SUPPORTING SECURITY TESTERS IN DISCOVERING INJECTION FLAWS[1]

Sven Türpe, Andreas Poller, Jan Trukenmüller, Jürgen Repp and Christian Bornmann

Fraunhofer-Institute for Secure Information Technology SIT, Rheinstrasse 75,
64295 Darmstadt, Germany
{tuerpe,poller,repp,truken,borni}@sit.fraunhofer.de

ABSTRACT

We present a platform for software security testing primarily designed to support human testers in discovering injection flaws in distributed systems. Injection is an important class of security faults, caused by unsafe concatenation of input into strings interpreted by other components of the system. Examples include two of the most common security issues in Web applications, SQL injection and cross site scripting. This paper briefly discusses the fault model, derives a testing strategy that should discover a large subset of the injection flaws present, and describes a platform that helps security testers to discover injection flaws through dynamic grey-box testing. Our platform combines the respective strengths of machines and humans, automating what is easily automated while leaving to the tester the artistic portion of security testing. Although designed with a specific fault model in mind, our platform may be useful in a wide range of security testing tasks.

1. INTRODUCTION

A number of specific faults are commonly associated with software insecurity as they commonly occur and are commonly exploited. One major class of vulnerabilities is injection flaws, comprising e.g. cross-site scripting, SQL injection and others that share the same root cause.

Security testing is often approached in a top-down manner, starting with a theory how to test under some idealized conditions and subsequently putting it into practice where it may or may not work well. This paper presents the Software Security Evaluation System (SSES), a tool that went the opposite way. The SSES has been developed from a set of auxiliary scripts towards a concept that may in the end yield a deeper understanding of vulnerabilities and security testing.

While having been developed with injection vulnerabilities in mind, our tool has grown to support a variety of security testing tasks: fuzzing; observation; event correlation; some aspects of reverse engineering; spoofing, replay and automated interaction; application of test cases to different protocols and interfaces; and others. It is really a workbench full of tools that are frequently needed in security testing, and capable of integrating more. Underlying all the individual functions is a common idea: the sensor-actuator approach.

The remainder of this paper is organized as follows: Section 2 discusses related literature. Section 3 outlines the fault model of injection attacks that motivated tool development. The tool itself is described in section 4 and examples of its application in section 6.

2. RELATED WORK

Attempts to support or automate vulnerability testing are manifold, and the existing body of research has inspired the design of our tool in many ways.

Fuzzing [1, 2, 3] and sandboxing (such as in *Holodeck*) are techniques that our tool is capable of integrating. What we add are security-specific considerations how to connect such tools. Testing using fault injection is discussed in [4, 5].

Commercially available vulnerability scanners for Web applications attempt to find unknown vulnerabilities. They use a crawler to explore the HTTP interface of the application, then use a database of typical attack inputs in a way similar to fuzzing. Besides being limited to Web applications they do not live up to their promises[6, 7]. More advanced is WAVES[8, 4], which aims at SQL injection and cross site scripting vulnerabilities. WAVES is a research prototype and uses artificial intelligence along with other interesting concepts.

Large amounts of research exist regarding specific types of flaws, most often in Web applications [9, 10, 11, 12] as well as on white-box testing methods. We attempt to be more generic. Finally, tainting [13, 14] is an important technique. The idea is to consider input as tainted until sanitized, and follow its whereabouts through the program.

3. FAULT MODEL AND TESTING STRATEGY

The design of the SSES is based upon a rough fault model of injection vulnerabilities as described in this section.

3.1. Injection Vulnerabilities

Injection vulnerabilities come in a variety of flavours, e.g.:

Path traversal. Input is used in a file name such that malicious input causes undesired changes to the path.

[1]This work was supported by CASED, www.cased.de

Command injection. An input value is embedded in a command to be executed within the program or elsewhere.

Cross site scripting (XSS). An input value is embedded in a dynamically created Web page such that malicious input is interpreted as JavaScript code by the Web browser.

Content spoofing. A URL received as input is used as a redirect or link target, or as a frame or image source.

SQL injection. Input is used to construct a database query, which can be altered by malicious input.

XPath injection. Comparable to SQL injection, but attacking XML databases instead of relational ones.

LDAP injection. Like SQL and XPath injection, the target being queries the application makes to an LDAP server.

