Fraunhofer Institut
Sichere Informations-
Technologie

Diploma Thesis
for obtaining the academic degree
Diplom-Informatiker

Approaches for Automated Software Security Evaluations

Andreas Poller

October 23, 2006

Supervisors: Dr.-Ing. Robert Baumgartl
Chemnitz University of Technology
Chair for Real-time systems

Dipl.-Inf. Sven Türpe
Fraunhofer Institute for Secure
Information Technology
Secure Processes and Infrastructures Group

Abstract/Zusammenfassung

As a consequence of the highly increasing cross-linking of computer systems in computer networks, the possibilities for accessing programs operated at these machines is becoming more and more independent from the possibilities of having physical access to them. Thus the former existing physical access controls have to be replaced by logical access controls which ensure that computer systems are only used for the intended purpose and that the stored data are handled securely and confidentially.

The efficiency of such logical protection mechanism is verified by applying software security tests. During such tests it is proved whether security functions can be bypassed especially by exploiting software errors.

In this diploma thesis approaches for the automation of software security tests are examined regarding their effectiveness and applicability. The results are used to introduce a requirement and evaluation model for the qualitative analysis of such security evaluation automation approaches.

Additionally, the assertion is made that a highly automated software security evaluation is not a sensible development goal referring to the estimated cost-benefit ratio which is gained by trying to realise this goal. Based on this assertion, this diploma thesis discusses how to join the capabilities of a human tester and a software evaluation assistance system in an efficient test process.

Based on this considerations, the design and implementation of a software security evaluation system which has been developed prototypically for this diploma thesis is described. This system significantly involves the human tester in the evaluation process but provides approaches for automation where possible. Furthermore this proof-of-concept prototype is evaluated regarding its practical applicability.

Durch die zunehmende starke Vernetzung von Computertechnologie wird die Möglichkeit des Zugriffs auf einzelne Computersysteme und den darauf ablaufenden Programmen zunehmend ebenso stark unabhängig von den physischen Zugangsmöglichkeiten des Zugreifenden zu diesen Systemen. Diese wegfallenden physischen Zugangsbarrieren müssen deshalb durch logische Zugriffsbeschränkungen ersetzt werden, die sicherstellen, dass Computersysteme nur zu den vorgesehen Zwecken verwendet und die darin gespeicherten Daten sicher und vertraulich verarbeitet werden.

Die Wirksamkeit dieser logischen Schutzmechanismen wird mit Hilfe von s.g. Softwaresicherheitstests verifiziert. Dabei wird insbesondere überprüft, inwiefern Schutzfunktionen durch Zuhilfenahme von in der Software vorhandenen Programmfehlern umgangen werden können.

Diese Diplomarbeit überprüft bestehende Ansätze für die Automatisierung solcher Sicherheitstests hinsichtlich ihrer Wirksamkeit und Anwendbarkeit. Aus den Resultaten dieser Untersuchung wird ein allgemeines Anforderungs- und Bewertungsmodell entwickelt, welches die qualitative Bewertung von Ansätzen zur Sicherheitstestautomatisierung zulässt.

Desweiteren wird die Behauptung aufgestellt, dass die Forderung nach einer zu starken Automatisierung des Testverfahrens sich ungünstig gegenüber des Kosten-Nutzen-Verhältnisses auswirkt, welches bei der Realisierung dieser Forderungen zu erwarten ist. Darauf aufbauend versucht die Diplomarbeit abzugrenzen, wie sich die Fähigkeiten des menschlichen Testers und eines teilautomatisierbaren Softwaresystems effizient in einem Sicherheitstestprozess verbinden lassen.

Basierend auf diesen Betrachtungen wird beschrieben, wie ein Sicherheitsevaluierungssystem, welches prototypisch für diese Diplomarbeit entwickelt wurde, den Menschen zur Erstellung von Testalgorithmen einbindet aber, wenn dies zweckmäßig ist, Automatisierungen ermöglicht. Dieses System wird daraufhin auf seine praktische Anwendbarkeit untersucht.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The "Computer Crime and Security Survey" report [GLLR05], which is yearly published by the Federal Bureau of Investigation and the Computer Security Institute based on the responses of around 700 computer security practitioners in United States companies, government, universities, financial and medical facilities, shows that the types of security incidents producing the highest costs in 2005 where mainly viruses, unauthorized access security breaks and the theft of proprietary information. Beside the fact that the total (reported) losses resulting from cybercrime compared with the preceding years is decreasing, the sectors of unauthorized access, theft of proprietary information and website incidents significantly increases.

In contrast to others, these types of security attacks are often based on flaws in the software applications used. Thus, the quality of software products is still an important aspect for secure computing and is becoming more and more important with the increasing computerisation of human environment and everyday life.

A possibility to verify and ensure a reasonable software security level after finishing the main development process is the evaluation through system testing. Compared with other engineering disciplines, testing of software products is much more difficult and the verification of the absence of errors and weaknesses is practically impossible. Especially the security evaluation where software is not tested against its specifications but is tested for proper functioning outside these well-known specifications is a complex process.

One of the tasks of the Secure Processes and Infrastructures Group at the Fraunhofer Institute for Secure Information Technology is to perform these security evaluations of different kinds of information technology systems. These security audits are primarily so-called black box tests. That means that the examined system is like a black box where the tester can only see the interactions of the system with its environment and not the processes inside. During these examinations, the auditor is quasi in the same situation like the most of the potential attackers which could try to penetrate the system. These tests are normally extensive and therefore a real need for and automation or semi-automation of several procedures exists. Whilst for other testing techniques there already exist various approaches like automatic code evaluation and control flow analysis, the available approaches for black box security evaluation are only a few one.

This diploma thesis wants to show the techniques already existing in this sector and where their limits are. Furthermore, based on these researches and consequential considerations, general limits for the automation of black box security evaluations are defined and approaches for a useful and efficient integration of this automation in the testing workflow of the (human) tester are outlined. In addition a software environment will be presented showing one possible practical solution for the previously formulated aspects. The abilities of this environment will be demonstrated on examples from tester's real life.

As an delimitation to other works in this research area, the main focus of this thesis is discussing methodologies to discover new and unknown security weaknesses in information processing systems and not to discuss different approaches for automatic testing of systems against already known security problems, which is often (but not exclusive) meant by the terms "security scans" or "penetration tests".

1.2 Guide through this diploma thesis

This section should provide a description of the structure of the diploma thesis and some additional annotations. It should help the reader to orient himself while reading the document.

Structure of this diploma thesis

Chapter 2 describes some general methods and aspects of software testing in general and especially the specific characteristics of black box testing. Furthermore, the coherences between software errors and software failures and the differences between software correctness, reliability and security are discussed. At the end of the chapter, typical causes for software failures and the resulting weaknesses and vulnerabilities are presented in a security context.

There already exist several approaches for automated software security evaluation through black box testing. Some of them are working completely automatically and others are semi-automatic. In chapter 3 these existing approaches are examined for their features and capabilities. At the beginning, the concept of the "fuzzers" as one important approach for automated software security evaluation is presented in different complexity levels whereby the "fuzzing frameworks" are the most sophisticated one. In the following, two commercial security scanners as representatives for many other similar applications are discussed. At the end, other approaches which are not clearly classifiable but which provide other notable approaches are described.

The consideration of the existing approaches in chapter 3 already implies that there are limits for the automation of software security evaluations. On the basis of a general model of a software security evaluation process these limits are discussed more extensively in chapter 4. The requirements for an implementation of automation in different complexity levels are outlined. Based on these levels the existing approaches are classified to provide a critique of the current state-of-the-art techniques. As a result, the assertion is made, that a strongly automated security evaluation system is not a reasonable objective for further researches. Instead, an evaluation system is outlined which should combine the abilities of a human tester and an evaluation assistance software in a sensible way. This system concept is the base for the further considerations in the following chapters.

In chapter 5 the evaluation system proposed in chapter 4 is described more detailed. The general design of the system components and the basic data structures are presented. The decision is made that a module based system where the modules can be arranged using a graphical user interface should be the main development goal. This arrangement is called the testing algorithm. The testing algorithms should become automatable by involving automation approaches in test case creation and failure detection. On the other side the sensors and actuators which are the direct connection between the test object and the testing algorithm should be implemented separately as a distributed application.

Based on the system design presented in chapter 5 an implementation of a software security evaluation system has been done for this diploma thesis. A description of this implementation can be found in chapter 6. The single parts of this implementation are the sensors, the actuators and the central control unit. These components together form the whole evaluation system as a centralized distributed system. Furthermore the modules which have been implemented exemplarily to demonstrate some common testing algorithm are explained.

The capabilities of this evaluation system are examined in chapter 7. Beginning with simple examples the test case creation and the failure detection are demonstrated. More sophisticated testing algorithm and strategies derived from tester's real life and practical results are considered as well.

In chapter 8 the results of the examinations in chapter 7 are considered and the capabilities of the evaluation system are compared with the capabilities of the existing approaches presented in chapter 3. It is discussed which deficits of the other state-of-the-art techniques could be compensated and which relevant problems could not be solved. Furthermore a brief outlook for further

researches on the topic is given.

Notation conventions

Few special notation conventions are used in this diploma thesis which are described in this subsection.

Paragraphs which appear in a gray box are belonging to an example which illustrates an issue mentioned in the previous part of the section. Usually the introduction of an example is also justified in the paragraphs before the grey box.

`Text which represents machine-processed information like source code, data in a description language, interface messages or user inputs and outputs is written using a typewriter font. In some cases a part of this text can also be coloured to increase the readability.`

In Appendix C the section titles are coloured. These colours represent specific data types. The same colours are used in the testing algorithm diagrams to illustrate the type of the data which is exchanged at the module links (see also 6.4).

Annotations

In this diploma thesis terms are used which are registered trademarks. The use of such a term always refers to the purpose the owner of the trademark intents to express. Thus these terms are not explicitly marked.

Diagrams and other images in this diploma thesis include materials from the Open Clip Art Library [OPE]. All of these materials are public domain which means that the author abandons his rights derived from the copyright by publishing these materials in the Open Clip Art Library. The included materials are only so-called clip arts which show hardware components, animals, humans or symbolize common technical terms. Therefore the use of these clip arts does not contribute anything relevant to the meaning of the diagrams and images. They only help to increase the readability.

This document is a reprint of the original thesis which has been handed in to the Chemnitz University of Technology at September 14, 2006. It is stored as publication 187/2006 [Pol06] in the MONARCH digital archive system of the Chemnitz University of Technology and digitally signed using the certificate of the author issued by the Fraunhofer Gesellschaft. The issuance chain can be verifyed by proving the fingerprint of the Fraunhofer root certificate [PKI].

The content of the CD which is enclosed to the print version of this diploma thesis (see also Appendix E) is provided as tarball and zip-file at the webpage of this publication in the MONARCH system. The hash values of these two files are the following:

```
AutoSecEvalCD.tar.gz

MD5  8464433222b86dbbcf7abd4a3e96d12a
SHA1 5393d46f6b6c98ba3a878dbafe2526f64a8a2eac

AutoSecEvalCD.zip

MD5  a65e9015480afadbfa0ad36e8fbc5acc
SHA1 e3d1592127521c6c7c056b7510625c7f7612282f
```

2 Fundamentals of black box testing

As a general introduction into the topic of this diploma thesis this chapter discusses the basic principles of software testing and at this especially the concept of black box testing. The origins of errors in software and the resulting failures are outlined as well. Additionally, at the end of the chapter security relevant issues are considered.

2.1 Concept of black box testing and security evaluations

In the traditional waterfall software development model, software testing is situated between the development phase and the delivery or customer implementation phase. Thus, it can be seen as one border between the software producer and the software consumer (also called software user) which should ensure the quality of the product and the conformance to the requirements claimed at the beginning of the software life cycle.

Testing has two major aspects: First it is an activity to find errors which had been inserted during one of the preceding steps in the software development process and to verify and validate the generated product. Second, it is a method to find these errors by involving an execution of the whole or parts of the software. Beside that, there also exist other methods to find errors without executing the software like special types of source code analysis.

Black box testing as one particular testing technique is a method where the tester has the same view on the software product like every "normal" consumer of the system. It only involves the requirement specification for the software. Therefore, black box testing can only discover errors which are potentially visible to the consumer. That is the reason why black box testing is often also called behavioural testing.

In the following, a list of common black box testing techniques is given exemplarily as a small excerpt from a large set of existing testing methods [Bei95][KFN99]:

- **Domain testing**

  Domain testing tries to handle the problem of a practical infinite input and test case space by splitting this space in several subspaces (or domains) and then generating a set of test cases which are typical representatives of each of these domains. This strategy is based on the assumption that the results of the execution of these test cases are similar to each possible test case in the same domain.

  Domain testing has different sub techniques like boundary value analysis, equivalence class partitioning or pairwise testing. The main difference is the dimension of the considered input space, its sectioning and the selection of the representatives.

- **Scenario testing**

  Scenario tests are based on stories about the use of the tested program and is therefore often also called use case testing. The assumed user has special motivations which should be derived from the real world. These scenarios should be sufficient complex to test many parts of the program under different conditions.

Thus, scenario testing itself is often very complex, because the coverage of all program functionalities and features must be ensured to provide an adequate test.

- **Function testing**

  The focus of this testing technique is on the different functions and features which are provided by the tested program. Each of this function is checked whether the claimed functionality has been successfully implemented.

  Function testing is often used to supplement earlier states in the software engineering process, because software functions are added one by one which does not allow the test of the whole system but the test of the already implemented functions.

- **Risk-based testing**

  The idea of risk-based testing is different from the other testing techniques. The base approach is not creating test cases to cover as much functionality and features of the tested program as possible but analysing the risks and failures being relevant for the program and how to create test cases to prove the possible presence of errors causing these failures.

  This testing method is often related to risk analysis, failure mode and effects analysis or cost-benefit analysis.

- **Specification-based testing**

  Specification-based testing uses the claims in the specifications of a software product to test against. Thus, such tests are often used to ensure the conformance of the product to contracts, agreements and other regulatory [anhren] obligations which are commonly a main base for the specifications. This has often a legal background regarding the relationship between the software producer and the software consumer.

- **State-transition testing**

  In state-transition testing the main objective is to construct a state-transition graph of the tested system in order to create test cases which ensure that after executing all of these test cases each transition has been executed at least once. This method is control flow oriented and is only applicable if the state and executed transitions are reflected to the systems environment and can be identified by the black box tester.

A term which was used but not clarified in the preceding paragraphs is the word “error”. By defining the word “error” the difference between black box testing and security evaluations will become understandable.

In the traditional black box testing an error is a behaviour of the regarded software system which is not conform to its specifications or implies that the software does not meet the requirements which are the base for these specifications. The conformance with the specification is also often expressed with the terms “correctness” or “the software is correct”. The second important expression in this context is “reliability”, which informally means that the user can rely on the software and the software meets the requirements. Per definition, if the formulated specifications are correct then correct software is also reliable.

The focus of “security” in the meaning of a software quality characteristic is the next step in the definition process. A software can be considered as secure, when it can not be affected by threats from its environment which can cause an unwanted malicious behaviour and especially a bypassing

of necessary trust and protection conventions and barriers which can result in data loss, loss of system integrity or data confidentiality.

Software which is not reliable is also often not secure. But software can be classified as reliable without being secure. This is a consequence of the fact that security flaws often have the characteristic that they exist without affecting normal software processes and workflows. Thus, the software with these types of errors is mostly reliable for its users.

The delimitation of the terms is explained in the following simple example:

A function is considered which should calculate the reciprocal of an integer number. The requirements, a specification and a corresponding implementation are shown in figure 2.1. The presented implementation is correct regarding the previous formulated specification. But the specification itself is faulty because it does not factor in that the division by zero is not defined. The program will fail when the zero, which is an integer number, is entered (the zero also appears when entering a string instead of a number). Thus the function is correct but not reliable.

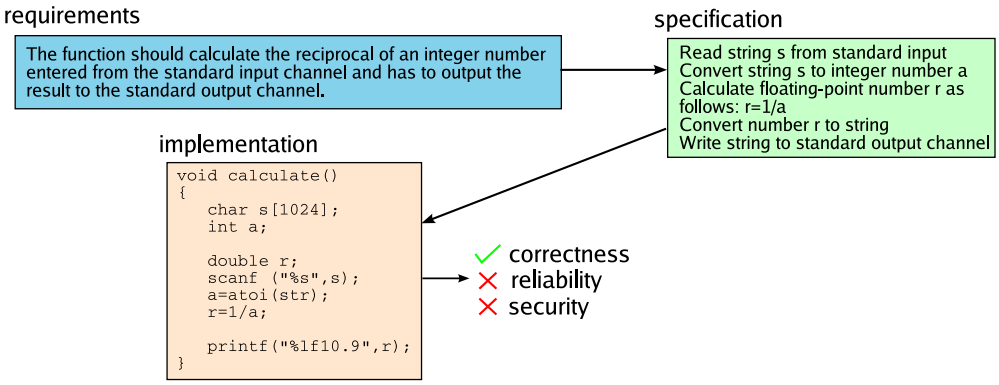


Figure 2.1: Correct implementation of the reciprocal function

A reliable implementation with a better specification is shown in figure 2.2. Here, the division by zero is considered and the program outputs an error when the zero or a string is entered. Beside that, there exists another issue in the function. The character array `s` which is used to temporary save the input has a fixed size. When entering a string with a length greater then 1024 characters the `scanf` function will overwrite data on the programs stack outside the array boundaries. This behaviour can be a security flaw which is very typical and which is called "buffer overflow" (see also 2.4). Nevertheless, the function can be seen as reliable, because the likelihood of the appearance such a string is very low in a non-malicious program environment.

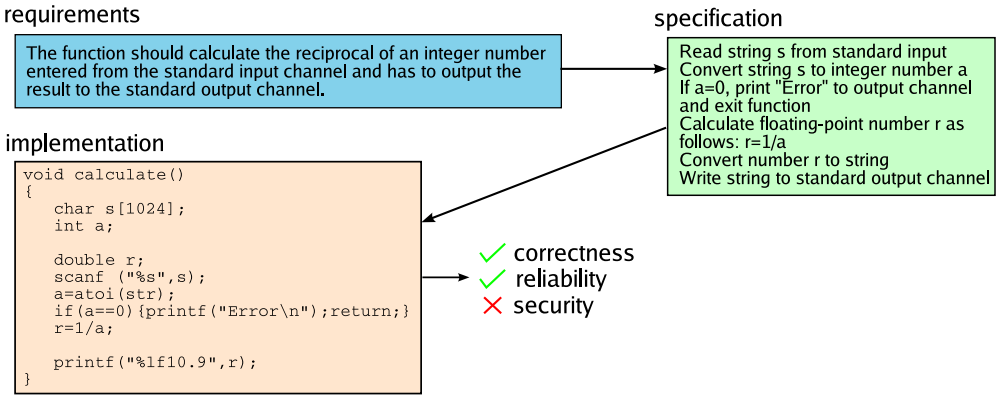


Figure 2.2: Reliable implementation of the reciprocal function

This problem is solved in the implementation shown in figure 2.3. The program with the function is linked with a special library which prevents buffer overflow security breaks in string functions. This implementation of the function can now be considered as secure.

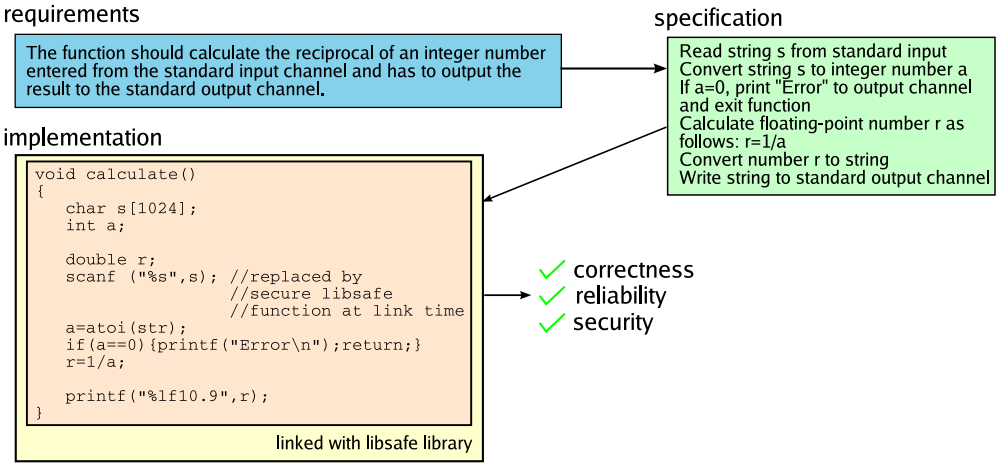


Figure 2.3: Secure implementation of the reciprocal function

Especially the delimitation of the terms “reliability” and “security” is not very sharp and subjective. Security is often considered as a feature of reliability, depending on the use of the software. In this diploma thesis the separation is defined as follows:

- Software is reliable, if the likelihood of the appearance of errors in non-malicious environments is low enough to provide an undisturbed use in common application scenarios.
- Software is secure, if the likelihood of the appearance of errors in malicious environments caused by security threats is low enough to ensure the following general security services [Org89]:
  - Authentication, which is the verification of the identities of involved entities and data origins
  - Confidentiality, which are services to protect data against disclosure by unauthorized entities
  - Integrity, which means that it is ensured that data are in the right form for the intended use and that they are protected against malicious modifications
  - Access Control (Authorisation), which is the granting or the denial of access to resources regarding the identity of the accessing entity
  - Availability, which means that an acceptable level of service availability is provided to authorized entities
  - Accountability, which refers to the possibility of the association of security relevant actions to the achieving entity
  - Non-repudiation, which means that actions related to legal interests can not be denied by involved parties after they have been performed

2.2 Reasons for performing black box tests

One of the questions often asked in this context is why to perform black box tests instead of simply using white box testing. Often it is argued that a white box test is more sophisticated because it allows a detailed observation and verification of the internal behaviour of the test object.

There exist several reasons why black box testing is more common especially for security evaluations. The most important are considered in the following:

White box testing requires the availability of the program source code and the analysis of this source code for an identification of security flaws. Often an execution is necessary to examine the collaboration of parts of the source code. This examination is more complete when done properly than in black box testing but the effort is very high in many cases. The reason is, that this examination is potential less goal-oriented than black box testing.

It is important to understand that security evaluations usually try to find lacks of the software implementation and specification which allow malicious entities in the environment of the software to compromise the system. There usually exist security flaws which are clearly identifiable by potential attackers and others which require a high effort to discover and exploit them. Because of the fact, that software evaluations always have to factor in a reasonable cost-benefit ratio the more important errors are those of the first mentioned category.

White box testing which considers the internal behaviour of the test object tends to discover security flaws which are not the most relevant one. This is the result of the fact that the potential attacker does usually not have this internal description of the compromised software. The attacks are usually derived from observations done from an outside point of view. This aspect is more sensible realized in black box testing.

Often there are also organisational reasons why white box testing can not be applied e.g. that the source code has to be kept confidential against the testing instance. Sometimes the specifications are also incomplete or completely missing.

2.3 Causes for software failures

The reasons for software failures can be generally categorized in three classes [VMKV97]:

- Erroneous input data
- Faulty code
- A combination of both

In other words, there can be anomalies inside the code and outside the code which can cause failures separately or in acting together.

A convenient model to describe the causes and effects of errors is provided by Andreas Zeller [Zel05] as follows:

A central point in Zellers model is the infection. An infection is the deviation of the resulting program state from the intended program state after the execution of erroneous code. This infection is necessary to be able to recognize the code defect by testing but it is possible to execute erroneous code without causing an infection.

Hereby, example reasons for defects in code can be the following [Gra90]:

- Numerical errors, which are caused when the special properties of computer arithmetic, like limited numerical precisions or domains, have not been considered.
- Incomplete conditions, which are the result of an incomplete or incorrect programmed evaluation of input or interim values.
- Unexpected variable values caused by incomplete variable definitions or validations
- Exceptional and borderline cases for selected algorithm where a special processing has not been factored

- Wrong hypotheses, which cause the building of algorithms which do not meet the program requirements or are not suitable for the problem to solve

  This error type can also appear when the application scenario is changed without a new consideration of the software applicability.

- Use of an improper version handling, which causes the unwanted interaction of code in incompatible development states

- Improper use of operating resources like memory leaks or overflow of assigned memory blocks

- Bad coding style which results in misinterpretation or incorrect use of functions and variables during programming

- Defects, which have been added intentionally for e.g. debugging or forced compatibility reasons

Back to Zellers infection model, the state of a program is defined as the values of the program variables and the current execution position in the program code. This definition is different from the normally better characterization of a program state as the summary of the states of all operating resources related to the program, but the Zeller model can be extended to this point of view easily[1].

An infected state normally causes other infected states, because it supplies subsequently executed program code with wrong input data. This infection propagation must not necessarily take place but normally it appears sooner or later. An example for such a possible infection propagation where by the time each operating resource has an infected state is given in figure 2.4.

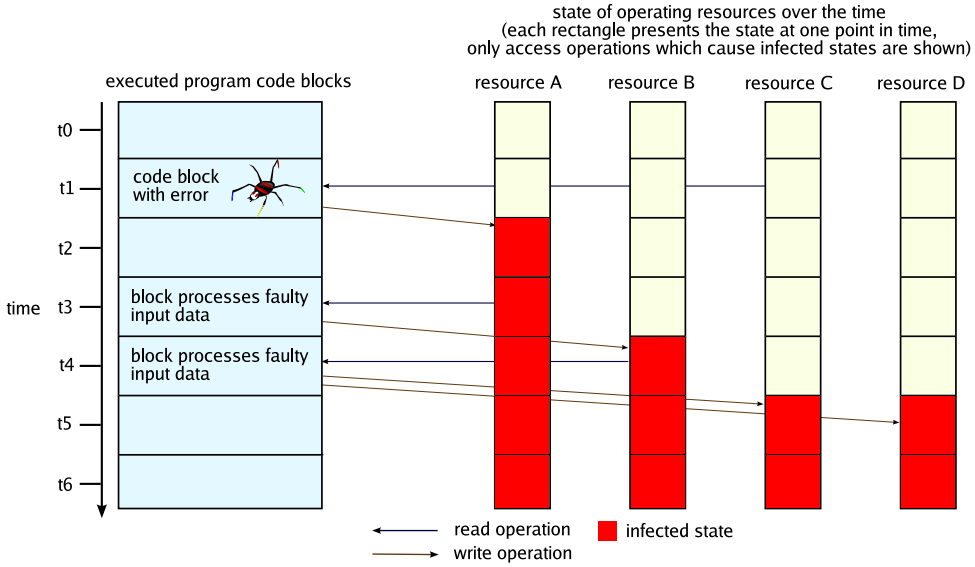


Figure 2.4: The propagation of an infection caused by an error in the program code

A failure is defined as an error in the program behaviour which can be identified by an outside observer. If a failure is not observed then an infection which can nevertheless appear can not be discovered.

[1] In fact, Zeller is more focused on debugging where variable values have a higher importance

2.4 Security relevant software failures and weaknesses

In the previous section different errors have been presented which can cause security flaws. In this section, several security relevant software failures which are also called vulnerabilities are considered. The delimitation of this term is not very sharp. The word "vulnerability" can also focus on the error which can cause a security breaking failure.

It is difficulty to introduce a consistent taxonomy to classify vulnerabilities. Thus, only some factors which can be used as a characterisation are listed below:

- Is the vulnerability directly affecting the attacked component or is the infection propagated to another component which then fails? This component normally trusts the erroneous component and is therefore often not able to prevent the security failure. A classification based on this model depends on the point of view and the abstraction level (interacting software functions and modules, front-end and back-end systems, clients and servers).
- Is the vulnerability utilisable to inject and to execute malicious code or to inject malformed data to provoke a change in the software control flow in order to invoke a malicious behaviour?
- Does the exploitation of the vulnerability require an active injection of malicious data or can the security break be performed by passive methods only using an analysis process? These different types of attacks are also called "active attacks" and "passive attacks".
- What are the effects of attacks against the vulnerability regarding the general security services? (see also section 2.1)

Different classification approaches can be found e.g. in [LBMC93] and [Inc06].

In the following, examples for common software vulnerabilities are presented and explained. This listening should also be a base for the further considerations in this diploma thesis:

- **Buffer overflow**

  A buffer overflow occurs when incoming data are written in an array buffer which is smaller than the size of the data. As a result the writing routine exceeds the array boundary and overwrites data in different areas of the program memory depending on the location of the buffer. A buffer overflow attack can be used to inject and execute malicious code or to modify data in the program variables.

- **Command injection**

  A command injection is a failure, where an attacker injects a malicious command sequence as data input and this data input is used to form a command or command parameters executed by another component. The error cause is hereby an insufficient check of this input to exclude specific data sequences which can result in an unwanted behaviour. A popular subtype of this security flaw is the SQL injection, where the attacker can manipulate commands which are generated for a SQL database server by injecting special data inputs in the generating application which is usually a web front-end program.

- **Format string attack**

  A format string attack can be performed when input strings are used as formatting strings for string input or output operations. Some string operations in selected programming languages allow e.g. tags to read data from normally not accessible parts in the program memory which can be used by an attacker to get unauthorized data access.

- **Cross-site scripting**

  Cross-site scripting is the injection of malicious code or data which is not taking effect on the erroneous system itself but intentionally causes failures on other systems which rely on data received from the infected system. Usually, this term is deriving from security flaws in the area of web applications where an attacking client uses a web server input channel to propagate malicious code to other web server clients.

