect vulnerabilities (Section V-B)? For convenience, in this section we designate the new version of the WAP tool by *WAPe*.

A. Real web applications

To assess the new version of the tool and to answer the first three questions, we run WAPe with 54 web application packages written in PHP and compare it with the prior version of the tool (v2.1).

WAPe analyzed a total of 8,374 files corresponding to 2,065,914 lines of code of the 54 packages. It detected 413 real vulnerabilities from several classes in 17 applications. The largest packages analyzed were *Play sms v1.3.1* and *phpBB v3.1.6_Es* with 248,875 and 185,201 lines of code. Table V summarizes this analysis presenting the 17 packages where these vulnerabilities were found and some information about the analysis. These 17 packages contain 4,714 files corresponding to 1,196,702 lines of code. The total execution time for the analysis was 123 seconds, with an average of 7.2 seconds per application. This average time indicates that the tool has a good performance as it searches for 15 vulnerability classes in one execution.

We run the same 54 packages with WAP v2.1. The tool flagged as vulnerable the same 17 applications. Table VI presents the detection made by the two tools distributed by the 10 classes of vulnerabilities and the false positives predicted

Web application	Version	Files	Lines of code	Analysis time (s)	Vuln. files	Vuln. found
Admin Control Panel Lite 2	0.10.2	14	1,984	1	9	81
Anywhere Board Games	0.150215	3	501	1	1	3
Clip Bucket	2.7.0.4	597	148,129	11	16	22
Clip Bucket	2.8	606	149,830	12	18	26
Community Mobile Channels	0.2.0	372	119,890	8	116	47
divine	0.1.3a	5	706	1	2	9
Ldap address book	0.22	18	4,615	2	4	1
Minutes	0.42	19	2,670	1	2	10
Mle Moodle	0.8.8.5	235	59,723	18	4	7
Php Open Chat	3.0.2	249	83,899	7	9	11
Pivotx	2.3.10	254	108,893	6	1	1
Play sms	1.3.1	1,420	248,875	19	7	6
RCR AEsir	0.11a	8	396	1	6	13
refbase	0.9.6	171	109,600	10	18	48
SAE	1.1	150	47,207	7	39	48
Tomahawk Mail	2.0	155	16,742	3	3	3
vfront	0.99.3	438	93,042	15	25	77
	Total	4,714	1,196,702	123	280	413

TABLE V: Summary of results for the new version of WAP with real web applications.

Web application	Version	WAP & WAPe real vuls.				WAPe real vuls.					WAP FP		WAPe FP	
		SQLI	XSS	Files*	SCD	LDAPI	SF	HI	CS	Total	FPP	FP	FPP	FP
Admin Control Panel Lite 2	0.10.2	9	72							81	8		8	
Anywhere Board Games	0.150215		1	1					1	3				
Clip Bucket	2.7.0.4		10	11				1		22	2	4	6	
Clip Bucket	2.8	4	10	11				1		26	2	4	6	
Community Mobile Channels	0.2.0	14	27		3			3		47	4		4	
divine	0.1.3a		4	2					3	9				
Ldap address book	0.22					1				1				
Minutes	0.42	9						1		10				
Mle Moodle	0.8.8.5		6	1						7	2	1	2	1
Php Open Chat	3.0.2		10			1				11				
Pivotx	2.3.10						1			1	9		9	
Play sms	1.3.1		6							6		2	2	
RCR AEsir	0.11a		9	3					1	13	1		1	
Refbase	0.9.6		46					2		48	7	4	11	
SAE	1.1	11	25	10	1			1		48		23	12	11
Tomahawk Mail	2.0	2	1							3	1	2	3	
vfront	0.99.3	23	28	16				10		77	26	20	40	6
	Total	72	255	55	4	2	1	19	5	413	62	60	104	18

*DT & RFI, LFI vulnerabilities

TABLE VI: Vulnerabilities found and false positives predicted and reported by the two versions of WAP in web applications.

and not predicted. The third to sixth columns show the number of real vulnerabilities that the tools found for the classes that both detect, i.e., the 386 vulnerabilities of classes SQLI, XSS, RFI, LFI, DT and SCD. This provides a positive answer to the second question: WAPe still discovers the vulnerabilities detected by WAP v2.1.

