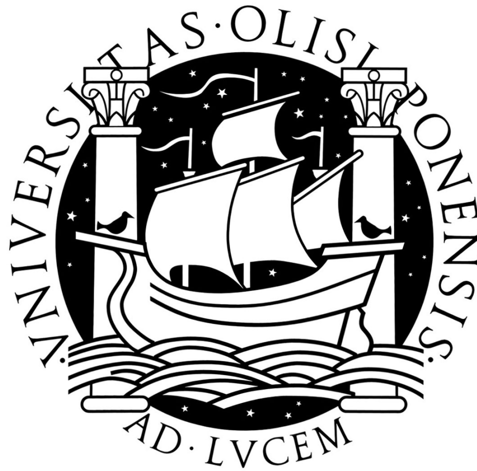
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



VULNERABILITY ASSESSMENT THROUGH ATTACK INJECTION

João Alexandre Simões Antunes

MESTRADO EM INFORMÁTICA

September 2006

VULNERABILITY ASSESSMENT THROUGH ATTACK INJECTION

João Alexandre Simões Antunes

Dissertação submetida para obtenção do grau de

MESTRE EM INFORMÁTICA

pela

Faculdade de Ciências da Universidade de Lisboa

Departamento de Informática

Orientador:

Nuno Fuentecilla Maia Ferreira Neves

Júri:

Henrique Santos do Carmo Madeira

Isabel Batalha Reis Gama Nunes

Miguel Nuno Pupo Correia

September 2006

Resumo

A nossa dependência nos sistemas computacionais para a realização das mais diversas actividades tem crescido ao longo dos anos. O progressivo aumento da complexidade dos problemas tratados também requer o desenvolvimento de soluções mais elaboradas. As aplicações tendem por isso a possuir uma maior dimensão e complexidade. Por outro lado, o sempre presente compromisso entre a realização de testes minuciosos e o lançamento no mercado afecta a qualidade do software. Por isso, as aplicações acabam por ser disponibilizadas ao público antes de terem sido suficientemente testadas. Os erros de software são continuamente detectados *à posteriori*, muitas vezes resultando em vulnerabilidades de segurança que poderão ser exploradas por adversários maliciosos, e assim comprometer a segurança do sistema. A descoberta de vulnerabilidades de segurança é assim uma actividade indispensável na construção de sistemas confiáveis. A ferramenta AJECT é apresentada como uma ferramenta para a aferição de vulnerabilidades, sem recurso ao código fonte nem a uma base de dados de vulnerabilidades actualizada. A metodologia por detrás desta ferramenta baseia-se na simulação das acções dos adversários, utilizando a injecção de ataques como meio de despoletar e detectar comportamentos anormais nas aplicações-alvo. Os resultados experimentais preliminares com servidores IMAP mostraram que o AJECT conseguiu não

só descobrir todas as vulnerabilidades publicadas no ano de 2005, como também uma desconhecida.

PALAVRAS-CHAVE: Injecção de ataques, injecção de faltas, testes de caixa preta, detecção de vulnerabilidades e supervisão de processos.

Abstract

Our reliance on computer systems for everyday life activities has increased over the years as more and more tasks are accomplished with their help. The increasing complexity of the problems they address also require the development of more elaborated solutions. So, applications tend to become larger and more complex. On the other hand, the ever present trade-off between time to deployment and thorough testing puts pressure on the quality of the software. Hence, applications tend to be released with little testing. Software bugs are continuously detected afterwards, resulting in security vulnerabilities that can be exploited by malicious adversaries and compromise the systems' security. The discovery of security vulnerabilities is then a valuable asset in the development of dependable systems. AJECT is presented as a new tool for vulnerability assessment, without requiring access to the source code or to any updated vulnerability database. The methodology utilized in the construction of AJECT emulates the behavior of an adversary by injecting attacks to trigger and detect abnormal behavior in the target systems. Preliminary experimental results in IMAP servers showed that AJECT was able to discover not only all known vulnerabilities, but also a previously unknown one.

KEY WORDS: Attack injection, fault injection, backbox testing, vulnerability detection, and process monitoring.

Acknowledgments

I would like to express my gratitude to my advisor, Professor Nuno Ferreira Neves, whose expertise, understanding, and patience helped and guided me in this starting academic journey. I appreciate his vast knowledge and skill in many areas, and his ever-present support and assistance, without which the quality of this thesis would certainly be affected.

I would like to thank the other members of the Navigators research group, namely, Professors Paulo Veríssimo, Miguel Correia, and António Casimiro, from whom I am still learning so much. This is also true for all remaining Professors of the Department of Informatics, whom so notably and capably pass the technical knowledge to the upcoming generations.

I would also like to extend my thanks to my present and past colleagues at the Large-Scale Informatics Systems Laboratory (LASIGE) research laboratory for their friendship and support. Thanks Henrique, Paulo, Jieke, Marco, Rui, Hugo, Leonardo, Bruno, Daniel, Nuno Cardoso, Marcírio, Liliana, Nuno Carvalho, Susana, Odorico, Giuliana, Fábio, and Alysson.

I must also thank my friend and girlfriend, Mara Santos, for the patience, understanding, and support she has demonstrated whenever it was required to sacrifice our time together and the attention she certainly deserved.

A final word of thanks to the institutions that supported the research work that culminates with this thesis: the FCT through the project POSC/-EIA/61643/2004 (AJECT), and the LASIGE and the EU by the project IST-2004-27513 (CRUTIAL).

Lisboa, September 2006

João Alexandre Simões Antunes

Dedico esta tese ao meu melhor amigo e pai, a quem tanto devo como exemplo pessoal de vida, e que sempre me apoiou e incentivou a ir mais além.

Obrigado, pai.

Contents

Contents i

List of Figures v

List of Tables vii

List of Pseudocodes ix

1 Introduction **1**

1.1 Contribution 5

1.2 Document Organization 7

2 Related Work **9**

2.1 Fault Injection 11

2.1.1 Hardware Fault Injection 11

2.1.2 Software Fault Injection 15

2.1.3 Simulation Model Tools 20

2.2 Static Vulnerability Analyzers 23

2.2.1 Lexical Analysis 23

2.2.2 Type Checking 24

2.2.3 Control-flow Analysis 25

2.2.4 Data-flow Analysis 26

2.2.5 Model Checking 28
2.3 Fuzzers 31
2.3.1 Fuzzer's Knowledge 32
2.3.2 Fuzzer's Specialization 32
2.4 Run-time Prevention Mechanisms 33
2.5 Vulnerability Scanners 36
2.5.1 Generations 37
2.5.2 Host- and Network-based Scanners 38

3 Attack Injection Architecture 41
3.1 Using Attacks to Find Vulnerabilities 43
3.2 AJECT's Design 48
3.3 Test, Attack and Packet Hierarchy 51
3.4 TPS Component 52
3.5 Injector Component 53
3.6 Monitor Component 54
3.7 Test and Attack Analyzer Components 55