- **Race condition**

  A race condition vulnerability can occur during the access to a resource which only supports serial access. The error is caused by a missing lock (insufficient synchronisation) on the resource during the validation and access granting process which allows a change of the resource by other instances after the successful validation.

- **Cryptographic errors**

  A cryptographic error or a weak encryption can allow an attacker to decrypt sensitive data without or with restricted knowledge about the encryption parameters.

- **Weak random numbers**

  If a routine which produces predictable random numbers is used in a software, a potential attacker can use this property to foresee specific application conditions and parameters.

- **Memory leaks**

  Memory leaks occur when allocated memory areas are not freed before deleting the references to them. An attacker who knows under which circumstances this behaviour takes place can perform a so-called denial of service attack which causes an unavailability of a computer resource for its intended purpose and group of users.

- **Malformed data**

  Malformed data attacks are any other possible attacks not described above which use the injection of malicious data sequences to generate a program behaviour not originally specified. The flaws which facilitate such attacks are always defective or incomplete validations of input data.

- **Faulty access control**

  Faulty access controls allow the escalation of privileges of unauthorized users or the bypassing of control mechanism. These types of attacks are often based on weak protection of credentials or tokens used for the authorisation. Also the defective processing of malformed input data can be the cause for these failures.

There also exist vulnerabilities which can enable a potential attacker to initiate so-called “social engineering attacks”. These types of attacks do not cause a failure in the attacked software but they can be used to mislead human users of the programs to start actions which can support the attacker to override installed security mechanism. At this, these vulnerabilities are not in the main focus of this diploma thesis, but they are considered whether they are relevant under special circumstances.

Vulnerabilities which can result because of e.g. faulty software configuration management or insufficient or inconsistent documentation or user guidance are also not main part of the considerations. Often these attack potentials can be discovered by an examination of documents and guidelines which have been delivered with the test object. These kinds of evaluations are not inside the scope of this diploma thesis.

3 Existing approaches for automated software security evaluation

This chapter classifies and discusses several approaches for software security evaluation and especially for the automation of the evaluation process. Beside software, which is available as open source tools, commercial programs are considered as well.

3.1 Protocol fuzzers

"Protocol fuzzers" belong to a class of software tools which emulate an interface regarding specifications of a selected information exchange protocol and which interact in an unusual or "forbidden" (in terms of the regarded protocol) manner. In other words, the tools behave outside the defined rules for data and control flow [Oeh05][vS05]. The test object tries to collaborate with this interface and should be able to recognize the malicious interactions and data. Otherwise it changes to an undefined or unwanted state which can result in vulnerability.

Software utilities that imitate malicious human user inputs belong to the first considered type of fuzzers. The second type is mainly used to test the correct implementation of computer network protocols at different layers of the OSI reference model. Another group of fuzzers manipulates files and checks the proper behaviour of file reading routines in error cases. The fourth presented group fuzzes interactions between the operating system and processes at the application level.

User input fuzzers

One of the first approaches to automated black box software evaluation was published in 1990 in connection with a reliability study of several UNIX utilities [MFS90]. The main motive for this study was the observation, that some applications which were operated by remote terminals crashed because of noisy telephone lines used for the interconnection of the remote and the local system (the line "fuzzed"). From that experience the assumption was deduced, that there exist different errors in the code of the used computer programs that can cause software failures, when these programs consume special (randomly) manipulated inputs.

The initiators of the study, Miller et al, developed a system for testing software for these failures. This system mainly consisted of a random generator, the so-called the "fuzz" program, which generated a random string of a specified length with particular features. The second program called "Ptyjig" provided a pseudo terminal which was used to send the random string as a pretended human user input to the test object. An overview of the system is given in figure 3.1. For the purpose of an automation, this process was performed in a loop and the test object was checked whether it crashed or not after each iteration. Depending on the used system platform between 25 and 33 % of the utilities (including popular UNIX tools like vi, make, emacs, telnet etc.) did not pass the evaluation successfully.

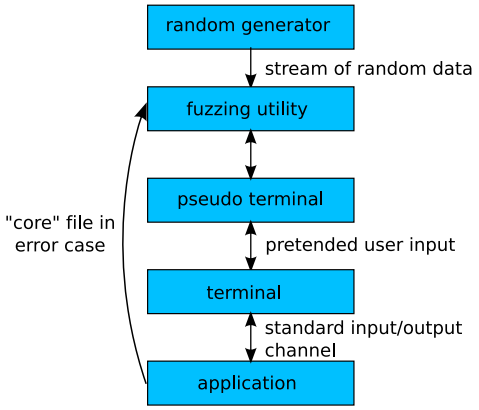


Figure 3.1: Fuzzing system for terminal-controlled applications proposed by Miller et al

In difference to today's use of fuzzers, the main aspect of this study was not especially in examining security relevant issues but in verification of the entire reliability of the software. Beside this different evaluation goal, the basic methodology is similar to many security relevant fuzzing tests performed nowadays.

Based on this first study, a second report was published in 1995, where the authors not only evaluated UNIX utilities but also examined UNIX network services [MKL+95]. A further development of this input fuzzer was published by Ghosh et al [GSS98] in 1998. The software environment (showed in figure 3.2) presented in this paper was able to generate random malicious user inputs based on a Backus-Naur grammar[1]. This approach was considered to provide a possibility for a more specific test of selected software functions and can be seen as a precursor of the fuzzing frameworks (see 3.2).

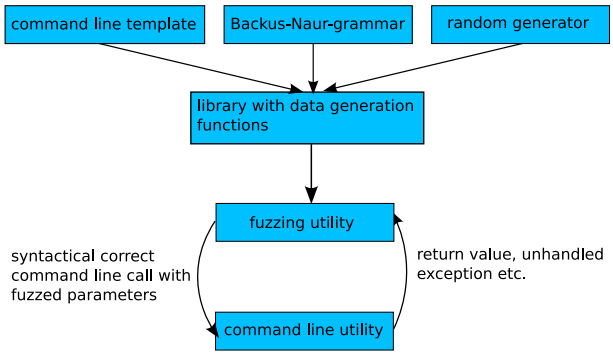


Figure 3.2: Fuzzing system for applications using command line options proposed by Gosh et al

Network protocol fuzzers

Network protocol fuzzers exist in many different versions. They are usually an implementation of a specific network protocol which is connected with a random generator to create malformed data entities within the protocol data units. Two popular representatives are mangleme [MAN] and ircfuzz [IRC].

Mangleme is a CGI[2] application which can be run with a web server. It generates a HTML page as an answer to a client request which consists of incidentally selected HTML tags with randomized tag option fields. The evaluation target, a web browser client, should be able to handle these malicious web pages. The mangleme developer has claimed that he found many bugs in almost every existing web browser (e.g. Microsoft Internet Explorer, Mozilla, Opera, Firefox, Lynx and Safari).

[1] Further information about formal languages and grammars can also be found in [HMRU00].
[2] See glossary for an explaination of the term

Ircfuzz is a similar concept but, at this, the fuzzing target is the IRC protocol. The evaluation objects are IRC clients which try to communicate with the ircfuzz server which produces anomalous server messages derived from the IRC protocol. The author also claims, that ircfuzz has broken many popular IRC clients (e.g. xchat, trillian, kopete, iRC).

These two examples should be considered as a demonstration of the base concepts of these fuzzers. It is often a static implementation of a widely used network protocol connected with random generators for producing malicious data to insert in the protocol data units. Restricted by the fact, that they only generate data, normally an analysis of the evaluation is not provided. A human tester or another monitoring system has to supervise the test and has to analyse failures. This concept is very simple. Nevertheless it seems that it has helped to find serious security flaws in popular software products.

File fuzzers

File fuzzers can be called the "offline versions" of network protocol fuzzers. These tools generate files including malicious data and check whether an evaluation target is able to read the file content.

A main difference to communication channels based on network protocols is that file input channels are normally not used in the context of a multistage conversation. Files are always considered as a whole context and are always changed as a whole context. During the manipulation of file data there is usually no access allowed for other activities. The activity which reads the file, communicates with a system driver which uses a simple protocol which allows to access different parts of the file. Thus, the communication partner (de facto the file itself, but the file is not an activity) can be seen as stateless related to the communication context. This charateristic makes the difference between the File fuzzers and the other fuzzers mentioned in this chapter

A popular representative is SPIKEfile [SPI] which is derived from the fuzzing framework SPIKE (see 3.2).

Operating system interactions fuzzers

Operating system interaction fuzzers are used to audit operating systems, whether they are vulnerable through their service access points. They usually randomly generate system calls or they embed invalid data in the system call parameters. Another common term used for these tools is "kernel stress test tools". An example can be found in [STR].

3.2 Fuzzing frameworks

Fuzzing frameworks are toolkits which allow a more sophisticated design of software tests by providing a development environment for software testers for modelling test cases.

There is an overlapping between approaches in this section and the ones in 3.1. Some of the protocol fuzzers from section 3.1 provide a sophisticated configurability and can therefore be considered as a kind of fuzzing framework as well.

But the distinctive feature for being a fuzzing framework is whether the tool allows testing based on adaptable rules for the data at the emulated interface. These rules should allow a complete adaptation of the data representation within the abstraction layer (e.g. the OSI layer for network protocol fuzzing frameworks) the fuzzing framework is working at. The input of this data description, independent from the form this input is done, is mandatory for the use of a fuzzing framework.

Because all the fuzzing frameworks listed below in this section are used for network protocol fuzzing, a partition regarding their function like in 3.1 is not reasonable.

PROTOS - Project: Security Assessment of Protocol Implementations

PROTOS is a well-known cyber security project at the Oulu University Secure Programming Group of the University of Oulu, Finland. The developed fuzzing framework itself is not freely available, but the researcher published many so-called "Test-suites" for different network protocols (e.g. SIP, SNMP and LDAP) which include test cases and the software necessary for the test case injection.

The main development behind PROTOS is the concept of the "mini-simulation method", which was originally designed for functional robustness testing and which covers the following application scenarios:

- Functional testing as a model testing of functions of a real-world entity
- Stub implementation as a model providing function stubs to a real-world entity
- Traffic analysis as a modelling of the interactions between two real-world entities
- Simulation as an interaction between two models
- Conversion as a mediator between the interactions of two real-world entities

The "mini simulation" is used to model protocol exchanges in interaction and syntax. Both, the behaviour specification and the syntax specification are produced using the same notation of an attribute grammar, which extends context-free grammars (Backus-Naur form) for associations of grammar symbols with attributes. For the creation of a mini-simulation such a grammar and a set of rules is given to construct the application by using parameterised components from the framework. The rules can be separated into different types as follows:

- Communication rules are used to associate elements of the given grammar with communication functions like opening a port, sending protocol data units etc.
- Replacement rules are used to modify the specification given by the grammar. They are e.g. used to inject a malicious behaviour.
- Semantic rules are used to extend the given grammar for functionality e.g. checksum, length field or padding generation.

The created application is a simulation of the protocol, which is able to communicate with its environment using functions of the mini-simulation toolkit and which can be intentionally faulty when configured with pertinent replacement rules. An overview of this process can be found in figure 3.3.

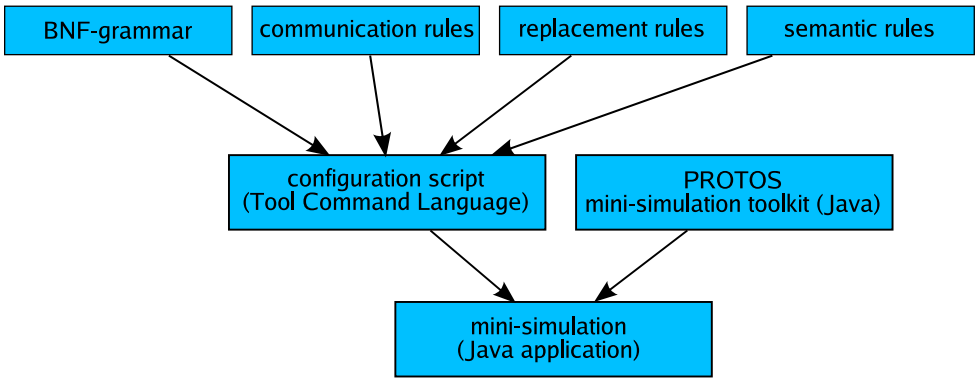


Figure 3.3: The creation of a PROTOS mini-simulation

The tests themselves are performed regarding the configuration script of the test suite (the mini-simulation application) where the test cases are implicit defined by so-called selection rules. For generating the test cases with anomalous behaviour, the given grammar is modified regarding these rules. Each set of grammar manipulations produces one ore more test cases which are summarized in one test group for each set. The monitoring of the test subject is not part of the mini-simulation application and has to be implemented using other external solutions.

A more extensive overview of the mini-simulation concept with an example for a test suite for TFTP can be found in [Kak01].

SPIKE fuzzer creation kit

In difference to PROTOS, SPIKE is a fuzzing framework that generates malicious input data using a block-based[3] sequential description of the protocol to fuzz. SPIKE has been developed mainly for the evaluation of network services to uncover security flaws. Thus the framework also includes many routines for different types of communications over network interfaces.

The language (also called SPIKE script) used for creating the protocol specification uses the following elements:

- Block commands to mark the beginning and the end of one semantic entity e.g. a higher level protocol unit encapsulated in a lower level protocol unit. The block commands are also used to name the semantic entity. Blocks can also be nested inside other blocks, which allows to define a hierarchy of data.
- Block information placeholders which are used to include information about a higher level block into a lower level block e.g. length fields. This functionality is deemed necessary, because many network protocols require additional information about embedded data or higher-level protocol units. Therefore, a manipulation of these additional data fields is essential when manipulating the embedded data itself.
- Data field insert commands are used to insert specific data into a block like strings, binary data or integer. Beside the information calculated by the block information placeholder functions, these data are output in the communication channel. A data field can also be marked as "variable" which advices the framework to fuzz it during the test.

An example of a description of a fictitious protocol can be found in figure 3.4.

```
s_string("SEND\r\n"); //A static inserted string
s_string("Information-Length:  ");
s_blocksize_unsigned_string_variable("information",7); //Insert the size of the
"information" block
s_string("\r\n");
s_block_start("information"); // Begin of the block "information"
s_binary_variable("123431242"); //Data pattern, which is variable
s_block_end("information"); // End of the block "information"
```

Figure 3.4: Example for a protocol description using the SPIKE script language

The SPIKE framework is written in the programming language "C". Thus the fuzzer to be derived also has to be written in this programming language. When creating a fuzzer with SPIKE then usually the framework is used to create messages regarding the block-based description (also called "SPIKE-file") given before and to provide the communication channels to send these messages to the evaluation target. The analysis of the answers from the test entity has to be implemented

[3] The term "block-based protocol analysis" is introduced in [Ait02].

separately and is not provided by the toolkit. A simple example of a SPIKE process is given in figure 3.5. This figure shows the different steps of the fuzzing algorithm and the corresponding function calls.

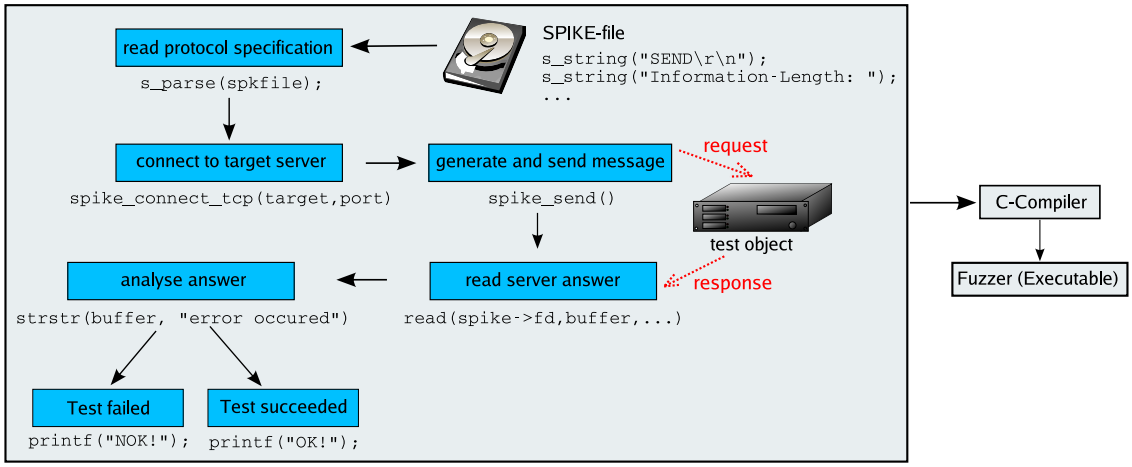


Figure 3.5: An example of a simple SPIKE fuzzer which evaluates a network service by sending a fuzzed request and analyzing the server response

An object-oriented extension of SPIKE is Peach [PEA], which is written in the programming language python. Its concept is similar, but several additional functions had been added (e.g. dictionary integration for fuzzing or groups for a better arrangement of the variable data fields).

Further information about block-based fuzzing in SPIKE can be found in [Ait02].

3.3 Commercial vulnerability scanners

Beside open source software and academic concepts and prototypes for automated software security evaluation, there exists an extensive market of commercial systems. For this diploma thesis, two of the most popular products has been chosen exemplarily to examine their features and methods:

- WebInspect by SPI Dynamics [WEBa]
- AppScan by Watchfire [APP]

(This selection has been done concerning the popularity of the software and whether it is possible to obtain a trial version.)

All of these products are limited in functionality to provide tests of web applications only. Because of this fact all other considerations in this section are explicitly reduced to this area of operation.

The developers of these products claim that they are able to automatically find security flaws in web applications:

- “Watchfire AppScan, the market-share leading web application vulnerability assessment software, automates security testing throughout the software development lifecycle.”, AppScan - product website
- “WebInspect is the most accurate and comprehensive automated Web application and Web services vulnerability assessment solution available today.“, WebInspect datasheet

It is difficulty to obtain detailed information about the software internals. Beside that, a complete evaluation of these products and especially of the promised features would exceed the scale of this thesis. Nevertheless, it is possible to provide a smaller and restricted evaluation based on a well-defined and simple application scenario. This test is used to study the systems and to give a short summary about the observed functionality and whether these tools are relevant in the following context of this diploma thesis. A complete description of the test is given in Appendix A.

The observations during the tests of the tools allow the following assumptions about their evaluation concepts:

First the tools use a crawler to explore the structure of the web application. Then these gained information is used to formulate requests to the test target where parameter patterns, which are well-known for causing security flaws in similar contexts, are inserted at different protocol levels (e.g. manipulated URLs, form parameters or HTTP header options). These patterns are usually derived from a database which is part of the product. The web application responses are analysed by an error and security flaw detection and the results are notified in a report which the user receives at the end of the test. The whole process is automated and does not need the assistance of the tool user. An overview is also given in figure 3.6.

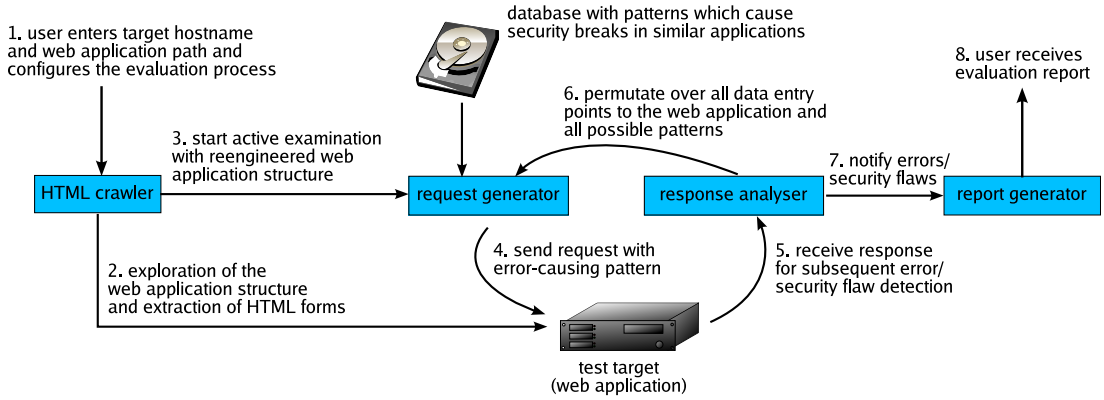


Figure 3.6: Automatic evaluation process performed by the tested commercial vulnerability scanners

The following observations have been made in addition:

- The sequence of the manipulations is not structured in that way, that the effects of a manipulation are traced over many state transitions in the web application. In many cases, the main assumption for those examination methods is probably that the security issue appears directly in the response to the anterior request with the error pattern.
- It seems, that not all web application structures can be successfully explored by the crawler.
- During the test many false positives (positive in the sense that a security issue has been found) appeared which are caused by a misinterpretation of the web application responses.

The efficiency of the presented products can not be assessed in this diploma thesis. But as a result of this test it can be questioned, that this approach is able to discover many relevant and more structured vulnerabilities. Especially the automatic interpretation of the web application interface semantic and the web application responses is a serious problem. The hit ratio during the executed tests can be rated as unsatisfactory.

3.4 Other approaches

Beside the approaches previously considered there exist other methods for automated security evaluation which does not match the other categories and which are mainly more academic and experimental. These concepts will be described in this section.

WAVES - Web Application Vulnerability and Error Scanner

WAVES was a project at the Institute of Information Science at Academia Sinica, Taipei, Taiwan. It was conceived as an open source project but since May 2006 the project website is no more available. So it is not possible to describe this project in detail but to give a general overview about the base concepts. The publications available for this project are [HTLK04] and [HHLT03].

As the name implies the project is mainly focused on the evaluation of web applications. In this context, web applications are a group of software systems which are operated by using services accessible through networks and based on HTTP and HTML. The web application is often a distributed system with different back-end applications. The main goal for the evaluations with WAVES is the detection of cross-site scripting and SQL injection vulnerabilities (see also 2.4).

An important part of WAVES is the crawling mechanism which is used for a reengineering of the user interface provided by the web application. The crawler identifies data entry points of the application for a further use in the error injection phase. The extraction algorithm is mainly focused on the HTML forms[4]. Thus, it has been considered as useful that the crawler is able to understand the semantics of these forms. It uses a dynamically adapted knowledge database to provide these data entry points with valid information in order to simulate a "normal" user and to allow a so-called "deep" or "hidden"[5] scan of the application. Also during the penetration test this database is used to obtain authentic values for the variables which are not fuzzed. The complete reengineering process is described in figure 3.7.

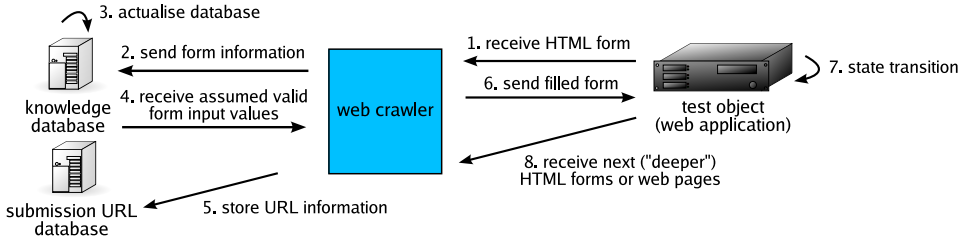


Figure 3.7: The WAVES web application reengineering process

Another approach suggested by Huang et al is the so-called negative response extraction. It is used to identify whether the error injection into the system has been discovered by a validation process inside the web application or not. The procedure which should ensure this correct error identification injects an input assumed as valid generated with the use of the knowledge database and an input assumed as invalid in order to study the systems behaviour. The result of these two inputs is compared with the result of a vulnerability test case. It was claimed, that this strategy improves the likelihood of the correct detection of successful and unsuccessful penetration attempts.

To test the examined system for cross-site scripting vulnerabilities the crawler executes the code embedded in the server response. The interactions between the script engine which is used for that execution and its environment are audited to detect successful attacks. Figure 3.8 shows the evaluation process.

[4] Forms which have to be filled out by the user to control the web application behaviour, see also the glossary
[5] "hidden" or "deep" means in this context that this crawler should be able to scan behind barriers which can not be passed by conventional other web crawlers used for website indexing

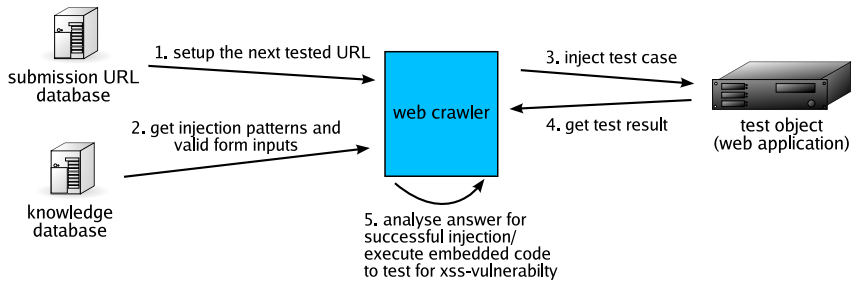


Figure 3.8: The WAVES web application evaluation process (the negative response extraction is not shown in the diagram)

The WAVES system is an interesting approach because it involves artificial intelligence methodology in the crawling and injection process e.g. a knowledge-based understanding of the form variable semantic. The estimation of the success of such a concept is difficulty because only few results have been propagated by the developers.

The researchers involved in the WAVES project also published papers about another project called WebSSARI (Web application Security via Static Analysis and Runtime Inspection) which should implement new approaches in white box testing.

WebScarab

WebScarab [WEBb], a project of the well-known Open Web Application Security Project (OWASP) is considered here as a representative for many other similar concepts and approaches which are widely available as open source tools.

As the name implies, WebScarab is a system for uncovering vulnerabilities especially in web applications. It was developed in Java and in difference to WAVES (see preceding subsection), WebScarab was mainly designed as a platform for several plug-ins for the creation and analysis of conversations. At this, a conversation in the context of WebScarab is a set of requests to a HTTP server and the corresponding responses. The framework itself mainly manages the conversation between a plug-in and the test target and the distribution of the conversations among the other plug-ins as shown in figure 3.9. Each plug-in can decide how to analyse the conversations received from other plug-ins through the framework.

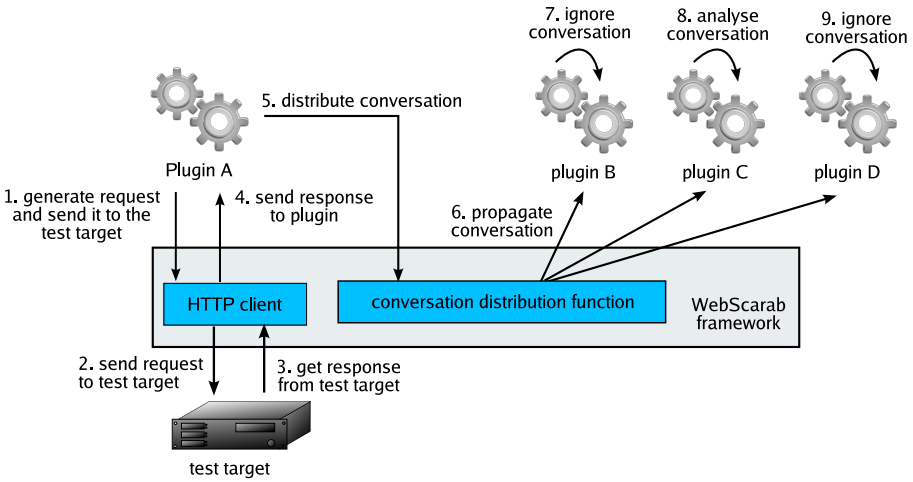


Figure 3.9: The WebScarab plug-in concept

WebScarab, as it is provided by OWASP, already contains different plug-ins which imply a specific use of the system. The most important tool is the proxy, which acts as a man-in-the-middle between a web browser and a web server. It intercepts requests which are sent by the browser and responses which are sent by the server to allow the manipulation and the analysis of these messages with one of the different other plug-ins.

As an automation approach also a fuzzer plug-in is included. It provides the automated generation of requests with fuzzed components in the HTML header or body. Thereby, the evaluation of the answer has to be done by the WebScarab user.

Other plug-ins allow a detailed examination of the conversations. To give some more examples, session identifier can be analysed for the quality of the randomisation of the used generator functions and HTML data can be explored for scripts and comments to find hidden information.

WebScarab is a popular toolkit, which is referred very often in connection with web application security.

Holodeck

Holodeck belongs to a class of security testing utilities which are called sandboxing tools which intercept and manipulate system calls of the evaluated system. The main idea is to wrap the target application on the entry points to the operating system. This is done by attaching a dynamic loadable library as a system call wrapper to the application. This technique is often also termed "system call hook". The behaviour of the wrapper library functions is parameterized by a control process which exchanges data with the hooked process via inter process communication channels (shared memory, pipes, signals etc.). The concept is also described in figure 3.10. Holodeck itself is designed for Microsoft Windows operating systems.

