SOFTWARE SECURITY ASSESSMENT THROUGH SPECIFICATION MUTATIONS AND FAULT INJECTION

Rauli Kaksonen
VTT Electronics
P.O.Box 1100, FIN-90571 Oulu, Finland
rauli.kaksonen@vtt.fi

Marko Laakso, Ari Takanen
University of Oulu, Department of Electrical Engineering, Computer Engineering Laboratory
Linnanmaa Box 4500, FIN-90401 Oulu, Finland
{fenris, art}@ee.oulu.fi

Abstract Numerous information security vulnerabilities exist in contemporary software products. The purpose of this paper is to present a practical approach for software security assessment based on fault injection. The approach has been introduced and applied in a real world case, Wireless Application Protocol gateways. The approach has been effective in systematically uncovering robustness problems in the components tested. The main impact is expected from early elimination of trivial vulnerabilities and elevated awareness in robustness problems and their security implications.

Keywords: Security assessment, protocol specification, fault injection, syntax testing, Wireless Application Protocol

Introduction

Numerous vulnerabilities exist in contemporary software products. Vulnerabilities can manifest themselves in crashes and hangs triggered by exceptional use of a product. Errors of this kind often expose the products to attacks against information security. Fault injection is a testing method for evaluating software robustness. The correctness of software responses against the specification is not evaluated, but rather the ability of the software to cope with the injected faults is under scrutiny. Fault injection with unexpected or erroneous input is suitable for security assessment, it simulates attack through external interfaces, e.g. network connections. According to our tests, there appear to be a large

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35413-2_36
R. Steinmetz et al. (eds.), *Communications and Multimedia Security Issues of the New Century*

number of products with errors that can be exposed by this method. Applying fault injection early in the life-cycle of products would lessen the number of errors in shipped products. More importantly the use of such tools could promote a trend of avoiding the introduction of errors in a pro-active fashion. The purpose of this paper is to present a practical approach for software security assessment based on the fault injection. First, background information is given about implementation testing, security assessment and fault injection. Second, a security testing approach is presented with help of an artificial testing example. Third, a real-world case in the form of a test suite against WAP gateway implementations is studied in detail. Finally, discussion follows and conclusions are drawn.

1. TESTING AND FAULT INJECTION

The international standard ISO/IEC JTC1: IS 9646, defines a methodology for *conformance testing* of protocol implementation (*implementation under test*, IUT). [2, p. 21]. The standard purposefully excludes activities such as security assessment, but provides useful terminology for this paper. Tests are executed by a *tester* that controls and observes the IUT. A *test case* demonstrates certain behavior or realization of some capability. Test cases are grouped into *test groups* which can be further grouped into higher level groups. The highest structuring level is *test suite*. The result of a test case is called a *verdict*. The verdict is either *pass*, *fail* or *inconclusive*. In passed test cases the IUT fulfills the test purpose and displays only valid behavior. In a failed test case the IUT violates a conformance requirement. Inconclusive means the inability to assign either a pass or fail verdict.

Software *fault injection* is a non-traditional testing method for evaluating the behavior of software under anomalous circumstances. [10, p. 34]. Fault injection can be applied to software input by feeding an IUT with data containing exceptional structures. The IUT is monitored for any deviation from acceptable behavior or for inconsistent internal state. This gives us information about the quality of exception handling code or *robustness* of the IUT [4, 10, p. 275]. Exceptional structures can be legal or illegal according the input specification. For example, the harmful and common *buffer-overflow* vulnerability is caused by the failure of the IUT to handle larger than expected chunks of input data [8]. Often a buffer-overflow anomaly can be used by an attacker to execute arbitrary code on a failed system.

Fault injection of input can be considered as a set of mutations applied to the syntax of software input. This method is also known as *syntax testing* [3, p. 284].

2. PROTOCOL SECURITY ASSESSMENT

Protocol implementations are logical targets for security assessment since messages are often transmitted over the Internet or other insecure networks. Insecure transmission media exposes the messages to malicious modification. Cryptographic protections are not effective against attacker who can negotiate a legal session before injecting malicious messages.

