Securing Energy Metering Software with Automatic Source Code Correction

Ibéria Medeiros
University of Lisboa
Faculty of Sciences, LaSIGE
Portugal
ibemed@gmail.com

Nuno F. Neves
University of Lisboa
Faculty of Sciences, LaSIGE
Portugal
nuno@di.fc.ul.pt

Miguel Correia
University of Lisboa
IST, INESC-ID
Portugal
miguel.p.correia@ist.utl.pt

***Abstract*—Industry is using power meters to monitor the consumption of energy and achieving cost savings. This monitoring often involves energy metering software with a web interface. However, web applications often have vulnerabilities that can be exploited by cyber-attacks. We present an approach and a tool to solve this problem by analyzing the application source code and automatically inserting fixes to remove the discovered vulnerabilities. We demonstrate the use of the tool with two open source energy metering applications in which it found and corrected 17 vulnerabilities. By looking in more detail into some of these vulnerabilities, we argue that they are very serious, leading to the following impacts: violation of user privacy, counter the benefits of energy metering, and serve as entering points for attacks on other user software.**

I. Introduction

The economic cost and environmental impact of energy production have been fostering the monitoring and analysis of electricity consumption information. A first aspect of this trend is the adoption of smart meters in substitution of electromechanical and electronic electricity meters. Smart meters have processing and communication abilities, which enable novel functionalities such as the transmission of the measurements to the electricity distribution system operator (DSO), the notification of power outages, and the monitoring of power quality. A second aspect of the trend is the adoption of plug-in (or plug-load) meters in home environments. These devices support the measurement of the energy consumed by the appliance(s) connected to an outlet, allowing a detailed assessment of the energy consumed. The third aspect is that industry is also using similar but more sophisticated and, in some cases, higher voltage power meters for similar purposes. The motivation is clear from an UK study that has shown that the use of these meters can lead to considerable energy savings for the organizations and carbon savings for the country [7].

Many of the meters provide energy consumption information that can be stored and processed in computing systems. This fact has been leading to the development of software to handle this information with several designations, such as *energy metering software* and *energy management software*. This software can be used for instance to store and access real-time and/or historical consumption data, plot this data in graphical format, analyze consumption trends, identify peak demand, and raise alarms about anomalous conditions (e.g., using email or SMS text messages). Many examples of such software exist, both commercial and open source. It is also being made available as software as a service (SaaS) in cloud computing environments. Examples include the high-profile but now discontinued Google PowerMeter and Microsoft Hohm services, or commercial products from companies like AlertMe and SilverSpring.

Many of these applications have web interfaces, i.e., use the browser as the interface to access the data. This simplifies the development of the software as designing graphical user-interfaces with web technologies is straightforward. However, this path also renders these applications vulnerable to common web application vulnerabilities: SQL injection, cross site scripting, command injection, file injection, etc. [26], [27]. These vulnerabilities allow malicious hackers and various kinds of cyber-criminals to modify the behavior of the application or tamper with its data. In the case of energy metering software, these attacks can have serious impacts:

- *violation of user privacy* – they can be used to access energy consumption data without authorization, which is a violation of privacy; there are even recent forms of activism against the adoption of smart meters in part due to concerns about privacy (e.g., stopsmart-meters.org);
- *countering the benefits of metering* – these attacks can modify the energy consumption information, countering the benefits of the uses of these applications; for instance, falsified data can prevent the reduction of consumption or even have the opposite result;
- *attack other user software* – these attacks can use the vulnerable software as a platform to attack users, for instance, by accessing private data in their computers or compromising their browsers.

Most of these vulnerabilities correspond to bugs in the application source code, so they can be avoided simply by writing secure software. However, programmers often do not have adequate knowledge about secure programming practices [11], [8], or simply make human errors.

In this paper we present an approach and a tool to solve this problem and their use with energy metering software. The proposed approach has mainly three aspects: (1) analyzing the application source code searching for vulnerabilities, (2) inserting fixes in the source code that correct these flaws, and (3) producing a report that assists the programmer in learning how to avoid inserting similar vulnerabilities. This approach is

implemented in the WAP (Web Application Protection) tool, which automates these steps. The current version of WAP runs with PHP code, which is a language designed specifically for web applications and currently the one that is used by the majority of such applications [14][1]. WAP analyzes the source code of a PHP application and outputs a corrected version and a report.

