Case study: automated security testing on the Trusted Computing platform*

Gergely Tóth
BME, Department of Measurement and Information Systems, SEARCH Laboratory
Magyar tudósok krt. 2.
H-1117 Budapest, Hungary
+36-1-205-3098
gergely.toth@mit.bme.hu

Gábor Kőszegi
BME, Department of Measurement and Information Systems, SEARCH Laboratory
Magyar tudósok krt. 2.
H-1117 Budapest, Hungary
+36-1-205-3098
gabor.koszegi@mit.bme.hu

Zoltán Hornák
BME, Department of Measurement and Information Systems, SEARCH Laboratory
Magyar tudósok krt. 2.
H-1117 Budapest, Hungary
+36-1-205-3098
zoltan.hornak@mit.bme.hu

ABSTRACT
This article is a case study summarizing the experiences gained during the execution of automated security testing with the Flinder framework within the EU FP6 OpenTC project. Usually, results of industrial security tests get little publicity due to the strict non-disclosure policies. The work presented in this paper is an exception, SEARCH Laboratory executed industrial-scale automated testing and the results could be published according to the open philosophy of the project.

The size of the task is best demonstrated by the following figures: more than 130 thousand tests have been carried out on the 250-thousand-line TSS (TCG Software Stack) implementation needing more than two weeks of continuous operation on four TPM-enabled machines revealing several security-relevant programming bugs and even some remotely exploitable vulnerabilities.

Categories and Subject Descriptors
D.2.5 [**Debugging and Testing**]: Testing tools.

General Terms
Reliability, Security and Verification.

Keywords
OpenTC, Trusted Computing, automated security testing, Flinder.

*Proceedings of the ACM SIGOPS European Workshop on System Security (EUROSEC), Glasgow, Scotland, March 31, 2008.

EUROSEC' 08, Glasgow Scotland.

1. INTRODUCTION
In most cases security vulnerabilities found in software applications are caused by minor programming failures, which can occur anywhere in the code – therefore filtering them out manually in large systems can be fairly tiring and time consuming. Fortunately in the typical, most dangerous cases (e.g. buffer overflows, integer overflows, printf format string bugs [4] [5] [6] [7]) this task is not hopeless at all. Automated security testing methods can be used for the most common vulnerabilities, with which most of the bugs can be detected efficiently.

The reason why developers paid less attention to these common failures in the past lies in the fact that only a fraction of the programmers' errors becomes security vulnerability and even fewer can be then exploited by an attacker.

Nevertheless, this "negligibly" small amount of common programming bugs is responsible for the majority of known malware: viruses, worms exist thanks to vulnerabilities based on these little coding errors; networks organized of hacked computers – so-called "botnets" – carry out the well known spam and phishing attacks based on social engineering.

Thus the popularity of testing-based error detection could rise in the future, as the complete formal verification of large, complex software is practically infeasible due to its time consumption and its high costs. The Flinder framework [1] used in this project employs error detection via dynamic testing; it is an efficient and fast tool for finding typical programming errors.

This article will summarize the results, experiences and final conclusions of an automated security testing task in which SEARCH Laboratory tested the Linux-based TSS (TCG Software Stack) implementation within the Open Trusted Computing (OpenTC) project funded by the EU 6th Framework Programme [3].

In literature it is relatively hard to find figures summarizing results of industrial-scale security testing – mostly due to the strict non-disclosure policies adopted by the large software developers. This case study is an exception, according to the open philosophy of the OpenTC project, detailed results of the automated security testing of the TSS implementation will be described below.

2. TEST SCENARIO
SEARCH Laboratory carried out automated security testing on the TSS (TCG Software Stack) implementation of the OpenTC project, which is based on the standard of the Trusted Computing Group (TCG) [2].

The TSS is a multi-layer stack providing access to the TPM (Trusted Platform Module). Applications run by the operating system use the different layers of the TSS to access Trusted Computing functionality.

Figure 1 shows the layers of the TCG platform: it can be seen that different layers need different levels of access to resources and different permissions.