In fault injection mutations can be applied to the syntax of individual protocol messages as well as the order and type of messages exchanged. For ease of discussion, we define protocol *element* to a part of a message, a single message or a sequence of messages. Mutations can add, move, delete and modify elements. The number of different mutations is unlimited. For creation of effective test cases for fault injection the semantics of individual messages and message exchanges should be preserved. This is in contrast to *random testing*, in which the input data is randomly generated [6]. The hypothesis is that carefully constructed input is more likely to find errors in the IUT [5].

During fault injection we identify three kinds of elements: *base* elements, *derived* elements and *exception* elements. A base element is an independent element of protocol while a derived element is computed from other elements. Examples of derived elements are length fields and checksums. Derived elements must be assigned values which preserve protocol semantics, i.e. they follow *semantic rules*. An exception element purposefully contains a fault or is otherwise unexpected or surprising. There may be one or more exception elements in a single test case. The design of test cases requires a protocol specification and semantic rules, after that it is a matter of combining suitable exception elements and base elements together.

2.1. TEST DESIGN

ISO 9646 identifies a protocol implementation capable of generating test cases as an *enhanced implementation* [2, p. 79]. We chose not to fully adopt the ISO 9646 approach of using a combination of SDL, ASN.1 and TTCN. We aimed to minimize the required training and to provide a simple integration with specifications of classic Internet protocols, not typically presented in ASN.1 form.

Instead we use *mini-simulations*. A mini-simulation is an executable model of partial functionality of some protocol entity. Our mini-simulation environment uses *Backus-Naur Form* (BNF), with extensions, as the protocol specification notation [1, p. 25]. The extended BNF with semantic rules is capable of modeling the behavior needed for test case execution [5]. A mini-simulation does not try to cover full functionality, e.g. testing of an HTTP server might use the mini-simulation of a client only capable of sending an HTTP request. The protocol specification is mutated to contain the required exceptional elements.

We are not interested about the correctness of the IUT responses according to protocol specifications. This reduces the needed effort significantly compared to traditional testing, because the expected responses do not need to be specified and verified. Our mini-simulation environment uses Java as the primary implementation language [9]. Semantic rules are implemented as Java classes.

The scientific community has proposed experimental tools drawing from the same principles as we are [4, 5, 6, 10].

2.2. "PHONE BOOK" PROTOCOL TEST DESIGN

Here, a test suite design for an imaginary *phone book protocol* is presented. A real-world example, the *WAP-WSP-Request test suite*, is presented later on. The phone book protocol enables a client process to request phone numbers from a server over a network. The purpose of the testing is to assess the ability of a server to withstand attack through request messages with malicious modifications. The BNF specification of the protocol follows.

```
<exchange> = <request> <response>
<request> = 0x01 <name>
<response> = 0x02 <number>
<name> = { <CHARACTER> } 0x00
<number> = { <DIGIT> } 0x00
```

Protocol exchange is made up of a request message and a response message. The request message is sent from a client to a server, and it consists of a prefix byte `0x01` followed by a null-terminated string. The string gives the name of a person whose phone number was requested. The response message is sent from a server to a client, and it consists of a prefix byte`0x02` followed by a null-terminated digit string. The digit string contains the requested phone number, empty string represents an unknown number.

First, we define a fixed name "John Smith" for all requests. The location to put the name into is specified using a *path*, which in this case is `<name>.0`. This path points to the first element in the right-hand side of the`<name>` definition, i.e., {`<CHARACTER>`}. The following **replace** operation performs the replacement.

```
replace <name>.0 "John Smith"
```

After a careful analysis, we decide to add two kinds of exception elements: invalid request prefix bytes and exceptionally long names. Boundary values `0x00` and `0xff` serve as exceptional prefix bytes. For name string we try artificially long names with 256, 1024 and 65536 characters. Exception elements are added by two **insert** operations.

```
insert <request>.0 (0x00 |0xff)
insert <name>.0 (256 x "a" |1024 x "a" |65536 x "a")
```