4 Implementation of the Tool 57
4.1 TPS Component 60
4.2 Injector Component 63
4.3 Monitor Component 67
4.3.1 Signal Monitoring 70
4.3.2 Resource Monitoring 71
4.4 Synchronization Protocol 72
4.5 Attack Tests 75
4.5.1 Delimiter Test 76
4.5.2 Syntax Test 77

4.5.3 Value Test 78
4.5.4 Privileged Access Violation Test 79

5 Evaluation **81**
5.1 Experimental Framework 83
5.1.1 IMAP Protocol 83
5.1.2 Testbed and Implementation Issues 85
5.2 Experimental Results 86
5.2.1 Applications Under Test 87
5.2.2 Vulnerability Assessment 88
5.2.3 Test Results 93

6 Conclusion **97**
6.1 Future Work 100

Bibliography **103**

List of Figures

2.1 MESSALINE's architecture. 13
2.2 RIFLE's architecture. 14
2.3 Xception's architecture. 17
2.4 FTAPE's architecture. 17
2.5 GOOFI's architecture. 19
2.6 DEPEND's architecture. 21
2.7 VERIFY's architecture. 22

3.1 Fault, error, and failure. 44
3.2 Composite fault model (attack, vulnerability, and intrusion). 46
3.3 The architecture of the AJECT tool. 49

4.1 Example for the IMAP protocol finite state machine. 60
4.2 UML class diagram for the ProtocolSpecification class. . . . 61
4.3 UML class diagram for the `DataValues` and `DataType` classes. 63
4.4 UML class diagram of the Injector component. 64
4.5 UML class diagram of the Monitor component. 67
4.6 Synchronization protocol messages. 73
4.7 Synchronization protocol between the Injector and the Monitor. 74

5.1 IMAP state and flow diagram. 84

List of Tables

5.1 Applications with vulnerabilities. 88
5.2 Attacks generated by AJECT to detect IMAP vulnerabilities. 90
5.3 Commands tested in each IMAP state. 92
5.4 Syntax test attacks sample. 94

List of Pseudocodes

4.1 Function run() of the Test class. 66
4.2 Monitor component 69

Chapter 1

Introduction

Software evolved over the years. It became more useful and easier to make, but on the other hand it also became more complex. Applications suffered dramatic improvements in terms of the offered functionality. These enhancements were achieved in many cases with bigger software projects, which can no longer be solved by a single person or a small team. As a consequence of this increase in size and complexity, software development frequently involves several teams that need to cooperate and coordinate efforts. Also, to speedup the programming tasks, most projects resort to third party software components (e.g., a cryptographic library, a PHP module, a compression library), which in many cases are poorly documented, unsupported, and sometimes contain security vulnerabilities.

In addition, the competitive software market requires applications to be deployed with full functionality as soon as possible. This ever present trade-off between time to deployment and thorough testing, affects the quality of the dispatched software. Hence, applications tend to be released with basic testing and software bugs are continuously corrected afterwards. These software bugs have also evolved and are more sophisticated, leading to many new and different ways in which software can be exploited.

The nature of the software and the reliance we place in it, makes us vulnerable to deviations from its correct behavior, in such diverse areas as e-government, health services, and critical control infrastructures. Dependability in this systems is of paramount importance and a security compromise potentially catastrophic.

Besides bad programming, there are other sources of vulnerabilities. In particular post-development errors, such as installation or configuration mistakes during the operational phase, allow for instance the disclosure

of secret information stored in the system (e.g., a password file). The interaction between software components is also another source of security-related problems.

Statistics published by CERT demonstrate that the number of reported vulnerabilities is not decreasing — in 2004 there were 3780 reported vulnerabilities, which correspond to a 346% increase in relation to the year 2000, or around 1095% when compared to 1996 (CERT, 2005).

The existence of a vulnerability *per se* does not cause a security hazard, and in fact they can remain dormant for many years. An intrusion is only materialized when the right attack is discovered and applied to exploit a particular vulnerability. After an intrusion, the system might or might not fail, depending on its capabilities in dealing with errors introduced by the adversary. Sometimes the intrusion can be tolerated (Veríssimo et al., 2003), but in the majority of the current systems, it leads almost immediately to the violation of its security properties (e.g., confidentiality, or availability).

Several tactics can be employed to improve the system's dependability with respect to malicious faults (Powell & Stroud, 2002). If one could develop error-free software, vulnerabilities would not exist and dependability would surely be increased. Actually, without vulnerabilities, applications could not be exploited. So, theoretically, if one devises methods and means to remove these vulnerabilities or even prevent them from appearing in the first place, we could create much more dependable software.

Vulnerability removal can be performed both during the development and operational phases. In the last case, besides helping to identify programming flaws which can later be corrected, it also assists the discovery of configuration errors. Intrusion prevention, such as vulnerability

removal, has been advocated because it reduces the power of the attacker (Veríssimo et al., 2003). In fact, even if the ultimate goal of zero vulnerabilities is never attained, vulnerability removal reduces the number of entry points into the system, making the life of the adversary increasingly harder (and ideally discouraging further attacks).

1.1 Contribution

This thesis proposes attack injection with extended monitoring capabilities as a method for detecting vulnerabilities. This method tries to detect software bugs as an attacker would, i.e., trial and error, by consecutively attacking its target. Attack injection does not depend on a database of known vulnerabilities, but it rather relies on a more generic set of tests. Through careful and automated monitoring, the results of the attacks can be later analyzed to pinpoint the exact vulnerability. This allows the detection of known and unknown vulnerabilities in an automated fashion.

This reasearch resulted in the development and evaluation of a vulnerability assessment tool called AJECT — *Attack in**JEC**tion **T**ool*. AJECT can be used for vulnerability detection and removal by simulating the behavior of an adversary. It injects attacks against a live system while observing its execution to determine if the attacks have caused a failure. In the affirmative case, this indicates that the attack was successful, which reveals the existence of a vulnerability. After the identification of the flaws, traditional debugging techniques can be employed to examine the application code and running environment, to find out the origin of the vulnerabilities and allow their subsequent elimination.

At this time, only servers were chosen for vulnerability removal be-

cause, from a security perspective, they are probably the most relevant components that need protection. They constitute the primary contact points of a network facility — normally, an external hacker can only enter into the facility by connecting to a server. Moreover, if an adversary compromises a server, she or he immediately gains access to a local account, which can then be used as a launch pad for further attacks. Although this work only accounts for remote services (e.g., servers or daemons), this vulnerability detection methodology can also be applied to any type of software application involving user input such as network data, command arguments, configuration or other loadable files (e.g., local daemons, command-line programs, web browsers, e-mail clients, etc.).

AJECT performs black box testing, so it does not require access to the source code to perform the attacks. However, in order to be able to generate intelligent attacks, AJECT has to obtain a specification of the protocol implemented by the target server (e.g., IMAP protocol specification for IMAP mail servers or HTTP protocol specification in case of web servers).