The paper shows the use of WAP with two open source energy metering applications: *emoncms* and *measureit*. *emoncms* is being developed in the context of the OpenEnergyMonitor project[2]. It is an open framework for processing, logging and visualizing energy, temperature and other environmental data. *measureit*[3] is a simpler application that allows storing and accessing voltage and temperature data. WAP discovered and corrected 17 confirmed vulnerabilities in these two applications, 3 SQL injection and 14 cross site scripting.

II. Energy Metering Software

An energy metering system can have different degrees of size and complexity. In the paper, we will consider a simple scenario, where meters input data into energy metering software. Although the evaluation section focus only on two open source energy metering applications, we analyzed others. From this analysis we concluded that most of them follow a three-tier architecture. The user interface is a browser in the user's computer and the application itself has modules that can be assigned to three tiers (in some cases, the first two tiers are difficult to separate):

- presentation tier – modules concerned with the direct interaction with the user and display of information;
- logic tier – modules that implement the core of the application;
- data tier – concerned with the storage of data; applications use for instance a MySQL database management system (DBMS) to save data.

The two applications that were tested were *emoncms*, which is part of the OpenEnergyMonitor project, and *measureit*. The OpenEnergyMonitor is an open source project that develops hardware and software for energy monitoring. *emoncms* is not a static software package, but a configurable framework with many modules that can be combined in different ways. Its main objective is the processing and visualization of energy data.

emoncms includes five core modules: input, feed, visualizations, dashboard, and user. The *input module* pre-processes meter input before it is inserted in the database, e.g., for creating histogram data. The *feed module* provides functionality for inserting, storing and retrieving time stamped data in the database. The *visualizations module* can analyze large data sets and generate graphics in different formats. The *dashboard module* includes the dashboard builder and viewer. The former is a visual editor that allows creating dashboards out of several widgets and visualizations. The *user module* handles user actions and data, including authentication and sessions.

emoncms provides other modules that are optional. For instance, there are Raspberry PI and Arduino modules to interface the application with these boards. Another example is the SAP calculator module that fills and helps the user to understand the worksheet of the UK's Standard Assessment Procedure for Energy Rating of Dwellings [5]. Most *emoncms* modules are implemented in PHP, but a few are in JavaScript and Python.

measureit is a much simpler software package, mostly focused on storing and visualizing voltage and temperature data obtained with Current Cost meters[4]. *measureit* can for instance, display last hour / last day / last 7 days / last 30 days energy consumptions and the associated costs. It can display graphics with the evolution of the measurements of a meter, the daily / weekly / monthly usage of energy, and several other forms of statistic data. The applications is mostly written in JavaScript, Python and PHP.

III. The WAP Approach and Tool

This section presents our approach to automatically detect and correct vulnerabilities in web applications. We present the approach in terms of the WAP tool. More details can be found on a technical report [19].

A. Input validation vulnerabilities

The highest risk vulnerabilities to which web applications are exposed are input validation vulnerabilities [27]. These flaws are avoidable by doing proper validation or sanitization of user input. Consider for instance SQL injection, usually considered the main web application vulnerability for several years. The flaw consists in inserting user input in an SQL query without adequate validation/sanitization. For instance, the following line in PHP is vulnerable to SQL injection:

```
echo mysql_query("SELECT * FROM users
WHERE username='$_POST['user']' AND
password='$_POST['password']'");
```

The problem with the line is that the two parameters, username and password, are embedded in the query with the expectation of real usernames and passwords being received. If that is the case, information about an user that provides his password is echoed. However, if a malicious user inserts as username `administrator' --` and an empty password, the information of the user *administrator* is printed because the query becomes[5]:

```
SELECT * FROM users WHERE
username='administrator' -- AND password=''
```

B. WAP tool

The WAP tool runs essentially three steps: (1) static analysis of the source code searching for input validation vulnerabilities; (2) correction of the source code by inserting fixes, i.e., instructions that validate the input; (3) report the vulnerabilities detected and how they were corrected. Figure 1 shows its main components.