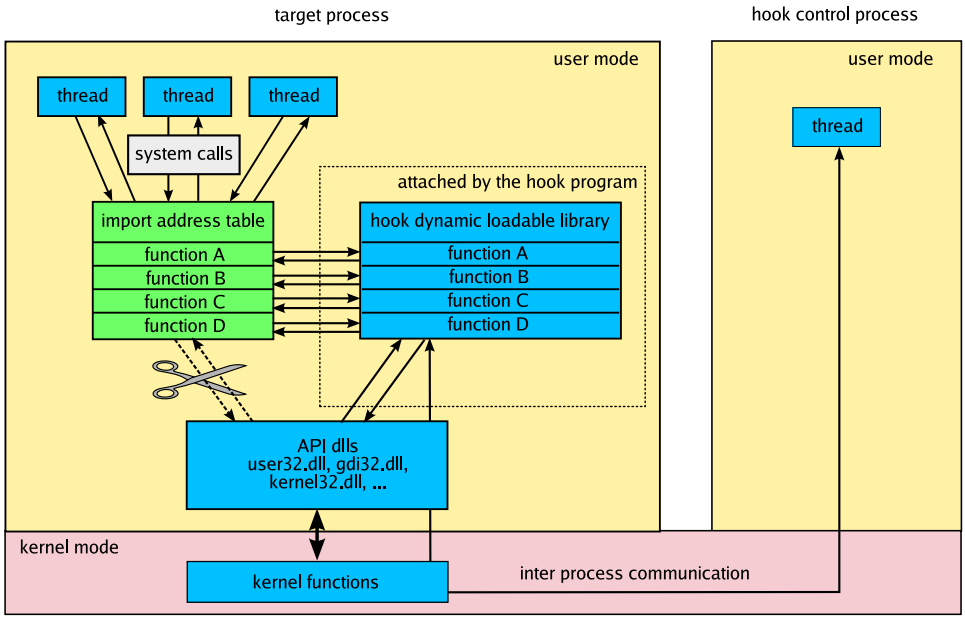


Figure 3.10: Concept of a system call hook based on the altering of the import address table (Windows NT)

Usually, Holodeck is used to test how the target application reacts on faulty data returned by a system call function. Faulty means in this context that the return value indicates that an error appeared during the system call function execution (e.g. faults while trying to access the memory or files, registry errors or network errors) or that data, which are returned by input functions

which has been read from data channels like sockets or files, were corrupted. Another type of tests allows to simulate limits for the access to several resources provided by the operating system like network and memory. This methodology is used to evaluate whether the test system is able to handle these errors properly or whether these manipulations are causing software errors.

Another application field for Holodeck is the monitoring of interactions between the tested system and the operating system while provoking a malicious application behaviour with other fault injection or testing tools. By using Holodeck it is possible to observe whether the test target invokes unusual or unwanted system calls to the operating system. In that case Holodeck can work as an error sensor (see also 5.6).

Holodeck is focused mainly on the technical aspects of intercepting and manipulating system calls. It does not include more sophisticated procedures for data analysis and error interpretation. It only provides the execution of implemented automated error generation scenarios.

Holodeck is mentioned here as a representative for many other similar tools based on different hook techniques and system platforms but with similar operational area and use.

4 Conditions, considerations and concepts for automated software security evaluation

In this chapter a general model of a software security evaluation process is outlined and the limits for a reasonable applicability of automation in this process model and the underlying criteria are discussed. The existing approaches from chapter 3 are assessed regarding their abilities to provide such an automated software security evaluation process.

As a result of this analysis a new model of a software security evaluation system is introduced which factors in the deficits of the existing approaches and the other results of the discussion of automation in software security evaluation.

4.1 Strategies and procedures for breaking software

For a further consideration of methods for security evaluation automation it is first necessary to examine the whole evaluation process and to describe the different steps which have to be absolved to gain applicable results which reflect the level of security of a test object.

In the next section, the objectives and potentials for software assistance and automation in these different steps are discussed based on the generic evaluation process model which is defined in the following.

Preliminary evaluation considerations

At the beginning of each software test, the common application scenario, the design and the main objectives of the evaluation have to be specified. A specialisation of the main security aspects already mentioned in 2.1 (authentication, confidentiality etc.) has to be formulated in security targets and protection profiles. Often these requirements can not be found in a sufficient form in the software specifications generated by the manufacturer. Thus, in difference to conventional reliability analysis the reflection of the purpose of the software, the purpose of the security related functions in the software and the main targets for potential security breaks is essential.

There exist common criteria for security evaluations of software, which can be used as a general standard to gain comparable test results. These criteria are derived from different national standards and are summarised in the "Common Criteria for Information Technology Security Evaluation" standardized by the ISO [ISO98].

Preliminary examinations of the test object

In the next step the test object itself is examined in a preliminary study. The different separable components and the interfaces and their protocols for the communication with the system environment have to be identified.

In the following, the interactions on the interfaces are observed while initiating common application scenarios. The data and control flow which has been monitored thereby is the base to create assumptions about the internal procedures of the test object and to determine used data entities.

In this stage of the evaluation (the passive analysis) it is already possible to find software vulnerabilities based on wrong interface usage like e.g. the weak encryption problem. Also failures which

can be provoked through conventional system use covered by the specifications can be identified. The other main objective is to find data entry points for the following active analysis.

Active analysis

Based on the observations from the preceding passive analysis the test object is analysed by creating data and control flow on the system interfaces which is usually outside its specifications. At this, a common method is to mutate the behaviour observed during the passive analysis. Of course, the number of possible producible test cases is almost infinite. Thus, a tester has to identify the interesting constellations. Clues for this identification can be:

- Which test cases fail in other similar software applications? What are the typical errors for selected algorithm or problem solutions?
- Which methods or data entities at the interface are supposed to functioning near security relevant functions in the software? Especially, which data entities are carrying parameters for these functions?
- Do there exist unusual solutions for problems which the software designer or programmer was confronted with?
- Are there possibilities to newly combine methods at the interface in a malicious manner which are normally uncritical in common application contexts and when considered separately?
- Are there fundamental software security concepts not respected?
- Are there methods or data entities appearing at the interface whose existence and function can not be explained in a logical way?
- Is the interface resistant against malformed input data?

The result of each test case has to be examined whether a security relevant error has been found or not. This sounds easily but it is not always. The failure can appear late and can be difficultly to notice by the tester. Often the tester can also not clearly identify the vulnerability but because of specific observations the possibility of the existence of security problems can be outlined.

Test results

The results of the analysis are passed to the software manufacturer in a test report. Often it includes recommendations how to solve the found errors.

The standard security evaluation process is also described in figure 4.1. Note that for reasons of simplification the test object in this figure is only pictured by a single computer system. Nevertheless it can be e.g. a whole distributed system as well.

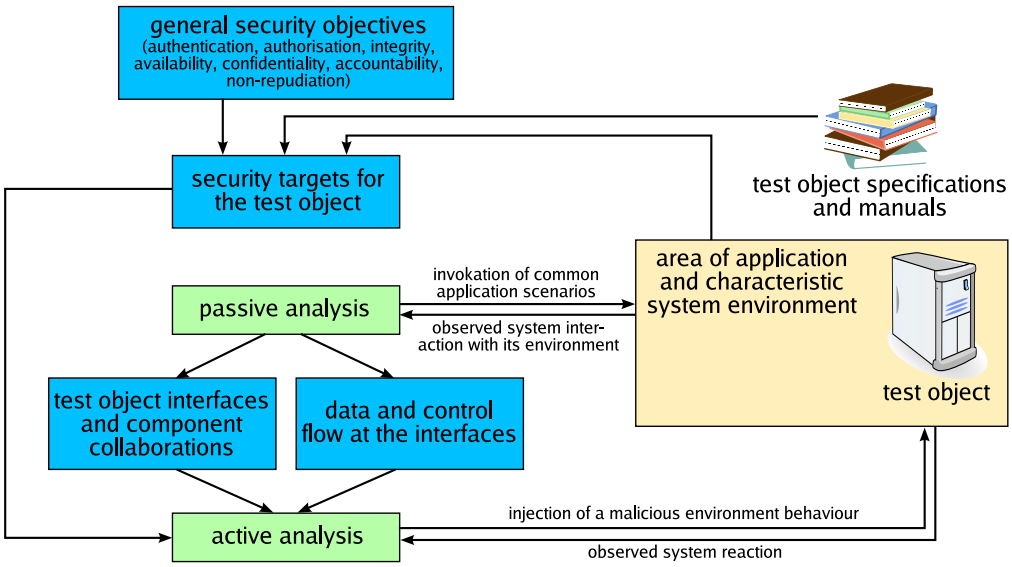


Figure 4.1: The software security evaluation process

4.2 The limits of automation of software security evaluations

Based on the evaluation process described in section 4.1 the requirements and approaches for automation in the different steps of this process model are discussed in the following.

The first step is mainly focused on the formulation of security targets for the test object based on the previously given information about the application scenario, the specifications and the general security requirements. An evaluation system (the fictitious system which performs the automated evaluation) needs these information in a computerized form to be able to derive these security targets. Beside the fact, that in some cases specifications are available described in a formal machine readable language the sufficient semantic understanding of this description is still an unsolved problem in many parts. Because of that, a further discussion of automated derivation of test requirements and criteria is not done here. In the following, the assumption is used that a human tester defines the specific security targets for the test object.

The more interesting discussion is about the possibilities for evaluation automation for the passive and active analysis phase. In the ideal case and strongly simplified the evaluation system must have the following capabilities:

- It must be able to capture data on the test object interfaces and to connect to these interfaces to invoke a data exchange with the test object. This capability is mainly a technical one.
- It must be able to comprehend the syntax and the semantic of the test object interfaces.
- It must be able to uncover passive detectable security flaws. This capability requires integration of a knowledge data base. During manual evaluations, this knowledge is provided by the experience of the human tester.
- It must be able to generate test cases and to interpret the test object reactions after initiating a test case. At this, “test case” does not only refer to a single data injection to the test object. Moreover, it can also be a complex use case scenario.

The “understanding” of the interfaces of the test object (the second required capability) can be specified more precisely by defining different levels of comprehension (each level includes the features of the previously defined):

1. The evaluation system is able to connect to the interfaces and to read and write data to them.
2. The evaluation system is able to apply a syntactical scheme to the interface data and to interpret received protocol data units by identifying the several data entities in the protocol messages on different protocol levels.
3. The evaluation system is able to identify the state of the control flow at the test object interfaces.
4. The evaluation system is able to comprehend the semantic of single interface messages independent from any other temporal succeeding or preceding messages.
5. The evaluation system is able to comprehend the semantic of a bulk of messages (a protocol session) at the interfaces and can automatically derive the internal state of the test object at any point in time.

The classification scheme and the data needed for each level are also described in figure 4.2. Note that each higher step requires the existence of the features of the previous level. In level 4 and 5 there can also be a feedback to the knowledge database symbolized by the double arrow.

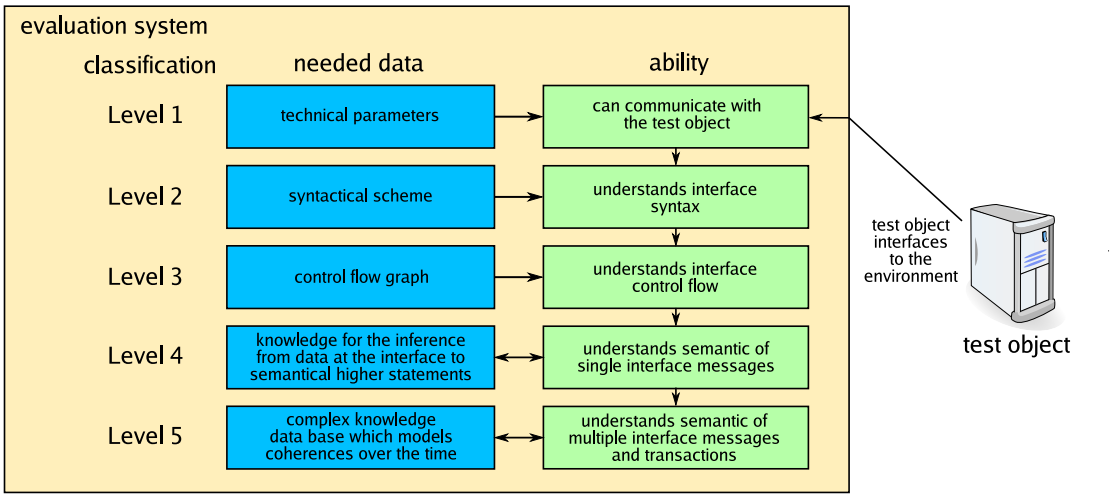


Figure 4.2: The different levels of test object interaction capabilities (the interface comprehension levels)

On the other side, similar levels can also be defined for the test case execution, which is in fact the data generation for the interface messages to the test object. In the following, these levels are also called “complexity levels of the test case execution”. They are defined as follows:

1. The evaluation system is technical able to send generated messages to the test object. The messages themselves must not comply to any criteria.
2. The evaluation system is able to send messages to the test object which are syntactical correct regarding the interface protocols. The test case generation algorithm typically changes data entities in the protocol units to prove the secure implementation of the data validation procedures of the test object.
3. The evaluation system is able to reach specific states in the interfaces control flow.

4. The evaluation system is able to generate a message exchange at the interfaces to realize specific target actions defined in a higher semantic. This ability also involves proper reactions on the test object responses.

5. The evaluation system is able to generate messages at the interface to execute whole use case scenarios which are defined in a higher semantic.

The complexity levels of the test case execution are also explained in figure 4.3. In this figure the test generator communicates with an injection mechanisms on a dedicated level which differs depending on the abilities of the evaluation system. The test case is formulated by the test case generator in an internal description language understandable by the injection routine.

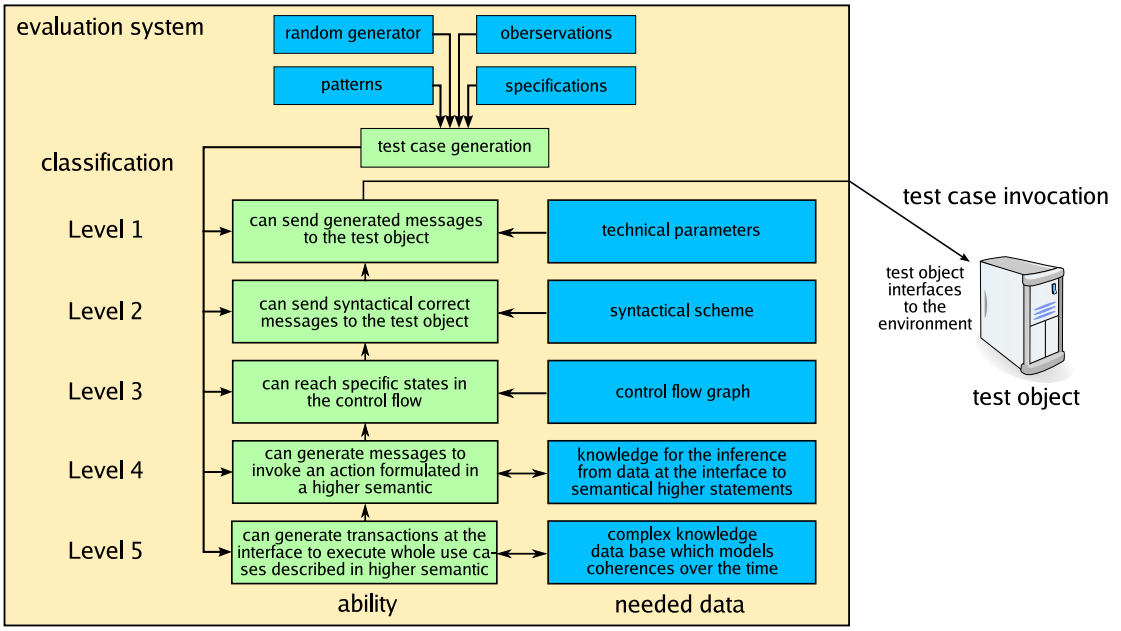


Figure 4.3: The different complexity levels of the test case execution (Note that the figure only shows that the evaluation system invokes the test cases at the test object and does not illustrate that there can be a consideration of the test object outputs for the test case creation)

The application of the different levels of interface comprehension and test case execution complexity is explained in the following example (as a simplification it is assumed, that the evaluation system has the same level of interface comprehension and test case execution complexity):

Imagine an evaluation system which should prove the secure implementation of a FTP service [PR85].

A level one evaluation system is able to connect to the service via TCP port 21 (standard port for FTP services). It can send protocol data units which can be e.g. produced by a random generator. It can also capture other protocol transactions if invoked by other systems and can replay them with random changes in the protocol data units. The syntactical correctness of these data units is not necessary.

A level two evaluation system is able to parse FTP messages for its command and to identify the command parameters. On the other side, it is also able to generate syntactical correct messages and to send them to the test object.

At level three the evaluation system can generate FTP messages regarding the actual state of the interface. For example, it knows that on a server reply code “331 User name okay, need password.”

it has normally to answer with a "PASS" command. It also knows e.g. how to reach the state to get the password input (for sending e.g. malicious password phrases) by first sending a "USER" command.

At level four the evaluation system recognizes the semantic meaning of FTP messages from the test object. E.g. it knows that a "532 Need account for storing files." reply means, that the currently logged in user is not privileged to store files in the current directory. It is also able to process actions with a description like "Prove whether you can send a file named 'password.txt' to the current directory" by generating the needed FTP commands.

At level five the evaluation system comprehends whole interactions with the FTP service. It is able to abstract the message flow to a meaning like "I'm currently logged in as user 'foo' and I tried to upload file 'password.txt' three times. The directories I have proved belong to the user 'root'.". It is also able to formulate the next step in a general way and on a semantic high level like "I know that this system is running an apache server. Because of that, I will try to upload files to directories normally belonging to the user 'apache'. If this doesn't work, I will try to break the password of the user 'apache'.".

Another point not considered yet, is the test case generation itself. At this, several approaches are mentioned here without a classification in different ability levels:

Random generator The test cases are generated by a random generator without any special strategy. It can randomize the contents of interface messages, their order or both.

Patterns The test cases are generated by involving previously entered patterns which are known to act malicious in similar application scenarios.

Observations The test cases are generated by manipulating previously captured transactions with the test object (e.g. information gained during the passive analysis phase).

Specifications The test cases are generated based on an inference of the previously given security targets with other available specifications.

These methods can also be used in combinations e.g. a conjunction of the specifications with observations derived from the passive analysis phase.

The lower the ability level of the evaluation software is the smaller is the set of types of software vulnerabilities which can be found automatically. This gap can only be filled out with the skills of a human tester who assists the software. Exemplarily some of the existing approaches from chapter 3 are classified in figure 4.4.

The system "fuzzed" is only able to connect to the test object and to send data produced by a random generator. It is not able to interpret any response data at the interfaces and the injected data flow does not belong to any syntax.

In contrast to "fuzzed", SPIKE[1] and mangleme are able to generate data regarding the syntax at the "attacked" interface (SPIKE needs a syntax description previously entered by a programmer) but they are also not able to interpret any received data or to follow the control flow at the interfaces.

The two tested commercial scanner WebInspect and AppScan are able to follow the control flow at the interfaces[2], but the injection mechanism does only syntactical changes in the dedicated server

[1] The consideration of SPIKE only refers to the scope of the framework. Of course, the programmer using this framework can add further data processing routines not considered here.

[2] Note that "interface" does not refer to the HTTP interface of the web server because the web server is not the test object. It refers to the interface of the examined web application.

complexity level of the test case execution	interface comprehension level				
	1	2	3	4	5
1	fuzzed				
2	SPIKE, mangeleme		WebInspect AppScan		
3			PROTOS		
4				WAVES	
5					

Figure 4.4: Classification of existing security evaluation systems

requests without a more complex examination of the interface behaviour and considerations of different interface states.

PROTOS allows to model a control flow and a data flow at the interfaces of the test object and to derive test cases from these specifications. It can follow the conversation at the interfaces and can retrieve the current position in the control flow graph.

WAVES includes several approaches for identifying the semantic meaning of the web pages of a web application. It tries to find HTML form fields where the user has to enter specific data. Thus it renders the visualisation of the web pages to gain this information. It also includes a self extending knowledge data base which provides data to generate test object inputs where some input parameters are filled out semantically correct and others contain malicious information.

(Note that these considerations are only referring to the base concepts of the regarded evaluation systems. They do not factor any practical implementations and their results. This should also not provide a general rating of the regarded software.)

Especially, creating an application which implements functionalities in interface comprehension and test case execution complexity level four or five is difficult because of the necessary involvement of knowledge representation and inference systems or other artificial intelligence techniques to map interface transactions to a higher semantic.

The commercial scanners WebInspect and AppScan try to “bypass this principle” by applying a huge data base of error patterns in manifold combinations with known and previously captured input sequences to the test objects. Unfortunately, without regarding the real capabilities to invoke software failures, already the evaluation of the system replies fails in the most cases, because of the insufficient semantic comprehension of the test object interfaces (see appendix A).

WAVES, which is the only evaluation system classifiable as level four, is restricted by its project scope to few security flaws (cross-site scripting and SQL injections). Nevertheless the effort to implement these functionalities is high and checkable results are not available for public.

Thus it is safe to say that only automation up to level three is practically accomplishable. This restricts the possible detectable vulnerabilities and increases the need of assistance by a human tester. The consequences of this claim for the design of an evaluation system which should automate as much steps as possible are discussed in the next section.

4.3 Software-aided security evaluation

As shown in the previous section, an automation of security evaluation is practically limited to the possibilities of getting access to the test object interfaces and analysis or creation of interface messages based on well defined and machine processible interface specifications. This base can be

extended by several other tools for further analysis or test case execution but the step of identifying semantic entities at the interfaces has to be done by a human tester who is only able to sufficiently associate the meaning of these entities and who has an understanding of the whole semantic of more complex interface transactions.

De facto, the human tester remains in the middle of the evaluation process and the evaluation system assists him as far as possible.

The model of such a system which respects all of these aspects is given in figure 4.5. The figure is a simplified illustration which only shows the general interactions between the involved actors. The more detailed depiction can be found in figure 4.6 which also involves the software structure discussed in the following.

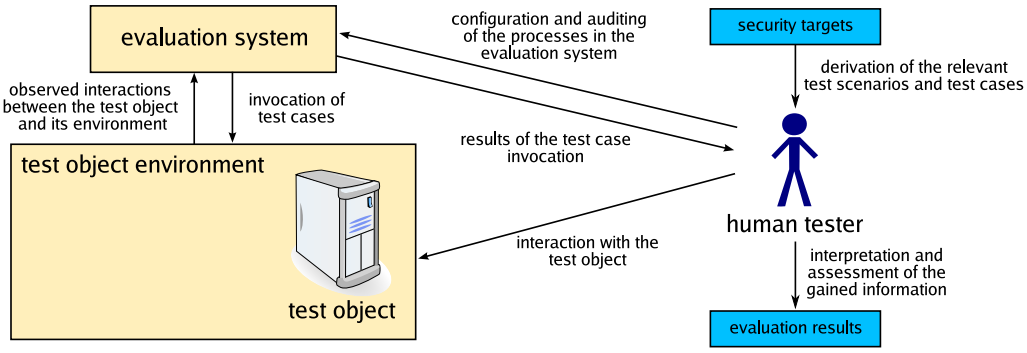


Figure 4.5: Proposition of software system for software security evaluation (Note that the picture is a simplification of figure 4.6.)

The message exchanges between the test object and its environment are captured by sensors at the communication channels. Note that in figure 4.6, because of simplification reasons, the test object is symbolized by a single computer system which is connected to other systems by a computer network. Actual, the sensors and actuators can also act between e.g. application and operating system.

The data of the different sensors are collected centralized in the evaluation system central control unit and are parsed involving knowledge about the technical nature of the captured interfaces (e.g. syntax, control flow). The resulting common internal data representation is now read and processed by different tools which are a part of the central control unit. The arrangement of these tools is done by the human tester who supervises the process. He has to decide which methods have to be applied to which data and he also selects the test cases. The tools also exchange data among each other (not shown in figure 4.6) if necessary.

The test cases again appear in the internal data representation and are compiled to a form suitable for the test object interfaces. The application of the test case data is done by actuators in the environment of the test object which emulate other systems which usually interact with the test object.

The human tester also receives output from the evaluation system about the interactions on the interfaces and especially the test object reactions on the test case data. These observations support his test case decisions and should be presented by using informative data views. The evaluation system can also provide tools with methods to unhide failures of the test object. The human tester has to validate these failure messages. At the end, the discovered vulnerabilities are registered in the test report.

In the proposed evaluation system automation can be applied on the tool execution level. The human user can model a set of test cases by creating an arrangement and parameterisation of a subset of the available tools. During the automatic evaluation phases it must be ensured that test

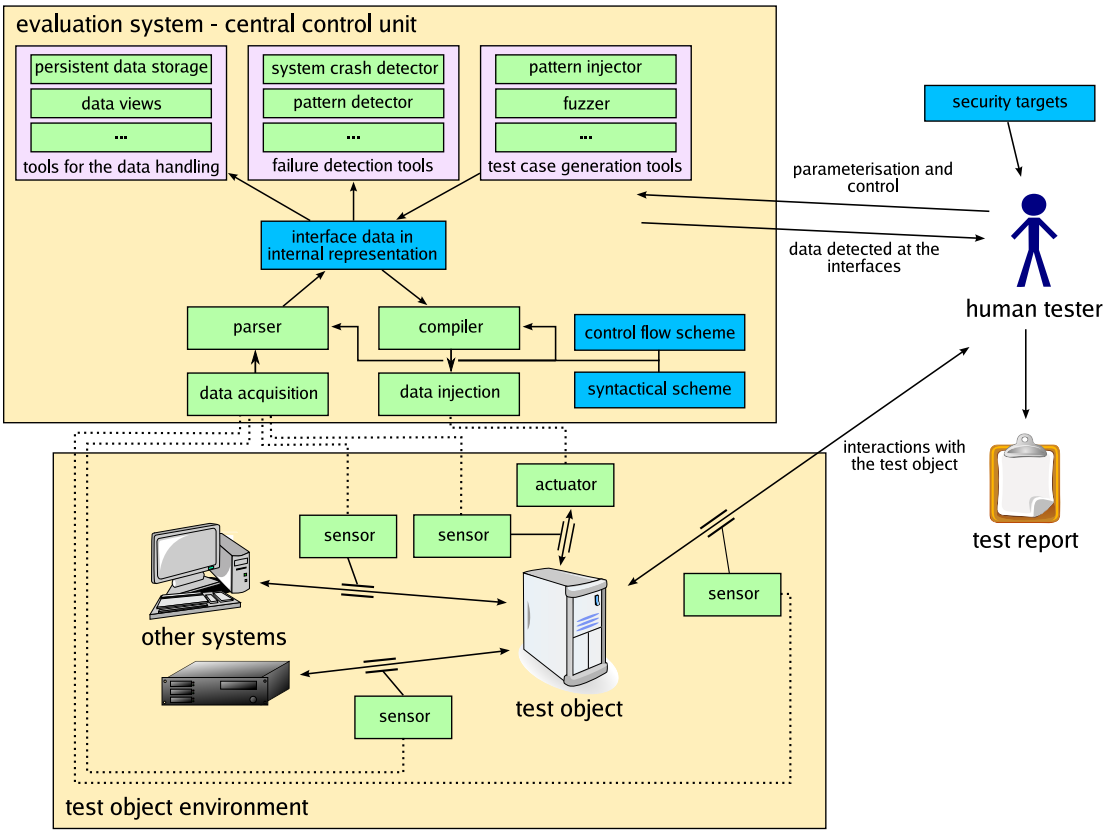


Figure 4.6: Detailed illustration of software system for software security evaluation (Note that the picture is an extension of figure 4.5.)

object failures are recognized and the related data are marked. The criteria for the interpretation of a test case result as potential vulnerability are also set up by the human tester.

The effectiveness of this model is based on the following assumptions about the special skills of the human tester and the assistance software as the two parties which perform the evaluation process:

- Human tester
  - The human tester is able to identify the data entities and states at the interfaces which are security relevant related to the security targets previously formulated for the test object. He can abstract to the internal state of the test object.
  - The human tester can understand the semantic of the data and states of the interfaces, especially this information which correlate with objects from the human vocabulary of concepts and when a detailed formal specification is not available, which is common for black box testing.
  - The human tester can easier identify the “interesting” test cases.
  - The human tester can make assumptions where no information is available.
- Assistance software
  - The assistance software is necessary to provide a helpful view of the actions at the test object interfaces to the human tester.
  - The assistance software can parse and compile a high amount of data at the interfaces easily when a machine readable specification is available.

- The assistance software can execute many test cases in a short time when the set of these test cases is previously specified by the human tester in a sufficient form.

5 Specification proposals for a software system for security testing

In chapter 4 only the general structure of a software security evaluation system is explained. Thereby, mostly the requirements on this system are formulated. This structure is refined in this chapter and the evaluation software is specified more detailed related to the needed data structures and the design of the involved components.

5.1 Preliminary considerations

As discussed in section 4.3 an software security evaluation process should involve a human tester in a useful way and should therefore combine the advantages of the human tester and an assistance software system.

Figure 4.6 also implies that the evaluation system includes components derived from different major functionality classes. These classes are the following:

- Sensors and actuators which are embedded in the environment of the test object and capture transmissions on the test object interfaces. The actuators inject the interface data of the test cases which are generated by components outside the test object environment.
- Data acquisition and injection components which are the link between the sensors or actuators and the security evaluation application. This application can be considered as a central control unit, central data collection and central data injection point for the network of the sensors and actuators.
- Parsers and compilers which transfer the interface data to and from their internal representation in the central control unit. This transfer is necessary to structure the data as much as possible by involving syntactical and interface control flow information. The subsequent processing mainly takes place on this structured data representation.
- Data flow control elements which coordinate the distribution of the structured data to and from the different tool components which are used to create data views, to store test cases and observations and to create new test cases.
- Test case generation utilities, to provide several methods for creating test case data.

5.2 Design of the sensors and actuators

The sensors and actuators are two classes of components which are not part of the central control unit itself and which are installed in the environment of the test object. This is similar to the use of devices with the same tasks in other engineering disciplines and makes sense because the superior system components should not be restricted by the requirements of this specific environment.