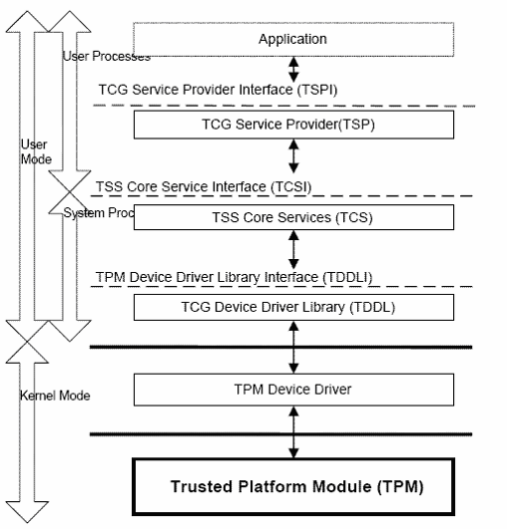

Figure 1. TCG Software Stack

From the figure above the main structure of the TSS can be observed:

- The TSP layer is implemented in a dynamic library, which is linked to the user application. This layer transforms the API calls into SOAP (Simple Object Access Protocol) messages, which are sent via TCP/IP to lower layers of the TSS.
- The TCS and TDDL layers are implemented in a separate daemon process. This process accepts incoming SOAP requests, processes them and responds in SOAP responses.

The main target of security testing carried out by SEARCH Laboratory was the Core Services (TCS) layer, which acts as an interface between user applications and system processes, device drivers; accordingly this layer needs elevated privileges to the host computer OS.

Moreover, this layer has a relatively complex functionality and is accessible by 3rd parties over SOAP, which runs over TCP/IP.

Considering these features it can be seen that the TCS layer has a wide attack surface, so finding common programming errors in it is especially critical from the view point of the system's security. These are the reasons why systematic and automated security testing was carried out on the TSS implementation.

Additionally, the TSP layer linked to the user application was also tested, since it can be directly accessed by rogue applications.

3. EVALUATION APPROACH

For the automated security evaluation of IT systems, three typical approaches exist:

- **Testing**, during which test vectors are created and dispatched to the Target of Evaluation (ToE). Assessment of correct/incorrect operation is done based on the observations.
- **Static source code analysis**, which tries to find weaknesses based on analyzing the source code (e.g. by finding patterns of typical mistakes).
- **Formal reasoning**, which tries to automatically deduce mathematical proofs of properties of the system (e.g. value of a certain item will always be between 1 and 10 in any state of the application).

In the OpenTC project for the verification and validation of the software components all three types of evaluation are used. Static source code analysis is used on a grand scale and formal reasoning is used for security-critical internal modules (such as kernel functions).

The automated testing approach was chosen for the evaluation of the TSS implementation because it is similar to the approaches of the attackers and can model efficiently the possibilities of external adversaries. SEARCH Laboratory used the testing tool Flinder [1], whose aim is to discover typical security-related programming bugs (e.g. buffer and integer overflows, printf format string bugs [4] [5] [6] [7]) in the Target of Evaluation. By detecting such bugs, Flinder increases the quality and security level of the tested software.

Although there are similar testing tools available on the market, Flinder offered several advantages, such as support for Linux, the possibility of developing custom test suites for testing or the ability to execute both black-box and white-box security testing.

For the completion of its task Flinder analyzes the dynamic behavior of the Target of Evaluation (ToE) during its execution. Flinder is capable of executing both white-box and black-box testing: white-box testing is done by function level fault injection via modified source code; as opposed to this, the black-box method uses only the binary executable of the program and observes the given responses for manipulated inputs and checks whether the ToE took some unexpected action (e.g. halting or freezing).

Flinder is capable of testing both applications and network protocol implementations. It contains customizable general-purpose modules, which can be set up according to requirements. To facilitate its applicability, Flinder supports a wide range of cryptographic and encoding algorithms for compression, encryption and digital signatures. Handling of input/output data is easily extensible by Python scripts.

For the testing Flinder relies on a method called fuzzing [8], which means the modification of binary data (usually input) handled by the Target of Evaluation. Several different approaches can be used for fuzzing, Flinder uses a combination of the following:

- random, i.e. random modification of binary data,
- pre-defined, i.e. modification of data with a fixed set of test vectors,
- algorithmic, i.e. algorithmic modification of the original data (e.g. truncating or expanding it), and
- reactive, i.e. when the reactions of the ToE are taken into consideration (e.g. successive approximation of the maximal size of a buffer).

The general sequence of one test case in the black-box scenario is the following (see also Figure 2):

- Every test case needs a valid message as the input for the manipulation. These are usually generated by a specific program (called Input Generator).
- The Capturer module intercepts these valid messages (e.g. from the network).
- The Parser module is supposed to translate valid messages into an internal Message Structure Description Language (MSDL) according to a given format description (MFDL).