[1]Supporting more than one language requires a great effort and is beyond the purpose of a proof-of-concept prototype as WAP.

[2]http://openenergymonitor.org/

[3]https://code.google.com/p/measureit/

[4]http://www.currentcost.com/

[5]Recall that -- indicates that the rest of the line is a comment, which should not be interpreted by the DBMS.

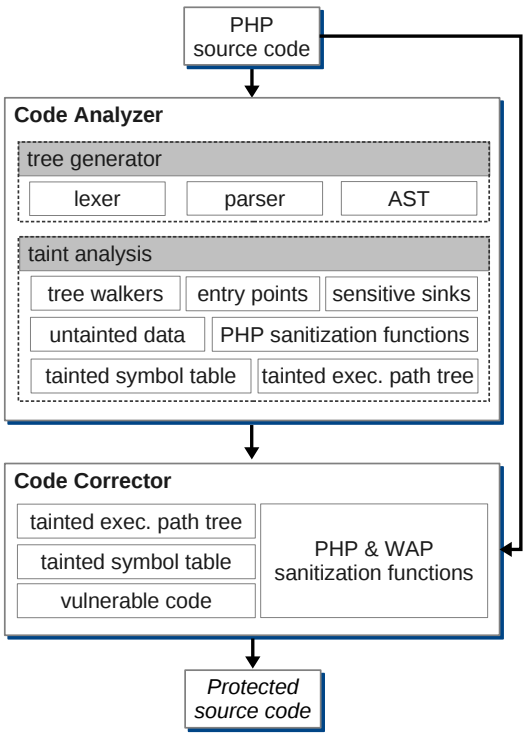

Fig. 1. Architecture of WAP with its main modules and data structures.

The first step, static analysis, starts with a phase similar to what is done by the front end of a compiler. It first parses the source code and generates an abstract syntax tree (AST) that represents that code. Then it does taint analysis: starting from an entry point (e.g., `$_POST`), it follows the code by walking through the AST to see if that input reaches a sensitive sink (e.g., `mysql_query`) without proper validation. If such a case is found, it is a vulnerability. In the trivial example above both the entry points and the sensitive sink are in the same line of code, but that is not usually the case as input can follow complex paths through the code.

The idea of taint analysis is to consider that inputs are tainted (not trusted, compromised) and to propagate this taintedness to variables that take this input. For instance, line `$u=$_POST['user']` propagates the taintedness of the input on the right hand side to the variable `$u` on the left. Sanitization functions, on the contrary, remove taintedness. For instance, if variable `$u` was used in a query to a MySQL database and the assignment was `$u=mysql_real_escape_string($_POST['user'])`, the variable `$u` would be untainted because the PHP function `mysql_real_escape_string` sanitizes or escapes a string (i.e., it substitutes dangerous characters like `'` by escaped versions, e.g., `\'`).

The second step, code correction, involves identifying the fix to insert for each vulnerability found, to identify the place in the source code where the fix needs to be inserted, and to modify the file where that place is. For instance, for the SQL injection vulnerability mentioned above the fix consists in inserting calls to `mysql_real_escape_string`.

The third step is simply to write a report describing the vulnerability and the inserted fix.

The tool has mainly two components: the code analyzer and the code corrector. The former includes a lexer and a parser created using ANTLR (ANother Tool for Language Recognition) [21] and written in Java. The other component of the code analyzer is the taint analyzer that is composed mainly of tree walkers, i.e., of code that walks through the ASTs to identify vulnerabilities.

The current version of the WAP tool addresses the 8 classes of vulnerabilities most common in PHP: SQL injection (SQLI), cross site scripting (XSS), remote file inclusion, local file inclusion, path traversal, source code disclosure, OS command injection and direct dynamic code evaluation (eval injection). The tool is configured with a list of around 10 entry points (like `$_POST`), more than 40 sensitive sinks (like `mysql_query`) and one or more sanitization functions per class of vulnerability (like `mysql_real_escape_string`).