The sensors are used to capture data at the interfaces of the test object. Each captured data block has the following attributes:

- The direction of the data flow, which can be an input or an output to and from the test object

Usually the interface alternates between reading input and sending output.

- A timestamp, which marks the point in time when the data transaction happens
- An identification of the involved communication partners

  Usually but not necessarily one of them should be the test object.

- A unique identifier for the communication channel

  A communication channel is a resource of the test object as well as a resource of the other communication partners where data can only be written or be read sequentially. With other words, on one communication channel no more than one communication event can happen at once from the view of the involved communication partners. There exist many other characterisations of a communication channel, but this is the one important in this context.

When the object is working in an environment abstracted by an operating system, a clue for capturing interface data are the messages between the application and the operating system. Unfortunately this often involves a considerable implementation effort and extensive technical knowledge.

In the application area of distributed systems the interesting interface transactions are often only the communications over the computer network. Thereby it is usually easier to capture these data because of the easy extension of these networks for network traffic sensors. The simplest form of such a sensor is also shown in figure 5.1.

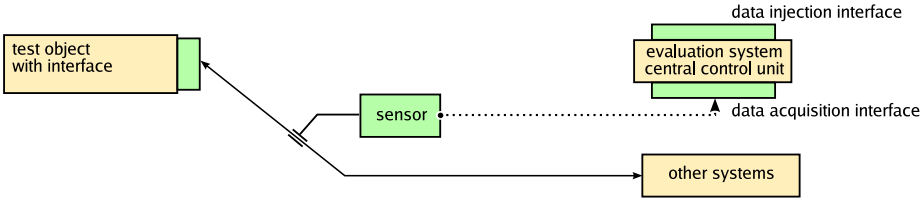


Figure 5.1: Eavesdropping sensor

Often a sensor is also working as a proxy, which means that the sensor provides an interface to other interacting systems where the messages are forwarded to the test object. From the view of the other systems this sensor appears as the test object. A graphical explanation can be found in figure 5.2

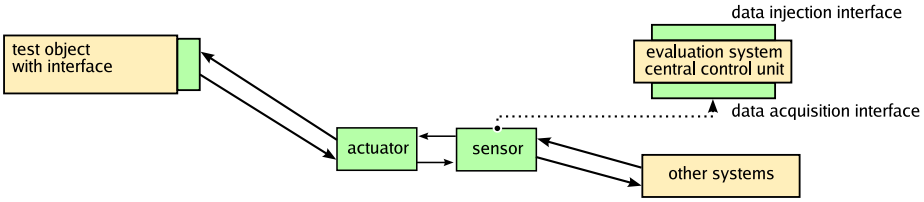


Figure 5.2: A proxy sensor

From the point of view of the test object, an actuator is a “normal” other system in its environment which uses the test object interfaces. From the view of systems which also communicate with the test object the actuator has normally no special meaning and is in fact invisible. The actuator is fed with the input data from the evaluation system central control unit which are forwarded to the test object. On the other side it can receive the answers of the test object and send them to the central control unit. In this function it is also used as a sensor (see also description in 5.3).

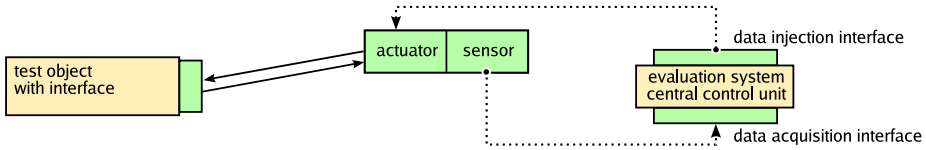


Figure 5.3: A typical actuator construct

Conceivable is also the combination of a sensor to intercept interface messages from other systems to the test object and an actuator to reinject these messages after they have been processed (and possibly manipulated) by the evaluation system central control unit, as shown in figure 5.4.

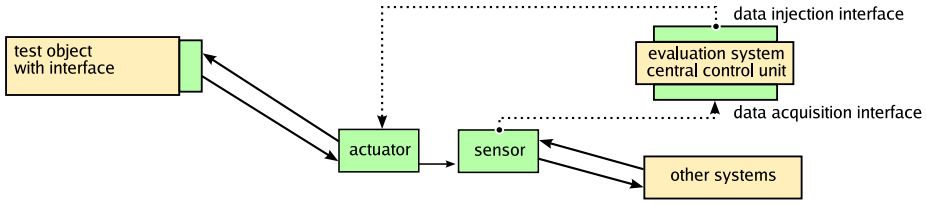


Figure 5.4: Sensor-actuator combination for message interception

A problematic issue is the generation of the timestamp to mark each interface transmission acquired by the sensors. For a correct functioning it is necessary to provide one single time base for all the sensors. Otherwise it is not possible to reconstruct the exact temporal order of the events which occur at the test object interfaces.

The time base is usually provided by a hardware clock whose time value is accessible through the operating system. Especially when the sensors are working on different machines, a precise synchronization is difficulty to apply (Note, that a huge bulk of transactions can be processed in a thousandth part of a second). Approaches for time synchronization are not discussed here but can be found e.g. in [Mil85].

5.3 Preprocessing of the interface data

Every sensor has a counterpart in the evaluation system central control unit to acquire the collected data. Together with the counterparts for the actuators, they form the interface of the central control unit to the test object. It makes sense that all sensors and all actuators use a uniform interface protocol for the communication with the corresponding mechanism in the evaluation system central control unit. This can reduce the implementation effort and can ensure an easy extensibility.

The main task of the first step in the processing of the input data is the transformation in an internal data representation which guarantee an adequate view of when, what and to whom data were transmitted on the test object interfaces. This is done by a classification of the data in a structure described in figure 5.5.

Hereby, a session is a collection of several communication channels connected to the test object. Each channel consists of a set of data transactions from and to the test object. For every data transmission the direction (inbound or outbound traffic related to the test object), the timestamp and the transmitted data are stored. Each connection includes several additional meta data which can be e.g. channel parameters. These are less important information for the further processing but helpful for the identification of the communication associations. It is assumed, that each complete data transmission is one semantic unit e.g. one protocol data unit or one operating system call (depending on the type of the sensor which has provided the information).

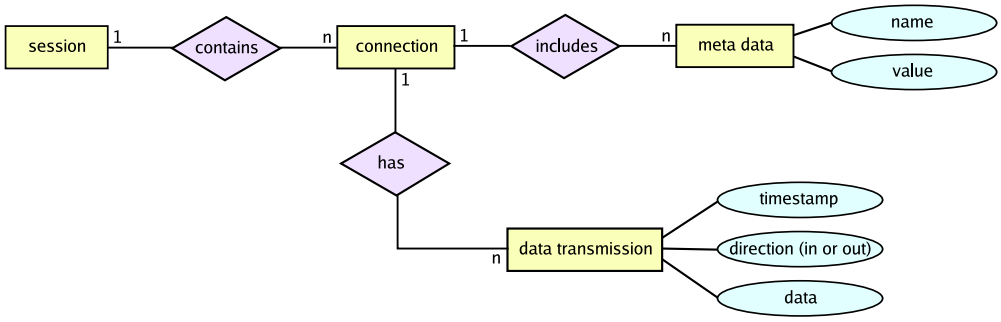


Figure 5.5: Entity-relationship diagram of the internal representation of the sensor data

In the next step, the syntax tree for each of these data transmissions is constructed sequentially. This requires a correct implementation of the test object interface protocols in the central control unit. A simplification can be done by using a universal syntax parser which can be fed with a syntactical description in a common language e.g. the Backus-Naur form.

To process the data transmissions (or also called interface messages) of a session sequentially a loop structure is introduced which allows to iterate over the data transmissions of all connections in the session which is the input for this loop. As mentioned above, it also includes the parsing algorithms which thus are applied to every message of the session.

In other words, one different message of the session is parsed and processed during each loop iteration. It makes sense to include other procedures in this structure which should be applied to single interface messages. The loop scheme is also described in figure 5.6.

Note that in this figure the case selection, which decides how to process each specific message or a group of messages with selected features can also be placed in other points of the control flow where the decision criterion is accessible.

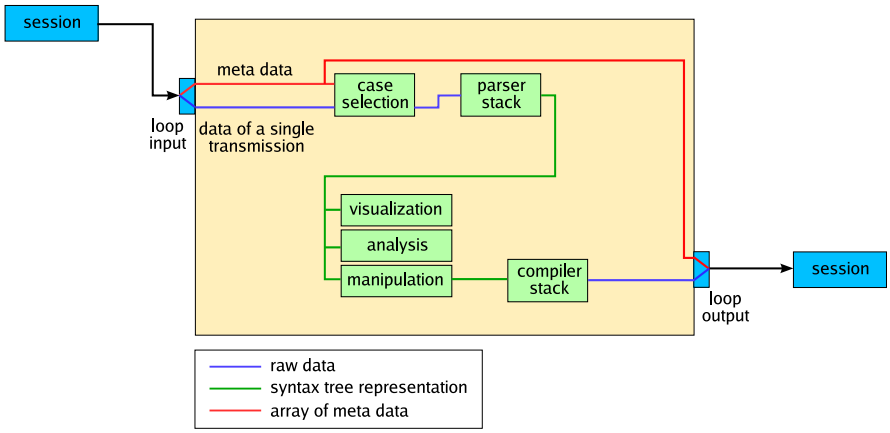


Figure 5.6: Loop architecture for sequential processing of data transmissions stored in a session

This loop structure provides possibilities for automation, because of the automated processing of a set of interface messages regarding a previously implemented scheme.

The central control unit should also allow the specific handling of single messages selected by the human tester who manages the evaluation process. It should provide capabilities to combine different decoding and encoding techniques and tools from other categories like visualisation, manipulation and analysis.

5.4 View of the interface data for the human tester

The central control unit should also provide methods to visualise processes at the test object interfaces in a form which is easy interpretable by the human tester. The data representations described in the previous chapter, the session model and the syntax tree already imply some requirements for these visualisation tools.

The user should be able to access the data of every data transmission of any connection at the different steps of the processing where the representation uses the session model. The view of data in the syntax tree representation should be easy to explore by the human tester which means to provide adequate methods to examine each branch and node of the tree.

Beside these straight forward concepts which are conceivable without any extensive considerations another approach for a helpful visualisation, the sequence diagram, should be introduced here. The sequence diagram is widely used, especially in the area of object-oriented programming to visualise the sequence of execution (the control flow) in collaborations of objects over the time. This concept is modified here, to show the sequence of message exchanges between the test object and other systems. In literature this type of the sequence diagram is often used to explain communication protocols, but it is not very common for system analysis software.

Figure 5.7 shows an example where a message from a system in the test object environment invokes a communication between the test object and a connected database server

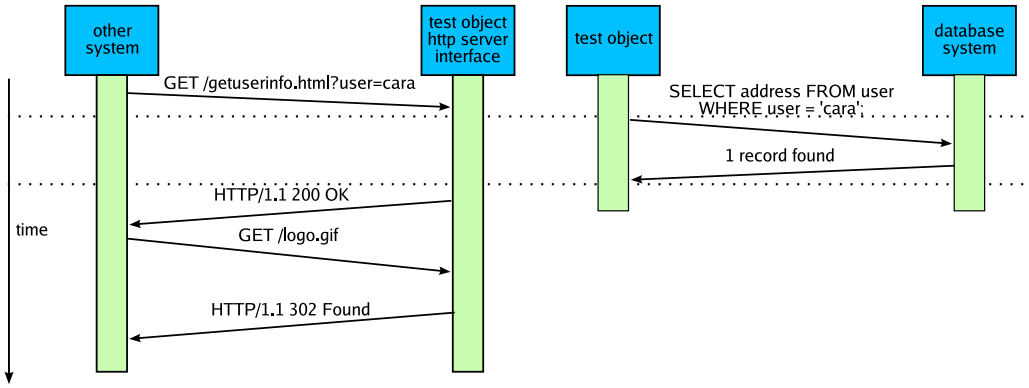


Figure 5.7: Example for a sequence diagram

The sequence diagram can visualise these coherences for the human tester in a useful manner. Furthermore, it is desirable that the human tester can select the message data shown in the diagram for each transmission by selecting nodes from the syntax trees of the parsed data unit.

This example also shows that a correct timestamp and the time synchronization of the sensors is important for creating a valid sequence diagram.

5.5 Test case creation

The previous sections consider the acquisition, internal representation, visualisation and the general steps of the processing of the interface data. This section discusses the generation of test cases which is a main task primarily prepared by the preceding steps.

In general there are two major possibilities how to create test cases for security tests:

- Test cases can be created by using specifications and documentations of the test object.

- Test cases can result from a mutation of a previously executed "normal" use case and the associated observations. This mutation can affect the data flow, the control flow or both in one test case.

At this, the main focus is on the second method. Nevertheless, the central control unit should also provide techniques to allow the human tester to design tests without involving any other input data acquired by the sensors.

For the use case mutation several methods are known which should be available in the evaluation system:

- Replacement of node values in the syntax tree of messages with random generated data, data derived from a pattern database or manually entered values
- Removal of selected messages from conversations
- Changing the order of messages
- Adding messages from other contexts or which have been newly generated
- Unmodified replay of data out of context

The targets for the mutation can be previously captured interface interactions which are reinjected as test cases or interactions between the test object and the other systems in the environment which are still running at the moment of manipulation. The second case requires an interception of messages in the environment. A sensor concept which allows this was already described in figure 5.4

5.6 Failure detection

Another aspect which is important especially to implement automation approaches is the failure detection. As already shown in 2.3, a failure is the mapping of an internal system error to the system environment. Usually an event (or also called an effect) appears in this environment which is not expected by the other involved entities.

As discussed in 4.1, it is not realisable with a reasonable effort that the evaluation system has applicable knowledge about security targets whose realisation has to be proved by the software test. Thus the focus is more on general failure detection in the sense of detecting abnormal program behaviour instead of trying to identify and classify specific security relevant failures (examples for such failure classes can be found in 2.4). The human tester which supervises the evaluation has to decide whether a discovered failure affects a claimed security target or not.

As a result of these considerations the evaluation system should be able to detect some typical events which lead to the assumption that a system failure appeared whereby the type or the exact specification of the failure is unimportant at this. Such typical events are:

- The test object spends an unexpected long time period on the processing of the previously entered test data. [1]
- The test object consumes an anomalous high amount of system resources e.g. memory, processor time, input/output operations or disk space.
- One or more of the test object processes terminates abnormal.
- A nominal-actual comparison of the test object output data identifies an unwanted gap. In the simplest case data appear unexpectedly in a specific form at interfaces.

[1] In systems which depend on the processing of data in other systems there often exists a timeout event which is triggered when the other systems do not provide the answer to input data in a time period of a specific length.

Some of these properties are not observable through the dedicated test object interfaces which are normally used for the test case injection. Thus it can be necessary to install sensors in the test object environment which monitor its "health" like e.g. the consumed system resources. An example for such an integration is shown in figure 5.8.

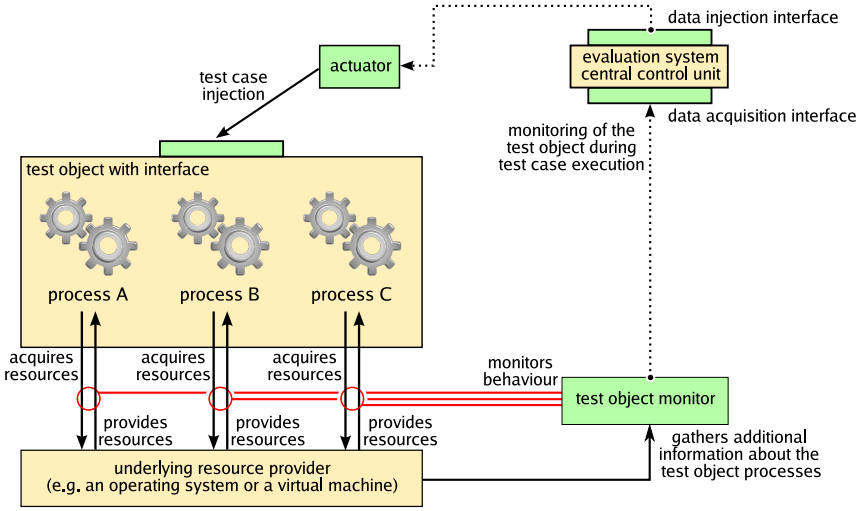


Figure 5.8: Proposal for an inclusion of a test object monitoring for failure detection

The difference between the monitor and the other sensors presented in 5.2 is that the data provided by the monitor are considered as dynamic features of the test object. The evaluation systems can prove whether these features are taking sensible values.

5.7 Component integration

As shown in the preceding sections, the evaluation system should consist of different functional units which should be combinable by the human tester. The number of possible combinations determines the size of the set of possible test cases which are producible but on the other side a good usability must be kept.

Thus, the first measure is the encapsulation of every function in a module with standardised module interfaces. Additional, the introduction of global data structure definitions is necessary to allow a data exchange between these modules.

The module interface elements are arranged in a directed graph. The nodes in this graph are the data inputs and outputs of the used modules. Each edge in the graph represents a data transfer connection between an output and an input. The head of the edge is always an output and the tail of the edge is always an input. Edges are only allowed between in- and outputs of the same data type. An output can be connected to multiple inputs by different edges. A module itself is a set of input and output nodes. When a module assigns data to one of its outputs these data are forwarded to the connected inputs.

All modules are stored in an ordered list. Each module has a routine which is called, when the data on the outputs of this module should be updated. This occurs in the order the modules are stored in the list. When the routine of the last module was called this process restarts with the first module.

An overview of the involved elements is also given in figure 5.9

Some of the modules are data sources which mean that they have no inputs, others are data drains

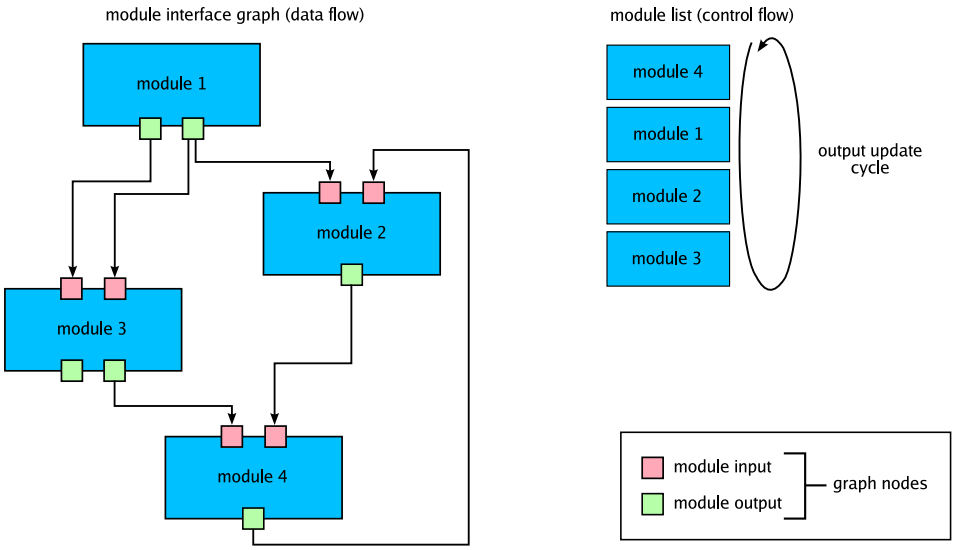


Figure 5.9: The graph of the modules inputs and outputs, the module list and the update loop

which mean that they have no outputs. A data source is normally a module connected to a sensor in the test object environment and a data drain is usually a module connected to an actuator. But also databases and data storages can be sources or drains.

The advantage of this concept is, that it is ensured that the update routine of a module is called again after each update routine of all other modules has been called once. This property helps to allow a kind of parallelizing of algorithm which is important here (e.g. interception and manipulation of messages of two interfaces at "the same time").

This advantage on the one side can be a disadvantage on the other side, because the construction of execution loops with a set of modules is more complicated. This has to be considered during the module design.

For providing better possibilities for test case generation and analysis, the system should also provide the ability to allow a dynamic change of the structure of the module processing (e.g. the module order) during the execution of the update loop. The human tester should be able to insert steps in the data processing anytime without reloading and restarting the whole module structure.

Another important aspect is the question of the description language the human tester should use to arrange the modules and to describe the data processing. One possibility is the use of scripts. The other approach supposed here is the direct creation and manipulation of the graph and module list by using a graphical user interface. This method exposed useful in other areas where the fast rearrangement of processing elements is important for testing different algorithm combinations. At this, one example from the area of image processing is given in [Mar05].

6 Implementation of a software system for security testing

To provide a proof-of-concept for the further considerations in this diploma thesis, the evaluation system which is discussed in its theoretical structure in chapter 4 and chapter 5 has been implemented prototypically.

In this context, the practical design of the proposed software security evaluation system is described in this chapter. Furthermore, the software implementation of the requirements and concepts claimed in the previous chapters is outlined.

6.1 System components

As described in chapter 5, the evaluation system is designed as a distributed system with a central control unit. This central control unit contains the graphical user interface where the human tester creates the testing algorithms. It also acts as the centre point for the data acquisition and the test case injection.

For a better portability the central control unit is written in the programming language Java. In contrast, the sensors and actuators which are the other nodes in the distributed system are written in the programming language C and C++. They have been designed platform-dependently for the Linux operating system. A platform-independent implementation of these components is not possible in this case because they use special functions provided by the operating system and due to other technical reasons.

Of course, because of the small functional complexity of these programs they are easily adaptable to other platforms and other methods of communication with the test object interfaces. In the test system each sensor or actuator is connected to the central control unit via a TCP/IP connection. Hereby, the sensors or actuators operate a TCP server where the central control unit connects to.

This concept takes two major implementation requirements into account. The first requirement is to ensure the platform independence wherever that is possible. The second requirement is the separation of the sensor and actuator components from the central control unit, to allow a “free positioning” of these elements in the test object environment. The other existing approaches presented in chapter 3 are always based on one single evaluation program which contains all the sensors, all the actuator and the test case creation and analysis.

An overview of a possible configuration of the whole evaluation system is given in figure 6.1.

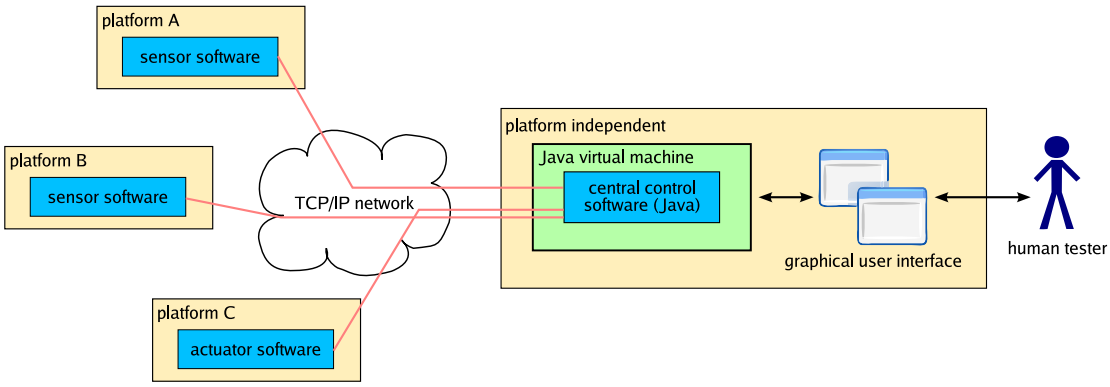


Figure 6.1: Example configuration of the evaluation system (Note that the test object is not shown but it is assumed that it is connected with the actuators and sensors)

6.2 Implementation of the sensors and actuators

For the studies of the evaluation system a set of sensors and actuators has been implemented. They are only usable to eavesdrop, intercept or inject TCP/IP connections but the concept is easily portable to perform this functionality on other types of communication channel and interfaces.

Each sensor works as a TCP server where the central control unit can only read data from. These data are directly captured at the TCP/IP interfaces of the test object or at the TCP/IP communication channels between the test object and other systems in the test object environment. They are encoded using the Base64 coding [Jos03], supplemented with meta information and embedded in an XML scheme. The major element in that scheme is the "packet" which represents one single message (and its meta information) which has been exchanged between the test object and objects in its environment. The XML data which are derived from this XML scheme are sent to the central control unit.

An example for such a XML data transmission to the central control unit during the detection of a connexion establishment to a HTTP server (the test object) is shown in figure 6.2.

On the technical side, the sensor uses the libpcap library which can be used to capture all data transmitted on the selected network interfaces. It is important that the data transmissions from and to the test object occur at this network interface.

Generally, the actuator is functioning in the same way, but the communication takes place in the opposite direction. The central control unit sends XML data to the actuator including "packets" with data encoded in Base64 and meta information. In this case, these "packets" are interface messages which have to be injected. The actuator initialises a TCP/IP connection to the test object and forwards the received data after decoding them from Base64. Response data from the test object are forwarded through a sensor server to whom the central control unit can be connected as well.

A more detailed overview of the used techniques and the correlations between the involved elements can be found in figure 6.3.

Another implemented method to evaluate test objects with a HTTP server interface is a HTTP intercepting proxy. A HTTP client in the test object environment connects to this proxy instead of connecting directly to the test object interface (the HTTP server). The HTTP request is then forwarded to the central control unit. In the following, the central control unit has to process the request and to manipulate its parameters in order to generate a specific test case. The request is sent back to the proxy which invokes the actual connection to the test object interface. The HTTP response created by the test object is forwarded to the HTTP client and can also be forwarded to the central control unit through a sensor server.

```
<packet>
        <client_ip>141.12.87.157</client_ip>
        <server_ip>193.108.95.40</server_ip>
        <client_port>42616</client_port>
        <server_port>80</server_port>
        <client_message>true</client_message>
        <message_size>0</message_size>
        <client_finnished>false</client_finnished>
        <server_finnished>false</server_finnished>
        <connection_id>0</connection_id>
        <timestamp>621219171</timestamp>
        <data></data>
</packet>
<packet>
        <client_ip>141.12.87.157</client_ip>
        <server_ip>193.108.95.40</server_ip>
                        ...
        <client_message>false</client_message>
                        ...
        <timestamp>621219175</timestamp>
                        ...
</packet>
<packet>
                        ...
        <message_size>429</message_size>
                        ...
        <timestamp>621219186</timestamp>
        <data>R0VUIC9uZXdzdGlja2VyLnJkZiBIVFRQLzEuMQ0KSG9zdDogd3d3LnRhZ2Vzc2NoYXUuZGUN
ClVzZXItQWdlbnQ6IE1vemlsbGEvNS4wICh YMTE7IFU7IExpbnV4IGk2ODY7IGRlOyBydjox
                        ...
        </data>
                        ...
</packet>
                        ...
```

Figure 6.2: Example of a sensor-to-control unit transmission during a connexion establishment at a test object interface (the three dots mark omissions made for better readability)

The concept of the HTTP intercepting proxy is illustrated in figure 6.4.

The sensor clients in the evaluation system central control unit transform the data in the internal data representation described in 5.3. The meta data are created from the additional information given in the XML data (see also 6.2). Note that the problem of the time synchronisation of the sensors and actuators is not solved yet and is not discussed in this diploma thesis (see also 5.1).

It is also expected that a message from a TCP client is always an input to the test object. The tag “client_message” in the XML data indicates this origin of the data packet. In other words, it is assumed that the test object interfaces are TCP servers. This simplification is only used here for the studies of the evaluation system. With an additional possibility for parameterisation at the sensor or the central control unit, client messages can be easily classified as test object output when necessary.

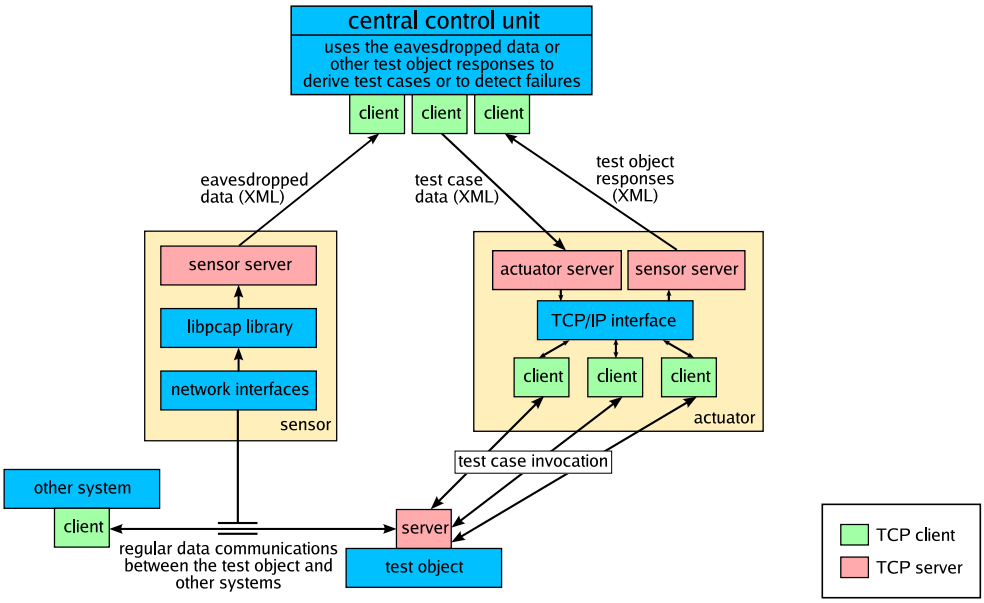


Figure 6.3: TCP/IP sensor and actuator implemented for testing the capabilities of the evaluation system

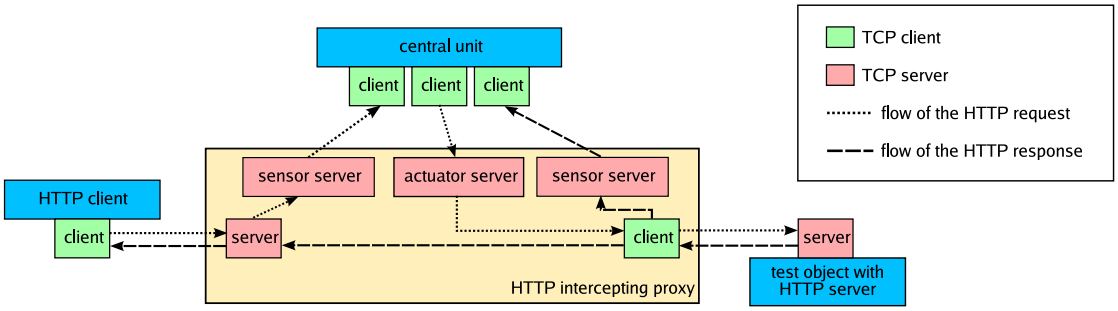


Figure 6.4: The concept of a HTTP intercepting proxy as it has been implemented for the evaluation system

6.3 Implementation of the basic structures of the central control unit

As described in the previous section, the data transmitted from the sensors to the central control unit and from the central control unit to the actuators are analysed or created in the central control unit which provides therefore the user interface for the human tester who designs and supervises these processes.