To demonstrate the usefulness of this approach, several attack injection campaigns were conducted against different IMAP servers. IMAP is a popular method for accessing electronic mail on remote servers. Users that need to manage email accounts from different machines resort to IMAP because it allows the manipulation of remote message folders in a way functionally equivalent to local folders. The main objective of the experiments was to show that AJECT could automatically discover a number of different vulnerabilities, which were described in bug tracking sites by various people. The tests managed to confirm that the tool could be used to detect many IMAP vulnerabilities, and also the discovery of a new vulnerability that was previously unknown to the security community.

1.2 Document Organization

This thesis is organized as follows. The next chapter gives some insight about the areas of research that influenced and contributed to the work presented here, such as fault injection, static vulnerability analyzers, and fuzzers. Chapter 3 describes the architecture of AJECT and its main components. The chapter also provides the methodology details for vulnerability detection. AJECT's implementation details are provided in Chapter 4. The experimental context and results are presented in Chapter 5, namely: a brief explanation of the IMAP protocol, the class tests used in the experiments, some details on the experimental testbed, and the results of the injection campaigns. The thesis ends with some conclusions and ideas for future work.

Chapter 2

Related Work

The thesis presented here describes a methodology for the discovery of vulnerabilities on services provided by network or local daemons, and a tool that implements those ideas. This work has been influenced by several research areas, such as fault injection, static vulnerability analyzers, fuzzers, run-time prevention mechanisms, and vulnerability scanners.

2.1 Fault Injection

Fault Injection is an experimental approach for the verification of fault handling mechanisms (fault removal) and for the estimation of various parameters that characterize an operational system (fault forecasting), such as fault coverage and error latency (Arlat et al., 1993).

Traditionally, fault injection has been utilized to emulate several kinds of hardware faults, ranging from transient memory corruption to permanent stuck-at faults. Faults can be either injected at the hardware-level (logical or electrical faults) or at the software level (code or data corruption) (Hsueh & Tsai, 1997). Hardware-implemented fault injection uses additional hardware to physically introduce faults into the target system's hardware, whereas software-implemented fault injection can be used to target applications, or even the operating system, through additional software instructions.

2.1.1 Hardware Fault Injection

This type of fault injection tries to provoke the effects of real faults at the hardware level. They can be either injected *with* or *without* physical contact.

Hardware fault injection with contact

Probably, the most common type of hardware-implemented fault injection is done with direct physical contact with the target system, often called *pin-level injection* (Crouzet & Decouty, 1982; Lala, 1983; Martínez et al., 1999, among others). Examples of this method use pin-level probes and sockets in order to alter the electrical current and voltage at the pins. Both of these methods provide good controllability of fault times and locations with little or no perturbation to the target system.

The injection of faults at the pin-level is not identical to traditional stuck-at and bridging fault models that generally occur inside the chip. Nonetheless, the same effects, such as the evaluation of error detection circuits, can be achieved.

In the simplest way, the electrical current of the pins is changed by some special probes attached to them. Though simpler, these active probes are only capable of generating stuck-at faults or bridging faults (by placing a probe across two or more pins). Active probes attached to the power supply hardware inject electrical disturbance faults. However, this technique risks damaging the injected device. Another method for inserting faults at the pin-level consist of inserting a socket between the target hardware and its circuit board. This socket then forces analog signals, representing the desired logic values, onto the pins. More complex logic faults can be achieved by composing the pin signals with adjacent ones or even with previous signals on the same pin.

MESSALINE (Arlat et al., 1989) is a fault injection tool capable of adapting to various target systems. This flexible tool performs physical fault injection at the pin-level, both through active probes and socket insertion. The former uses the *forcing* technique in which the fault is directly applied

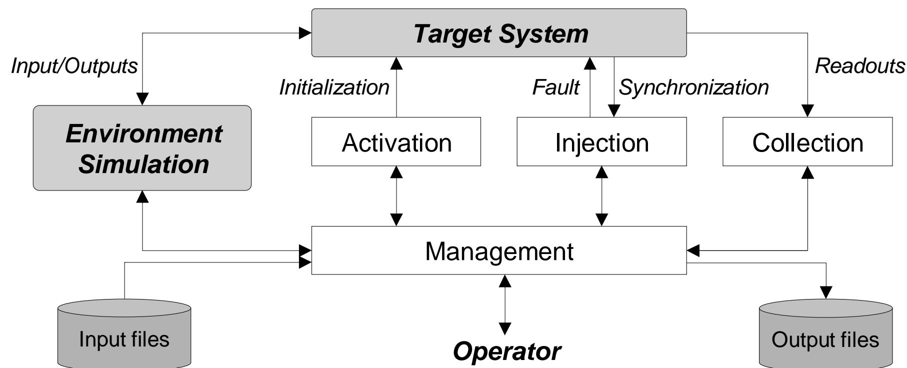


Figure 2.1: MESSALINE's architecture.

by means of multipin probes on the integrated circuit (IC) pins. The *insertion* technique is employed in the latter, by placing the IC under test on a specific box where transistors switches ensure its proper isolation. Current sensing (forcing technique) or comparison of signals on both sides of the switch (insertion technique) are used to identify the activation of the fault, i.e., the occurrence of an error. This detection also ensures that the injected faults are actually activated, and is useful in measuring their dormancy.

The components of the MESSALINE tool are presented in Figure 2.1. The tool's adaptation to a target system is accomplished through a board description file (that specifies the functional interconnections of the ICs on a board), an IC library (that describes the number and type of pins), and a injection devices description (describing the relationship between each injection element and each pin of the injection devices).

RIFLE (Madeira et al., 1994) combines trigger and tracing techniques traditionally used in digital logic analyzers and can inject almost all types of pin-level faults. Figure 2.2 shows the general architecture of the tool. The target IC is replaced by a piggy-back circuit, which holds the IC itself and intercepts the pin signals. The fault injection is triggered when some predefined unique conditions arise.

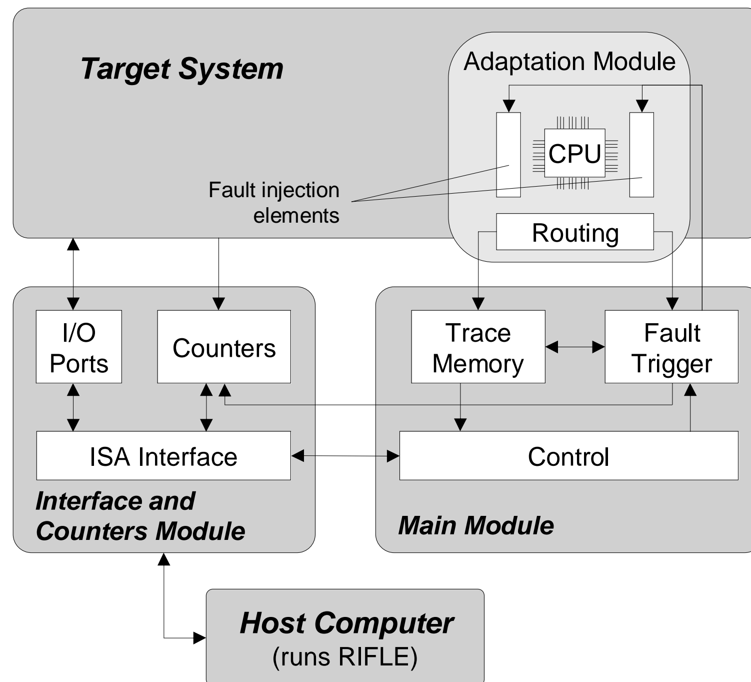


Figure 2.2: RIFLE's architecture.