Vertical bar "|" separates alternative elements. Construct *n* x *e* repeats the *e* element *n* times, e.g. `5 x "a"` is equal to `"aaaaa"`. The resulting specification with exception elements is as follows.

```
<exchange> = <request> <response>
<request> = (0x01 |0x00 |0xff) <name>
<response> = 0x02 <number>
<name> = ("John Smith" |256 x "a" |1024 x "a" |65536 x "a") 0x00
<number> = { <DIGIT> } 0x00
```

The default value for each alternative is the leftmost choice, i.e. `0x01` for the request prefix and `"John Smith"` for the name. We decide to have test cases for typical request, requests with invalid prefix bytes, requests with long name strings and requests with both kinds of exceptions. These test cases are selected by the **combine** operation. The operation takes zero or many paths as parameters, each path specifies one or more elements. Multiple elements can be specified by separating their segments using vertical bar. When multiple paths are used, elements specified by the paths are permutated to get test cases, e.g. selecting 2 elements with first path and 3 elements with second results total of 6 test cases. The test cases for the example are selected by four **combine** operations.

```
combine                                          ;# case 0
combine <request>.0.(1|2)                        ;# cases 1-2
combine <name>.0.(1|2|3)                         ;# cases 3-5
combine <request>.0.(1|2) <name>.0.(1|2|3)       ;# cases 6-11
```

The first operation has no path, so all defaults are chosen. The second operation selects `0x00` and `0xff` as prefix bytes, and the name will take the default value. The third operation specifies each long name string, it generates three test cases all having the default prefix byte. The final operation generates all six combinations of invalid prefix bytes and long name strings. The test suite has a total of 12 test cases, the request messages created by each test cases are shown in Table 1. Further tests could be designed, e.g. by omitting the trailing zero byte supposed to terminate the name string.

Table 1 Generated messages for "Phone Book" protocol testing.

Test case	*Request message*	*Test case*	*Request message*
0	`0x01 "John Smith" 0x00`	6	`0x00 (256 x "a") 0x00`
1	`0x00 "John Smith" 0x00`	7	`0x00 (1024 x "a") 0x00`
2	`0xff "John Smith" 0x00`	8	`0x00 (65536 x "a") 0x00`
3	`0x01 (256 x "a") 0x00`	9	`0xff (256 x "a") 0x00`
4	`0x01 (1024 x "a") 0x00`	10	`0xff (1024 x "a") 0x00`
5	`0x01 (65536 x "a") 0x00`	11	`0xff (65536 x "a") 0x00`

2.3. TEST CASE GENERATION

The combining of exception elements and base elements is done manually by a designer. Subsequently, the test cases are generated automatically in three phases.

1 Resolve those alternatives where the choice is determined by manually selected elements.

2 For the input elements into the IUT, resolve remaining alternatives by making the default choice.

3 Compute derived elements using semantic rules.

The resulting number of test cases tends to be larger than in traditional testing, since there are many exception categories to test and test design is not burdened by having to specify the expected output. Several test cases have the same base elements, the only variation being different exception elements.

Assignment of a verdict in a traditional test case is based on the equivalence of expected and actual behavior. In the security assessment the verdict is based on acceptability of the actual behavior. This can be implemented by a *postcondition checker*, which detects undesirable behavior [10, p. 276]. We can simply require that the IUT must stay running (not to crash) or the IUT must respond to further input (not to hang). Postcondition checkers can also monitor for dangerous file manipulation or other more complex failure modes.

3. WAP-WSP-REQUEST TEST SUITE

WAP is a family of protocols for delivering advanced data services and Internet content to wireless terminals [11] [12]. A WAP gateway mediates traffic between terminals and actual content providers, i.e. WAP and HTTP servers. Relevant components of the WAP infrastructure are shown in Figure 1. Security of WAP gateways is essential since even encrypted WAP traffic will be exposed as plain text inside a gateway. The purpose of the *WAP-WSP-Request test suite* is to assess the ability of a WAP gateway to handle maliciously formatted *Wireless Session Protocol* (WSP) messages [13]. Using a workstation or a laptop with a modem and a phone an intruder can sent a malicious WSP message to a gateway. Since the focus is on the robustness of WSP message handling, the overall security of a WAP system is not assessed. However, a single vulnerable point is sufficient to totally compromise the security of a system.