In section 5.7 a structure for the collaboration of so-called modules has been presented. These modules should provide the capability for implementing testing algorithms by a human tester in an efficient way. In a first step, the implementation of this structure is described here.

Figure 6.5 shows an UML diagram of the main elements of the central control unit implementation which realise the mentioned module structure (cp. figure 5.9). Note that the class description is incomplete because less important class attributes and methods have been removed for reasons of simplification.

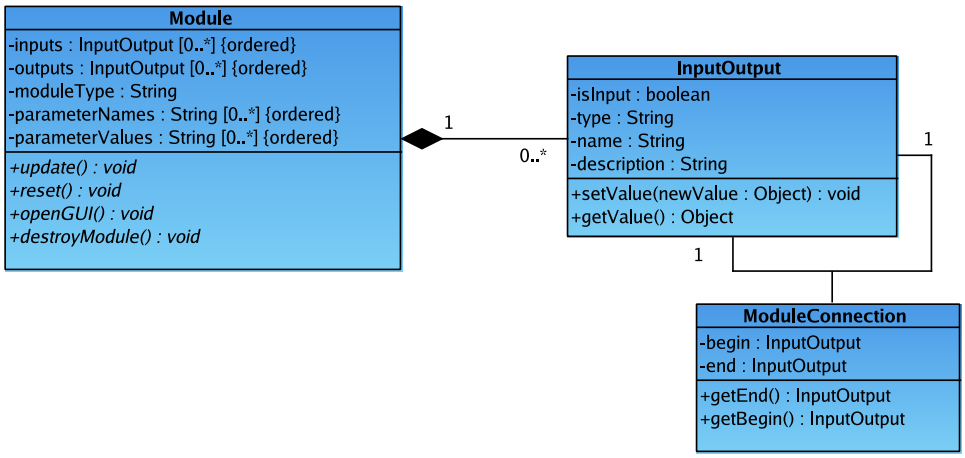


Figure 6.5: UML diagram of the main elements of the central control unit

The class “Module” represents the superior class for the derived module types which implement specific data processing functions for modelling testing algorithms. Each derived class consists typically of a characteristic number of inputs, outputs and parameters. The parameters describe the current configuration of the module and are necessary for the implementation of an persistent configuration storage. The class also includes a set of abstract methods which shall be overridden by derived classes. The method “openGui” triggers the instance of the class to initialise an individual provided graphical user interface to allow the human tester to set up the module. The method “update” is called during the loop execution of all modules (see also figure 5.9) to signalise that it is the modules turn to rewrite its output values by executing its data processing algorithm.

The following module type classes have been introduced for the further work in this diploma thesis:

- Data input and output modules, which are used to connect the central control unit to the sensors and actuators in the test object environment and to exchange data with them (see also 6.2)
- Viewer modules, which are used to provide the human tester information about the data at module outputs
- Parser and compiler modules, which are used to convert raw byte data to and from a more structured form concerning the syntax and semantic of these data
- Session management modules, which are used to handle sessions (a set of conversations at a test object interface captured by a sensor)
- Element operation modules, which are used to operate on the data structures created by the parsers (at this, the term “element” refers to node contents in parser trees)
- Boolean and trigger modules, which are used to activate and deactive modules in the testing algorithm and to handle the data flow

A more detailed overview of the modules can also be found in 6.5.

The module inputs and outputs are realised by a separate class “InputOutput”. Each instance of this class has a specific type of the data which can be exchanged through this interface. This type is stored in the attribute “type”. The attribute “isInput” signals whether an instance of the class is a module output or an input. A “ModuleConnection” creates an association between two instances of the class “InputOutput”. This link implies that the data appearing at the output (begin of the link) have to be forwarded to the input (end of the link). By using multiple instances

of “ModuleConnection”, one output can be connected to multiple inputs but one input can always get data from only one single output.

For the implementation of the graphical user interface the graphic library “G” [GLI] is used. This library has the advantage, that by using a G class as superior class for the classes shown in figure 6.5, these classes can be extended for a graphical representation on a G canvas. The G canvas can be inserted to the application user interface allowing the user to see an illustration of the relationship between the modules and their interface links.

The implementation of the class structure for the sensor data (see also 5.3) is described in figure 6.6.

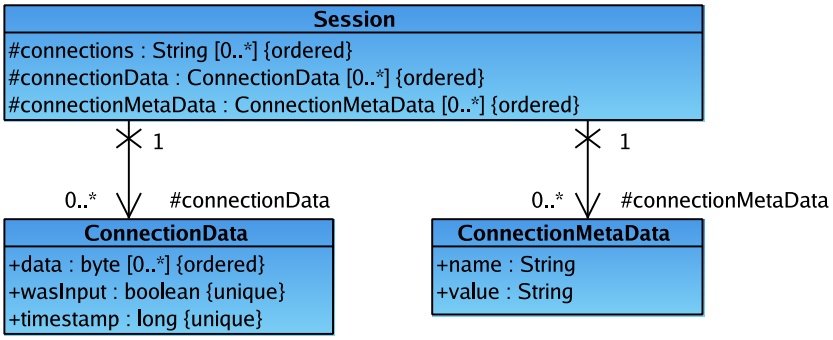


Figure 6.6: UML diagram of the classes for storing the sensor data

The main class “Session” stores the transmitted data of multiple communication associations in one context (that is the reason why it is named a “Session”). Each of these associations can have an identifier stored in the attribute “connections”. The attributes “connectionData” stores multiple instances of the class “ConnectionData” in a 2-dimensional list which is not described in the diagram. The first dimension is the list of the connections and the second is the list of the data transmissions at this connection. The structure of the “connectionMetaData” attribute is similar.

6.4 Graphical user interface of the central control unit

The graphical user interface of the central control unit consists of three main parts: the testing algorithm diagram, the execution control and the graphical user interfaces of the single modules.

Using the testing algorithm diagram the human tester can design module collaborations to process sensor data, to analyse communications or to create test cases for an injecting using an actuator. A description of the symbol of a single module is given in figure 6.7 a).

Figure 6.7 b) shows such a diagram. The “Net sensor” is a module which is connected to a sensor server and the “Net injector” is connected to an actuator server (see also 6.2). Both modules are processing data on the network interfaces of a test object. The “Session Viewer” can be used to visualize the captured data, which have been transformed in the “Session” structure (see also 6.3 or 5.3). The algorithm implemented here forwards captured data for a reinjection by an actuator in the test object environment. In fact this algorithm is a kind of copying and reinvocation of communications without a manipulation of the data.

The colours of the links, the inputs and the outputs of the modules show the type of the forwarded data. The numbering on each module represents the position in the list of all modules und consequential their execution order (or also called update order).

The control of the execution loop by the human tester is done using a special panel below the diagram which is shown in figure 6.8. Using the toggle button “Go” it is possible to start and

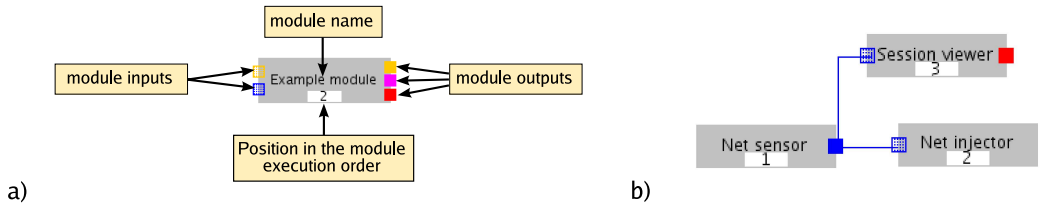


Figure 6.7: Example for a) a single module and b) a module collaboration

stop the execution. The execution can be suspended by activating the "Hold" button. The "loop delay" is a latency time inserted after each completely finished loop execution. By activating "Slow Down" the execution switches in a debugging mode where the data flow is illustrated in the testing algorithm diagram to help the tester to identify the current algorithm behaviour.

Figure 6.8: Execution loop control panel

Each module in the testing algorithm diagram can also have its own graphical user interface for parameterisation. It can be shown and hidden when necessary.

A complete overview of the graphical user interface is given in figure 6.9.

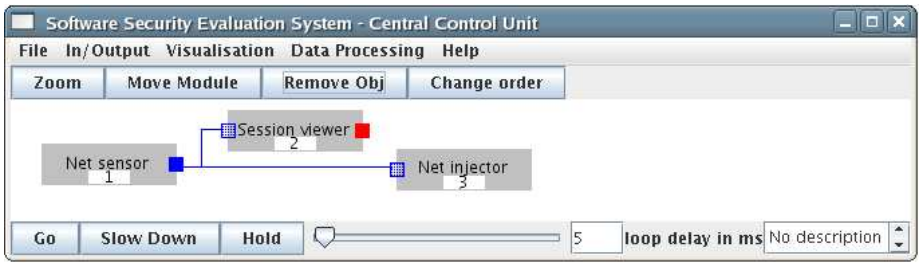


Figure 6.9: Overview of the whole graphical user interface of the central control unit

The menu bar at the top of the window provides some general application control functions like saving and loading of the workspace and a categorisation of the available modules in several sub menus.

All in all, the graphical user interface consists of only few main elements. Nevertheless the different modules can provide additional complex graphical user interfaces not regarded here. This design takes the claim for an easy extensibility into account.

6.5 Description of the implementation of selected modules

In this section, the implementation of different modules for the central control unit is described. This description is split in the several functional classes where the modules are categorized in. The presented modules are only examples implemented for the proof-of-concept done in the scope of this diploma thesis. Nevertheless each of these module classes should outline functions which are deemed necessary to be implemented to design testing algorithm and the automation of them.

Each module has several inputs and outputs (also called pins) shown in the testing algorithm diagram. Each pin has a specific colour representing the type of the data provided or accepted

at this pin. A classification of these data types is given in Appendix C. The modules shown in the diagrams of this section always have their inputs on their left side (symbolized by a non-solid rectangle) and their outputs on their right side (symbolized by a solid rectangle).

Data input and output modules

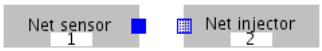


Figure 6.10: Data input and output modules

The data input and output modules designed for this diploma thesis are only for the data acquisition and injection on TCP/IP interfaces. As also described in 6.2, they are a kind of front-end for the sensors and actuators in the environment of the test object. Their function is to provide a link between the data representation in the central control unit and the data representation at the interfaces of the autonomic sensor and actuator programs.

Thus, these modules have only one pin. The sensor has an output pin for the captured data and the actuator (injector) has an input pin for the test case data (also shown in the figure 6.10). Each module has its own graphical user interface where to set up the hostname and the port for the sensor or actuator server (see also figure 6.3).

Viewer

The viewer modules are tools which should help the human tester to get an overview of the processes at the test object interfaces and during the test algorithm execution. As also shown in table 6.1, they are usually specialized for one data type as input. The data viewing itself is provided by the graphical user interface of each module.

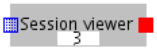
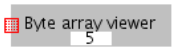
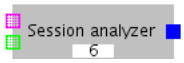
Name and symbol	Input(s)	Output(s)	Purpose
Session viewer 	session which content should be shown	content of a selected interface message	shows the conversations of the entered session with the transfered messages and their content
Byte array viewer 	byte array which content should be shown		shows the content of a byte array
Data explorer 	data abstraction which content should be shown	content of a selected tree node	shows the tree structure of a data abstraction input and the content of selected tree nodes
Session analyzer 	data abstraction of a interface message and the corresponding meta data	session created from selected messages	creates a sequence diagram from all entered messages and their meta data

Table 6.1: The viewer modules

The simplest module regarding its function is the “Byte array viewer” which only shows the data of a byte array without any other special options or functions.

The “Session viewer” is used to show the connections, metadata and interface data of a session. The graphical user interface is also shown in figure 6.11. The picture shows an example session with two connections. The first connection is selected and therefore the meta data are shown in the meta data list box. The third box on the first row of boxes lists the data transmissions. An “I”

shows that a transmission was an input to the test object and an "O" shows that a transmission was an output of the test object. The text box in the bottom of the window shows the content of the selected transmission. In that example the first listed data exchange (a welcome message from the FTP server) has been sent from the test object to the other system.

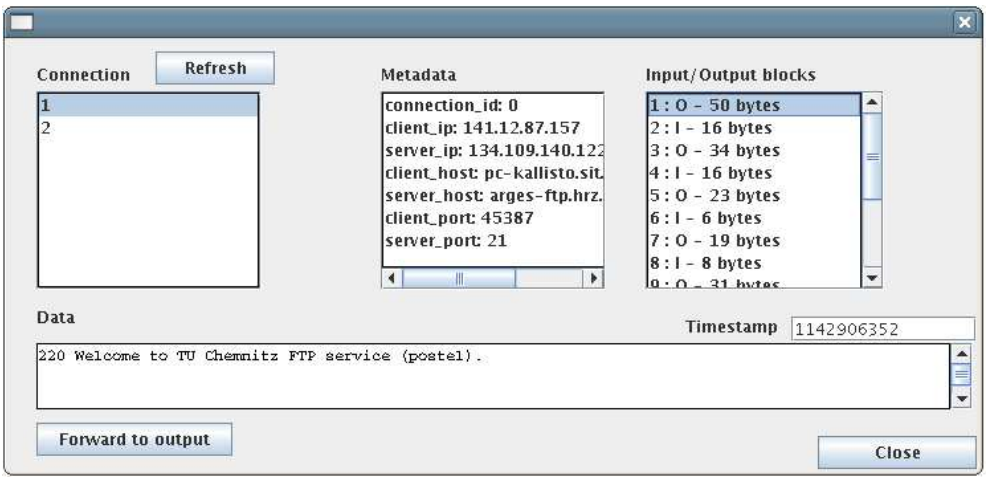


Figure 6.11: Graphical user interface of the "Session viewer"

Another important viewer is the "Data explorer". In difference to the "Session viewer", the "Data explorer" only accepts structured data of one data transmission as an instance of the "data abstraction" type. The viewer visualises the parser tree as shown in figure 6.12. The picture shows the parser tree for a data packet which has been sent to a HTTP server.

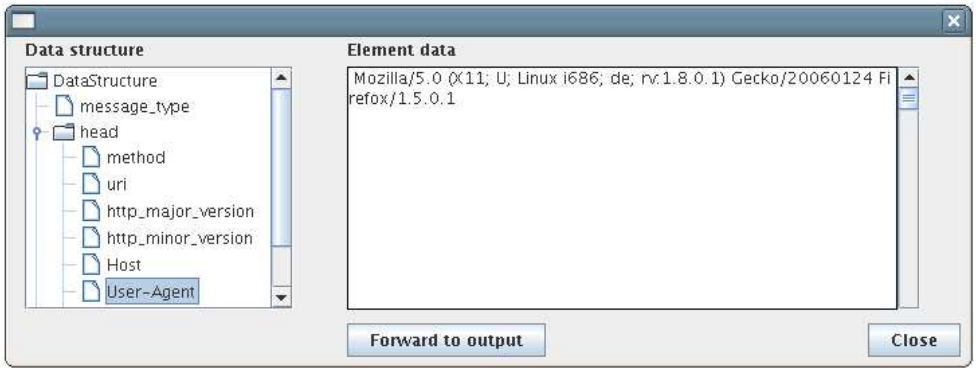


Figure 6.12: Graphical user interface of the "Data explorer"

The human tester can use the viewer to explore the tree and to see the contents of the leafs.

Another useful method to show the processes on the test object interfaces is the sequence diagram which is provided by the module "Session analyzer". Like the "Data explorer" it also accepts a data packet in the structured "data abstraction" type. Each packet is stored and added to the shown sequence diagram. Additionally, the meta data for each packet are required, because they include the timestamp which is important to show the chronological sequence of the interface transmissions.

An example for such a sequence diagram provided by the "Session analyzer" module is given in figure 6.13. The figure shows a FTP login process and the reading of a directory. Note that the directory information is transmitted by using a separate communication channel shown at the end of the whole sequence. The timestamp is shown on the left side of the diagram. The red

horizontal lines symbolise test object outputs and the green horizontal lines symbolise test object inputs. The vertical bars are representing the processes involved in the communications. The red bars depict processes of the test object, the green bars depict processes of the other system. The content of each transmission is shown above the corresponding horizontal line.

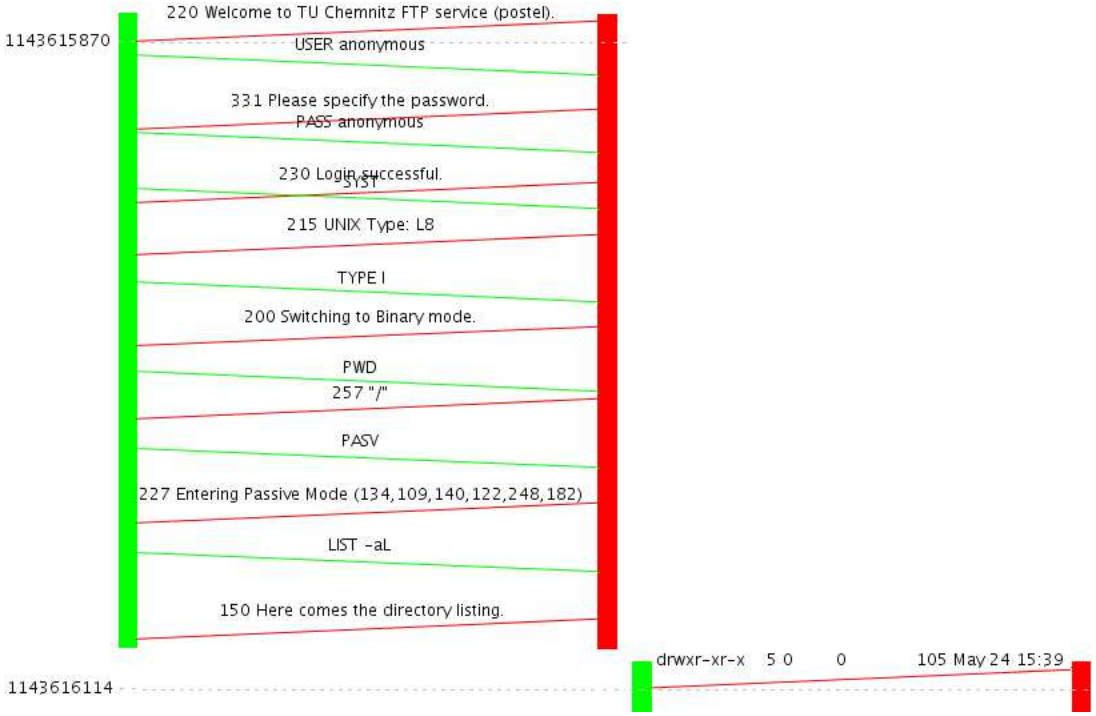
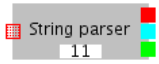
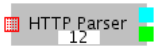


Figure 6.13: An example for a sequence diagram produced by the "Session analyzer" module

Parsers and compilers

Parsers are used to structure the raw data of data transmissions by taking into account available data protocols. They are used to transform byte arrays or parts of a parser tree into a data representation of the type "data abstraction".

The parsers which have been implemented for this diploma thesis are shown in figure 6.2.

Name and symbol	Input(s)	Output(s)	Purpose
String parser 	byte array which contains the message to be parsed	used grammar as a byte array output, the state of the last parsing process and the generated data abstraction	parses the incoming byte array which is interpreted as a string by applying an entered grammar
HTTP parser 	byte array which contains the message to be parsed	state of the last parsing process and the generated data abstraction	parses an interface message using the HTTP protocol specifications

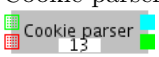
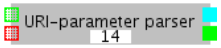
Cookie parser 	byte array which contains the message to be parsed or a data abstraction generated by the HTTP parser	state of the last parsing process and the generated data abstraction	parses the cookie information in the incoming data abstraction or the byte array interface message
URI-parameter parser 	byte array which contains the message to be parsed or a data abstraction generated by the HTTP parser	state of the last parsing process and the generated data abstraction	parses the parameter information transfered by a HTTP message

Table 6.2: The parser modules

There exist three parsers which can be used to handle messages of the HTTP protocol. These modules are helpful to evaluate web applications.

The "String parser" is an approach for a generic parser. "Generic" means, that the human tester can enter specific parser rules which are interpreted and applied to the raw data. At this, the Backus-Naur form[1]is used as the language for creating these rules. It is a context-free grammar where each production rule is a mapping of a string of non-terminal and terminal symbols to one non-terminal symbol. Hereby, a terminal symbol is a constant value (constant string). There exists one non-terminal symbol in each formal grammar which is the starting non-terminal symbol.

An example for a grammar for describing the syntax of FTP messages is shown in figure 6.14

```
*answer=status_code SPACES answer_description [NEWLINE]
*status_code=NUMBER NUMBER NUMBER
*answer_description=somestring

*commandline=command SPACES command_option [NEWLINE]
*command=CAPITALLETTERS
*command_option=somestring

somestring=VISIBLECHAR [SPACES] [somestring]

start=answer|commandline
```

Figure 6.14: A grammar for parsing FTP messages

The example grammar processed by the module has some modifications in comparison with the original Backus-Naur form. Some non-terminal symbols are marked with an asterisk. This means that the value of these symbols should appear in the parser tree which is produced by the "String parser" module.

The starting non-terminal symbol is per definition `start`. The non-terminal symbols which are written upper case are predefined.

The opponent module for the "String parser" is the "String compiler". It is fed with the same grammar like the parser through a direct link from the parser module. It is shown together with the other static compiler modules in table 6.3.

[1]Further information about formal languages and grammars can also be found in [HMRU00].

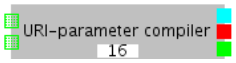
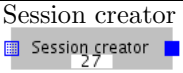
Name and symbol	Input(s)	Output(s)	Purpose
HTTP Compiler 	data abstraction containing a parser tree of a HTTP message	state of the last compilation process and the compiled interface message	compiles an interface message from a given HTTP parser tree
URI-parameter compiler 	data abstraction of the HTTP message where the parameters derived from and the data abstraction with a parser tree of the transmitted parameters	state of the last compilation process, data abstraction with recompiled HTTP parser tree and the parameters as byte array	compiles parameters parsed from a HTTP message data abstraction and reinsert them into the HTTP parser tree or generates a byte array interface message
String compiler 	grammar for the corresponding parsing process as byte array, the original parsed message as byte array and the data abstraction with the parser tree	state of the last compilation process and the compiled interface message	recompiles an interface message previously parsed with the "String parser"

Table 6.3: The compiler modules

These modules create raw byte data or merge two parser trees by compiling the data in one tree to data sequences which are added as content to leaf nodes in the other tree.

Session management modules

The implemented set of session management modules is shown in table 6.4.

Name and symbol	Input(s)	Output(s)	Purpose
Session splitter 	session which messages should be forwarded individually and a trigger to provoke each iteration	interface message and the corresponding meta data of the currently selected data transmission in the incoming session, the trigger is updated when the forwarding restarts with the first message	splits a session in their single interface messages
Session joiner 	interface message and the corresponding meta data	reassembled session	reassembles a session from incoming interface messages and the corresponding meta data
Session creator 	session to be manipulated	derived session	creates a session by using elements of the entered session

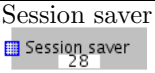
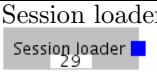
Session saver 	session to be saved		saves session information to a persistent memory
Session loader 		session which has been loaded	loads session information from a persistent memory

Table 6.4: The session management modules

The session management modules “Session splitter” and “Session joiner” are used to split sessions in their single interface transmissions and to recreate them with a step by step adding of single messages to a whole session again (joining of the single transmissions). This separation and reintegration is necessary because each message has to be transformed individually for test case generation.

The module called “Session creator” allows the user to create a new session by a rearrangement of messages of another session. The graphical user interface of this module is shown in figure 6.15.

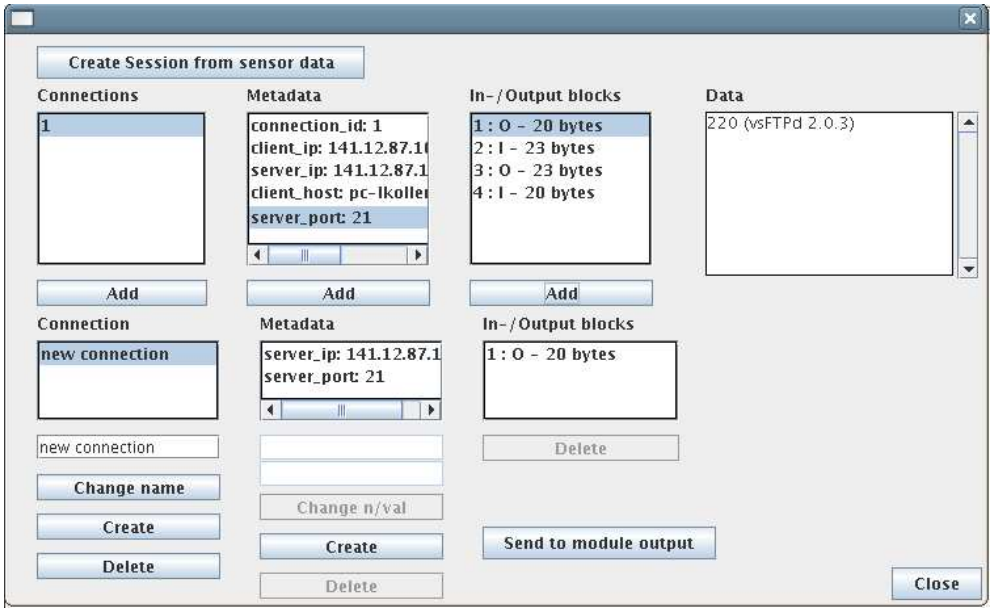


Figure 6.15: Graphical user interface of the “Session creator” module

The session data of the existing session (the input of the module) are listed in the upper list boxes. They can be added individually to the data of the new session represented by the lower list boxes. The meta data parameter values can also be changed by using the text fields below the meta data list box. The “Data” text box shows the content of the currently selected transmission for control reasons.

The “Session saver” and “Session loader” modules are used to save and load session data to and from a persistent memory e.g. a hard disk. These features allow an analysis or replay of older sessions.

Element operation modules

This class contains modules which provide operations on elements. Elements in this context are leaf nodes of a parser tree or parameter values embedded in the meta data.

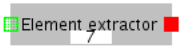
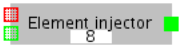
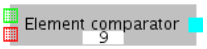
Name and symbol	Input(s)	Output(s)	Purpose
Element extractor 	data abstraction where the leaf node data should be extracted from	byte array with the extracted leaf node content	extracts the content of a selected leaf node of a data abstraction parser tree
Element injector 	data abstraction where the data should be injected into and the byte array which should be injected	data abstraction with the injection	injects the value of the incoming byte array as content into a selected leaf node
Element comparator 	data abstraction which should be examined and a comparison value as a byte array	result of the comparison as boolean value	compares the content of a leaf node of a data abstraction with the entered byte array content using adjustable criteria
Metadata injector 	meta data where a data pattern should be injected into and the byte array which should be injected	meta data with the injection	injects the value of the incoming byte array as content into a selected record of the meta data

Table 6.5: The element operation modules

The “Element extractor” converts an element from a parser tree to a byte array. This byte array can be manipulated or analysed in other modules e.g. another parser.

The “Element injector” is used to replace the value of a leaf node of a parser tree with another value which is provided by a byte array input to the module. This value can be e.g. generated by a random generator or can be taken from a pattern data base. This module is common to use for creating test cases by modifying existing sessions.

The evaluation of values in parser tree leaf nodes can be done by the module “Element comparator”. It creates a boolean value depending on the result of the evaluation. This value can be used e.g. to control the test case invocation.

Changes on the connection parameters can be applied by the “Meta data injector”. Its functioning is similar to the “Element injector”.

Boolean and trigger modules

The boolean and trigger modules provide different help functions to control the data processing. An overview of the set of modules is given in table 6.6.