The fault injection module also controls the recording of the tracing information, needed for a complete characterization of each fault. This monitoring allows RIFLE to receive error feedback (i.e., to detect whether the injected fault has produced an error) and detailed behavior analysis (i.e., the impact of the fault in the target system). The analysis of the tracing information is automatically performed after the injection of each fault.

Hardware fault injection without contact

This type of fault injection tries to mimic some natural physical phenomena by causing electrical interference on the target hardware. One way to create such anomalies can be accomplished by using heavy-ion radiation. An ion passes through the depletion region of the target device and generates electrical current. Creating an electromagnetic field in or near

the target device also injects faults. However, the exact moment of heavy-ion emission or electromagnetic field creation cannot be predictably controlled. Hence, these techniques have low controllability of the time and locations of the faults.

FIST (Gunneflo et al., 1989) is a fault injection system for studying transient faults effects on ICs. It uses heavy-ion radiation from a ^{252}Californium source to inject single event upsets (i.e., bit-flips at internal locations in ICs). The heavy ions emitted from the source are highly ionizing particles capable of creating transients when they pass through a depletion region on the IC. When a heavy-ion penetrates a semiconductor material, it creates a track of electron-hole pairs. The resulting high electric field will then cause the deposited charge to induce pulse in the associated circuits. If the particle hits a sensitive region in a memory element, the current pulse will change the state of that memory element. The same workload is executed and analyzed on two different processors: the radiated processor and a golden chip, not submitted to radiation. The comparison of the output signals from the two circuits during each clock cycle reveals the errors (i.e., the manifestation of the fault). However, the heavy-ion technique does not gives the ability to control the time or location of the injection faults with great precision.

Later developments also studied the application of such fault injection system in more complex VLSI circuits (Karlsson et al., 1995, 1994).

2.1.2 Software Fault Injection

Software-implemented fault injection (SWIFI) can be used to target applications or even operating systems. It provides a low cost and easily controllable alternative to physical fault injection. SWIFI is less expensive

because it does not require special hardware instrumentation to inject the faults. SWIFI is usually achieved by changing the contents of memory or register based on specified fault models to emulate the consequences of hardware faults (e.g., see Arlat et al., 2003). And though it can only inject faults into locations that are accessible to software (e.g., by altering bits in instructions, address pointers or data, or by changing whole sequences of instructions), it can emulate the error behavior caused by most faults. However, changing the target system itself (e.g., by adding special instructions, software traps) can disturb the workload of the system.

Several techniques for injecting software-implemented faults exist. They can be injected in run-time, or during the compilation.

Run-time injection

This technique requires that special conditions arise during the execution of the target system to trigger the injection of the fault. The injection is done by either modifying the system's interrupt handler vector, or by actually inserting special instructions in the target program. In the former mechanism, the injection can be activated after the expiration of a predetermined timeout or if a particular event occurs, such as an access to a specific memory address. When those conditions arise, the execution is transferred to the exception/trap handler, which injects the fault. In the latter technique, the actual instructions that inject the fault are inserted in the target program.

Figure 2.3 depicts the fault injection architecture of Xception (Carreira et al., 1995, 1998). This tool resorts to the advanced debugging and monitoring capabilities present in modern processors. The injection of the faults is achieved by modifying the interrupt handler vector. Xception uses the

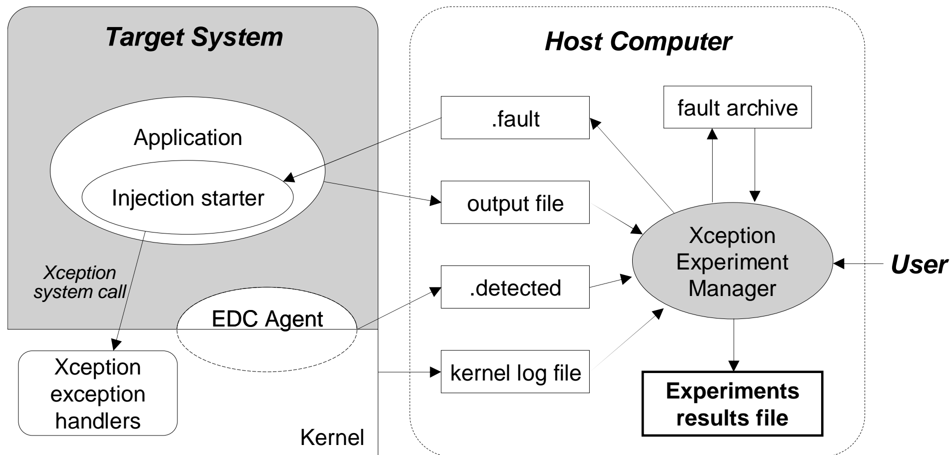


Figure 2.3: Xception's architecture.

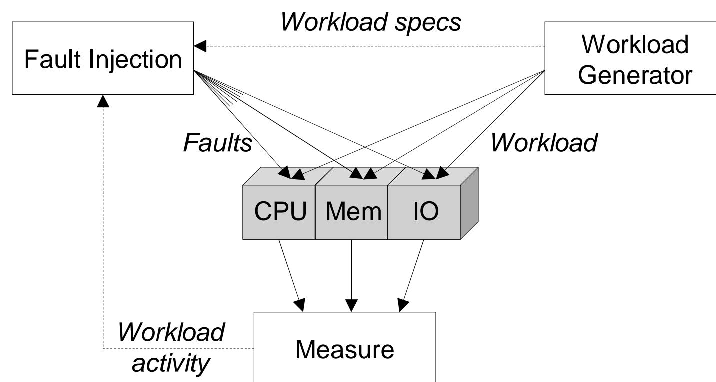


Figure 2.4: FTAPE's architecture.

processor's hardware exception triggers, based on access to specific memory addresses, to activate the injection. This method alleviates the need to insert software traps or modify the application software.

FTAPE (Tsai & Iyer, 1996, 1995) is a fault tolerant and performance evaluator tool. By combining the knowledge of the workload activity and stress-based injection, the tool can achieve a high level of fault propagation. The tool then measures the fault/error ratio, number of failures, and performance degradation of the system under test.

The block diagram of FTAPE is presented in Figure 2.4. The tool is

composed by a workload generator, a fault injector, and a measure module. Software-implemented fault injection is used to emulate the effects of underlying physical faults (CPU, memory, or I/O faults). The workload generator gives FTAPE an easily controllable workload that can propagate the injected faults, such as repeated CPU intensive tasks, large memory usage, and continuous file system access. And finally, the measure module, which is necessary to determine the actual activity caused by the workload.