Although these classes of vulnerabilities are often treated as different and unrelated, they have in common their underlying fault model and root cause.

3.2. Common Root Cause

While details vary, we assume the general setting for injection flaws to be always the same:

1. There are at least two distinguishable components.
2. One of the component, the front-end, is exposed to the the attacker while the other, the back-end, is not.
3. The front-end controls the back-end through some interface, e.g. an API or a command language or a combination thereof.
4. The front-end uses attacker-supplied input as a parameter when controlling the back-end.
5. The front-end fails to sanitize input in such a way that control would always be with the front-end and never with anyone who supplies (malicious) inputs.

For this scenario to turn into a vulnerability only one more property is required: there must be at least one input leading to an effect that an attacker would desire. Note that the component view and front-end/back-end distinction are imposed upon the target and may differ from the deverlopers' model.

3.3. Testing Strategy and Techniques

Above description yields a testing strategy. There are two components, so there is interaction that we can observe. Only the front-end has interfaces that an attacker can access, so the vulnerabilities must be exploited through these. As the front-end controls the back-end, there must be a cause-effect relationship between inputs made to the front-end and control activity between the front-end and the back-end.

Given these preconditions, injection vulnerabilities can only exist where input into the front-end controls interaction with the back-end. We call any known causality of this type a vulnerability *candidate*. Whether such a candidate really is a vulnerability depends on two more factors: the handling of particular characters or input patterns, and the potential effect that manipulating the particular inputs has on the back-end and the overall application.

Assessing the effects in the end remains a task best suited to humans, whereas the other two aspects can be supported with tools. Specifically, a tester can and should be supported in finding candidates and in testing the details of input handling for each candidate. This is what the SSES does. It helps the tester to see what is going on at the front-end interfaces, relate events from different locations to each other, derive test cases from observations and execute them.

There are obvious limitations: it is possible to construct cases of vulnerabilities that cannot be found by observing data flow between components. Detection may be inhibited e.g. by transcoding or by time lag between cause and effect. However, our assumption is that the tester is looking for bugs, not carefully hidden backdoors, and the hope is that such bugs have a tendency to be simple.

4. THE TOOL: SSES

4.1. Design Rationale

The design of our tool is rooted in practical considerations along with the fault model outlined above. Existing tools, such as network monitors, fuzzers and network interfaces, allow testers to interact with the test target and observe some aspects of its behavior. The most flexible way of automating is using a script language on top of such tools to implement specific testing strategies and test cases.

To provide more specific support to testers, several obstacles must be overcome. First, simple standard tasks should not require programming. Second, security testing – at least the kind assumed and discussed here – remains highly interactive and explorative. The tester will modify testing based on prior results and observations. Third, creating test cases and executing them is often the easy part. Interpreting observations is much more difficult, and many tools seem to fail on this part [6, 7]. Fourth, while ad-hoc programming is flexible, its results are hardly ever reusable even if faults and testing strategies are similar.

Out of these needs we designed a tool that is running in parallel to the test target, connected to it through sensors and actuators. The tester interacts with both the target and the tool at will. The tool may simply observe, filter and interpret, or actively execute test cases, particularly those suggested by our fault model. The tester may modify the working of the tool at any time during the testing, or leave some observation routine in place while interacting with the target in varying ways.

4.2. Key Concepts

The key concept of our architecture are sensors and actuators. *Sensors* are passive listeners that observe communication channels between the test target and entities in its environment. *Actuators* simulate communication partners and other sources of input in the targets environment and are responsible for test case invocation.

Sensors and actuators can be placed around a component wherever interactions with other components can be observed or input be manipulated, for instance on the network, at programming interfaces, to watch log files and so on. They might even correspond to a hardware interface.

Messages captured by a sensor are transformed into an internal representation using a stack of parsers. Messages that belong to the same interface context are collected in a session. Based on this internal representation, the tester can devise test cases and execute them. To do so, the tester connects sensors, actuators, and other tool modules into a setup that, in principle, could perform arbitrary computing tasks on the data captured by sensors, injected by actuators or provided by the tester. Common test strategies can thus be implemented, e.g.: observation; replay of unmodified or modified sessions; fuzzing on any protocol level; stress testing; recognition of patterns in messages; and event correlation. These strategies can be combined freely. For instance one may apply fuzzing to an input channel while observing the output of a component.