3.1. TEST DESIGN

The following explains the creation of the WAP-WSP-Request test suite. The specification of the WSP protocol was available from the WAP-Forum

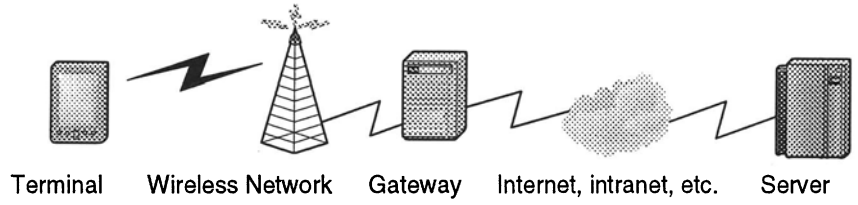

Figure 1 Components of the WAP infrastructure relevant to the WAP-WSP-Request test suite.

[11]. The specification was manually rewritten into a BNF specification. For brevity, only a subset of the whole specification is shown below.

```
<wsp-pdu> ::= <tid> ( <get> | <post> )
<tid> ::= <BYTE>
<get> ::= 0x40 <get-url-len> <get-url> <headers>
<get-url-len> ::= <UINTVAR>
<get-url> ::= <URL>
...
<URL> ::= <URL-PROTOCOL> <URL-DELIMITER> <URL-HOST> "/" <URL-FILE>
<URL-PROTOCOL> ::= { <CHARACTER> }
...
```

A WSP *get* message is sent by the terminal to the gateway to fetch a page specified by an *universal resource locator* (URL). The message contains one byte `<tid>` followed by a `<get>` element. The separation between different WAP sessions is made by different values of `<tid>`. The `<get>` element is a sequence of the byte `0x40`, the length element `<get-url-len>`, an URL string `<get-url>` and headers. The URL is made of a protocol identifier `<URL-PROTOCOL>`, a delimiter `<URL-DELIMITER>`, a host name `<URL-HOST>` and so on.

A mini-simulation based on the relevant parts of the WSP specification was created. Two semantic rules were needed, one for the creation of length elements (e.g. `<get-url-len>`), and another to create the increasing values for `<tid>`. The incrementation of `<tid>` by one after each test case was required in order to guarantee the separate handling of each test case by the IUT. Base element values were set to correspond to the fetching of "`index.wml`" from the local host (ip-address `127.0.0.1`) by using HTTP. Now, the mini-simulation produces the following message

```
0x01 0x40 0x1a  "http://127.0.0.1/index.wml"
```

Four categories of exception elements were used in the message mutation. The exception categories, the number of different mutations per category and some sample elements for each category are shown in Table 2. String exceptions are character arrays of "a", with lengths from 2 to 64000 characters. Length exceptions are boundary values for length fields. Delimiter exceptions are special mutations of the delimiter characters ":", ";" and "/" in the URLs. The

Table 2 The WAP-WSP-Request test suite exception categories, the number of different mutations in a category and sample elements for each category.

Exception category	#	*Category sample*
String	13	`<STRING-E> ::= 2 x "a" |... |64000 x "a"`
Length	10	`<LENGTH-E> ::= 0x00 |0x01 |0xff	...`
Delimiter	19	`<DELIMITER-E> ::= "::/" |...`
End of string	1	`<EOS-E> ::= "a"`

end of string exception is the replacement of the null byte with the character "a" at end of a string. A total of 4236 test cases were created by combining the presented categories. For example, the<URL-PROTOCOL> element identifies the used protocol. The default value for the<URL-PROTOCOL> is set to be "http" for *hypertext transfer protocol.* The value is mutated by string exceptions to test the capability to handle very long protocol strings. Another example is the length element <get-url-len>, which is mutated by boundary values. The following operations add the exception elements as alternatives for the "http" string and the length in <get-url-len>.

```
insert <URL-PROTOCOL> <STRING-E>
insert <get-url-len> <LENGTH-E>
```