Compile-time injection

To perform fault injection at compile-time, the program instructions must be changed before the program image is loaded and executed. The modified instructions will cause the injection of the fault. When the program, and the altered instructions, are executed, the fault is activated. This injection method is very simple since all faults are hard-coded into the target program, thus causing no perturbation during execution. However, it requires access to the source code, or assembly code, of the target system. Also, special attention must be taken in order to know *which* particular instructions to modify.

Another flexible tool for fault injection is GOOFI (Aidemark et al., 2001). This tool is designed to be adaptable to various target systems and is capable of different fault injection techniques. The architecture of the tool is presented in Figure 2.5. New fault injection techniques can be added to the tool by providing the respective fault injection algorithm. Also, for each supported target system there is a target system interface. GOOFI supports software-implemented fault injection at compile-time. Using this technique, the tool injects the faults into the program and data areas of the

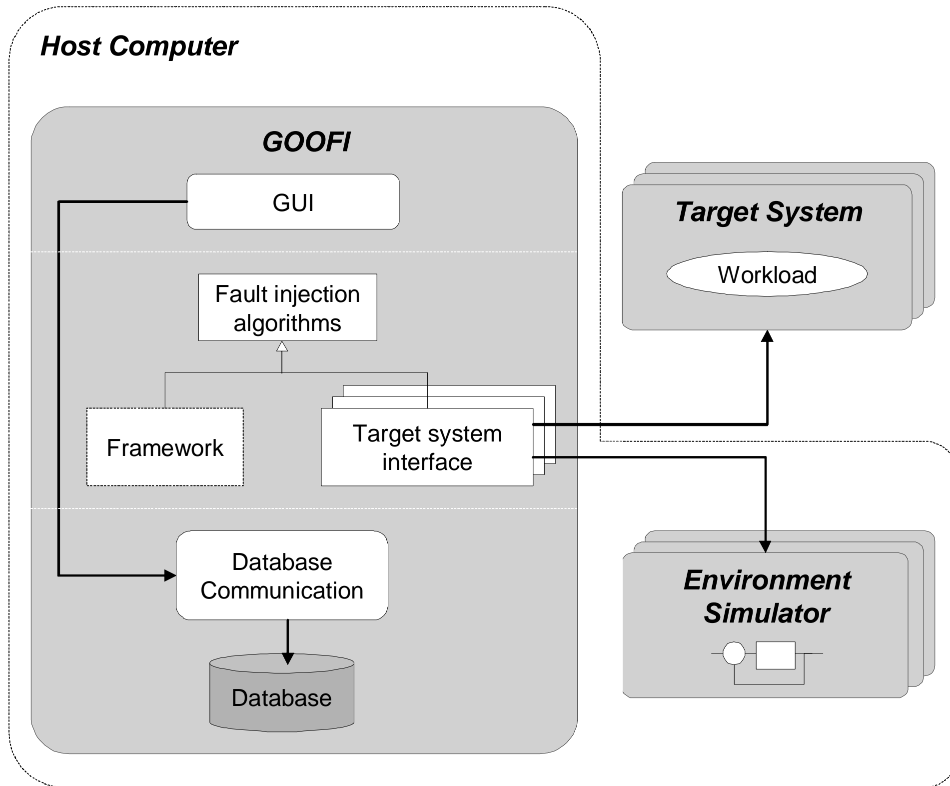


Figure 2.5: GOOFI's architecture.

target system before its execution. The tool also supports scan chain implemented fault injection, which enables faults to be injected into the pins and many of the internal state elements. This method uses the built-in test logic present in many modern VLSI circuits, primarily intended for testing ICs or printed circuits boards.

The emulation of other types of faults has also been accomplished with fault injection techniques, for example, software and operator faults (Brown et al., 2002; Christmansson & Chillarege, 1996; Durães & Madeira, 2003). Robustness testing mechanisms study the behavior of a system in the presence of erroneous input conditions. Their origin comes both from the software testing and fault-injection communities, and they have been applied to various areas, for instance, POSIX APIs and device driver interfaces

(Albinet et al., 2004; Koopman & DeVale, 1999).

2.1.3 Simulation Model Tools

The physical injection techniques discussed earlier are adequate to evaluate the dependability of small systems, such as hardware components, or processors. However, the current trend of integrating more and more components on-chip makes it difficult for pin-level injection to cover the internal faults adequately. SWIFI can be used against more complex systems, but it requires access to a prototype of the hardware for which the faults have to be examined. So, hardware description models were developed and simulated in functional simulations tools, which could be employed to study the behavior of systems, starting from the early stages of design (Choi & Iyer, 1992; Clark & Pradhan, 1993; Goswami et al., 1997; Hein & Goswami, 1995; Jenn et al., 1994; Kumar et al., 1994).

The main advantage of functional simulation is that it can model the behavior of hardware and software architectures with greater accuracy, without any target prototype or any special evaluation hardware. These simulation models can provide different levels of abstraction (e.g., hardware device, network, I/O subsystems), with great observability of all the modeled components. Realistic fault scenarios can be accommodated by injecting faults onto the simulated model and analyzing its effects.

DEPEND (Goswami & Iyer, 1990; Goswami et al., 1997) is an integrated simulation environment for the design and dependability analysis of fault tolerant systems. Fault tolerant architectures can be easily modeled in DEPEND and subject to extensive fault injection studies. Figure 2.6 shows the simulation environment of the tool. DEPEND can model hardware and software components at the functional level. An library of objects is used

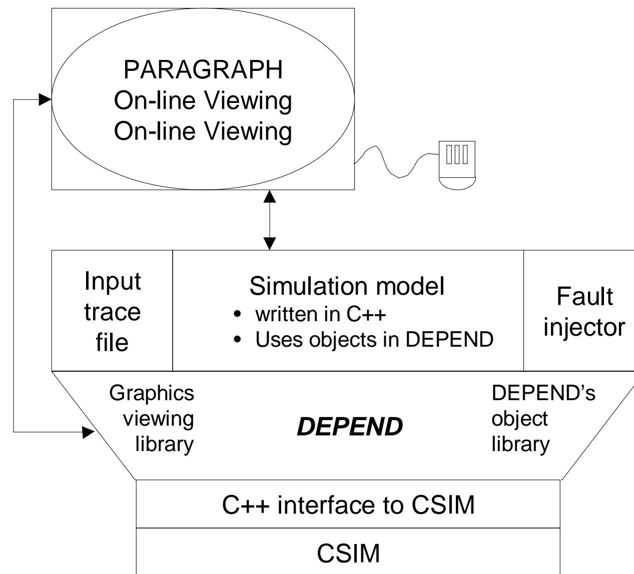


Figure 2.6: DEPEND's architecture.

to simulate hardware components (e.g., CPUs, communication channels, and disks), whereas software components are modeled by C++ routines, written by the user. The simulation environment of DEPEND is based on a process-based simulation language, CSIM (Schwetman, 1986), whereas the system behavior is described by a collection of processes that interact with each other. This process-based approach is an effective way to model the system, its behavior, its repair schemes, and the software in detail. It eases the inter-component modeling, their dependencies and interactions, specially in more complex and larger systems. Also, it allows the execution of real programs inside the simulation tool.