The tester's role is to connect sensors and actuators to the appropriate interfaces; connect them into a specific test setup; supervise operations; and interpret results. Test procedures are mapped to algorithm as a collaboration of several individual tool modules, which can be adapted easily to different test scenarios by exchanging e.g. parsers, compilers or the involved sensors and actuators.

4.3. Implementation

The approach described above has been implemented in a platform called Software Security Evaluation System (SSES) with the following features:

- Java-based central control software which is used by the tester to create the test case generation and failure detection algorithms
- Sensors and actuators, which are independent programs connected to the control software via a TCP/IP network; this makes it easy to add support for further interfaces
- Graphical user interface to design algorithms by combining certain modules in algorithm diagrams

An overview of the SSES in a possible configuration is given in figure 1.

Sensors capture data blocks at interfaces of the test target, which are the basis for test case generation or failure detection. Consequently they are the input into a test case generation or failure detection algorithm. The output of these algorithms are test case data to be injected into the target using an actuator (test case generation) or statements whether a fault has been detected or not (fault detection).

The tester uses the GUI of the control software to create and parameterize the algorithms and to supervise their execution. During the execution they can be adapted at any time.

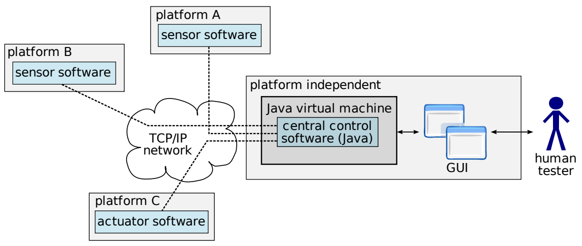

Fig. 1. SSES example configuration (without test target)

To specify a test setup the tester uses components like data input and output modules, viewer modules for visualization, parser and compiler modules to convert interface data, session management modules to organize interface messages in contexts and others. A screenshot of the user interface containing a simple algorithm as a collaboration of three modules with specific tasks is show in figure 2.

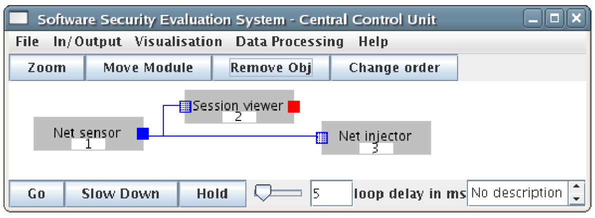

Fig. 2. User interface of the SSES central control

The SSES allows arbitrary data flows between modules to be specified by the tester. The output of any module can be fed as input into one or multiple arbitrary other modules. Loops are permitted. The only limitation is that each input must have a single source. One could, however, extend the tool with modules that mix or interleave inputs from multiple sources if this is desired. It is up to the tester to make sense of the tools available and to use them to carve out just the information needed about the behaviour of the test target.

5. TESTING EXAMPLE

This section describes two SSES test scenarios that demonstrate the flexible use of the testing framework: a test for SQL injection flaws and a test for cross site scripting vulnerabilties.

5.1. SQL Injection Test

Depending on the design and implementation of the test target, the test strategy varies. If the interfaces of the test target can be sufficiently described in their syntax and semantic, the SSES can perform the test automatically.

According to the assumptions made in 3.2 and 3.3 the test target is defined as follows:

The front-end is a web application which is connected to a back-end database system using a TCP/IP-based database query protocol. In a regular use case a human interacts with the web application using a HTTP/HTML-based user interface. According to our fault model, vulnerability candidates are indicated by correlated data in front-end input and database communications.

Considering this scenario, a conceivable grey-box test is shown in figure 3. The SSES generates HTTP requests containing malicious patterns which are sent to an actuator working as an HTTP client. The client forwards the request to the test target. The web application (test target) creates the SQL query and sends it to the database. This communication is captured by a sensor, which forwards the data stream to the failure detection algorithm. The detection algorithm stops the test when malicious patterns re-appear in the queries.

