Implementation of Tainted Mode approach to finding security vulnerabilities for Python technology

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

***Abstract*—Most web applications contain security vulnerabilities that result in high rate of SQL injection and cross-site scripting attacks present-day. The most of vulnerabilities is the result of improper or none input validation by web application. The Tainted Mode approach is widely used for detection of such vulnerabilities. This paper presents the implementation of Tainted Mode approach to finding security vulnerabilities for Python technology.**

***Index Terms*—Input validation, Python, Tainted Mode, Web security, Web vulnerabilities**

I. INTRODUCTION

Web applications have widely spread in present-day information systems. Web applications have such advantages as intuitive user interface, ability to deliver its services over Internet, and speed of the development process. As web applications are developed by programmers with little or no security in mind and thereafter become available over public networks, many problems of security concern are emerged. Security vulnerabilities in web applications potentially let attacker to steal confidential data, break data integrity or affect web application availability. The task of hardening web applications security is one of the most urgent for now: according to Acunetix survey [1] 60% of found vulnerabilities address web applications.

The most common way of hardening web applications security is searching for security vulnerabilities therein in order to fix them up. This paper presents the implementation of the Tainted Mode approach to automatically finding vulnerabilities in web applications developed using Python technology.

II. CASE STUDY

According to OWASP [8] researches, the most efficient way of finding security vulnerabilities in web applications is manual code review. This technique is very time-consuming, requires high expert skill, and is prone to overlooked errors. Therefore the security society actively develops automated approaches to finding security vulnerabilities.

Automated approaches to finding security vulnerabilities can be divided in two wide categories: black-box and white-box testing.

The first approach is to analyze web application functioning from the user side, assuming that source code of an application is not available [2]. The idea is to submit into application's web forms various malicious patterns (implementing for example SQL injection or cross-site scripting attacks) and to analyze its output thereafter. An assumption of possible vulnerability is made if any application errors are observed. This approach does not guarantee any precision or completeness of its reports.

The second approach is to analyze web application from the server side, assuming that source code of an application is available. In this case dynamic or static analysis techniques can be applied. The widely known Tainted Mode paradigm is used to find security vulnerabilities when the source codes are available. Tainted Mode approach makes an assumption that all user input through HTTP-request is untrustworthy and should be sanitized by special functions before it is used in security critical operations, e.g. database queries or construction of HTTP responses.

Perl [3] was the first scripting language where Taint Mode was implemented in interpreter by means of dynamic analysis. Other implementations of Tainted mode were done later for Ruby [4], PHP [5], and Java [6]. After all Tainted Mode was implemented in PHP [7, 15] and Java [9] by means of static analysis.

However, there is no implementation of Tainted Mode for Python technology. Although Python is spreading fast inside web development (see Trac [10], Wiki [11], LiveJournal [12]). The purpose of this work was to choose between dynamic and static applications of Tainted Mode to Python and its implementation thereafter.

III. DYNAMIC VERSUS STATIC ANALYSIS

Dynamic analysis is the process of extracting program's properties by executing and monitoring it on different input data. The main problem here is to prepare such input data that would cover the most part of possible executable paths.

On the contrary, static analysis is the process of extracting program's properties from the source code solely, without executing it. The main drawback of the static analysis is the algorithmic undecidability of such problems as Must-alias and Path infeasibility [13]. The undecidability leads to imprecision of the reports made by static analysis. Hence, the expert is required to check the correctness of every result.

There is one more drawback of the static analysis technique in application to web technologies: mostly all of them are based on interpreted languages (PHP, Perl, Ruby, Python, Java, MSIL) that have an important ability to evaluate string variables as code. This feature causes incompleteness of the static analysis in application to interpreted languages.

The choice for Tainted mode implementation by means of dynamic analysis was made due to former reasoning about static analysis.

IV. Tainted Mode Implementation

In order to define more precisely the sense of security vulnerability caused by improper (or absent) input validation the following assumptions were made:

1) All data received from the client via HTTP-requests is considered untrustworthy (or tainted). This is the most restrictive assumption about possible uses of web application.
2) All data that is local to web application is considered trustworthy (or untainted). This assumption states that there are no malicious users that can affect system's security from inside. Every communication is made solely via HTTP requests and responses.
3) Any untrustworthy data can be made trustworthy by special kinds of processing named sanitization. There are such functions in every web application development technology.

With these assumptions made, security vulnerability is defined as 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 files in future.
3) Untrustworthy (tainted) data should not be used 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.

In order to implement the stated rules, a modification was made to Python interpreter. The representation of built-in *String* class was modified to contain Boolean flag that tracks taintedness of the corresponding string value. The functions that should not receive untrustworthy data are named critical functions.