The user writes the system model in C++ using the object library. The model is then executed in the process-based simulation environment where the assortment of objects, including the fault injector, CPUs, and communication links, execute simultaneously to simulate the functional behavior of the architecture. The faults are injected according to some predefined

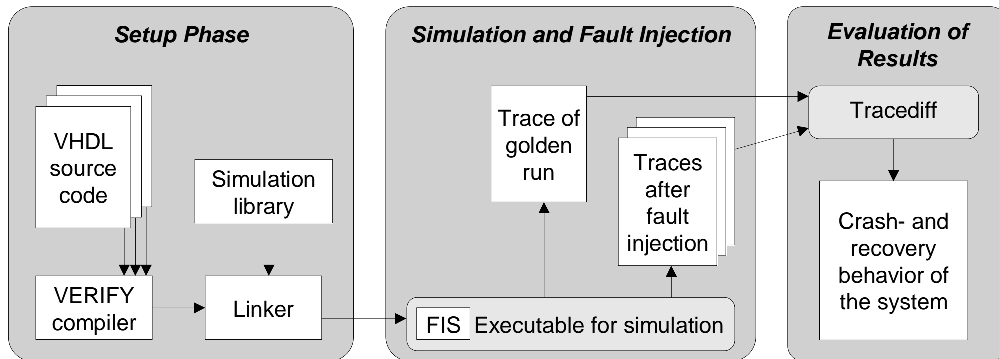


Figure 2.7: VERIFY's architecture.

schemes (i.e., at constant rate, based on an exponential distribution, or at high probability under heavy workload). Statistics of the behavior of the simulated system, such as the repair mechanisms, are then collected.

VERIFY (Sieh et al., 1997) is simulation based fault injector, that extends the capabilities of the VHDL language. VHDL (Ashenden, 1990) is a very well established hardware description language for building ICs. However, it lacks some important features related to the analysis of the system's fault tolerance and dependability, such as reliability parameters checking. Being VHDL-based, VERIFY is able to cover all level of abstractions. This allows the system designers to use a single language for designing and testing the functional and temporal properties of the system.

Figure 2.7 gives an overview of the different phases and components of VERIFY. The tool possesses a specific compiler that supports some VHDL extensions, and a simulator environment for running the fault injection experiments. The fault parameters are extracted from the VHDL source code by the compiler and supplied to the simulator. The resulting executable, with fault injection signals (FIS), is then executed on the simulator. A golden run is performed without any fault injection and compared against the state of the faulty runs.

2.2 Static Vulnerability Analyzers

Static vulnerability analyzers look for potential vulnerabilities in the source code of the applications. The majority of these works has focused on the most common type of vulnerability, the buffer overflow. Typically, these tools examine the source code for a fixed set of dangerous patterns, or rules, and then provide a listing of their locations (Haugh & Bishop, 2003; Larochelle & Evans, 2001; Viega et al., 2000; Wagner et al., 2000). Then, the programmer only needs to go through the warned parts of the code to determine if a problem actually exists. More recently, this idea has been extended to the analysis of binary code (Durães & Madeira, 2005). Static analysis has also been applied to other kinds of vulnerabilities, such as race conditions during the access of (temporary) files (Bishop & Dilger, 1996). A comprehensive list of static analyzers can be found in (Chess & McGraw, 2004). A few experiments with these tools have been reported in the literature, showing them as quite effective in locating programming problems. However, if a pattern, or rule, has not been written to find a particular problem, the tool will never find that problem. Also, these tools have the limitation of producing many false warnings.

2.2.1 Lexical Analysis

This is one of the simplest form of static code checking. These tools usually look for unsafe library functions or system calls, such as *gets()* or *strcpy()*. The source code, fed into the analyzer, is parsed and tokenized. The resulting tokens are then matched against a database of dangerous constructs.

ITS4 (Viega et al., 2000) is tool a for statically scanning C and C++ source code for security vulnerabilities. This simple tool tries to automate

a lot of the grepping, usually done by hand, when performing security audits. ITS4 uses a vulnerability database of dangerous patterns, such as functions susceptible to buffer overflows. The tool parses the source code into lexical tokens which are matched against the pattern database. Anything that is in the database gets flagged, possibly resulting in a large number of false positives.

2.2.2 Type Checking

Static analysis could be leveraged by mimicking some of the features that make programming languages, such as Java, more secure. One of these features is strong type checking, which C lacks. C has a reputation of being an unsafe language. It was designed with space and performance considerations in mind, in a time where security was not a big concern. It allows direct pointer manipulations without any bounds checking. So, it is up to the programmer to do the checks himself.

By providing the language with harder type checking, some of the vulnerabilities could be detected statically. Usually this is done by supplying additional keywords (e.g., type qualifiers), treated as commentaries by the compiler, but recognized by the analyzer. However, this approach can only enforce the values to be used in accord with their type, such as integer variables that could overflow to negative values, or unsigned integers that could underflow to large values.

CQual (Foster et al., 1999) developed a framework for adding type qualifiers to a programming language, allowing, for instance, types to be polymorphic. It uses type qualifiers (e.g., `dynamic nonzero int`) to perform type checking. The framework extends the programming language so that it can also infer the qualified types and their relationships,

such as polymorphism. So, whenever a function expects some particular type of data in one of its parameters, it can also receive any of its subtypes. Important static analysis tools evolved from this framework, such as CQual extensions (Shankar et al., 2001) and Carillon (Elsman et al., 1999).

Other authors formulate the buffer overrun detection problem as an integer constraint problem. This methodology is incorporated in BOON (Wagner et al., 2000). This tool uses simple graph theoretic techniques to construct an efficient algorithm for solving the integer constraints. An abstract data type was created for defining strings in C. Also, these buffers are defined as pairs of integer ranges (their allocated size and length), regardless of their contents. Detecting buffer overruns is then a question of tracking those integer ranges. First, a constraint language is used to model string operations. Then, a fast and scalable integer range analysis solves the constraint system. Any constraint violation indicates a possible buffer overflow vulnerability.

2.2.3 Control-flow Analysis

A more elaborate analysis requires the examination of the application's flow of control, i.e., the semantics of the program. Again, more compiler features are lent to static analyzers. Abstract syntax trees (AST) can be built in order to study the various relations between the different modules and functions of the program. The scope of the analysis can be done at three levels: *local level* (each function is analyzed separately), *module level* (the relationships and interactions within a particular module, or compilation unit, are analyzed), and *global level* (analyzes the program globally, including the interactions between the different modules).

Control-flow analysis can detect problems like invalid pointer references, the use of uninitialized memory, or improper operations on system resources (e.g., trying to close a closed file descriptor). First, the static analyzer parses the source code and builds the AST. Second, the analyzer traverses the entire tree, determining the control-flow paths. Finally, the paths are simulated and any inconsistencies (i.e., potential vulnerabilities) identified.