The next four columns correspond to the new vulnerabilities that WAPe was equipped to detect and the following column is the total of vulnerabilities detected by WAPe (413 vulnerabilities). WAPe detected 26 zero-day vulnerabilities of the LDAPI, HI, and CS classes, plus one known SF vulnerability. The vulnerabilities found in the *Pivotx v2.3.10* and *refbase v0.9.6* (for XSS) packages were previously discovered and registered in Packet storm [18] and CVE-2015-7383. The *Community Mobile Channels v0.2.0* application was the most vulnerable mobile application with 47 vulnerabilities (SQLI and XSS mostly). This seems to confirm the general impression that the security of mobile applications is not always the best. Also interesting is the fact that the most recent version of *Clip Bucket* contains more 4 SQLI and the same 22 vulnerabilities than the previous version.

WAP v2.1 reported more vulnerabilities than WAPe, but they were false positives. The last four columns of the table show the number of false positives predicted (FPP) and not predicted (FP) by WAP (the first two columns) and WAPe (the next two columns). The original tool correctly predicted

62 false positives and incorrectly 60 as not being so. WAPe predicted 104 false positives: the same as WAP plus 42 that WAP classified as not being false positives. This means that the data mining improvements proposed in this paper made the tool more accurate and precise in prediction of false positives and detection of real vulnerabilities.

We analyzed the 18 cases reported by WAPe as not being false positives; some of them had function calls that we did not consider as symptoms, such as calls to functions *sizeof* and *md5*, whereas others contained sanitization functions developed by the applications' programmers. For example, the *vfont v0.99.3* application contains 6 of these cases, using a function named *escape* to sanitize the user inputs. To demonstrate the extensibility of the tool for such functions, we fed it with that non-native PHP function (*escape*) as being an external sanitization function and belonging to the sanitization list (see Section III-A), and we run the tool again for that application. The tool correctly did not report these 6 cases. We recall that WAP does not report candidate vulnerabilities that are sanitized. This example shows that a user can configure WAPe for a specific web application during its development, feeding WAPs with user functions developed for that application and helping the user revising the code of the application.

B. WordPress plugins

To answer the first and last questions, and to find previously-unknown (zero-day) vulnerabilities, we run WAPe with a set of 115 WordPress (WP) plugins [27], 5 of which with vulnerabilities registered in CVE [5]. WordPress is the most adopted CMS and supports plugins developed by many different teams. We selected 115 plugins from different tags (arts, food, health, shopping, travel, authentication, popular plugins and others) and distributed by several ranges of downloads, from less than 2000 to more than 500K. The popular plugins fit in this last range, having some of them more than 1M downloads. Fig. 4(a) shows the number of downloads of these plugins and Fig. 4(b) the number of web sites that have these plugins active.

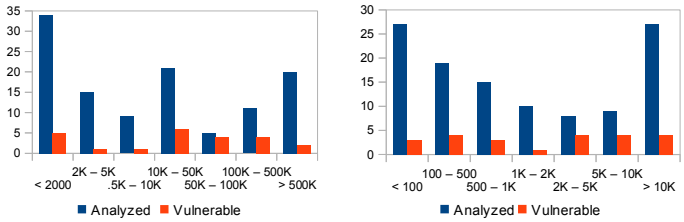
WAPe discovered 153 zero-day vulnerabilities and detected 16 known vulnerabilities. Table VII shows the 23 plugins with vulnerabilities, distributed by 8 classes. The *wpsqli* weapon detected 55 SQLI vulnerabilities, while the other detectors found the remaining 114 vulnerabilities of the XSS, RFI, LFI, DT, HI and CS classes (last 2 are new). For the known 5 vulnerable plugins (*appointment-booking-calendar 1.1.7*, *easy2map 1.2.9*, *payment-form-for-paypal-pro 1.0.1*, *resads 1.0.1* and *simple-support-ticket-system 1.2*), we confirmed the vulnerabilities using the information about them published in BugTraq [3]. However, for the *simple-support-ticket-system 1.2* plugin WAPe detected more 13 SQLI vulnerabilities than those that were registered.

The 23 plugins fit in all ranges of downloads, as depicted by the orange columns of Fig. 4(a). 16 of them have more than 10K downloads, reaching more than 500K downloads. All ranges of active WP installations contain vulnerable plugins, as shown by the orange columns of Fig. 4(b). 12 plugins are used in more than 2000 web sites. The vulnerable *Lightbox Plus Colorbox* plugin is active in more than 200,000 web sites (the most used plugin), making these web sites vulnerable to XSS attacks.