After this the test cases are created by using **combine** operations. In the operations the asterisk ("*") matches zero to many continuous path segments and the question mark ("?") any single segment. The first combine operation tries all string exceptions within the <URL-PROTOCOL> element. The second combines the same exceptions with length exceptions within the<get> element.

```
combine <URL-PROTOCOL>.*.<STRING-E>.?
combine <URL-PROTOCOL>.*.<STRING-E>.? <get>.*.<LENGTH-E>.?
```

As seen from Table 2 there are 13 different exception elements in the string category, which yields 13 test cases for the <URL-PROTOCOL> element. Messages generated for the first four test cases, with 2, 4, 8 and 16 character strings are shown below.

```
0x02 0x40 0x18 "aa://127.0.0.1/index.wml"
0x03 0x40 0x1a "aaaa://127.0.0.1/index.wml"
0x04 0x40 0x1e "aaaaaaaa://127.0.0.1/index.wml"
0x05 0x40 0x26 "aaaaaaaaaaaaaaaa://127.0.0.1/index.wml"
```

The 13 different string exceptions and the 10 different length exceptions generate a total of 130 test cases for the combined mutations of<URL-PROTOCOL> and <get> elements. The test cases created by a combine form a test group. The design of all test cases in the WAP-WSP-Request test suite required a total of 36 such combines, i.e. there are 36 test groups.

3.2. TEST EXECUTION AND RESULTS

The test suite was executed against seven different WAP gateways from different vendors. Test runs against all seven gateways contained failed test cases. The number of failed test cases varied from 10 to 622 in 2 to 20 test groups. The total number of test cases, failed test cases, total test groups and failed tests groups and shown in Table 3. Results from all seven gateways indicate that they were potentially vulnerable for an attack through the found vulnerabilities. The vulnerability beyond denial-of-service of four gateways was verified by implementing an *exploit* for each of them. The exploits took advantage of buffer overflow vulnerabilities found by the test suite. Each exploit executed assembler code piggybacked into a WSP message. The code is executed, after successful injection, in the target machine using privileges of the gateway process. Total compromise of the gateway services based on any of these four implementations was demonstrated.

Table 3 Failed test cases out of 4326, failed test groups out of 36 and availability of fixes for the seven tested WAP gateways.

Gw	*Failed cases/groups*	*Fix?*	Gw	*Failed cases/groups*	*Fix?*
1	569/10	no	5	664/8	yes
2	141/18	yes	6	622/14	yes
3	10/2	yes	7	148/20	partial
4	385/16	no			

3.3. TEST REPORTING

Detailed test results and test cases accompanied with Java code to execute them were sent to the vendors of the tested gateways. An exploit code was included in bug reports about gateways for which an exploit was constructed for. The purpose of the reports was to get the vulnerabilities fixed, and more importantly, make vendors to rethink their software process to prevent the appearance of such vulnerabilities in the first place. Four vendors have provided a fixed version of their product, one provided a partial fix and two have not provided any fixes. Vendor reactions are also shown in Table 3. All responses received were positive and some vendors indicated that they will take actions to prevent vulnerabilities of this kind in their future products.

The test suite is publicly available on the Internet, excluding the exploits and the names of the products tested [7]. The aim of the publication is to make the test material available for vendors and their customers and to promote public discussion about security vulnerabilities.

4. DISCUSSION

The presented fault injection approach is based on the long known principle of feeding the IUT with "garbage" and observing its behavior. The ability of the approach to systematically uncover robustness problems of tested components has been surprising. In addition to the aforementioned WAP-WSP-Request test suite, we have been testing implementations of various Internet-related protocols. A majority of the tests have found serious problems in the tested implementations. In several cases, the discovered problems can be used to compromise system security.