PREfix (Bush et al., 2000) is an error detection tool for C and C++, based on simulating the execution of individual functions. Models of the functions that describe the behavior of the functions as a set of conditionals, consistency rules, and expression evaluations, are automatically generated. Whenever a function call is encountered during the path execution, the model for that function is used to determined which operations to apply. PREfix sequentially simulates the action of each operator and function call, and traces distinct execution paths. Inconsistencies to the model are detected when a execution path violates some constraint. By tracking the state of the memory during the path execution, and applying consistency rules of the language to each operation, inconsistencies can be detected and reported. Also, detailed tracking information of the paths and values enlighten the conditions in which such inconsistencies have manifested. Some of the inconsistencies that PREfix can warn are: using uninitialized memory, dereferencing uninitialized, null, or invalid pointers, leaking memory, etc.

2.2.4 Data-flow Analysis

Another approach to statically analyze programs consists in observing the possible paths the program data can pursue. More information is added to

some particular types of data, such as those originating from the outside. This data can then be tracked down by simulating its possible paths. Rules of type inference are then used to detect inconsistencies.

LCLint (Evans et al., 1994) is an annotation-assisted static checking tool for finding buffer overflows vulnerabilities. More expressive annotations were added to LCLint, allowing programmers to explicitly state function pre- and post-conditions (Larochelle & Evans, 2001). It combines traditional data-flow analysis with constraint generation and resolution. These annotations describe assumptions about the buffers that are passed to functions. They also specify the state of the buffers as the functions return. For instance, the programmer can annotate the function as to restrict the maximum size of a particular buffer to a global variable used in the code, or to specify the minimum and maximum buffer indices that can be read. LCLint then generates constraints for the C statements. The tool builds an AST, storing each constraint in the corresponding node. Then, as the tree is traversed, constrain-based analysis techniques are employed to resolve and check those constrains. LCLint takes into account the value of the predicates on different code paths in order to detect buffer overflow constraint violations. However, it fails to detect all types of buffer overflow vulnerabilities, and also generates many spurious warnings.

Other extensions (Shankar et al., 2001) to the previously mentioned CQual provide the tool with ability to detect format string vulnerabilities. Built on top of its framework, it uses type qualifiers to perform type checking. By utilizing a method based on Perl's taint mode, the tool can identify and track data originated from the outside. The types are either marked with `tainted` or `untainted` qualifiers, which are combined with C qualified types (e.g., `int`, `tainted int`, `untainted char *`). Type quali-

fiers naturally induce a sub-typing relationship on qualified types. CQual considers data types originating from the outside as tainted data (e.g., program parameters, or network interfaces). Also, it establishes the following relationship: untainted data is a particular subtype of tainted data. Much like polymorphism, whenever a function expects tainted data in one of its parameters, it can also receive untainted data. However, CQual complains if tainted data is being passed to a function that expects untainted data.

Another feature of these extensions to CQual is that it performs type inferring. So, not all types are required to be annotated, also because the tool provides annotated versions of most standard library functions. Only a small number of annotations is required in a few key places in the program.

Additionally, CQual analyzes taint-flow paths by tracking the propagation of tainted data. For instance, if there is an execution path in which tainted data is interpreted as a format string, CQual raises an error. Constraint graphs are created, such as those where tainted data is directed to untainted data types. This helps to identify the unsafe sequence of operations that lead to the error.

2.2.5 Model Checking

Model checking (Clarke et al., 1994, 2000) is a formal verification technique that systematically enumerates possible states of the system, exploring non-deterministic events in the system. Starting from an initial state, model checking recursively generates successive system states. The model, a representation of the system under evaluation, is defined by an abstract specification of the system. This high-level description is a simplified description of the code, which abstracts away many details of the

actual implementation.

Explicit state model checking systematically searches for errors in a state graph, which represents the behavior of the system. It is usually used to prove that a system satisfies a specified property, i.e., the system's specification. Also, its thoroughness at exploring the state space of the system makes model checking a good method for finding errors in unusual system states.

However, this technology has some drawbacks. First, there is the difficulty in describing the intricate model of the system. Also, errors can only be detected if the system properties they violate are explicitly and correctly specified.

MOPS (Chen et al., 2004; Chen & Wagner, 2002) is a static analysis tool that checks if a program can violate some specified security property, such as if a setuid-root program does not drop root privileges before executing an untrusted program. Security properties are modeled by finite state automatons and supplied to the MOPS. Examples of such properties are: drop privileges properly, create chroot jails securely, avoid race conditions when accessing the file system, avoid attacks on standard file descriptors (e.g., standard input, standard output, and standard error), and create temporary files securely. The tool then exhaustively searches the control-flow graph of the program to check if any path may violate a safety property. If MOPS finds violations, it reports execution paths that can cause such violations. MOPS found errors in large network-related programs, such as Apache, Bind, OpenSSH, PostFix, Samba, or SendMail.

CMC (Musuvathi et al., 2002) is a model checker for C and C++ that executes the implementation code directly. Hence, it does not require a separate high-level specification of the system. The system is modeled as

a collection of interacting concurrent processes. Similar to Java PathFinder (Visser et al., 2000), CMC emulates a virtual machine or operating system. Each process runs unmodified C or C++ code from the actual implementation, which is scheduled and executed by the CMC. Different scheduling decisions and other nondeterministic events provides CMC the ability to search many possible system states for violations to the correctness properties (e.g., the program must not access illegal memory, or a particular function should not return an invalid object). However, the user must supply those correctness properties. The user is also responsible for building the test environment that adequately represents the behavior of the actual environment. This environment model fakes an operating system and anything outside the system under evaluation. For instance, the test environment can be a collection of substitute API functions that simulate the workload of the system.

Later on, CMC was used to create a model checking infrastructure for file systems called FiSC (Yang et al., 2004). This tool actually runs a Linux kernel in CMC. The model checker starts with an empty, formatted disk, and recursively generates and checks successive states by executing state transitions. These transitions are file system actions induced by a file system test driver, such as creating, removing, or renaming files, directories, and hard links; writing to and truncating files; or mounting and unmounting the file system. As each new state is generated, FiSC intercepts all disk writes, checking if the disk is in a state that cannot be repaired (i.e., invalid file system).

2.3 Fuzzers

This testing technique was inspired by noisy dial-up lines that sometimes scrambled command line characters and crashed the application. Some researchers were surprised to find that these spurious characters were causing programs to crash, including a significant number of basic operating system utilities — applications that should be robust enough not to crash upon receiving unusual input. They created Fuzz (Miller et al., 1990), a tool for generating random characters (similar to those created by noisy telephonic lines) and feed it to several Unix command line utilities. This extremely simple and naive method was able to crash several programs at a low effort.