The interpretation process of a particular Python program can be viewed as follows:

1) Assign user input via HTTP parameters to program variables. Mark them tainted;
2) If any string variable interacts with tainted string values through assignments, concatenation, etc., mark it tainted too;
3) If any tainted string value is passed through a sanitization routine, mark the value untainted:
4) If any tainted string value is passed to a critical function, raise an error reporting about possible vulnerability.

In order to operate, this process of interpretation should be configured with list of functions that return user input, that sanitize user input, and that are critical to user input. Since web applications are usually developed under certain frameworks, function names used to retrieve HTTP input, construct HTTP output or to communicate with external services may be distinct in different web applications. Hence, the implementation of Tainted Mode should be flexible enough to be configured for different frameworks used by web applications under analysis.

The developed configuration scheme contains three lists of function names:

1) The first list contains names of functions that return user input from HTTP-requests. Every string that is returned from such function gets tainted flag in signaled state.
2) The second list contains names of functions that sanitize user input. String value gets tainted flag in unsignaled state if a function from this list was applied to it.
3) The third list contains names of critical functions, such as wrappers to system calls, database queries, etc. If string value with signaled tainted flag is used as an argument to such function, an exception is raised, reporting about possible vulnerability.

It can be seen that if there is need to analyze web application developed under certain framework, the only work to be done is to populate the named function lists with appropriate values.

V. Conclusion

This paper has described the implementation of Tainted Mode approach to finding security vulnerabilities caused by improper (or absent) input validation.

Necessary changes were applied to Python interpreter in order to achieve the desired features of Tainted Mode: setting of Taint value, its propagation through program statements, and reaction of passing Tainted values into security critical functions. The tool uses three configuration files. The first file contains settings which functions are considered to return user input, the second file – which functions are considered to sanitize user input and the third – which functions are considered critical. Thus, the tool could be easily reconfigured to support different frameworks if needed. The initial configuration was made to support analysis of web application developed for Mod_python [14] framework.

As ongoing work, there are plans to extend configuration files of the tool with the class Id, attached to every sanitization

and critical function listed therein. The idea is that not every sanitization function is suitable for processing of untrustworthy input data. For example, function that encodes HTML tag symbols (‘<’ and ‘>’ to ‘<’ and ‘>’) and is suitable for sanitizing data before HTTP response construction will not sanitize data used in SQL queries construction. By this extension the tool will be able to produce more accurate results in security vulnerabilities detection.

REFERENCES

[1] M. Andrews, “The State of Web Security”, IEEE Security & Privacy, vol. 4, no. 4, pp. 14-15, 2006.

[2] L. Auronen, “Tool-Based Approach to Assessing Web Application Security”, Seminar on Network Security, 2002.

[3] D. Ragle. “Introduction to Perl's Taint Mode.” [Online]. Available: http://www.webreference.com/programming/perl/taint/

[4] D. Thomas, Ch. Fowler, A. Hunt, “Programming Ruby: The Pragmatic Programmer’s Guide”, Addison Wesley Longman, Inc, 2001.

[5] N.-T. Anh, S. Guarnieri, D. Greene, J. Shirley, D. Evans, “Automatically Hardening Web Applications Using Precise Tainting”, IFIP Security Conference, 2005.

[6] V. Haldar, D. Chandra, M. Franz, “Dynamic Taint Propagation for Java”, Proceedings of the 21st Annual Computer Security Applications Conference, 2005.

[7] N. Jovanovic, Ch. Kruegel, E. Kirda, “Pixy: A Static Analysis Tool for. Detecting Web Application Vulnerabilities”.

[8] M. Curphey, A. Wiesman, A. Van der Stock, R. Stirbei, “A Guide to Building Secure Web Applications and Web Services”, OWASP, 2005.

[9] V. Livshits, M. Lam, “Finding security errors in Java programs with static analysis”, In Proceedings of the 14th Usenix Security Symposium, 2005.

[10] The Trac Project. [Online]. Available: http://trac.edgewall.org/.

[11] MoinMoinWiki Engine. [Online]. Available: http://moinmoin.wikiwikiweb.de/.

[12] LiveJournal. [Online]. Available: http://www.livejournal.com/.

[13] W. Landi, “Undecidability of static analysis”, ACM Letters on Programming Languages and Systems, 1992.

[14] Mod_python – Apache/Python Integration. [Online]. Available: http://www.modpython.org/.

[15] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, S.-Y. Kuo, “Securing web application code by static analysis and runtime protection”, In WWW '04: Proceedings of the 13th International Conference on World Wide Web, 2004.