C. Challenges

WAP does static analysis of source code, which is known to be an undecidable problem for non-trivial languages [16]. To circumvent this result, WAP and, to the best of our knowledge, all other static analysis tools do only partial analysis of some language constructs, so they are not sound [2], [6]. In WAP's code analyzer this issue appears in conditional statements like if-then-else, in which in some cases it is not possible to know which branch will be executed. The impact of this is that if, for instance, a variable is marked tainted in a branch but not in the other, WAP is pessimistic and marks it tainted. A similar issue happens with loops, which may be executed zero, one or more times.

WAP can raise false positives – detect nonexistent vulnerabilities – for a number of reasons. An example are functions written by the programmer that validate or sanitize input by manipulating this input (typically a string). Since it is difficult to understand if such function does the check correctly, WAP does not untaint the variable(s) affected, leading to false positives. This kind of false alarm can be burdensome with static analysis tools that produce lists of potential vulnerabilities to be validated by humans. This is not the case in WAP as the tool corrects the vulnerabilities automatically. If WAP adds a fix to an nonexistent vulnerability, this does not modify the behavior of the application, at least based in our current experience.

To reduce the number of false positives, the tool performs advanced forms of analysis. It does global, interprocedural and context-sensitive analysis, which means that data-flows are followed even when they enter new functions and other modules/files. This involves handling several data structures, global variables, and resolving module names (which can contain paths taken from web application environment variables).

IV. Vulnerabilities Discovered

We used the WAP tool to analyze *emoncms* and *measureit*. The tool focus the analysis on the PHP files of those applications, as most of the interaction with the users is performed in this code. Therefore, they contain most of the attack surface of the applications, and this form of analysis should find highly relevant vulnerabilities from a security standpoint.

Table I displays the modules that were analyzed and summarizes the results. The table presents the number of PHP files analyzed, the lines of code (LOCs) per application and per file with vulnerabilities, the number of SQL injection and cross site scripting vulnerabilities found (no vulnerabilities of the other classes were discovered). The vulnerabilities that we managed to attack are in bold; the others 3 may be false positives (see Section IV-D). In the following sections we will discuss three representative vulnerabilities.

webapp / vuln. files	files	app. LOCs	file LOCs	SQLI	XSS
emoncms - modules	7	1,089			
- igraph3.php			63		**3**+3
emoncms - extras	7	291			
- embed.php			48		**1**
- kwhdstacked.php			47		**1**
- kwhdzoomer.php (old)			44		**1**
- kwhdzoomer.php			72		**3**
emoncms - examples	62	5,496			
- user.php			177	**2**	**1**
measureit v.1.14	2	967			
- measureit_functions.php			915	**1**	**4**
total	78	7,843	1,366	3	17

TABLE I. SUMMARY OF THE ANALYSIS OF EMONCMS/MEASUREIT.

A. emoncms: reflected cross site scripting

WAP found several cross site scripting (XSS) vulnerabilities in *emoncms*, four of which we managed to attack. Figure 2 shows an excerpt of the output of the tool for file `kwhdzoomer.php` of the visualizations module. The figure shows the two vulnerabilities and the respective fixes. Interestingly the two involve lines quite far apart in the source file, which make them hard to find manually: the first in lines 18 and 69; the second in lines 17 and 70.

A reflected XSS vulnerability is a piece of code that essentially sends input back to the user without sanitization. For instance, in the first vulnerability in the figure the input parameter `kwhd` is written in a page sent back to the user. This type of vulnerability is usually considered very serious, and appears in the second place in the OWASP top 10 rank of web vulnerabilities [27]. An attacker can exploit it to make the user send a malicious script – usually written in JavaScript – to the web server that is reflected and executed in the browser. The solution for this vulnerability is to encode the input in such a way that the script is interpreted in the browser as text to be displayed, not as code to be executed. This is done by a function that is specific of the WAP tool, `san_out`, which calls functions of the OWASP PHP Anti-XSS Library[6].

An attacker can exploit a vulnerability like these two to achieve any of the three impacts mentioned in the introduction. The execution of the malicious script can *violate user privacy* by accessing user data in the *emoncms* server and sending it to some server controlled by the attacker. The script can *counter the benefits of metering* by sending a request to the server causing the modification of the data stored there. Finally, the script can *attack the software user* directly for instance by stealing his cookies and sending them to some server, or by running an exploit against the browser or some add-on (e.g., to the Java runtime in which many vulnerabilities have been discovered in recent months). This user can be, for instance, an engineer or an administrator of a company, so this can be used as platform for another more pernicious attack.