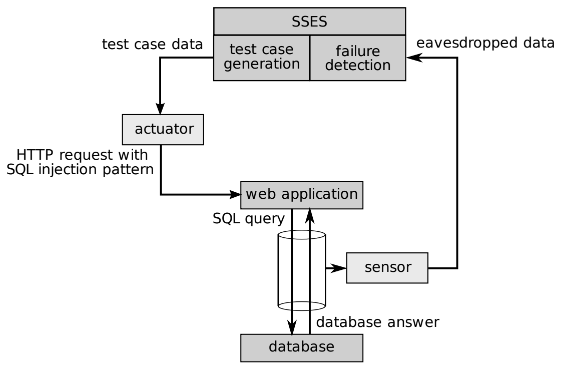

Fig. 3. SQL injection test with test case execution and failure detection performed by the SSES

In practice, machine-processable descriptions are often not available, incomplete or otherwise insufficient. In such cases the human can fill out the gaps as shown in figure 4. He can invoke the test cases using appropriate software e.g. a web browser, when the web application interface can not be controlled by the SSES in a sensible way. At this, the SSES only acts as a failure detection system which alerts the tester when a vulnerability candidate "appears" at the communication channel between web application and database.

The third case is the combination of the two others. The tester interacts with the web application but communication between both is manipulated to invoke the test cases. These manipulations can be performed randomized and with changing malicious patterns. The SSES acts here as a manipulating proxy as shown in figure 5.

This method is useful if the interface of the web application is only partially describable for the test case generation algorithm. Thus, the tester has to support the process by providing the necessary interactions with the web application.

A detailed description of SSES algorithms for discovering SQL injection vulnerabilities can be found in [15].

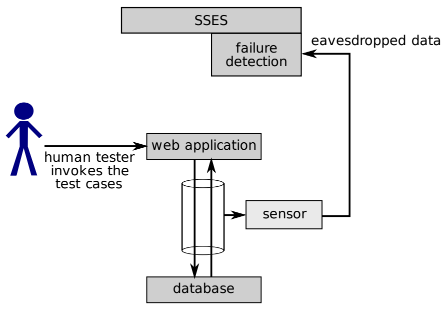

Fig. 4. SQL injection test with test case executed by the tester

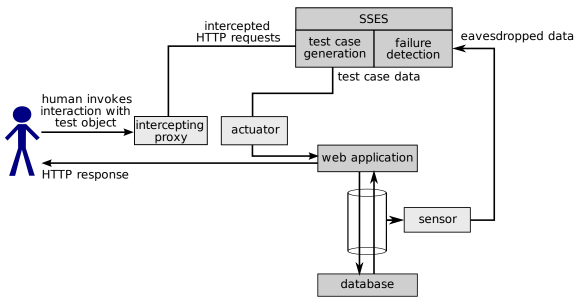

Fig. 5. SQL injection test with SSES intercepting requests

5.2. Cross Site Scripting Test

For cross site scripting flaws in scenarios similar to those described in 5.1, the back-end system is the browser of some web application user. This definition of "front-end" and "back-end" seems unorthodox but according to 3.2 the web application controls the user's web browser by submitting HTTP responses. Thus the web browser is the downstream component in the chain where the effect occurs. A detailed description of SSES algorithms for discovering XSS and other injection vulnerabilities can be found in [16].

Consequently, the sensor connected to the output communication channel in 5.1 has to move its position. Now, it has to capture the data transmissions between web application and web browser. Also changing are side conditions such as input patterns. However, the failure detection remains similar. The XSS test scenario is shown in figure 6.

6. CONCLUSION

We described the design of a security test tool developed out of need by security testers. Design principles are a) to support a human security tester rather than automating all tasks; b) to provide specific support for analysis tasks; c) few side conditions and assumptions about the test target; and d) a generic grey-box testing strategy suggested by the fault model of injection attacks. We demonstrated how to use this tool to find

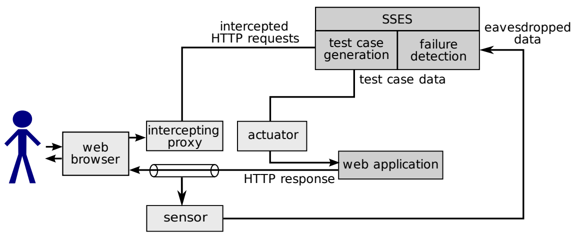

Fig. 6. XSS test with request interception by the SSES

XSS and SQL injection flaws in Web applications. The key distinguishing feature of our SSES is the sensor-actuator approach, which in particular is different from what security scanners typically do.