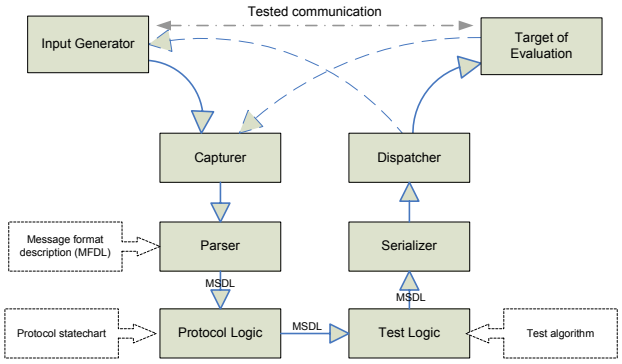

Figure 2. Flinder architecture for black-box testing

- The Protocol Logic updates the protocol statechart based on the type of the parsed message. Additionally different actions can be taken according to this state information (e.g. timeout setup).
- Afterward, the Test Logic can execute modifications on the message (e.g. change integer values, enlarge buffers and change their contents) in order to provoke symptoms of the typical errors during the execution of the ToE.
- Then the Serializer reconstructs the message from the internal representation, which now includes the modifications made by the Test Logic. Note, that the Serializer can reconstruct internal rules of the modified message, e.g. re-calculate size fields or re-sign a modified packet.
- Finally the Dispatcher forwards the recreated message to the Target of Evaluation (ToE) and then observes the reactions of ToE. It is tested, whether it successfully finishes, whether it sends an error message or acts inappropriately in any way (e.g. terminates, exhausts system resources or simply stops responding).

Beyond testing an application as a black-box, Flinder also supports white-box mode testing. In this source code based testing some parts of Flinder need to be compiled together with the ToE. Aim of inserting these code snippets is to extract the data structures to-be-modified (e.g. function parameters or global variables) from the tested software and afterwards inject the manipulated data back to the program's internal state. Communication with Flinder goes through inter process communication. Thus, in case of white-box testing the following is carried out:

- the ToE is started and it sends the variables to be tested to Flinder;
- Flinder modifies these data similarly to black-box testing and sends them back;
- the ToE receives the modified data and injects them back into its internal state and continues operation;
- finally, Flinder observes the operation of the ToE.

4. EXECUTION OF TESTING

SEARCH Laboratory was looking for bugs at two levels of the TSS implementation: the first approach was the black-box man-in-the-middle testing of the SOAP-based network interface of the TCS layer, which actually meant manipulation of Remote Procedure Calling (RPC) parameters in captured XML messages originating from the TSP layer.

The second approach was the white-box testing of the platform's Trusted Service Provider Interface (TSPI), which is actually is a function library. In this case we used functional test applications of the implementation as Input Generators enhanced with Flinder's fault injection technique. This approach also allowed us to observe the behavior of the TSP layer (the one above the Core Services). Figure 3 shows the locations of the two approaches.

In order to carry out the tests, SEARCH Laboratory performed the following tasks:

- generating the format descriptors (MFDLs) for the different API calls,
- creating Input Generators, i.e. applications making use of the different API calls,
- creating an Actuator module, detecting problems (such as time-outs, program crashes or memory exhaustions).

Using the Flinder framework together with the above customizations, SEARCH Laboratory executed the tests summarized in the next section. The customization took approximately 1 month, the actual test execution around 2 weeks continuous operation on four TPM-enabled personal computers (HP DC7700p).

5. TEST RESULTS

During the initial test execution 135 237 test cases were evaluated (38 106 in black-box and 97 131 in white-box mode). Amongst this huge amount there were only 403, which caused an error in the service.