Name and symbol	Input(s)	Output(s)	Purpose
Boolean sign 	boolean value which should be shown		shows the state of the input
Boolean negator 	boolean value which should be negated	negation of the boolean input value	inverts the boolean input value
Boolean-to-trigger converter 	boolean value which should be proved whether it is “true”	generated trigger	creates a trigger when the boolean input value is “true”

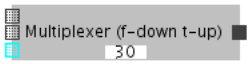
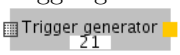
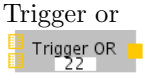
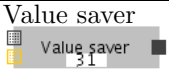
Multiplexer 	two generic input values which can become output and the boolean value to control the switch	one of the input values	switches between the two input values as a function of the boolean input value
Trigger generator 	input value of a generic type which causes the trigger generation when updated	generated trigger	generates a trigger when its input is updated
Trigger or 	two trigger values which can cause the creation of a new trigger	generated trigger	generates a trigger when it receives a trigger on at least one of the both inputs
Value saver 	input value of a generic type and the trigger which causes the storage	stored value	stores the value of its generic input when it receives a trigger

Table 6.6: The element operation modules

The first module, the “Boolean sign”, is helpful for the indication of boolean values at the outputs of other modules. It appears red when its input value is “false” and green when the input value is “true”. It can be used to monitor the evaluation process because modules are often designed with a boolean output that reflects the state of the last operation performed.

The “NEG” module is a negator. The module “B->T” (boolean-to-trigger) creates a trigger when its input pin receives a new boolean “true” value. The “Multiplexer” is used to forward one of the two inputs to the single output pin of the module depending on the value of the third input pin of the type “boolean”.

The “Trigger generator” creates a trigger when the value at its input pin is renewed. The value of the new input is not considered. The “trigger or” fires a new trigger when it receives a trigger on one of its inputs.

The “Value saver” stores a value when a trigger at its input appears. The value can be recalled by using the output of the module.

The “Multiplexer”, the “Trigger generator” and the “Value saver” have inputs and outputs which are coloured grey. That symbolizes that the data type of these pins is unspecified at the beginning and is assigned when the first link with another pin is created.

6.6 Additional annotations

As already discussed in section 5.7 the considered system architecture of the central control unit also implies several specialities and also problems.

One special feature is, that all modules are executed sequential in a loop whereby the order of execution is set up explicitly (shown by the numbers at the module symbols). Only modules which have not got new input data since the last loop run are overridden. Apart from that it is impossible that modules are starving during the algorithm execution.

This special feature implies new requirements for creating loops to process data of multi-element data types like e.g. the “Session” type which contains information about several test object interface transactions. The elements of these sets often need to be processed one by one in a similar manner. In other programming languages this is done by “for-to“ or ”while” loops.

In the presented system this kind of loops can not be created explicitly. Instead there are modules available like e.g. the “Session splitter” which split up multi-element data types and forward one part in each module loop run. When all parts have been forwarded to the output pins of the “Session splitter”, this module stops the processing which means that it does nothing when its update method is called. This causes modules which depend one these data to stop their own execution respectively they are overridden in the module execution loop because their own inputs are not updated.

Beside the higher implementation effort, the advantage of this methodology is that the multi-element data can be complemented during the iteration. Also after e.g. the “Session splitter” has stopped because all data have been forwarded it can restart automatically after its input data have been complemented.

Another advantage is the easy creation of infinite loops without an exclusion of other algorithm parts from the execution. The “Session splitter” can be set up to restart the data forwarding with the first element after the last element has been assigned to its output pins.

7 Practical examination of the software security evaluation system

In this chapter several solutions for security evaluations with testing algorithms implemented in the previously described security evaluation system are presented.

Because of the limited scale of this diploma thesis these evaluations only consider test objects with TCP/IP network interfaces. These test objects are mainly applications operated on the layer five (session layer), six (presentation layer) or seven (application layer) of the OSI reference model. The test objects examined in this chapter are fictitious, except the cross-site scripting test object.

7.1 Modelling of standard security threats

In this section testing algorithms for the invocation of standard security attacks are considered. These algorithms are the framework for the more specific attacks in the following sections and should familiarize with basic concepts necessary to implement such software security testing algorithm.

Simple replay attack

This algorithm is the most uncomplex one. It is based on a capture of data at a network interface and the replay of these data or a subset by an actuator in the test object environment. The result of this replay is a new conversation which is captured by a sensor at the communication interface of the actuator (the associated actuator concept is also explained in figure 5.3).

The implementation as a testing algorithm diagram is shown in figure 7.1

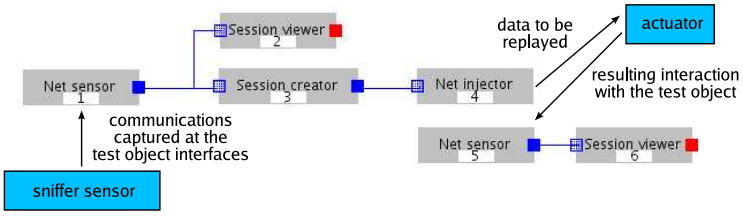


Figure 7.1: Simple replay attack - testing algorithm with additional symbols for the sensor and the actuator

The “Net sensor 1” receives the captured data and sends them to the “Session viewer 2” and “Session creator 3”. “Session creator 3” can be used by a human tester to select the messages which should be replayed. It can also be removed from the algorithm and as a result the data can be directly forwarded from “Net sensor 1” to “Net injector 4”. “Net injector 4” sends the data to the actuator tool in the test object environment.

“Net sensor 5” receives the whole conversation with the test object inputs and outputs which results from the replay and sends them to the “Session viewer 6”. The human tester can use “Session viewer 6” to examine transmitted and received data.

Replay attack with modification

An extension of the replay attack is the insertion of a loop over all single transmission in the session (compare with figure 5.6). A testing algorithm which realises that is shown in figure 7.2.

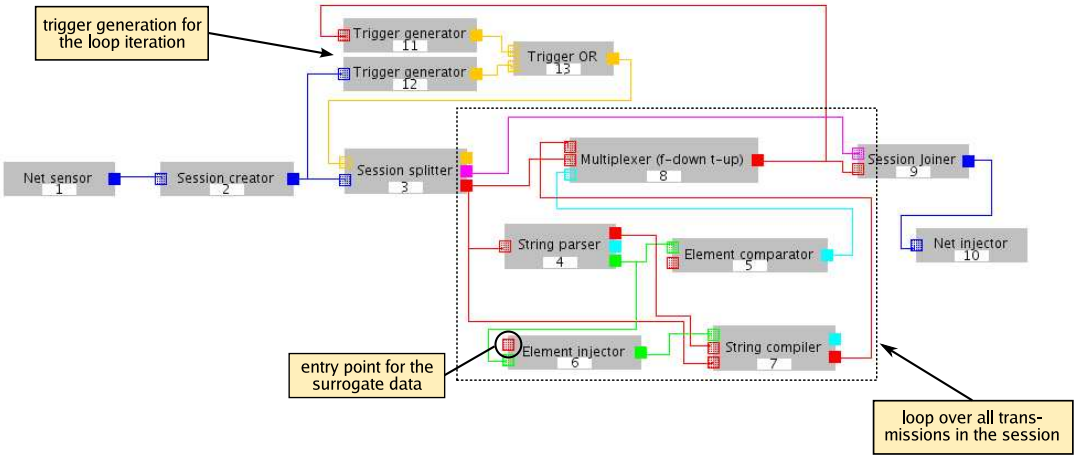


Figure 7.2: Replay attack with data modification - testing algorithm

For that insertion, the algorithm proposed in figure 7.1 has been split at the connection point between the "Session creator" and the "Net injector". As parser and compiler, two generic modules for processing strings have been chosen exemplarily. The newly inserted loop allows the classification of each single interface message by the "Element comparator" whether it should be reinserted with modifications or unmodified. To perform this task it is controlling the "Multiplexer" which switches between the modified and the unmodified data.

The trigger which is generated by the "Trigger generator" modules in the top of the diagram is necessary to start each new iteration of the loop. It acts as a kind of feedback which creates triggers as long as the "Session splitter" or the "Session creator" output messages. When no trigger appears, the "Session splitter" does not assign the data of a new message from the session to its outputs.

The data which should be used as surrogates to replace others in the parser trees of the single messages can be inserted via the "Element injector". In the diagram in figure 7.2 the corresponding input pin which receives these data is not connected but could be fed with any other imaginable byte pattern.

The "Session splitter" can be configured to continuously forward messages to the loop. This is achieved by infinitely repeating the conversations of the input session and makes it possible to create a large set of test cases ad hoc. The variation among these test cases is hereby achieved by the manipulation modules inside the loop which should change their parameterisation after each loop run. This method can provide a kind of automatic test case creation.

In the testing algorithm diagram in 7.2 there exist no modules for the analysis of the resulting communications between the actuator and the test object. It is possible to insert another combination of a "Net sensor" and a "Session viewer" which allow a human tester to examine the results of the test case injection.

Man-in-the-middle attacks

A so-called man-in-the-middle attack can be simulated by using a sensor-actuator combination for message interception (explained in section 5.2 and figure 5.4). The testing algorithm structure is similar to that used for a replay attack with modification. The difference is, that the "Net sensor"

and the “Net injector” are connected with this special sensor-actuator combination instead of processing eavesdropped data and to reinject them as new connections.

At this, the test case invocation is done by another system in the test object environment which communicates with the message interceptor. Usually the sequence of the messages is not changed and only the message data are manipulated before being forwarded to the test object immediately.

Stress test

A stress test algorithm can be used to evaluate the behaviour of the test object while it is fed with a large amount of other input data. The main target is to observe whether the high data processing load can cause unexpected effects like race condition flaws or operating resources leakages.

An algorithm which allows to perform such a stress test is shown in figure 7.3.

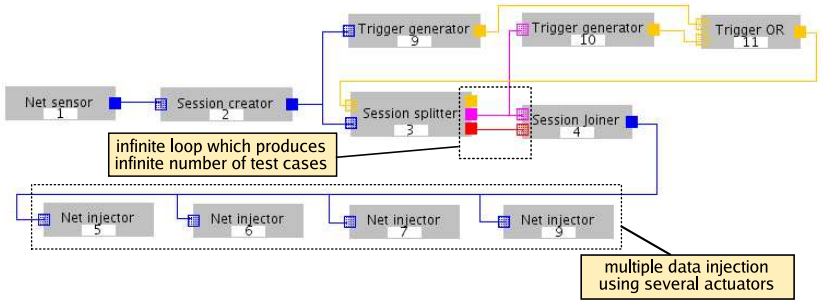


Figure 7.3: Stress test algorithm

The loop which is created with the “Session splitter” module and the “Session joiner” module is set up to be infinite which means that it infinitely replicates the communications in the session. The data are forwarded to the set of the “Net injector” modules at the bottom side of the diagram where each module is connected to an individual actuator. This allows a multiple injection of the same data from different actuators and therefore additionally increases the data flow to the test object. This method also allows to generate a so-called distributed denial of service attack against the test object.

The algorithm in figure 7.3 can be extended for data manipulations inside the infinite loop (see also figure 7.2). An algorithm which tests the proper behaviour of the test object under the stress conditions is not shown in figure 7.3 but can be added as a derivation from one of the other algorithm templates presented in this chapter.

7.2 Recognition of test object failures

The preceding section lists implementation patterns testing algorithm in the software security evaluation system to generate malicious test cases. The methods which can be applied to identify software failures are also another important aspect which is discussed in this section.

Unfortunately this problem is often excluded in the existing approaches and the successful recognition of failures often relies on the human evaluator who performs the evaluation.

Detection of erroneous output data of the test object

One way to identify software failures is by analyzing the test object output for data patterns which indicate that an error appeared during execution. This could be e.g. an error message content or malformed components of the data.

When data patterns are known it is possible to detect the failures. An algorithm which exemplarily implements such detection is shown in figure 7.4.

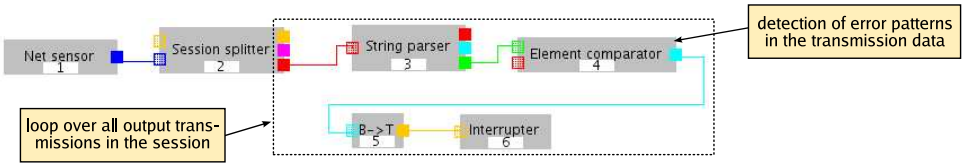


Figure 7.4: Algorithm for the detection of error patterns in messages

The "Net sensor" captures messages on the test object interfaces and bundles them in a session. This session is splitted again in their single messages by the "Session splitter" module which is configured only to forward test object outputs (this configuration is not visible in figure 7.4). The "Element comparator" examines parts of the messages whether they contain the previously entered error pattern. In such a case the algorithm is interrupted.

Note that the algorithm presented in figure 7.4 is usually combined with a test case injection. This test case injection will be interrupted as well when the pattern detector triggers. Then the human tester has to examine the data flow which has caused the test object failure.

Timeout at the test object interfaces

Another clue that something unexpected has taken place during the test case execution at the test object is a large delay of the output data at the test object interfaces. The observation of such behaviour is often used to indicate broken conversations between communication partners. The system which waits for receiving data usually reacts on the delay after a certain time which is called timeout. Such a reaction could be e.g. the resetting of the communication channel or a repetition of the last message.

In the security evaluation system, a timeout can also be used to suspect test object failures. An algorithm for detecting timeouts is presented in figure 7.5.

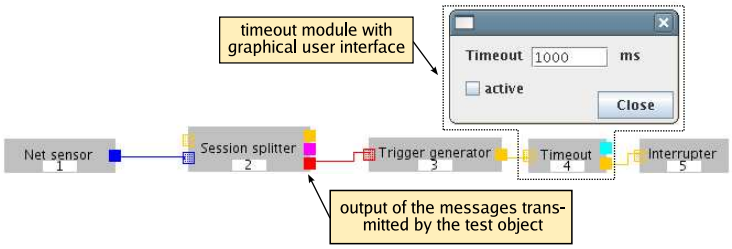


Figure 7.5: Timeout detection algorithm with the user interface of the timeout detector module

The "Session splitter" forwards the outputs detected at the test object interfaces which has been collected by the "Net sensor". Each appearance of any test object output resets the up-counting timer of the timeout module to zero. When the timer reaches the set up timeout, the module generates a trigger event which stops the testing algorithm process by activating the interrupter module.

To prove the appearance of a specific message it is possible to extend the algorithm with additional filters between the "Session splitter" and the "Trigger generator". This can be e.g. an "Element comparator" module in combination with an upstream parser module.

7.3 General security testing algorithm

The basic algorithm structures shown in the two previous chapters can be used to implement test procedures proposed in the approaches in chapter 3. By adapting these procedures, the capabilities of the presented evaluation system can be outlined. In the most cases the suggestion will be extended to demonstrate additional features.

Simple fuzzer

The simplest software testing proposed in chapter 3 is the fuzzing of interfaces by sending randomised data without any structure to them (see also 3.1). An implementation of such a method as a testing algorithm for the evaluation system is shown in figure 7.6.

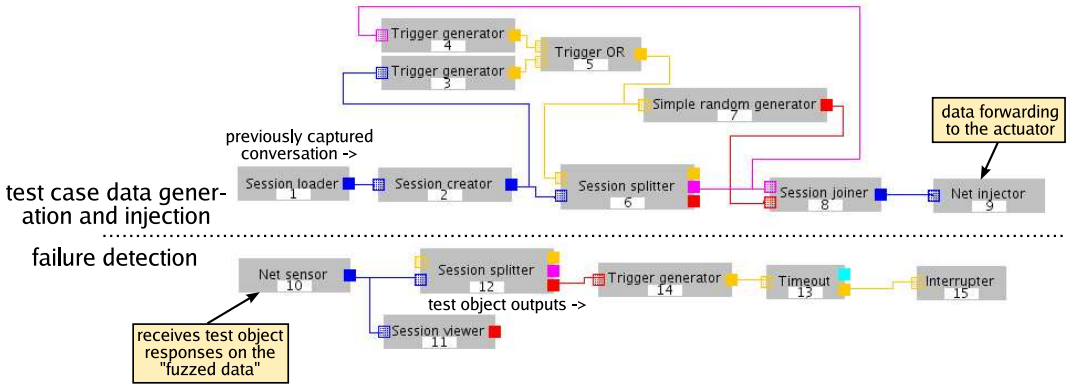


Figure 7.6: A simple fuzzer created with the evaluation system

The algorithm is separable in two parts: the test case injection and the failure detection. In this example, it is a combination of a replay attack with modification and a timeout error detection. Furthermore it is assumed that there already exists a captured conversation with the test object which is used as a base for the test case data.

This conversation data are loaded by the “Session loader 1” module and forwarded to the “Session creator 2”. By using the “Session creator”, the human tester selects the communications which should be modified in the following. The “Session splitter 6” extracts the meta data of each message in the created session. The message data are not forwarded. They are replaced by randomized data from the “Simple random generator 7”. De facto only the control flow but not the data flow of the existing conversation is used.

The result is a conversation session with the test object which is similar in structure to the session created by the human tester with the help of the “Session creator” and which is deriving from the session previously captured. Beside the structure, the data of this session have been completely generated by the “Simple random generator”.

This session is invoked as a test case by the “Net injector 9” and its connected actuator. The test object usually replies to the injected data. These outputs are captured by the actuator and are loaded into the testing algorithm through the “Net sensor 10”. In the following, timeout detection is used to stop the execution when the test object does not reply anymore.

Alternatively also the use of a failure recognition based on pattern detection is possible when such patterns are known for the test object. It is also thinkable to create an infinite test case generation loop by repeating the session data.

Protocol fuzzer

The simple fuzzer can be extended to mutate selected parts in the data flow on the test object interface regarding a communication protocol. This approach is also implemented e.g. by the network protocol fuzzers described in 3.1 and the fuzzing frameworks presented in 3.2. The simple fuzzer from figure 7.6 is therefore extended amongst others for a parser and a compiler as shown in figure 7.7.

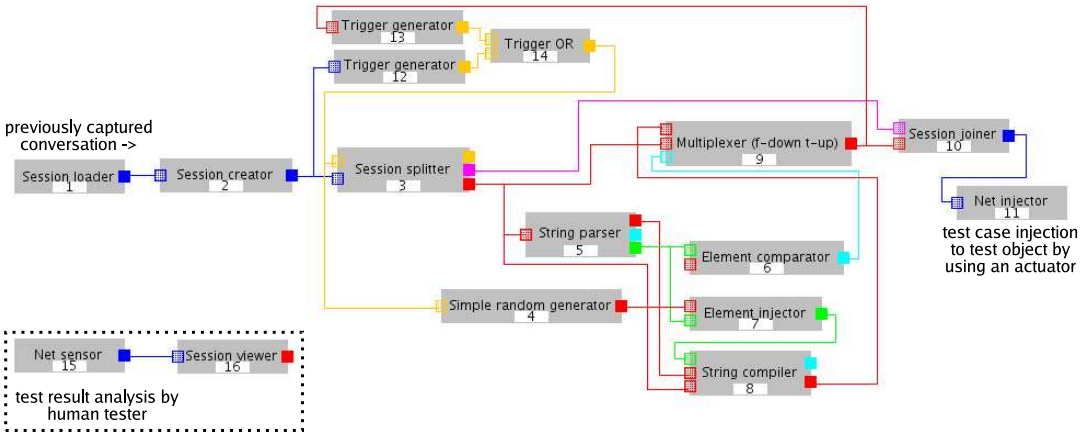


Figure 7.7: A protocol fuzzer created with the evaluation system

The random generator which was directly connected to the "Session joiner" in figure 7.6 is now the input module for an "Element injector" to mutate a selected element of the parser tree of the currently processed message. The "Element comparator" is necessary to decide which messages should be changed. For a better understanding, a more detailed description of an excerpt of the algorithm is given in figure 7.8.

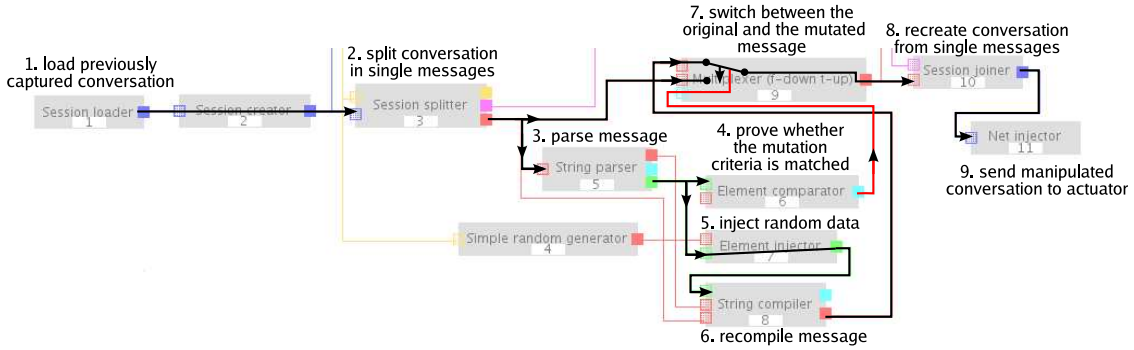


Figure 7.8: Protocol fuzzer with detailed description (data flow lines are broken when data type is changed when passing a module)

This algorithm design is based on the assumption that the initial data for the modification derive from a previously captured conversation with the test object. This conversation should already contain the messages which are used to create the test cases. The recombination of several messages using the "Session creator" is also applicable in order to create a new prototype session for the manipulation.

It is also evident from figure 7.7 that the result analysis and thus the failure detection is performed by a human tester. If possible it is also conceivable to use a method described in 7.2.

An example of the application of the protocol fuzzer algorithm presented in figure 7.7 is given below.

The test object is a FTP service with one user account for the user `foo` with password `foopwd`. The evaluation should prove the input validation of the FTP server program.

Before starting the test case injection, it is necessary to provide a grammar to parse and recompile the FTP protocol data units. Hereby the FTP grammar is used which has already been presented in figure 6.14. It is entered in the "String parser 5" and is forwarded by this module to the "String compiler 8".

In the next step a conversation between the FTP server and another system is captured which should be the base for the test cases. The capturing can be performed easily by a connection of a "Net sensor" and a "Session saver" module (for explanations of both module types see also 6.5). The single transmissions of the eavesdropped connection are the following:

(1) from FTP server to client: `220 (vsFTPd 2.0.3)`
(2) from client to FTP server: `USER foo`
(3) from FTP server to client: `331 Please specify the password.`
(4) from client to FTP server: `PASS foopwd`
(5) from FTP server to client: `230 Login successful.`
(6) from client to FTP server: `CWD test_dir`
(7) from FTP server to client: `250 Directory successfully changed.`
(8) from client to FTP server: `QUIT`
(9) from FTP server to client: `221 Goodbye.`

This conversation (or also called session) is loaded by the "Session loader 1" and forwarded through the "Session creator 2".

Another necessary configuration is the selection of the parser tree element which should be "fuzzed" and the criterion which should be matched by the messages which should be reinjected mutated. To help understanding the necessary parameterisation, the parser tree of the second message of the conversation (`USER foo`) which is produced using the mentioned FTP grammar is given here:

```
root
|-> data
    |-> commandline_0
    |   |-> command_0 :  USER
    |   |-> command_option_0 :  foo
    |-> commandline_0 :  USER foo
```

The test which should be performed is to combine the `CWD` command with arrays of randomized data of different sizes as command options. Thus, the "Element comparator 6" has to check whether the current processed message is the `CWD` message (message 6 of the conversation). The appropriate criterion which has to be set up is `root.data.commandline_0.command_0 = CWD`.

The parser tree node which has to be changed is `root.data.commandline_0.command_option_0`. Thus, this node name is entered to the "Element injector 7".

After applying this parameterisation, the testing algorithm can be executed. During execution the type of patterns generated by the "Simple random generator 4" can be changed by choosing other pattern types. A conversation forwarded by the "Net injector 11" to the actuator could be e.g. the following:

(1) from FTP server to actuator: 220 (vsFTPd 2.0.3)
(2) from actuator to FTP server: USER foo
(3) from FTP server to actuator: 331 Please specify the password.
(4) from actuator to FTP server: PASS foopwd
(5) from FTP server to actuator: 230 Login successful.
(6) from actuator to FTP server: CWD wq32,-32,m134iÃJX09saRVjnfc9317
(7) from FTP server to actuator: 550 Failed to change directory.
(8) from actuator to FTP server: QUIT
(9) from FTP server to actuator: 221 Goodbye.

By changing the criterion in the "Element comparator 6" to `root.data.commandline_0.command_0 = USER` it is possible to generate the following communication:

(1) from FTP server to actuator: 220 (vsFTPd 2.0.3)
(2) from actuator to FTP server: USER ds983:Ap2b2s13,2_3qwIwsIÃ§02Ãűjs2oj
(3) from FTP server to actuator: 530 Permission denied.
(4) from actuator to FTP server: PASS foopwd
(5) from FTP server to actuator: 530 Please login with USER and PASS.
(6) from actuator to FTP server: CWD test_dir
(7) from FTP server to actuator: 530 Please login with USER and PASS.
(8) from actuator to FTP server: QUIT
(9) from FTP server to actuator: 221 Goodbye.

In the similar way the manipulation of other command lines can be performed. Also the already mentioned variation of the injected data will lead to different test cases with potentially other reactions or also failures of the test object.

The presented procedure can be classified as semi-automatic. Once the algorithm is created and properly configured it can be triggered to infinitely inject communications to the test object. During this execution, the human tester can change the parameterisation without stopping or reinitiating the algorithm execution. By involving more coincidence in the process by e.g. applying random changes of the parameters of the "Element comparator", the "Element injector" or the "Simple random generator" the assistance through the human tester can be reduced and thus the automation can be increased. These functions are not implemented yet but can be an outlook for further improvements.

If special patterns are well known for often causing failures in combination with specific FTP commands in similar FTP server applications, it is helpful to provide these patterns as injection data from a special data storage unit. Such a module was not implemented for this diploma thesis but can be added easily.

At this, a problem which is still unsolved is the failure recognition. Especially when methods are used which improve the automation in the test case creation and the injection, an automatable failure detection strategy is more and more necessary. Conceivable is hereby a timeout monitoring (see also figure 7.5). In general, other procedures are difficulty to introduce because the appearances of potential failures is mostly unknown. In such a case sometimes a nominal-actual data or control flow comparison is applicable as another proposal for automatic failure detection.

7.4 Web application security tests

In the previous section, testing algorithms for test case injection and failure detection have been presented which are generally usable to evaluate several types of network applications.

In contrast, in this section the focus is specifically on so-called web applications. This takes into account that nowadays especially the sector of such software is becoming increasingly relevant.

Web applications are mainly faced with two types of security flaws: vulnerabilities against cross-site scripting and SQL injections. A comprehensive discussion of both security issues can not be given in this diploma thesis but a short summary is given in the particular subsections. For additional information see also [PC04] and [KP06].

Cross-site scripting

The principle of a cross-site scripting attack is already described in a general way in 2.4. This principle is now explained more detailed for web applications:

Strongly simplified, a client used to interact with web applications like e.g. a web browser usually receives two types of messages from the web server:

- The first type of messages are data in a description language which are used to describe the appearance of the web application output in a human readable form at the viewing device the user of the client uses to interact with the web application.
- The second type of messages are program codes which are executed on the machine of the web application user.

Interesting for cross-site scripting attacks is the second type. Because an arbitrary execution of external code per se leads to security flaws this process is normally restricted. Usually program code can only access data (including security relevant information e.g. credentials, tokens etc.) which derive from the same web site.

A common approach to bypass this limitation is to embed malicious code in pages of web sites where e.g. sensitive information should be theft from the associated data storage at the user's web client. This embedding can be performed when the web applications of a web site on their part filter incoming data insufficiently. Because the pages are often generated by using this previously entered information an attacker can exploit this lack of input validation.

In practice, the scripting language JavaScript is mostly used for such attacks. JavaScript codes are tried to be injected in parameters the web application receives from its clients. A defective web application reflects this code unmodified to its outputs and therefore distributes it to other clients which trust in the security of the web site.

An examination for cross-site scripting vulnerabilities is in many cases uncomplex and easy to apply (when comparing with other issues) because especially the failure detection is really simple. Such a testing algorithm can be described in a very abstract way as follows:

1. Identify web pages which transfer input data to the web application when specific actions take place. This step is also called "identification of the data entry points".
2. Inject code sequences at this data entry points in various combinations with normally intended input data.
3. Prove whether the injected code sequences appear unmodified in the test object outputs.

An implementation of this algorithm using the proposed software evaluation system is shown in figure 7.9.

At this, the data basis for the generation of the test cases are previously captured conversations which reproduce normal interactions with the test object involving transfers of input data to the web application.

These conversations are split in their single messages by the "Session splitter 2" and selected parameter values are replaced by a JavaScript code pattern stored in the "Constant string 5"

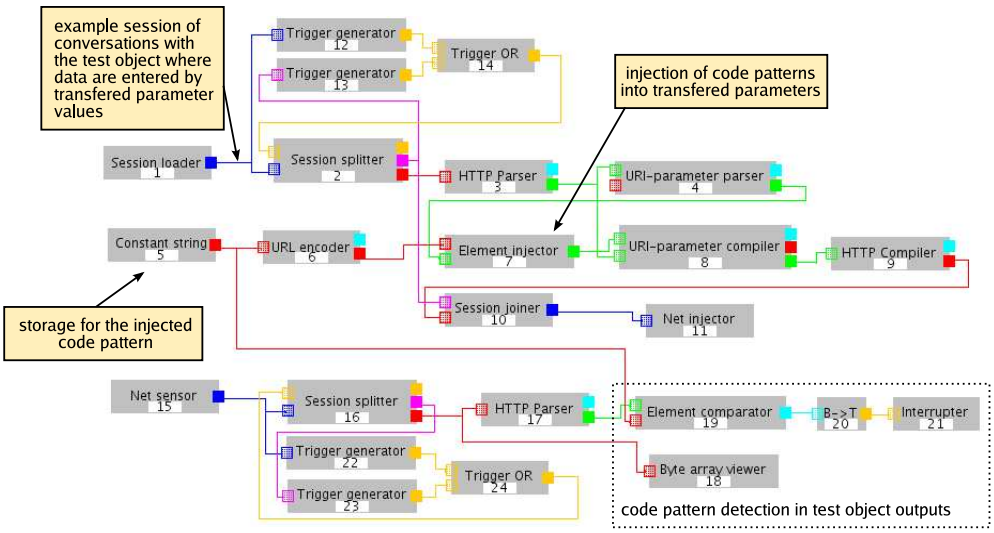


Figure 7.9: Algorithm for cross-site scripting test case injection and failure detection

module. This replacement is applied by the “Element Injector 7” module. The messages are assembled again to a session which is invoked as a test case by the “Net injector 11”.