Plugin	Version	Real vulnerabilities						Total	FPP	FP
		SQLI	XSS	Files*	SCD	CS	HI			
Appointment Booking Calendar**	1.1.7	1	3					4	1	
Auth0	1.3.6		1					1		
Authorizer	2.3.6		2					2		
BuddyPress	2.4.0							0	1	
Contact formgenerator	2.0.1	11						11		
CP Appointment Calendar	1.1.7	2						2		
Easy2map**	1.2.9		1	2				3		
Ecwid Shopping Cart	3.4.6		1					1		
Gantry Framework	4.1.6		3					3		
Google Maps Travel Route	1.3.1	1	2					3		
Lightbox Plus Colorbox	2.7.2		8					8		
Payment form for Paypal pro**	1.0.1		2					2		
Recipes writer	1.0.4		4					4		
ResAds**	1.0.1		2					2		
Simple support ticket system**	1.2	18						18		
The CartPress eCommerce Shopping Cart	1.4.7	8	17					25		
WebKite	2.0.1	1						1		
WP EasyCart - eCommerce Shopping Cart	3.2.3	13	6	29	5	2	5	60		
WP Marketplace	2.4.1		9					9		1
WP Shop	3.5.3		5					5	1	
WP ToolBar Removal Node	1839		1					1		
WP ultimate recipe	2.5							0		1
WP Web Scraper	3.5		3					3		
	Total	55	71	31	5	2	5	169	3	2

*DT & RFI, LFI vulnerabilities
**plugins with vulnerabilities registered in CVE-2015-7319, CVE-2015-7320, CVE-2015-7666, CVE-2015-7667, CVE-2015-7668, CVE-2015-7669, CVE-2015-7670

TABLE VII: Vulnerabilities found by new version of WAP in WordPress plugins.



(a) Downloads (b) Active installs

Fig. 4: Downloads and active installed plugins of 115 analyzed (blue columns) and 23 vulnerable (orange columns) plugins.

Fig. 5 presents the vulnerabilities detected by class for the 17 web applications and 23 WP plugins. Clearly SQLI and XSS continue to be the most prevalent classes. Moreover, it is possible to observe that WAPe detects correctly the vulnerabilities it was extended to detect. In both analysis it detected HI and CS vulnerabilities, while LDAPI and SF were only detected in the web applications (not plugins).

All these vulnerabilities were reported to the developers of the web applications and WP plugins. Some already confirmed their existence. All were confirmed by us manually.

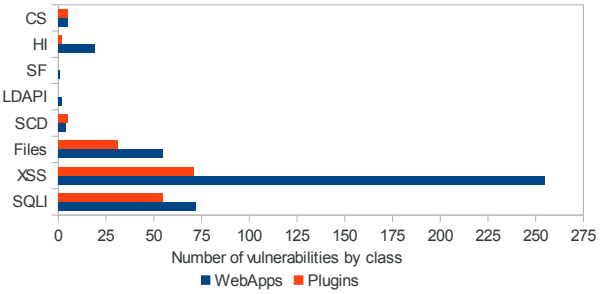


Fig. 5: Number of vulnerabilities detected by class in the vulnerable web applications and WordPress plugins.

VI. Related work

This section summarizes the main related work in the areas of static analysis and data mining used to detect vulnerabilities.

1) Static analysis: Taint analysis is a form of data flow analysis that tracks sensible data and verifies which points of the code it reaches. This form of analysis is usually used to detect vulnerabilities in source code, tracking the entry points and checking if they reach a sensitive sink. The technique uses two states – tainted and untainted – that may change during the data flow analysis. WAP [11] is a tool that performs this type of analysis to detect input validation vulnerabilities in PHP web applications. Pixy [10], phpSAFE [15], and RIPS [6] are other tools that apply the same technique to discover vulnerabilities. The first two only detect SQLI and XSS vulnerabilities. RIPS detects the same vulnerabilities as WAP v2.1, but cannot analyse object-oriented source code. phpSAFE has been configured to detect SQLI vulnerabilities in WordPress plugins, but this configuration was made in its source code, whereas in the present work we modified WAP to be configured without modifying the source code. On the contrary of the tool we present in this paper, none of these tools is modular and extensible for new vulnerabilities classes. Moreover, only WAP corrects the vulnerabilities found, fixing the web application source code and removing them.