Numerous vulnerabilities are published daily in security related mailing lists and bulletin boards. Vulnerabilities are often found by individuals who have intentionally looked for or, by mistake, noticed the vulnerable behavior in software. This is uncoordinated and unsystematic fault injection. The presence of these problems indicates that many software developers do not pay attention to the software robustness and fault injection methods are not used in the testing of products.

Widespread use of fault injection for security assessment could enhance the security of software components by eliminating robustness problems. The narrow focus is a limitation of the fault injection testing approach, as only simple implementation errors which manifest themselves as crashes and hangs can be found. Software cannot be made robust by simply fixing the found errors. Rather, numerous problems in the code indicate poor quality and should raise concerns about the software process used to develop the code. The main impact of the testing would be the elevated awareness of robustness problems and their security implications. Software should be implemented to be robust and secure in the first place. A security assessment of products by potential purchasers and independent third parties is a prerequisite for improving the market of robust and high quality software components.

5. CONCLUSIONS

This paper has presented a practical approach for testing the security of software by using fault injection. The main motivation behind the development of the approach has been the large number of robustness problems present in contemporary software products. The approach is based on mutations of the original protocol specification for creating messages containing exceptional structures. These messages are fed into the IUT. Failures of the IUT to handle the input are noted as indications of potential vulnerabilities. A tool using the approach has been used for testing various software components, e.g. seven WAP gateways. Each of the tested gateways had robustness problems. Four of the gateways were verified to contain vulnerabilities that may lead to a total system compromise. Due to the narrow scope of errors uncovered by fault

injection, a comprehensive security evaluation of software is not possible by using the proposed approach. However, a wide use of such tools early in the life-cycle of a software product could cut down the number of errors in shipped products. Such testing would promote awareness of issues related to robustness problems of products.

Acknowledgments

This work is done as part of the project *Security Testing of Protocol Implementations* (PROTOS) funded by the National Technology Agency (TEKES) and parter companies [7]. Addition to project personnel, the authors wish to thank contribution of Petri Mihönen, Marko Heikkinen, Zach Shelby and Marko Palola from VTT Electronics and Juha Röning from University of Oulu.

References

[1] Aho, A. V., Sethi, R. and Ullman, J. D. (1985). *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1985, ISBN 0-201-10088-6.

[2] Baumgarten, B. and Giessler, A. (1994). *OSI Conformance Testing Methodology and TTCN.* Elsevier Science B.V., 1994, ISBN 0-444-89712-7.

[3] Beizer B. (1990). *Software Testing Techniques.* Second Edition, Van Nostrand Reinhold, 1990, ISBN 0-442-20672-0.

[4] Koopman, P. and DeVale, J. (2000). *The Exception Handling Effectiveness of the POSIX Operating Systems.* IEEE Transactions on Software Engineering, Vol. 26, No. 9, September 2000, pp. 837-848.

[5] Maurer, P. M. (1990) *Generating Test Data with Enhanced Context-Free Grammars.* IEEE Software, Vol. 7, No. 4, July 1990, pp. 50-56.

[6] Miller, B. P., et. al. (1995) *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services.* Technical report, Computer Sciences Department, University of Wisconsin, 1995.

[7] Project: Security Testing of Protocol Implementations (PROTOS) `http://www.ee.oulu.fi/research/ouspg/protos/`.

[8] Takanen, A., Laakso, M., Eronen, J. and Röning, J. (2000) *Running Malicious Code By Exploiting Buffer Overflows: A Survey Of Publicly Available Exploits.* EICAR 2000 Best Paper Proceedings.

[9] The Source for Java^{TM} Technology `http://java.sun.com/`.

[10] Voas, J. M. and McGraw Gary. (1998) *Software Fault Injection.* John Wiley & Sons, 1998, ISBN 0-471-18381-4.

[11] WAP Forum `http://www.wapforum.com/`.

[12] WAP Forum. (1998). *WAP Architecture Specification.* 20-Apr-1998.

[13] WAP Forum. (2000). *WAP WSP Specification.* Approved 4-May-2000.