The responses of the test object are received by the “Net sensor 15” and are analysed for the existence of the injected pattern in the output messages by the “Element comparator 19”. The algorithm is interrupted when such previously injected code has been detected. The human tester can examine the suspicious data by using the “Byte array viewer 18”.

To achieve an automatable process, the modules “Element injector 7” and “Element comparator 19” have to be supplemented with additional features. On the test case injection side it is necessary to provide mechanism to randomly insert an element of a set of relevant code patterns as web application input parameter. By looping this procedure it is possible to create many various test cases.

The failure detection on the other side has to check whether one ore more of these codes are part of the test object output.

In the current version of the software evaluation system used for the practical examinations in this diploma thesis only the random selection of the parameters which has to be manipulated has been implemented yet. The injected code is not randomly selected from a set of patterns but such functionality can be added without any significant effort when necessary.

The procedure of finding cross-site scripting vulnerabilities with the proposed algorithm is also explained in the following example from tester’s real life.

The test object is a web application which provides a guestbook for visitors of a website. The user interface of this software is shown in a screenshot below this paragraph. The user can enter some data in the edit fields and can deliver his new guestbook entry by clicking the “Absenden” button.

The message which is sent to the HTTP server looks as follows:

```
POST /guestbook-php/index.php HTTP/1.1
Host:  tools.guestbook-provider.de
User-Agent:  Mozilla/5.0
Accept:  text/html;q=0.9
Content-Type:  application/x-www-form-urlencoded
Content-Length:  173

entry%5Bname%5D=test&entry%5Bemail%5D=test%40test.de
&entry%5Burl%5D=http%3A%2F%2F&entry%5Bcountry%5D=Deutschland
&entry%5Bcomment%5D=test&add=Absenden&id=xushuette&PHPSESSID=
```

There are several parameters which reflect the values like name, e-mail address or country which the web application user enters before performing “Absenden”. The session which is used to apply the tests only includes this message.

The “URI-parameter parser 4” generates the following parser tree from the transferred parameter information:

```
root
|-> parameters
    |->       entry%5Bname%5D : test
    |->       entry%5Bemail%5D : test%40test.de
    |->       entry%5Burl%5D : http%3A%2F%2F
    |->       entry%5Bcountry%5D : Deutschland
    |->       entry%5Bcomment%5D : test
    |->       add :  Absenden
    |->       id :  xushuette
    |->       PHPSESSID :
```

For the cross-site scripting vulnerability test the following code patterns are used:

- Pattern 1: `<script>alert(0)</script>`
- Pattern 2: `<scRipt>alert(0)</scRipt>`

Both patterns are entered successively in the “Constant string 5” module. Note that this module not only supplies the “Element injector 7” with the injected pattern but also provides the comparison value for the “Element comparator 19” in the failure detection section of the testing

algorithm. The "Element comparator 19" proves whether the pattern is contained in the element `body.data` of the HTTP parser tree of the test object response messages. The "Element injector 7" is set up to inject the pattern randomly in one of the elements of the parser tree shown above.

After entering one of the two patterns the testing can take place without any further assistance. The algorithm interrupts automatically, when the entered pattern is found in the responses.

In this example, the parameter `entry%5Bcountry%5D` is susceptible for pattern 1 and the parameter `entry%5Bname%5D` is susceptible for pattern 2 which means that the test object does not change the parameter values "to be harmless" before inserting them as content in the output messages.

For this example application the attacking of users of the guestbook is not the critical issue. More problematic is that a guestbook administration which is also affected by these security flaws is part of the account management for a domain. By thieving sensitive information from users logged in at this management it is possible to break in into their accounts.

SQL injection

An SQL injection is a command injection attack (see also 2.4) which is also used as a threat against web applications. In difference to cross-site scripting the main objective is hereby a change in the behaviour of a web application which was not intended by the application developer.

Like the cross-site scripting attack as well, the underlying flaw is a lack of validation of the application input parameters. When these parameters are used to create a SQL command executed by a data base it is possible to provoke a malicious behaviour.

A simple example could be the creation of an SQL statement using the following rule (`name` is the parameter previously received as a web application input):

```
sql_cmd := "SELECT * FROM users WHERE username = '" + name + "';"
```

Such a statement could be used e.g. to extract and show all information concerning the user `name` from the data base. By entering a parameter value like `Max' OR '1'='1'` the following command will result by applying the same rule mentioned above:

```
sql_cmd := "SELECT * FROM users WHERE username = 'Max' OR '1'='1';"
```

In that example, the web application could possibly handle the data base answer in such a way that it shows all the records stored in the table `users`.

To set up a testing algorithm for detecting SQL injection vulnerabilities the test case creation from the cross-site scripting algorithm can be used (see also figure 7.9). At this, instead of entering a JavaScript code pattern like it is used to detect cross-site scripting security flaws, the injected pattern should be a part of a SQL statement which is compatible to the other part given by the web application code.

The failure detection is much more complicated. Depending on the intention of such an SQL injection the appearance of an error can be manifold. Some web applications forward resulting error messages, others intentionally suppress any messages which could help to conclude on the internal processes.

Thus it is difficult to establish criteria. In difference to the cross-site scripting algorithm it can be more helpful to perform tests of the absence of specific patterns in the test object responses. E.g. when a test should prove whether a login page can be overridden by an SQL injection it makes sense first to prove what the test object usually answers when an attempt fails and then to use this knowledge to identify successful passes.

Another approach could use the concept of the intercepting proxy (see 6.2 and figure 5.4). Hereby, the SQL injection pattern is not applied to a previously captured message which is repetitively

manipulated and invoked as a test case but is inserted to HTTP requests from a web client operated by the human tester and which are sent to the test object through the evaluation system.

The responses are received by the web client and can be examined by the human tester whether a security relevant event occurred or not.

At the first sight this method seems not very reasonable but it has advantages when the evaluation system holds a set of patterns which can cause SQL injection security vulnerabilities. Then the human tester just has to reload the same web pages which are target of the evaluation and by selecting another injection pattern the evaluation system generates a new test case.

A completely other method could also be not only to prove the data transmissions between the web application and the web client but also to capture and to analyse the conversations between the web application and the data base server. This transmissions can also be checked for "illegal" patterns.

Of course, this approach is only realisable when it is possible to place a sensor between these two systems. But the analysis effort is less than by trying to interpret the responses of the web application.

7.5 Additional annotations

One may ask for the sense of a second implementation of testing procedures from already known approaches and solutions. The answer is that in this chapter it should be outlined that the proposed evaluation system is able to perform the same methods the existing approaches already provide but to realise these methods in one single concept and one single system.

Nevertheless these considerations should only be an entry point for further experiments and more complex testing procedures. The main aspect is to show that the evaluation system concept is appropriate to allow a human tester to create testing algorithms in manifold variations and to apply these algorithms automatically when this is possible in principle.

Many other issues could not be discussed because of the limited scope of this diploma thesis. Interesting could be e.g. the combination of different testing algorithm for the test case data injection. The software security evaluation system allows e.g. to perform stress tests and meanwhile to execute test cases using another technique like a protocol fuzzer attack. On the other side the application of different failure detection methods at the same time is not considered as well.

The application of the test object monitoring proposed in section 5.6 is also not included in the proof-of-concept. In the same way the simultaneous monitoring of different test object interfaces resp. different communication channels using one testing algorithm is a topic for further considerations.

8 Summary, conclusions and further considerations

8.1 Conclusions about the existing approaches

As described in chapter 3 there exist several theoretical and practical approaches for automatic and semiautomatic software security evaluation. The following results are arising from the examination of the presented methods:

Only few approaches provided a full automated evaluation of the test object. Often the interpretation of the results of test cases is incomplete or missing. The main focus is on the generation of malicious test case data and not on the recognition of system failures.

The commercial web application scanners which claim to feature an automated evaluation of web applications provide highly insufficient evaluation results.

From a theoretical point of view, those approaches which include automatic evaluation routines are only able to discover uncomplex security flaws because of their uncomplex test case generation. Uncomplex means in this context, that the security flaws exist very early in the processing of the input data in the test object. The most often affected algorithm parts in the test object are input validation routines.

Unfortunately, these kinds of security flaws are probably becoming less important in the future because of the decreasing proportion of software written in high level but machine-oriented programming languages like standard C or C++. Nowadays, less pretentious programming languages e.g. Java, C++ with Microsoft .NET extension or scripting languages like PHP or Perl are abstracting memory management, process synchronisation or type conversations. This abstraction prevents that the programmer can generate some classes of typical erroneous code like wrong buffer handling (which can cause buffer overflows) or format string weaknesses.

None of the approaches include methods for automated exposure of security flaws which are detectable by a passive analysis like e.g. weak encryption or predictable random numbers. Only few approaches include semiautomatic methods to handle these issues.

On the other side, a comprehensive but unspecific trial and error strategy like it is performed by the commercial web application scanners does not produce convenient results. The definition and consideration of individual security targets for each test object is deemed necessary. Approaches which provide an automated generation of these security targets from any kind of available specification are unknown.

Finally, the assertion can be made that the use of artificial intelligence approaches like automated reasoning or knowledge representation in security evaluation software is relatively unknown (not considering very simple artificial concepts).

8.2 How the presented security evaluation system improves the existing approaches

The examination in chapter 3 shows that currently there exist only few sophisticated approaches which allow an automated security evaluation with certain qualifications.

This most unsatisfactory development of this research sector leads to a theoretical discussion of

the question for the limits of automated security evaluation and which level of automation should be aimed in order to perform evaluations with a reasonable cost-benefit ratio. This aspect is discussed in chapter 4 and as a result an approach for a distributed software system which realises the concept of software-aided security evaluation is proposed.

The following elements of this system are deriving from the discussed existing approaches:

- Collaboration of modules which implement single data processing functions (see also Web-Scarab in 3.4)
- Test case generation by embedding random patterns into interface messages (see also 3.1)
- Interpretation or creation of interface messages regarding communication protocols and other predefined rules (see also 3.2)
- Specification of message syntax using a context-free grammar (see also PROTOS in 3.2)
- Consideration of test object monitoring by observing the resources consumed by the test object during the test case injection (see also Holodeck in 3.4)

These elements are complemented with the new approaches, considerations and features which outline the specific character of the solution proposed in this diploma thesis.

- Separation of the sensors and the actuators from the central control mechanism which is used to create the test cases, to control the test case injection, to analyse the interactions of the test object with its environment and to detect failures (see also 4.3)
- Development of a common data exchange protocol for the communication between the central control and the sensors and actuators (see also 5.2 and 6.2)
- Selection of existing data structures or development of new data structures which are helpful to design a software security evaluation system where several modules which individually implement specific functions form a testing algorithm through their collaboration (see also 5.3)
- Creation of a graphical user interface which allows a human tester to implement testing algorithm and, more important, to intuitively visualise the processes during the testing algorithm execution and which allows interventions of the human tester like parameter changes at any time during this execution (see also 6.4)
- General possibility to apply a testing algorithm automatically, when both of the following criteria are fulfilled by this testing algorithm:
  - The test case execution is designed in that way that it can create test cases automatically, also when this process involves a necessary consideration of the messages received from the test object
  - The failure detection is able to automatically detect test object failures and in the context of this diploma thesis especially security flaws
- Possibilities for semi-automatic testing algorithm application when the automation criteria for the testing algorithm can not be fulfilled completely or partially
- Discussion of methods for failure detection and consideration of failure detection algorithm as a part of a whole testing algorithm (see also 5.6 and 7.2)
- Capability for a combination of different test case creation methods with several failure detection mechanisms in one testing algorithm.

- Independence of the security evaluation system concept and especially the design of the central control from any communication method or protocol used between the test object and its environment. The system can be adapted simply by replacing the sensors and actuators by appropriate ones. Adaptable also means in this context that the testing algorithm can be used with other protocols only by applying few changes like replacement of parsers and compilers.

8.3 Problems of automated software security evaluation which are not solved in this diploma thesis

As already discussed in chapter 4 a complete automation of tests for specific security vulnerabilities is often not a sensible research goal. The failure of such concepts can be exemplarily studied on the commercial web application scanners. Other concepts like the different fuzzers and fuzzing frameworks are often incomplete regarding the automatic failure detection.

Thus the concept of the software-aided security evaluation is introduced in 4.3. "Software-aided" in this context is not referring to the use of assisting tools which is, of course, an already widely applied method. The term is more focussed on the discussion of the design of a software evaluation system which should supply the human tester with adequate test creation and analyse functions. When the human tester can entirely specify a security test by using the means of the evaluation system, this specification respectively this testing algorithm can be applied automatically by the system (an example is the cross-site scripting testing algorithm in 7.4).

None the less, the involvement of a human tester is mandatory to fill out the gaps in a test procedure where processes are vague and can not be described in a machine readable representation.

Other problems for which approaches or solutions are not provided in this diploma thesis but which are still important in the context of testing automation are the following:

- Interpretation of data, which include contents created for the interpretation by a human like information in human language, pictures, colours and text arrangements
- Concepts for automatical selection of test cases based on information like the specification of the test object and the particular security targets or a knowledge database with experiences about similar testing scenarios
- Automatic classification or evaluation of the relevance of test results
- Methods for automated or semi automated exposure of security flaws by analysing non-malicious (common) interactions with the test object (passive analysis)

8.4 Possible enhancements of the security evaluation system

Because of the early-prototype character of the proposed software system, there exist several points for further improvements.

The usability of the user interface of the central control unit is currently not completely satisfying and needs some improvements. This could be e.g. a better module management, presentation of the testing algorithm in a script language or better testing algorithm debugging capabilities respectively monitoring support.

Another problem is that the central control unit currently does not provide sufficient feedback capabilities between the test object outputs which are a result of a test case data injection and the creation of further test case data. This feature should be implemented in further versions. Therefore, synchronisation mechanisms are required because for the generation of specific input data it can be necessary to wait for the output of needed data by the test object.

Useful could be also the integration of existing parser and compiler frameworks which already provide a large set of available grammars. These tools can replace the universal Backus-Naur grammar parser and compiler which are implemented in the current software version.

In addition for further considerations it could be helpful to implement new sensors and actuators for different platforms to provide more possibilities of accessing other test object interfaces.

Finally the creation of additional testing algorithm and a further application of the proposed methodologies should help to outline the applicability and also the limits of the presented approaches. Such considerations should help to claim requirements for the further development of software security evaluation systems.

Bibliography

[Ait02] Dave Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. Technical report, Immunity, Inc., 2002.

[APP] AppScan. http://www.watchfire.com/products/appscan/appscan.aspx. last visited on 2006/05/09.

[Bei95] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems.* John Wiley & Sons, Inc., New York, NY, USA, 1995.

[GLI] G - 2D Graphics Library and Rendering Engine for Java. http://geosoft.no/graphics. last visited on 2006/06/12.

[GLLR05] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn, and Robert Richardson. *2005 CSI/FBI Computer Crime and Security Survey.* Computer Security Institute, 2005.

[Gra90] Timm Grams. *Denkfallen und Programmierfehler.* Springer-Verlag, 1990.

[GSS98] A. Ghosh, M. Schmid, and V. Shah. Testing the robustness of Windows NT software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE '98)*, pages 231–235. IEEE Computer Society, November 1998.

[HHLT03] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung 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. ACM Press, 2003.

[HMRU00] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HTLK04] Yao-Wen Huang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Kuo. Non-Detrimental Web Application Security Scanning. In *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE'04)*, pages 219–230. IEEE Computer Society, 2004.

[Inc06] Secure Software Inc. CLASP - Comprehensive Lightweight Application Security Process - Version 1.2. Technical report, 2006.

[IRC] ircfuzz. http://ilja.netric.org/files/fuzzers/. last visited on 2006/05/09.

[ISO98] ISO/IEC Standard 15408. *Common Criteria for Information Technology Security Evaluation*, Version 2.0 edition, November 1998.

[Jos03] S. Josefsson. The Base16, Base32, and Base64 Data Encodings, July 2003.

[Kak01] Rauli Kaksonen. *A Functional Method for Assessing Protocol Implementation Security.* Technical Research Center of Finland (VTT - Valtion teknillinen tutkimuskeskus), 2001.

[KFN99] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[KP06] Christopher Kunz and Peter Prochaska. *PHP-Sicherheit. PHP/MySQL-Webanwendungen sicher programmieren*. Dpunkt Verlag, 2006.

[LBMC93] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. Technical report, Naval Research Laboratory, November 1993.

[MAN] Mangleme. http://freshmeat.net/projects/mangleme/. last visited on 2006/05/09.

[Mar05] Mathias Markert. Methoden und Algorithmen zur Texturanalyse bei der optischen überwachung von Gefahrenräumen, 2005. Diploma Thesis.

[MCA] McAfee Foundstone Division. http://www.foundstone.com/index.htm. last visited on 2006/09/11.

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[Mil85] David L. Mills. RFC 958: Network Time Protocol (NTP), September 1985.

[MKL+95] Barton Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical report, 1995.

[Oeh05] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[OPE] Open Clip Art Library. http://www.openclipart.org/. last visited on 2006/05/09.

[Org89] International Standards Organization. Information processing systems - Open Systems Interconnection - Basic Reference Model. Technical Report 7498-2:1989, ISO, 1989. Part 2: Security Architecture.

[PC04] Cyrus Peikari and Anton Chuvakin. *Security Warrior*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[PEA] The Peach Fuzzer Framework. http://peachfuzz.sourceforge.net/. last visited on 2006/07/18.

[PKI] Certification instance of the FhG. http://pki.fraunhofer.de/EN/.

[Pol06] Andreas Poller. Approaches for automated software security evaluations. http://archiv.tu-chemnitz.de/pub/2006/0187, 2006. Diploma Thesis.

[PR85] J. Postel and J. K. Reynolds. RFC 959: File Transfer Protocol, October 1985.

[SPI] Spikefile. http://labs.idefense.com//labs-software.php?show=14. last visited on 2006/05/09.

[STR] FreeBSD Kernel Stress Test Suite. http://www.holm.cc/stress/. last visited on 2006/09/11.

[VMKV97] Jeffrey M. Voas, Gary McGraw, Lora Kassab, and Larry Voas. A 'Crystal Ball' for Software Liability. *IEEE Computer*, 30(6):29–36, 1997.

[vS05] Ilja van Sprundel. Fuzzing: Breaking software in an automated fashion, 2005. Paper published for 22C3 Chaos Communication Congress.

[WEBa] WebInspect. http://www.spidynamics.com/products/webinspect/. last visited on 2006/05/09.

[WEBb] The WebScarab Project. http://www.owasp.org/software/webscarab.html. last visited on 2006/05/09.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.

Index

Glossary

API	Application Programming Interface, source code interface between a program and another software system
application layer	layer seven (top layer) of the OSI reference model where the application processes are situated which use the underlying protocol stack, often this layer also includes the user interface
Backus-Naur form	a language to create context free grammars
black box testing	software testing method which is applied without a consideration of the internal structure of the test object
central control unit	in the context of this diploma thesis a central control unit is the central component of the distributed software security evaluation system
CGI	Common Gateway Interface, protocol which is used to interface external applications through a web server
checksum	data pattern which is added to allow a check of the integrity of other data units
cost-benefit analysis	analysis method which weights the benefit of an option against its costs
crawler	software which is used to explore (to crawl) the structure of a web application by following hyperlinks inserted in the web pages of this web application
failure mode and effects analysis	method which analysis the potential failures of a system and their effects and which tries to identify counter measures
form parameters	parameters which are generated from the user inputs into a HTML form and which are transferred to a web server to allow a reaction on this user input, also called “form values” or “HTML form parameters”
FTP	File Transfer Protocol, protocol for the file exchange between an FTP server and an FTP client
C/C++	C is a procedural programming language and C++ is its object-oriented derivate, it is one of the most widely used programming languages which uses a compiler to translate the C/C++ code in machine code for a dedicated platform
fuzzer	software which is used to provoke software failures by sending data units to the test object interfaces which are completely or partially created by a random generator
HTML	Hypertext Markup Language, description language which is commonly used to describe the appearance and structure of web pages, based on HTML information a web browser renders a view of a web page with its text, images, hyperlinks etc. for a human user

HTML form	element of HTML pages which allows a human user to interact with the mechanism which provides the page (e.g. a web application)
HTTP	Hyper Text Transfer Protocol, network protocol which is commonly used to transfer data between a web server and a web client
IRC	Internet Relay Chat, network protocol which is used for instant messaging
Java	an object-oriented programming language, Java source code is usually compiled in a machine code for a so-called Java Virtual Machine (JVM)
length field	data pattern which is added to store the length of other data units
LDAP	Lightweight Directory Access Protocol, network protocol which allows to access data stored in a directory service
man-in-the-middle attack	an attack where the attacker compromises a running conversation between two communicating parties
network service	service installed on a dedicated system which is accessible from other systems by using a computer network
OSI reference model	Open Systems Interconnection reference model, is a model for network protocol design which is based on the assumption that network protocols are organised in a protocol stack where each protocol in a layer of this stack uses the functions of the lower level stack protocol
padding	data pattern which is used to increase the size of a data unit with random data to reach a specific unit length
plug-in	is software which usage depends on a main application where it is added to provide specific (plug-in) functions
presentation layer	layer six of the OSI reference model used to format information in a representation which is comprehensible by the communicating processes
risk analysis	method to evaluate risks regarding their probability
script	algorithm description created by using a scripting language which is usually interpreted by a script interpreter which executes each script statement immediately after reading it instead of controlling a compiler to create a transformation of the source code into machine code
session layer	layer five of the OSI reference model used to organise the conversation between the communicating processes
service access points	interface which can be used by software to access the services of the underlying operating resource provider
SIP	Session Initiation Protocol, signalling protocol to manage interactive user sessions like telephone calls or instant messaging
SNMP	Simple Network Management Protocol, network protocol for managing and monitoring of network devices
software back-end	part of a software system which processes the data received from the front-end

software failure	effect of a software error which is visible for an observer outside the considered software system
software front-end	part of a software system which interacts with the software user
software life cycle	process model for software engineering
stress test	test method which tries to provoke a data processing inside the test object near the resource limits refering to e.g. processor load or memory usage
system environment	all other systems and devices which have logical access to the considered system
TCP/IP	common combination of the Transmission Control Protocol with the Internet Protocol in a protocol stack used for connection-oriented data transmissions
testing algorithm	algorithm which describes a procedure for the generation and injection of test case data (test case creation), the analysis of the test object reactions and especially the detection of software failures
TFTP	Trivial File Transfer Protocol, simplification of the File Transfer Protocol which provides file transfer capabilities based on small command set
UNIX	an operating system
URI	Uniform Resource Identifier, a string of characters which identifies a resource
URL	Uniform Resource Locator, a specialisation of an URI which additionally identifies the location of the mentioned resource by describing the access mechanism to it
vulnerability	weak point of software implementations which can be exploited in a malicious manner
waterfall model	software engineering model which is based on the assumption that a software life cycle consists of the sequential execution of several software development steps (requirements analysis, design, implementation, testing, integration and maintenance) whereby each step starts after finishing the previous step
web application	application which is accessed by a human user using a web browser
web browser	software which is used by a human to view web pages and to interact with the web server which delivers these web pages
web server	program which acts as a network service to provide access to web pages and web applications
white box testing	software testing method which involves knowledge about the test object internals e.g. source code

A Commercial scanner test

A.1 Test conditions

The tested commercial vulnerability scanners have been:

- WebInspect by SPI Dynamics (version 5.8.276)
- AppScan by Watchfire (version 6.0.346)

Both applications have been installed as trial versions on a test system with the operating system Windows XP Professional with Service Pack 2.

The test targets for the evaluation have been two dummy applications which are used to demonstrate web application vulnerabilities:

- Hacme Books v1
- Hacme Bank v2

(Both are provided by the McAfee Foundstone Division [MCA])

In addition to run the target applications the following software has been installed on the target system:

- Java Development Kit 1.5.0.06
- Microsoft IIS 5
- Microsoft .NET Framework 1.1
- Microsoft MSDE 2000 Release A
- Apache 2.2.2

The test scenario is also described in figure A.1

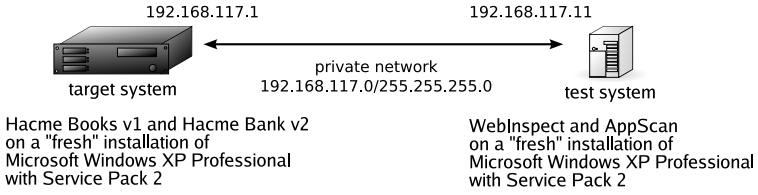


Figure A.1: The test scenario for the commercial evaluation tools

The trial versions are limited to access fixed IP addresses only. The systems reachable under these addresses are web servers with test applications provided by the software vendors. To bend this condition the IP addresses in the test scenario has been adapted to meet these requirements. Another problem was, that Hacme Bank only allows connections from “localhost” with IP address “127.0.0.1“. To solve this problem, an Apache web server has been installed at the target system as reverse proxy to forward request over the ”local loop“ network interface.

A.2 Test procedure

The test procedure has been similar for both vulnerability scanners and both dummy applications. The URL of the single web application has been entered to the vulnerability scanners. If possible, the scanner rules has been adapted to apply as much test techniques as possible and to found as much security flaws as possible.

For evaluating Hacme Bank as a test target, it was necessary to pass a login screen to enter the "bank account" in the web application. Thus, for both scanners it has been necessary to teach them the login procedure to provide the username and password.

Each dummy application has been scanned separately by each scanner. There was no intervention by the user necessary during the scan execution. Each scan finished with the creation of a scanning report which was the base for the test results. The test target has been reset after each scan (system reboot).

A.3 Test results

A.3.1 WebInspect

Hacme Books

Security flaws found:

- blind SQL injection, which has been a false positive
- directory listing (possibility to list a directory via a URL)
- admin section require authentication (possibility to list a directory named with "admin")
- GET open proxy access allowed (to use a web server as a proxy by entering an absolute URL in the HTTP request), which has been a false positive

Hacme Bank

Security flaws found:

- directory listing (possibility to list a directory via a URL)

Beside the listed security flaws, many other so-called "security relevant" issues have been found as well which has been classified as "medium risk" or "low risk" vulnerabilities. The most of them has been false positives. They are not listed here, because they would extend the list of detected failures significantly.

A.3.2 AppScan

Hacme Books

Security flaws found:

- two correct found SQL injections
- cross-site scripting vulnerabilities, which have been misinterpretations of server answers
- several other SQL injection vulnerabilities, which all have been false positive

Hacme Bank

Security flaws found:

- several SQL injections, which all have been false positive

A.4 Test summary

As a summary of the test results the following statements can be made:

- Both vulnerability scanners have not been able to find any security flaw which was implemented in the dummy applications in order to teach security testers in application evaluations.
- The scanners have often misunderstood server responses which resulted in false positives.
- The scanners have not been able to explore the structure of both web applications successfully.

Of course, this analysis is only related to the trial versions of the presented products, the automated scanning mode and the presented dummy applications.

B Usage of the implemented sensors and actuators

The sensor and actuator programs presented in this chapter have been implemented to assist a testing of the capabilities of the evaluation system which is described in this diploma thesis. They have been implemented for the Linux operating system but the implementation is portable to other platforms with few changes in the source code. In summary they should only provide suggestions for possible solutions of the problem of capturing data at the interfaces of a test object and hereby especially at network interfaces.

B.1 The NetSensorLinux sensor

Purpose

The NetSensorLinux sensor is a so-called sniffer based on the libpcap library. It can capture data on all network interfaces or on one selected network interface of a computer system.

Usage

Command line:

```
NetSensorLinux server_port [interface]
```

The parameter `server_port` requires an integer value which represents the port number where the sensor server should listen for incoming connections from the central control unit or other clients. The captured data are sent to the connected client using the XML format described in 6.2.

The optional parameter `interface` specifies a selected interface where the sniffer should capture data on.

B.2 The InjectorLinux actuator

Purpose

The InjectorLinux actuator can be used to initiate TCP/IP connections to invoke test cases on the network interfaces of a test object. It consists of an actuator server for collecting the data which have to be sent to the test object, TCP clients to connect with the test object and a sensor server which repeats the test case data and also sends back the responses of the test object.

Usage

```
InjectorLinux injector_port sniffer_port
```

The parameter `injector_port` requires an integer value which represents the port number where the actuator server should listen for incoming connections from the central control unit or other

clients. This actuator server receives the data of the test cases which should be invoked at the test object network interfaces using the TCP/IP network protocol.

The parameter `sniffer_port` requires an integer value which represents the port number where the sensor server should listen for incoming connections from the central control unit or other clients. The sensor server repeats the test case data received at the actuator server but with the parameters of the newly created communication channel. In addition it also reflects the answers of the test object.