Table 1. Test results

Test type	**Number of test cases**		
	Total	**Passed**	**Failed**
White-box testing of the TSP layer	97 131	96 977	154 (0.2%)
Black-box testing of the TCS layer	38 106	37 859	249 (0.7%)
Total	**135 237**	**134 834**	**403 (0.3%)**

The number of failed and executed test cases had a difference of 3 orders of magnitude, meaning that less than only 0.3 percent of the test cases failed. Taking into account that we tested more than hundred thousand test cases, this is a very important result, because it makes clear that these errors could not have been found

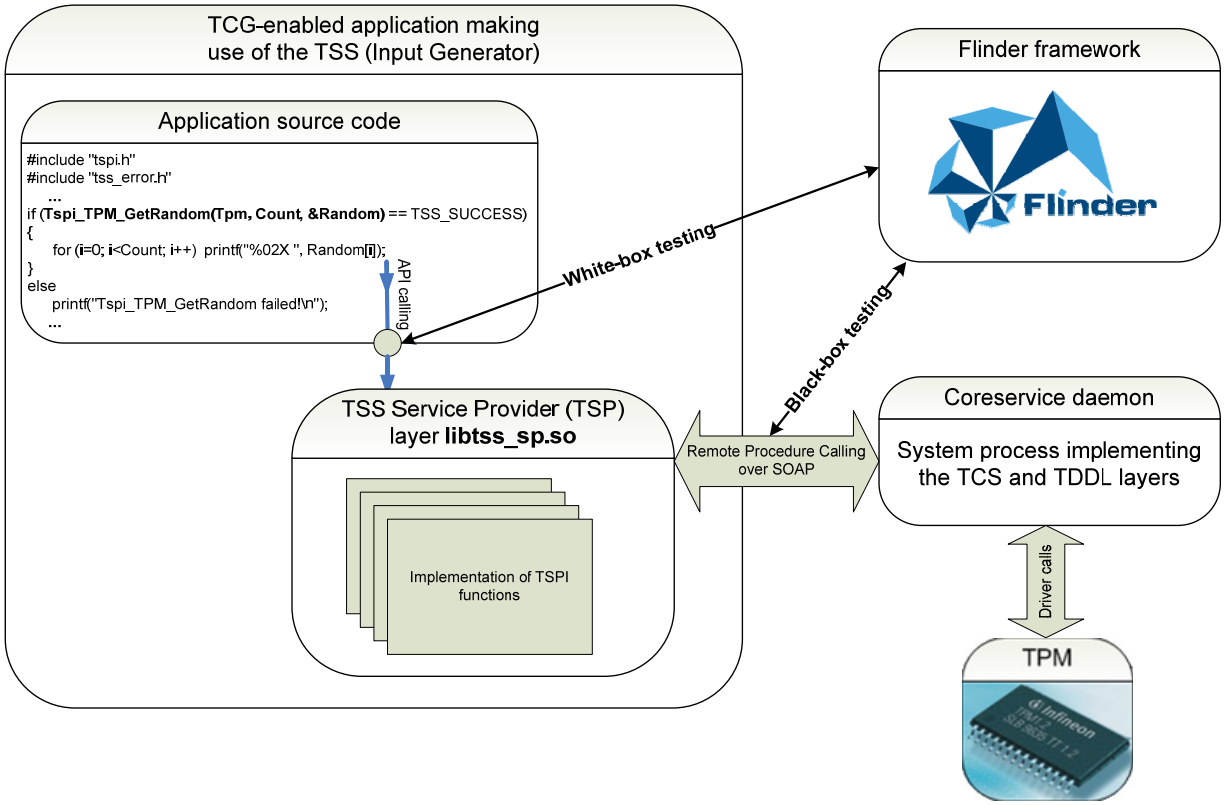

Figure 3. Locations of the two approaches in the testing environment

with pure manual testing, which would have been equal to "looking for a needle in a haystack".

We have to note here that due to the fuzzing-type testing, Flinder detected the same programming bug with several different test vectors, thus the number of failed test cases is significantly larger that the number of weaknesses identified.

On the other hand from the 65 functions and 36 SOAP messages tested there were 3 functions (4.6%) and 4 messages (11%), which could have been used in an attack against the platform according to our analysis. This showed that common security-relevant programming errors cannot be excluded with absolute confidence even during the development of such a security-sensitive application.

Table 2. Weaknesses by attack point

Attack point type	Tested		
	Total	No bug found	Potential bug found
TSP API function	65	62	3 (4.6%)
TCS SOAP message	36	32	4 (11.1%)

Such results confirm the importance of automated security testing: the software developer could not have otherwise eliminated these errors from the final software efficiently – and these errors could have been used for different kinds of attacks from 'simple' Denial of Service (DoS) to arbitrary remote code execution in the context of the administrator.

Compared to the above numbers it has to be mentioned that typical numbers with Flinder in industrial projects range between 10 and 40 per cent failed test cases. The significantly smaller ratio of failed test cases was due to the usage of extensive input validation of all API parameters.