[6]http://code.google.com/p/php-antixss/

```
= = = =  Vulnerability n.: 1  = = = =
Vulnerable code:
18: $kwhd = $_GET['kwhd'];
69: echo $kwhd;

Corrected code:
18: $kwhd = san_out($_GET['kwhd']);
69: echo $kwhd;

= = = =  Vulnerability n.: 2  = = = =
Vulnerable code:
17: $power = $_GET['power'];
70: echo $power;

Corrected code:
17: $power = san_out($_GET['power']);
70: echo $power;
```

Fig. 2. Output of the WAP tool showing two reflected XSS vulnerabilities found in *emoncms*' file `kwhdzoomer.php`.

B. emoncms: SQL injection

emoncms provides a set of example files for user authentication. These files are not part of the core system, but can be set as the entry page as they are or after being modified. WAP found the same two SQL injection vulnerabilities in several of these files, for instance in `user.php`. The output of the tool for one of these vulnerabilities can be seen in Figure 3. SQL injection was already explained in Section III-A. In this case, the code inserts supposedly an username provided by the user into the WHERE clause of an SQL query, but this username is not validated.

```
Vulnerable code:
140: $username = $_POST['username'];
144: $result = db_query("SELECT id,password,
salt FROM users WHERE username = '$username'");
16: return $result = mysql_query($query); (/home/
iberiam/Desktop/Grib/emoncms_1/emoncms_examples
-master/feed01/includes/db.php)

Corrected code:
140: $username = mysql_real_escape_string($_POST
['username']);
144: $result = db_query("SELECT id,password, salt
FROM users WHERE username = '$username'");
16: return $result = mysql_query($query); (/home/
iberiam/Desktop/Grib/emoncms_1/emoncms_examples
-master/feed01/includes/db.php)
```

Fig. 3. Output of the WAP tool showing one of the two SQLI vulnerabilities found in *emoncms*' file `user.php`.

Interestingly the structure of the query in line 144 does not allow the most common SQLI attacks: (1) circumventing the process of logging in (as in the example of Section III-A); (2) changing a SELECT query to extract more/other information than what was desired by the programmer (because the application does not print the output of the SELECT). However, it does allow attacks. For example, in the attacker can insert as username:

```
' OR 1=1 INTO OUTFILE '/var/www/html/vulnsite/
login-info.html' --
```

The resulting query is:

```
SELECT id,password, salt FROM users
WHERE username = '' OR 1=1 INTO OUTFILE
'/var/www/html/vulnsite/login-info.html' --
```

The substring `OR 1=1` creates a tautology, so the query writes all user identifiers, salts and hashed passwords in file `login-info.html`. The attack requires that the attacker knows the document root directory of the web site (`/var/www/html/vulnsite/`) but this is normally easy to find by trial and error or by using a spider tool. It also requires that this directory can be written by the application, which is often the case.

The second step is the attacker accessing the `login-info.html` file (e.g., by opening in the browser the URL `http://www.vulnsite.com/login-info.html`) and doing a brute force attack or a dictionary attack to find user passwords in a few seconds or minutes. From there the attacker can impersonate a valid user and do anything that he can do. The attacker can *violate user privacy* by accessing user data in the *emoncms* server or *counter the benefits of metering* by modifying this data.

C. measureit: stored cross site scripting

measureit has also several XSS vulnerabilities, one of them a stored XSS (file `measureit_functions.php`). This variant of XSS involves two steps: first the attacker inserts the malicious script in the application's database; second, the script is sent to one or more users. In the *measureit*'s stored XSS vulnerability, the first step of the attack consists in inserting a malicious script in the parameter `sensor_name`, which should take the name of a sensor. Then this parameter is inserted in the database by the following line:

```
$db->query("INSERT INTO measure_sensors
( sensor_id, sensor_title ) VALUES (
'$params[sensor_id]', '$params[sensor_name]'
)");
```

The second step happens whenever an user accesses this field of the database. Consider the case in which the attacker inserted the following input in `sensor_name`:

```
<script>alert(\'Sensor 1 - XSS\')</script>
```

In the panel of the application, whenever an user clicks on Sensor 1, the script is executed. In this example the script is not really malicious: it simply shows a window with "XSS" written. Figure 4 shows the effect of this action.