2) Data mining: Recently data mining has started to be explored to predict the existence of vulnerabilities in software. This technique uses machine learning classifiers that are trained with data sets containing instances composed by attributes. Some tools use attributes collected from source code while others from attack vectors. PhpMiner [22] detect SQLI and XSS vulnerabilities in PHP source code. It collects attributes from excerpts of code that end in a sink, independently of where they start. It does not perform the data mining process itself, which has to be run by the user using the WEKA tool [26]. Nunan et al. retrieve attributes from a large collection of XSS attacks vectors, using document- and URL-based features, to learn how to characterize and detect XSS attacks [14]. The WAP tool also uses data mining, but in contrast with these tools, it uses that approach to predict false positives.

VII. Conclusion

The paper presents the extension of the WAP tool to detect new vulnerabilities. It addresses the difficulty of extending these tools by proposing a modular and extensible version of the WAP tool, equipping it with "weapons" to detect (and correct) vulnerabilities of new classes. The approach involved restructuring WAP to make it modular and the creation of a new module to generate weapons, i.e., to generate automatically detectors and fixes to detect and remove new classes of vulnerabilities. To predict false positives the precision and accuracy of the data mining process has been improved, adding more symptoms about false positives and instances. The new version of the tool was evaluated with 7 new vulnerability classes using web applications and WordPress plugins. The results show that this extensibility allows WAP to find many new (zero-day) vulnerabilities.

Acknowledgment

This work was partially supported by the EC through project FP7-607109 (SEGRID), and by national funds through Fundação para a Ciência e a Tecnologia (FCT) with references UID/CEC/50021/2013 (INESC-ID) and UID/CEC/00408/2013 (LaSIGE).

References

[1] OWASP WAP – Web Application Protection. https://www.owasp.org/index.php/OWASP_WAP-Web_Application_Protection.

[2] WAP. http://awap.sourceforge.net/.

[3] BugTraq. http://www.securityfocus.com.

[4] CSO Online. Ashley Madison hack exposes IT details and customer records, July 2015. http://www.csoonline.com/article/2949902/vulnerabilities/ashley-madison-hack-exposes-it-details-and-customer-records.html.

[5] CVE. http://cve.mitre.org.

[6] J. Dahse and T. Holz. Simulation of built-in PHP features for precise static code analysis. In *Proceedings of the 21st Network and Distributed System Security Symposium*, Feb 2014.

[7] DB-Engines. http://db-engines.com/en/ranking.

[8] Imperva. Anatomy of comment spam. hacker intelligence initiative. May 2014.

[9] Imperva. Web application attack report #6. Nov. 2015.

[10] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, pages 27–36, June 2006.

[11] I. Medeiros, N. F. Neves, and M. Correia. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the International World Wide Web Conference*, pages 63–74, Apr. 2014.

[12] I. Medeiros, N. F. Neves, and M. Correia. Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69, March 2016.

[13] MongoDB. https://www.mongodb.org/.

[14] A. E. Nunan, E. Souto, E. M. dos Santos, and E. Feitosa. Automatic classification of cross-site scripting in web pages using document-based and url-based features. In *Proceedings of the IEEE Symposium on Computers and Communications*, pages 702–707, July 2012.

[15] P. Nunes, J. Fonseca, and M. Vieira. phpSAFE: A security analysis tool for OOP web application plugins. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015.

[16] OWASP. Session fixation. https://www.owasp.org/index.php/Session_fixation.

[17] OWASP. Testing for NoSQL injection. https://www.owasp.org/index.php/Testing_for_NoSQL_injection.

[18] Packet storm. https://packetstormsecurity.com.

[19] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[20] D. Powers. Evaluation a monte carlo study. *CoRR*, abs/1504.00854:843–844, 2015.

[21] J. Scambray, V. Lui, and C. Sima. *Hacking Exposed Web Applications: Web Application Security Secrets and Solutions*. Mc Graw Hill, 2011.

[22] L. K. Shar and H. B. K. Tan. Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1293–1296, 2012.

[23] The Hacker News. 600tb MongoDB database 'accidentally' exposed on the internet, Nov. 2015. http://thehackernews.com/2015/07/MongoDB-Database-hacking-tool.html.

[24] WhiteHat Security. Website security statistics report. Nov. 2015.

[25] J. Williams and D. Wichers. OWASP Top 10 2013 – the ten most critical web application security risks, 2013.

[26] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

[27] WordPress. https://wordpress.org/.