Still missing is an empirical evaluation of our approach. Such evaluation is made difficult by the involvement of humans in the testing process. We resist the temptation to set up a quick ad-hoc evaluation and leave it to further research to devise an appropriate evaluation method. We hope to see improved performance over HTTP-only vulnerability scanners.

Another path of further research that our tool points to is the development of a more detailed fault model for injection flaws, particularly one that is suitable for testing under real-world conditions. A third possible continuation is integrating SSES with complementary approaches such as concepts of AMNESIA [17], or with advanced visualization techniques. It seems also worth investigating what other testing tasks our tool could support, be they security-related or not.

7. REFERENCES

[1] B. Miller; D. Koski; C. P. Lee; V. Maganty; R. Murthy; A. Natarajan; J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services," Tech. Rep., 1995.

[2] A. Ghosh; M. Schmid; V. Shah, "Testing the robustness of Windows NT software," in *Proc. of the Ninth Intl. Symposium on Software Reliability Engineering (ISSRE '98)*. IEEE, November 1998, pp. 231–235.

[3] R. Kaksonen, *A Functional Method for Assessing Protocol Implementation Security*, Technical Research Center of Finland (VTT - Valtion teknillinen tutkimuskeskus), 2001.

[4] Y.-W. Huang; S.-K. Huang; T.-Po Lin; C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring," in *WWW '03: Proc. of the 12th intl. conf. on World Wide Web*, New York, NY, USA, 2003, pp. 148–159, ACM Press.

[5] W. Du; A. P. Mathur, "Testing for software vulnerability using environment perturbation," in *DSN '00: Proc. of the 2000 Intl. Conf. on Dependable Systems and Networks*, Washington, DC, USA, 2000, pp. 603–612, IEEE.

[6] S. Peine, H.; Mandel, "Sicherheitsprüfwerkzeuge für Web-Anwendungen," Fraunhofer IESE, Forschungsbericht.

[7] A. Wiegenstein; F. Weidemann; S. Schinzel; M. Schumacher, "Web application vulnerability scanners - a benchmark," `http://www.virtualforge.de/web_scanner_benchmark.php`.

[8] Y.-W. Huang; C.-H. Tsai; D. T. Lee; Sy-Y. Kuo, "Non-Detrimental Web Application Security Scanning," in *Proc. of the 15th Intl. Symposium on Software Reliability Engineering (ISSRE'04)*. 2004, pp. 219–230, IEEE.

[9] W. G.J. Halfond; Alessandro Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," in *Proc. of the Third Intl. ICSE Workshop on Dynamic Analysis (WODA 2005)*, St. Louis, MO, USA, may 2005, pp. 22–28.

[10] T. Pietraszek; C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation," in *Recent Advances in Intrusion Detection 2005 (RAID)*, 2005.

[11] Y.-W. Huang; C.-H. Tsai; T.-Po Lin; S.-K. Huang; D. T. Lee; Sy-Yen Kuo, "A testing framework for web application security assessment," *Comput. Networks*, vol. 48, no. 5, pp. 739–761, 2005.

[12] W. D. Yu; D. Aravind; P. Supthaweesuk, "Software vulnerability analysis for web services software systems," *iscc*, vol. 0, pp. 740–748, 2006.

[13] W. Xu; S. Bhatkar; R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a range of attacks," .

[14] W. G. J. Halfond; A. Orso; P. Manolios, "Using positive tainting and syntax-aware evaluation to counter sql injection attacks," in *SIGSOFT '06/FSE-14: Proc. of the 14th ACM SIGSOFT intl. symposium on Foundations of software engineering*, New York, NY, USA, 2006, pp. 175–185, ACM.

[15] C. Bornmann, "Prototypical extension of the SSES security testing framework," May 2007.

[16] A. Poller, "Approaches for automated software security evaluations," November 2006.

[17] W. G. J. Halfond; A. Orso, "Amnesia: Analysis and monitoring for neutralizing sql-injection attacks," in *ASE '05: Proc. of the 20th IEEE/ACM intl. Conf. on Automated software engineering*, New York, NY, USA, 2005, pp. 174–183, ACM.