The impact of this vulnerability is similar to the reflected XSS in *emoncms*: possible violation of user privacy, countering the benefits of metering, and attacking the software user. However, the risk is higher because any user that accesses the application can be affected.

D. False positives

In Section III-C we discussed the challenges of doing static analysis of source code and explained that the problem is undecidable. After running WAP we tried to attack all the

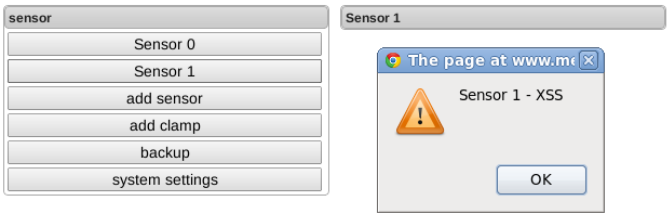

Fig. 4. Execution of the example script inserted exploiting the stored XSS vulnerability in *measureit*.

vulnerabilities found. We were successful with the vulnerabilities in bold in Table I, but not with the 3 in normal font (all in *emoncms*). This can mean one of two things: these were false positives as they are not vulnerabilities; or they are vulnerabilities but for some reason we were not successful exploiting them.

The 3 possible false positives happened in file `igraph3.php`. The issue is that the input comes through the `$_SERVER` entry point and is concatenated with other data and manipulated in such a way that the script no longer runs in the browser, even if there is no explicit sanitization.

V. Related Work

We use the output of static analysis to remove vulnerabilities automatically. We are aware of a few works that search for the specific case of SQL injection vulnerabilities, but usually without attempting to insert fixes in a way that can be replicated by a programmer. For instance, AMNESIA, does static analysis to discover SQL queries, not necessarily vulnerable, and in runtime checks if the call being made satisfies the format defined by the programmer [9]. Buehrer et al. do something similar by comparing in runtime the parse tree of the SQL statement before and after the inclusion of user input [4]. WebSSARI is much simpler than WAP (e.g., does not do interprocedural and context-sensitive analysis) but inserts some kind of fixes, although no information is given about what those fixes are [13].

There are several other techniques to find vulnerabilities other than static analysis. Testing finds bugs or vulnerabilities by executing the code [12]. Web vulnerability scanners use signatures of vulnerabilities to detect if they exist in a web site, many discrepancies between the results of different scanners have been identified [25]. Fuzzing and fault/attack injection search for vulnerabilities by injecting well-chosen inputs and trying-out a wide range of possible inputs [3], [1]. Neither of these techniques correct the code.

Dynamic taint analysis tools do taint analysis in runtime. PHP Aspis does dynamic taint analysis of PHP applications with the objective of blocking XSS and SQLI attacks [20]. ARDILLA observes the execution of a PHP web application then generates inputs trying to exploit XSS or SQLI vulnerabilities [15].

There are also tools and mechanisms that protect applications in runtime without searching for vulnerabilities. For instance, CSSE protects PHP applications from SQLI, XSS and command injection by modifying the platform to distinguish between what is part of the program and what is user input,

defining checks to be performed to the user input [22].WASP does something similar to block SQLI attacks [10].

We were unable to identify other work on the analysis of security vulnerabilities of energy metering software. There is however a considerable literature on the privacy of the use of smart meters. A recent study has shown that smart meters that use wireless communication can let anyone monitor the energy usage of hundreds of homes with a modest technical effort [24]. The authors propose defenses at communication level involving jamming. Lisovich et al. go further and show that it is possible to extrapolate activity information from power-consumption data [17]. Other authors proposed a protocol based on zero-knowledge proofs to allow the DSO to bill consumers about their energy consumption without the consumers disclosing consumption data [23]. McLaughlin et al. present electrical mechanisms to reach the same goal [18].