A further important finding of the testing was that we have examined, whether commercial automated static analysis tools would find the bugs we have found with automated testing. We concluded that some types of bugs detected using testing were not found using static source code analysis, such as the example below.

Code snipped containing an exploitable flaw:

```
memcpy(fixedSizeTarget.ptr,
   externalInput.ptr, externalInput.size);
```

Instead of:

```
memcpy(fixedSizeTarget.ptr,
   externalInput.ptr, fixedSizeTarget.size);
```

In the above code an attempt was made to copy an external string (coming from a SOAP message) to an internal fixed-size buffer. Unfortunately, the number of bytes to copy was determined not by the size of the target buffer, but rather by the length of the external input – which could lead to an exploitable buffer overflow in case the external input was longer than the internal buffer.

6. REGRESSION TESTING

Based on the findings of the initial tests, thorough bugfixing was carried out on the TSS implementation addressing all the weaknesses discovered by the first test run.

In order to verify the correctness of the bugfixes, all failed test cases have been re-executed on the updated software version.

In the end SEARCH Laboratory could verify that no identified weaknesses remained in the final TSS implementation and thus this important TC component reached and significant level. Passing Flinder tests means that typical security-relevant programming bugs, accountable for the majority of exploitable weaknesses, have been systematically eliminated. Naturally, no exhaustive testing was carried out, however the systematic testing on all external interfaces greatly reduces the chances of remaining weaknesses.

7. CONCLUSION

An open EU FP6 Research & Development project as OpenTC is a good opportunity to introduce the achievements of security testing to the public, which usually fall under strict non-disclosure regulations in case of industrial projects.

From the testing the following three main lessons could be learnt:

- Up to this day **developers commit typical security-relevant bugs** even in security-critical applications due to human mistakes, even though these errors are well-known and analyzed for more than 15 years.
- Complete manual testing is practically infeasible for high-complexity software like the just shown TSS implementation, therefore **automated techniques are absolutely necessary**, which are capable of carrying out such huge tasks efficiently.
- On the other hand **automated methods provide useful tools** for the protection against these typical security-relevant errors. They can systematically evaluate the functionality of the target systems, typical errors can be eliminated and therefore the overall security level and the quality of applications can be improved.
- **Fuzzing-based testing detects bugs, which widely-available static analysis tools do not find**, and therefore complements those tools effectively. The best approach is to use the combination of formal, static analysis and testing methods.

8. ACKNOWLEDGEMENT

This work has been partially funded by the EC as part of the OpenTC project [3] (ref. nr. 027635). It is the work of the authors alone and may not reflect the opinion of the whole project.

The detailed technical report based on which this paper was created can be found at [9].

The authors would like to thank the OpenTC team for valuable comments for the article, especially Hans Brandl, Armand Puccetti, Arnd Weber and Dirk Weber.

The OpenTC TSS was developed by Infineon. Please contact Hans Brandl for more information on the TSS: hans.brandl (at) infineon.com

9. REFERENCES

[1] G. Tóth and Z. Hornák. Flinder test methodology overview. 2006. http://www.flinder.hu/library/index.html

[2] Trusted Computing Group. http://www.trustedcomputinggroup.org

[3] OpenTC – Open Trusted Computing, EU FP6 IP. http://www.opentc.net

[4] Aleph1. Buffer overflow. *Phrack magazine*, Volume Seven, Issue 49, File 14, 1996. http://www.phrack.org./archives/49/P49-14

[5] M. Conover. Heap overflow, w00w00 Security Team, 1999. http://www.w00w00.org/files/articles/heaptut.txt

[6] Blexim. Basic integer overflows. *Phrack magazine*, Volume 0x0b, Issue 0x3c, Phile #0x0a, 2002. http://www.phrack.org./archives/60/p60-0x0a.txt

[7] scut , team teso. Exploiting format string vulnerabilities. 2001. http://julianor.tripod.com/bc/formatstring-1.2.pdf

[8] I. Sprundel. Fuzzing – Breaking software in an automated fashion, 2005

[9] A. Puccetti (ed). D07.2 V&V Report #2: Methodology definition, analysis results and certification. 2007. http://www.opentc.net/deliverables2007/Open_TC_D07.2_V_and_V_report_2.pdf

[10] P. Emanuelsson and U. Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Linköping University Electronic Press*, 2008.