Both servers use the XML data format described in 6.2.

B.3 The SnifferProxyLinux intercepting proxy

Purpose

The SnifferProxyLinux is an intercepting proxy which can be used to intercept HTTP requests generated by a HTTP client in the environment of the test object. Instead of sending these request directly to the test object HTTP interface they are redirected through the connected central control unit. In this central control unit these request can be mutated to invoke test cases. In the following, these new data are sent back to the proxy which forwards them to the test object.

Usage

```
SnifferProxyLinux proxy_port absorber_port injector_port
```

The parameter `proxy_port` requires an integer value which represents the port number where the HTTP client has to send its HTTP request to. Thereby the HTTP proxy protocol is used.

The parameter `absorber_port` requires an integer value which represents the port number where the sensor server should listen for incoming connections from the central control unit or other clients. This sensor server forwards the intercepted HTTP requests. The proxy connection of the HTTP client is delayed until the request is received back through the actuator server. This concept is also illustrated in figure 6.4.

The parameter `injector_port` requires an integer value which represents the port number where the actuator server should listen for incoming connections from the central control unit or other clients. The actuator server receives the HTTP request data back. After the reinjection of the data, the communication channel to the HTTP interface of the test object can be established and the data can be forwarded. This action is a test case invocation when the data have been mutated by the central control unit.

C Data types at the module interfaces

This chapter presents the different data types at the inputs and outputs of the modules of the evaluation system proposed in this diploma thesis. The colour of each heading represents the colour in the testing algorithm diagram.

Session

The session data type represents data structured in the session data format described in 5.3 and 6.2.

Byte array

As the name implies the byte array type provides a data array where each array element is interpreted as a byte. This type is the most unstructured representation of data transmitted at test object interfaces. It is also called the "raw data" type.

Data abstraction

The data abstraction type is the structured representation of data units transmitted at test object interfaces. This structure is created by a parser, which interprets the raw data in a byte array. The representation itself is the parser tree resulting from this process. After the processing of the parser tree (e.g. manipulation of parser tree elements) it is necessary to convert it back to a byte array for the further test case invocation.

The relationship between the types session, byte array and data abstraction is also described in appendix D.

Boolean

This type represents a single boolean value which can be "true" or "false".

Trigger

This is a special type where its instances have no real value itself. Only the event of the creation of an object of this data type without a value is recognized by modules which process trigger inputs. Usually it is used to signal specific events in a testing algorithm where a module has to react on when its update function is called.

Meta data

The instances of the meta data type contain specific parameters for each connection in a session. These parameters could be e.g. identifier of the communication source and drain. Each parameter entry consists of a parameter name and a parameter value.

D Example for data exchanges between several modules in the evaluation system

Figure D.1 shows an example testing algorithm which captures a conversation between a HTTP server and a HTTP client (in this example a web browser) and which reinjects the HTTP request through an actuator again to a HTTP server. The conversation consists of two connections.

The picture is a snapshot which was taken after the first run of all modules. The loop for the processing of the messages in the session at the output of "Net sensor 1" which is established by the "Session splitter" and the "Session joiner" has just finnished its first iteration in this snapshot.

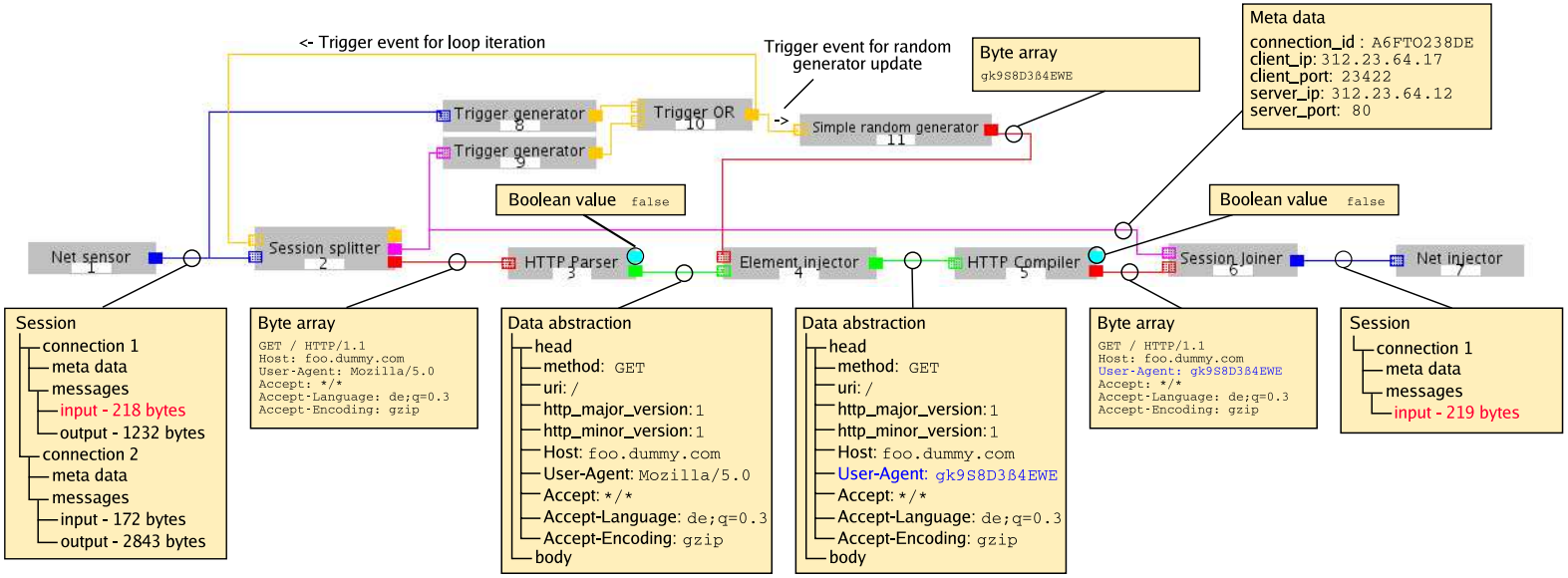


Figure D.1: Example for data exchange between modules

The yellow boxes show the data at the outputs of selected modules. Note that the number in the bottom of each module represents the order of execution.

The "Session splitter 2" has assigned the content of the first message of the first connection in the session received from the output of "Net sensor 1" to its own outputs. This message is marked with red letters.

In the following, the message (a HTTP request) has been processed by a parser. The "Element injector 4" has replaced the "User-Agent" data of the request head by a random value which was generated by the "Simple random generator 11". The message has been recompiled and added to a new session by the "Session joiner 6" involving the meta data created by the "Session splitter 2". This new session has been input for the "Net injector 7". The two boolean values show whether an error has appeared during the parsing and compiling process.

The next iteration of the message loop has been triggered by the event generated by the two "Trigger generator" modules. In the further processing the other messages of both connections will be edited when they also include the element `head.User-Agent`.

This criterion does not match for the HTTP responses. So these responses will pass the loop without a manipulation. The loop iteration stops when all messages have been processed and restarts again when the data at the output of "Net sensor 1" change.

It is also possible to exclude test object outputs from being forwarded by the "Session splitter" by changing the parameterisation of this module. At this, the processing of such messages is not problematic, because the "Net injector" does not forward data to the actuator which have been outputs of the test object during the preceding capture by the sensor.

E Content of the enclosed CD

The content of the CD enclosed to this diploma thesis is the following:

Directory	Content
`\CentralControl\`	central control unit software as NetBeans IDE project
`\CentralControl\javadoc\`	Java API documentation for the central control unit software
`\Examples\`	example testing algorithms which can be loaded as workspaces in the central control unit and additional files
`\Examples\AddOns\`	additional files
`\Examples\AddOns\ftpgram.txt`	grammar for the FTP protocol
`\Examples\AddOns\ftp.sdt`	session data for a FTP conversation which can be loaded into the central control unit using a "Session loader" module
`\Examples\PatternDetection\`	pattern detection algorithm
`\Examples\ProtocolFuzzer\`	simple protocol fuzzer
`\Examples\Replay\`	simple replay algorithm
`\Examples\ReplayModified\`	replay with modification algorithm
`\Examples\SimpleFuzzer\`	simple fuzzer
`\Examples\StressTest\`	stress test generation algorithm
`\Examples\TimeoutDetection\`	timeout failure detection algorithm
`\Examples\XSS\`	simple algorithm for detecting cross-site scripting vulnerabilities
`\Exe\`	executables of the developed programs
`\Exe\CentralControl\`	distribution build of the central control unit, use `java -jar "CentralControl.jar"` to execute
`\Exe\InjectorLinux\`	executable binary of the InjectorLinux actuator, use `InjectorLinux injector_port sniffer_port` to execute
`\Exe\NetSensorLinux\`	executable binary of the NetSensorLinux sensor, use `NetSensorLinux server_port [interface]` to execute
`\Exe\SnifferProxyLinux\`	executable binary of the SnifferProxyLinux intercepting proxy, use `SnifferProxyLinux proxy_port absorber_port injector_port` to execute
`\Libraries\GLibrary\`	G - Java 2D Graphics Library which is used to implement the central control unit; this software is licensed under the GNU Lesser General Public License
`\Libraries\Libb64\`	libb64 library which is used to implement the sensors and actuators; this library is licensed under the Creative Commons Public Domain License
`\Libraries\Libpcap\`	libpcap library which is used to implement the NetSensorLinux sensor; this library is licensed under the BSD License

\SensAct\	sensors and actuators developed for this diploma thesis as Eclipse projects
\SensAct\InjectorLinux\	InjectorLinux actuator (see also B.2)
\SensAct\NetSensorLinux\	NetSensorLinux sensor (see also B.1)
\SensAct\SnifferProxyLinux\	SnifferProxyLinux intercepting proxy (see also B.3)
\VulnScanners\	test reports generated by the commercial vulnerability scanners (see also appendix A)
\gpl.txt	a copy of the GNU General Public License, all source codes which have been written for this diploma thesis are licensed under the GPL

The following tools have been used for the development of the software:

- central control unit
  - J2SE Development Kit 5.0 with NetBeans IDE 5.0 / Java 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)
  - G - Java 2D Graphics Library version 1.0
- Linux sensors and actuators
  - libb64 version 1.1
  - libpcap version 0.3 (only used for the NetSensorLinux sensor)
  - Eclipse SDK version 3.1.1
  - GNU Compiler Collection (GCC) version 4.0.2
- All programming and testing work has been done on a Fedora Core 4 platform with a Linux kernel version 2.6.17-1.2142.

F First steps with the evaluation system software

This chapter will describe the first steps with the evaluation system software using a simple application scenario.

The following software is required to apply this guideline:

- Linux operating system
- Java 2 Runtime Environment, Standard Edition version 1.5.0 or higher
- Web browser

The evaluation system is designed as a distributed system but for simplification reasons it is assumed that all software is executed on one (Linux) machine. All other necessary software not listed above can be found on the enclosed CD. Additionally a HTTP web application must be provided as a test object which can run on any other system connected to the machine with the evaluation system via a TCP/IP network.

F.1 Installation of the sensors and actuators

For the example given here, the NetSensorLinux sensor and InjectorLinux actuator are used to perform the security test. Both can be found as executable binaries in the CD directory `\Exe\NetSensorLinux\` resp. `\Exe\InjectorLinux\`.

To execute the two programs it is necessary to find unused TCP/IP ports. At this, it is assumed that every port with a port number between `1000` and `1002` is disposable.

First, the NetSensorLinux sensor is executed by invoking the following command:

```
NetSensorLinux 1000 &
```

(Maybe it can be necessary to execute this command as user `root`.)

Secondly, the InjectorLinux is executed by invoking the command:

```
InjectorLinux 1001 1002 &
```

These two actions are necessary to prepare the following steps which involve the evaluation system central control unit.

F.2 Data acquisition with the central control unit

Now everything is prepared to run the central control unit. This program can be found in the CD directory `\Exe\CentralControl\`. It is started by invoking the following command:

```
java -jar "CentralControl.jar"
```

First a component which reads the data from the NetSensorLinux sensor is needed. Therefore a "Net sensor" is created by activating the menu item "Net sensor" in the menu "In/Output". The sensor module is placed on the workplace by moving it to the intended position and pressing the left mouse button.

To view the acquired data, a second module is created from the menu "Visualisation" using the menu item "Session viewer". It is useful to position the "Session viewer" near the "Net sensor" on the workspace.

Both created modules should be connected with a link. It can be created by clicking at the blue output of the "Net sensor". The linking mode becomes active and by clicking at the blue input of the "Session viewer" the link can be completed. By clicking on the free workspace during linking it is possible to set up fix points for the created link route. By pressing the right mouse button these fix points can be removed or the linking mode is left when there is no more fix point available to remove.

At the top of the workspace, there is a panel with four buttons which are described in the following:

- By toggling the "Zoom" button the zoom mode can be activated or deactivated. The left mouse button is used to zoom in, the right button is used to zoom out. The middle mouse button activates the standard view.
- By toggling the "Move Module" button the move mode can be activated or deactivated. By clicking and holding the left mouse button on a module while this mode is activated it is possible to change the position of the module at the workspace.
- By toggling the "Remove Obj" button the remove mode can be activated or deactivated. By clicking at a module or a link while this mode is activated the module or link is removed.
- By toggling the "Change Order" button the execution order editing mode can be activated or deactivated. By clicking at a module using the left mouse button a module can be moved forward in the execution order. By clicking with the right mouse button the module is moved backward in the execution order.

  Forward means in this context that the module is executed later, backward means that the module is executed earlier. Note that the number in the middle bottom side of each module changes during execution order editing.

By clicking at the "Net sensor" module without any other modes activated, it is possible to open its graphical user interface. To connect to the previously started sensor it is necessary to fill out the edit fields "Address" with `localhost` and "Port" with `1000`. By activating the "Start" button the module connects to the NetSensorLinux sensor which is confirmed by the text "active" in the "State" field. Now, the workspace should be similar to the one shown in figure F.1.

The network conversations captured by the sensor are consequently acquired by the "Net sensor". The forwarding of the data to the "Session viewer" by using the created link must be executed in order to examine these data. This is performed by activating the testing algorithm by toggling the "Go" button. By toggling the "Slow Down" button this process can be made visible. The processor time which is spent on the execution is regulated by changing the "loop delay" time.

After starting the algorithm execution every TCP/IP network traffic on the captured network interfaces is available in the central control unit. It can be viewed by opening the graphical user interface of the "Session viewer" and activating the "Refresh" button to create a current view of the data. For the following security test it is useful to create some conversations with the evaluation target (the web application) where parameters are transmitted from the web client to the web server.

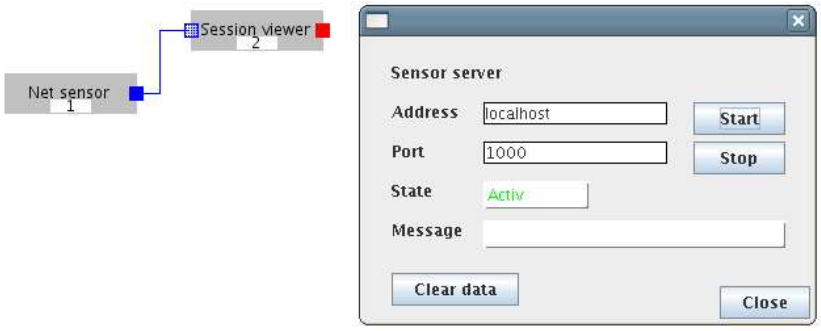


Figure F.1: The first algorithm created with the evaluation system used to acquire sensor data (the graphical user interface of the "Net sensor" module is shown as well)

F.3 Parsing and recompilation of the captured data

To manipulate a captured message it is necessary to parse it in order to work on the resulting structured data abstraction in the following. This feature is demonstrated on a single HTTP message previously captured.

The "Session viewer" module provides one single output where the raw content of one selected single message of the session entered in the module can be assigned as a byte array. This can be done by choosing a message in the "Input/Output blocks" list and activating the "Forward to output" button.

After assigning the message, a "HTTP parser" module from the sub menu "Parsers" in the menu "Data processing" can be placed on the workspace and its byte array input can be connected with the "Session viewer" output. In the same way a "URI-parameter parser" module is created and its green data abstraction input is connected with the data abstraction output of the "HTTP parser". The outputs of the modules can be monitored by creating several "Data explorer" modules from the menu "Visualisation". The workspace should now appear like in figure F.2.

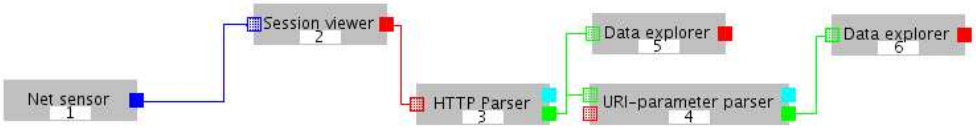


Figure F.2: The parsing of a single message

When opening the graphical user interface of the "Data explorer" modules by simply clicking on the module symbol, the structure of the message assigned to the output of the "Session viewer" module can be explored. Note that it is first necessary to activate the "Go" button to start the message forwarding.

By adding compilers from the sub menu "Compilers" in the menu "Data processing" the message can be reassembled. The first necessary compiler is the "URI-parameter compiler" module. This module has two inputs for a data abstraction. The description of each is shown in the text box in the lower right corner of the graphical user interface of the central control unit which usually outputs the message "No description". One of the inputs has to be connected to the "HTTP parser" module and the other has to be connected to the "URI-parameter parser" module because the parameters transmitted in the URI of the HTTP request have to be inserted into the structure of the HTTP message again.

The whole HTTP request data abstraction is sent again as input in a “HTTP compiler” module which recreates the message raw data as a byte array. With the “Byte array viewer” from the “Viewers” menu, this output can be monitored. It should have the same content like the message which was the input for this processing.

The workspace with the compilers should now be similar to figure F.3

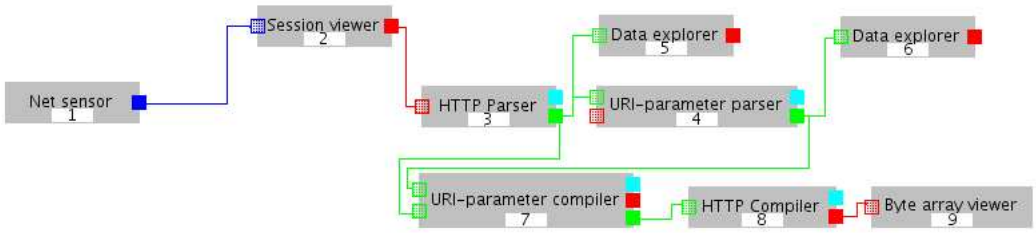


Figure F.3: The parsing and recompilation of a single message

F.4 Creation of the data processing loop

For the next explanations it is first necessary to delete all the modules created in the previous step. Only the modules “Net sensor” and “Session viewer” remain on the workspace which should now appear again like in figure F.1.

Hereafter, the “Net sensor” is connected with a newly created “Session creator” module which can be found in the sub menu “Sessions” in the menu “Data processing”. Additionally a “Session splitter” module has to be added which should be connected with the “Session creator”. The workspace should now be similar to figure F.4.

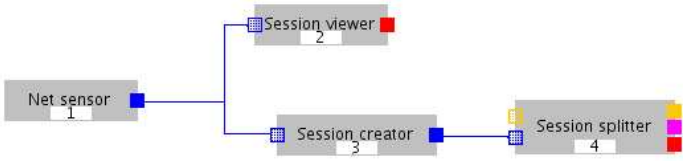


Figure F.4: “Session creator“ and “Session splitter“ module collaboration

In the following the ”Session creator“ is used to create the raw data for the test case injection as a derivation from the previously captured network data. Therefore, the graphical user interface of this module provides capabilities to derive a new session from another one. The upper and the lower part of this dialog is similar to the graphical user interface of the ”Session viewer“. Note that the content of the original session is shown in the upper side and the content of the new session is shown in the bottom side. Elements from the upper side can be added to the new session by using the ”Add“ buttons.

For the next step it is necessary to choose a whole connection from the captured session where some parameter values have been transmitted. By adding them to the new session the base for the further test case data can be created. The output message can be removed from the connection because it is not factored in the further processing (a test object output of an already closed conversation can not be used to create test cases). The new session can be sent to the output of the module by activating the ”Send to module output“ button. Finally, it should contain one single connection with one single HTTP request message.

For the security test it is necessary to continuously create connections which derive from the connection in the newly created session. This should be done with the "Session splitter" module.

The "Session splitter" module requires the input of a trigger to forward the next message to its outputs. In this example, there are two situations in which a trigger should be generated:

- The "Session splitter" itself has forwarded a message in the previous test algorithm run
- The "Session splitter" has received a new session from the "Session creator"

Thus, it is necessary to create two trigger generators from the "Trigger management" sub menu of the "Data processing" menu. The input of the first one is connected with the output of the "Session creator" and the input of the second is connected with the meta data output of the "Session splitter". This structure ensures that when one of the outputs is updated the trigger is generated. Both trigger signals are joined using a "Trigger OR" module which is connected with the trigger input of the "Session splitter".

In the following it is necessary to configure the "Session splitter" by using its graphical user interface:

- The "Loop output data" checkbox should be activated because the test case generation should be infinite.
- The "Fire only on trigger" checkbox should be activated.
- The "Forward inputs/outputs" checkboxes should be activated.
- The "Loop output data as new connections" checkbox should be activated.
- The "Create new connection IDs" checkbox should be activated.

The whole structure should be similar to figure F.5.

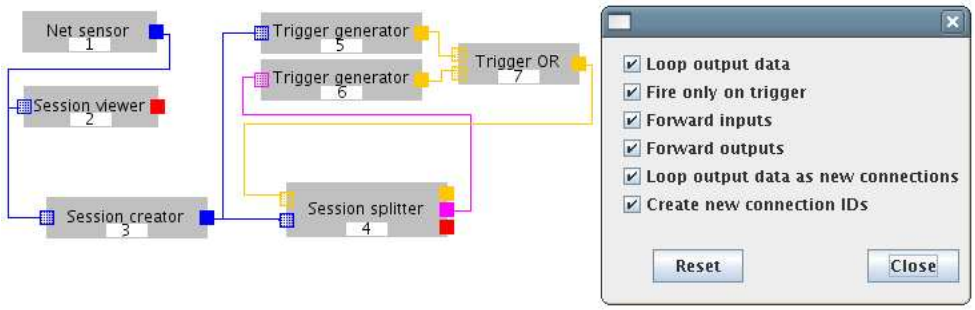


Figure F.5: The "Session splitter" and the trigger modules (the graphical user interface of the "Session splitter" is shown as well)

F.5 Implementation of the test case generation

The parser and compiler structure mentioned in F.3 is now added to the "Session splitter" structure with one small modification: An "Element injector" module is inserted between the compilers and the parsers in the data flow to allow a mutation of the test case data (this module can be found in the "Data processing" menu). Note that the execution order of the modules symbolized by the number in the bottom side of each module pictogram is important and should be similar to the one in figure F.6. To change it, see also F.2.

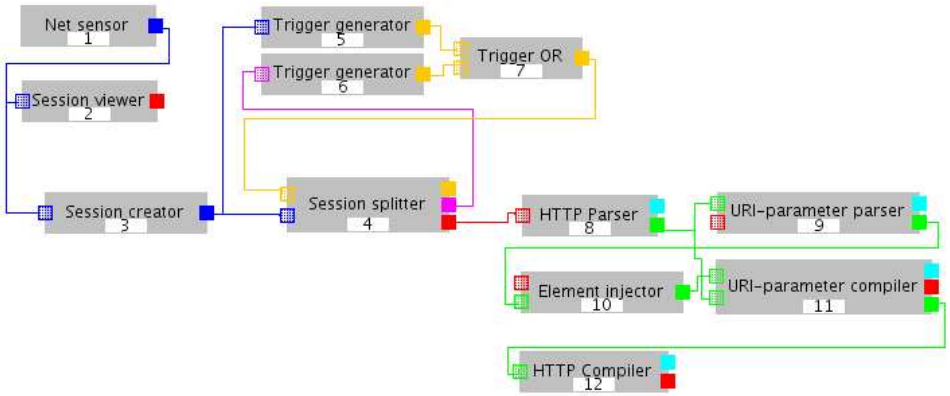


Figure F.6: The test case generation added to the "Session splitter"

In the next step the graphical user interface of "Element injector" module has to be opened and the "random injection" option should be activated. In this context, that means that the parameter which is manipulated in each test case is chosen randomly.

To complete the test case injection some more steps are necessary. First some data are needed to be injected as parameter values of the web application through the "Element injector" module. Thus, a "Simple random generator" from the sub menu "Random generators" of the menu "Data processing" is created and its output is connected with the input of the "Element injector". Additionally, a trigger input is necessary to perform the generation of the next random pattern. At this, the easiest way is to connected the input of the "Simple random generator" with the output of the "Trigger OR" module. The generator parameters can be changed by using the graphical user interface of the "Simple random generator".

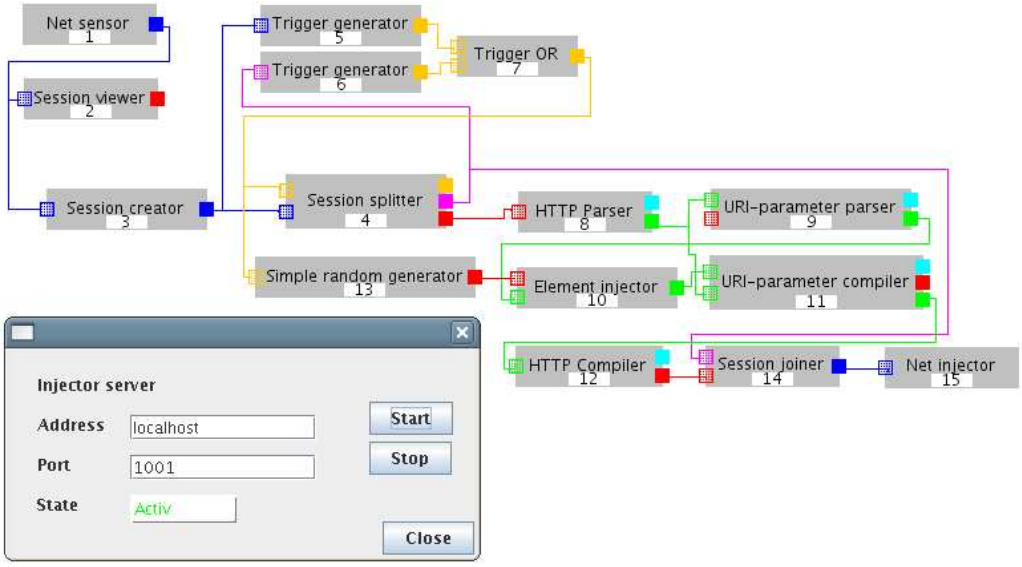


Figure F.7: Final test case invocation structure

Furthermore, to recreate a single session from the different messages created by the "Session splitter" loop it is necessary to add a "Session joiner" module. This module is fed with message raw data and the corresponding meta data and uses this information to produce a new session. Thus, the inputs of this module should be connected with the meta data output of the "Session splitter" module and the byte array output of the "HTTP compiler" module. This step completes

the creation of the test case data.

Finally it is necessary to send the data to the test object. This is performed by placing a "Net injector" module at the workspace which is connected with the "Session joiner". It should connect with the InjectorLinux actuator which has been created in F.1. The configuration and the final structure are shown in figure F.7.

By activating the whole algorithm with the "Go" button the test cases are created and invoked continuously. If there is no activity in the test algorithm it is necessary to generate one single trigger impulse by hand by clicking at one of the trigger generators. During the execution it is reasonable to change the parameterisation of the "Simple random generator" in order to change the characteristics of the injected pattern.

It is also possible to apply many mutations on one message by stringing many "Element injectors" together whereas the injected pattern of each injector module can differ.

The responses of the test object can be acquired by a "Net sensor" module. As also mentioned in F.1, the InjectorLinux actuator provides a sensor server which listens on port `1002`. Thus another "Net sensor" module can be created and connected to this port. The resulting web application answers can be viewed by creating another "Session viewer" or can be analyzed when a failure detection criterion is known.

It makes sense to experiment with the given algorithm to become familiar with the evaluation system and to develop other software security testing algorithms.

G Declaration of authorship / Selbstständigkeitserklärung

I hereby declare that in accordance with subsection (6) of section 10 of the examination regulations of September 12th, 1997 for the course of studies "Applied Computer Science" at the Chemnitz University of Technology the present diploma thesis was made by my self. Only the aids specified in the diploma thesis have been used. All parts which are taken over word-to-word or analogous from literature and other publications are quoted and identified.

In Übereinstimmung mit Abschnitt 6 des Paragraphen 10 der Prüfungsordnung vom 12. September 1997 für den Studiengang "Angewandte Informatik" an der Technischen Universität Chemnitz erkläre ich hiermit, dass die vorliegende Diplomarbeit selbstständig von mir verfasst wurde. Alle verwendeten Hilfsmittel wurden angegeben. Textabschnitte die wortwörtlich oder analog aus der Literatur oder anderweitigen Publikationen übernommen wurden, wurden eindeutig als solche gekennzeichnet.

Andreas Poller

Digital unterschrieben von Andreas Poller
DN: cn=Andreas Poller, c=DE, o=Fraunhofer, ou=SIT, People
Grund: I am the author of this document
Speicherort: Darmstadt, October 23, 2006
Datum: 2006.10.23 15:00:44 +02'00'

Andreas Poller
Darmstadt, October 23, 2006