VI. Conclusion

The paper presents an approach and a tool called WAP to automatically identify and correct vulnerabilities in web applications. We used the tool to analyze two open source energy metering applications, *emoncms* and *measureit*. WAP identified and corrected 17 vulnerabilities in these two applications, 3 SQL injection and 14 cross site scripting vulnerabilities. It also identified other 3 XSS vulnerabilities in *emoncms* that may be false positives since we did not manage to attack them. We discuss three of the vulnerabilities found: a reflected XSS in *emoncms*, a SQL injection in *emoncms*, and a stored XSS in *measureit*. We provide an argument that these vulnerabilities can allow a malicious hacker or a cyber-criminal to achieve three impacts: violation of user privacy, countering the benefits of energy metering, and attacking other software of the user.

Acknowledgment

This work was partially supported by the EC through project FP7-257475 (MASSIF) and the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009), the Multiannual Program (LASIGE), and contract PEst-OE/EEI/LA0021/2013 (INESC-ID).

References

[1] J. Antunes, N. F. Neves, M. Correia, P. Verissimo, and R. Neves. Vulnerability removal with attack injection. *IEEE Transactions on Software Engineering*, 36(3):357–370, Mar. 2010.

[2] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *Proceedings of the 30th International Conference on Software Engineering*, pages 211–220, May 2008.

[3] R. Banabic and G. Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*.

[4] G. T. Buehrer, B. W. Weide, and P. A. G. Sivilotti. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, pages 106–113, Sept. 2005.

[5] Building Research Establishment. The government's standard assessment procedure for energy rating of dwellings. http://www.bre.co.uk/filelibrary/SAP/2012/Draft_SAP_2012_December_2011.pdf, Dec. 2011. Draft 2012 edition.

[6] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30:775–802, 2000.

[7] CarbonTrust. Advanced metering for SMEs: Carbon and cost savings, May 2007.

[8] M. A. Davidson. The Supply Chain Problem, Apr. 2008. http://blogs.oracle.com/maryanndavidson/2008/04/the_supply_chain_problem.html.

[9] W. Halfond and A. Orso. AMNESIA: analysis and monitoring for neutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183, Nov. 2005.

[10] W. Halfond, A. Orso, and P. Manolios. WASP: protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34(1):65–81, 2008.

[11] M. Howard and D. LeBlanc. *Writing Secure Code. 2nd edition*. Microsoft Press, 2003.

[12] Y. Huang, S. Huang, T. Lin, and C. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th International Conference on World Wide Web*, pages 148–159, 2003.

[13] Y. W. Huang, F. Yu, C. Hang, C. Tsai, D. Lee, and S. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th International World Wide Web Conference*, pages 40–52, May 2004.

[14] Imperva. Hacker intelligence initiative. Technical Report 8, Imperva, Apr. 2012.

[15] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, May 2009.

[16] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 1992.

[17] M. A. Lisovich, D. K. Mulligan, and S. B. Wicker. Inferring personal information from demand-response systems. *IEEE Security and Privacy*, 8(1):11–20, Jan. 2010.

[18] S. McLaughlin, P. McDaniel, and W. Aiello. Protecting consumer privacy from electric load monitoring. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 87–98, 2011.

[19] I. Medeiros, N. Neves, and M. Correia. WAP: Automatic detection and correction of web application vulnerabilities. Technical report, INESC-ID, 2013. http://homepages.gsd.inesc-id.pt/∼mpc/pubs/wap-tr.pdf.

[20] I. Papagiannis, M. Migliavacca, and P. Pietzuch. Php aspis: using partial taint tracking to protect against injection attacks. In *Proceedings of the 2nd USENIX Conference on Web application Development*, 2011.

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

[22] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID'05 Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, pages 124–145, Sept. 2005.

[23] A. Rial and G. Danezis. Privacy-preserving smart metering. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society*, pages 49–60, 2011.

[24] I. Rouf, H. Mustafa, M. Xu, W. Xu, R. Miller, and M. Gruteser. Neighborhood watch: Security and privacy analysis of automatic meter reading systems. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 462–473, 2012.

[25] M. Vieira, N. Antunes, and H. Madeira. Using web security scanners to detect vulnerabilities in web services. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks*, July 2009.

[26] Web Application Security Consortium. WASC Threat Classification. Technical report, Jan. 2010. Version 2.00.

[27] J. Williams and D. Wichers. OWASP Top 10 - the ten most critical web application security risks (2010). Technical report, OWASP Foundation, 2010.