Fuzzing can find odd oversights and defects that manual testing often fails to find. Awkward test cases are hard to imagine for human test designers and prohibitive for thorough and exhaustive testing. By automating testing with fuzzing, millions of iterations can cover a significant number of interesting permutations for which it could be difficult to write individual test cases (Oehlert, 2005). Fuzzing can provide such test cases without intricate knowledge. It does not require any preconception about the system's behavior. However, it can only generate negative test cases, i.e., situations in which the application does not do something it was specified to. Fuzzing only provides a random sample of the system's behavior. Nevertheless, bugs detected by this technique are usually found to be compromising and exploitable vulnerabilities, so it is particular useful for security assessment.

Though its techniques are simple, fuzzers can be rather complex tools. They can be divided into different categories by taking into account two aspects: the knowledge loaded into the fuzzer, and its specialization or

generalization.

2.3.1 Fuzzer's Knowledge

There is a whole spectrum of knowledge that can be loaded into the fuzzer. *Thin* fuzzers are simple fuzzing tools with very little knowledge or assumptions about the system under evaluation. They typically send random and invalid data to the target's interface, without any regard to the interface specification, i.e., protocol's messages, file formats, etc. Another type of data these fuzzers can generate are produced by mutating templates of valid data, such as randomly changing individual bits of the normal workload of the system. A totally different approach is to provide means and knowledge in the fuzzer to create valid and semi-valid data. These *fat* fuzzers are able to perform a perfectly legal interaction with the target, generating data that is valid enough to be accepted by parsers, but still irregular to cause problems. Proceeding further along the parser's code path can achieve better code coverage. Thus, fat fuzzers can potentially catch more errors because they can go beyond the initial states of the system. However, by restricting the sparseness of the fuzzer, some more unusual, but nevertheless interesting, test cases could be left out.

2.3.2 Fuzzer's Specialization

Another aspect that differentiates fuzzing is related to the specialization of the tool. Fuzzers can be implemented with one target in mind, whether it is an application, a network protocol, or a file format. This *specialized* type of fuzzer is very target-dependent, but intelligent enough to create and assess test cases autonomously (Betouin, 2006; Biege, 2005; Sutton, 2005).

Usually, these tools are also fat fuzzers, which comprehend a large understanding of the target's interface format. On the other end of the spectrum, the *generalized* fuzzers are fuzzing tools that can be applied against a large set of different targets. For instance, it can be a simple tool with none or little knowledge about its targets (Miller et al., 1990). These fuzzers can be used against many targets that share some few assumptions the fuzzer makes, like the format of the interface (e.g., ASCII, binary, etc.). Fuzzer frameworks are another generalized type of fuzzers. These more complex tools can create custom-made and target-specific fuzzers. They provide means for manually adding knowledge for fuzzing some specific target, such as using a Backus-Naur form (BNF) dialect for a protocol specification (University of Oulu, 1999–2003), or through a scripting language for file format generation (Greene, 2005).

2.4 Run-time Prevention Mechanisms

Run-time prevention mechanisms change the run-time environment of programs with the objective of thwarting the exploitation of vulnerabilities. The idea here is that removing all bugs from a program is considered infeasible, which means that it is preferable to *contain* the damage caused by their exploitation. Though these mechanisms are usually implemented at compile-time, they perform detection or prevention at the time of execution. Most of these techniques were developed to protect systems from buffer overflows. Instead of solving the problem at the source (the vulnerable program) they try to resolve it at the destination (the stack being overflown).

Buffer overflow attacks are malicious exploitations to hijack the pro-

gram's flow of control. For instance, by overflowing the return address with one provided by the attacker, the program will execute the instructions located in that attacker's controlled address, which usually spawns a shell. Several examples exist in the literature to protect the integrity of the stack, or at least to detect any violation.

StackGuard (Cowan et al., 1998) is a simple compiler extension that provides means for detecting invalid changes in the return address and for preventing those changes from occurring. These stack smashing attacks explore the fact that the return address is located very close to a local variable with weak bounds checking. So, the attacker has to overwrite all memory, from that variable to the return address. By placing a *canary* word (i.e., a randomly chosen value) next to the return address of a function, StackGuard can detect buffer overflows on the stack. It first checks if the canary word is intact before jumping to the address pointed by the return address, i.e., *before* the function returns. If StackGuard detects any change to the canary word, it probably means that the return address is also modified, thus aborting the execution of the program (i.e., fail-safe stop).

However, StackGuard can also prevent such modifications from occurring, thus continuing the normal execution of the program. It resorts to a fine grain memory protection, provided by MemGuard (Cowan et al., 1997). This memory protection tool can restrict access to individual memory addresses via a special API. StackGuard protects the return address by designating it read-only when the function is called. The return address' protection is only raised when the function returns. Since the only way to overwrite the return address is through the MemGuard API, the attack can be prevented.

Nevertheless, StackGuard can only stop buffer overflow attacks that overwrite everything along the stack. Stack Shield (Vendicator, 2000) is a compiler patch that implements both return addresses and function pointers protection. It provides two techniques for protecting the stack. The first technique duplicates the stack's return addresses of functions in a global array. Before a function is called, its return address is pushed onto the stack. Stack Shield then copies this address into the global array. Before the function returns, the return address in the stack is replaced by the global array copy, thus overwriting any changes made to the stack by the attacker. Another approach involves storing only the return address of the current function in a global variable. Before returning, the return address in the stack in compared with the one in the global variable. If there is a difference the program's execution is halted. Unlike the former technique, this approach will detect the buffer overflow attempt, instead of ignoring it and continue the program's execution.

Stack Shield also protects frame and function pointers from being changed. It inserts checking code before all function calls that make use of function pointers. Stack Shield declares a global variable in the data segment using its address as a boundary value. Only pointers to functions below that address are valid. So, the additional checking code compares any dereferenced function pointer with the boundary value. If it points to addresses above the boundary, the process is terminated. Note that legitimate dynamically allocated function pointers are misinterpreted as invalid function pointers.

More sophisticated techniques mitigate pointer corruption exploits. PointGuard (Cowan et al., 2003) adds special code to the program that prevents an attacker of producing predictable pointer values. It encrypts

pointers while they are in memory, and decrypts them immediately before dereferencing. A key, generated at run-time, is used to encrypt pointers in memory. When a pointer is dereferenced, its value is decrypted from memory and loaded into the CPU register. If an attacker overwrites the pointer value, it will jump into an unpredictable memory address, thus making the attack impracticable.

However, only PointGuard's compiled and instrumented code can be effectively defended against pointer corruption attacks. Standard library functions and data structures, such as `malloc`, are not currently supported.

A recent study compares the effectiveness of some of these techniques, showing that they are useful only to prevent a subset of the attacks (Wilander & Kamkar, 2003).

2.5 Vulnerability Scanners

Vulnerability scanners or vulnerability assessment tools (VAT) allow system administrators and security managers to check applications, computer systems, or entire interconnected networks against a vulnerability/misconfiguration database. After investigating the system for its flaws, these tools generate comprehensive reports with well-documented, sorted, ranked, and cross-referable data. Some vulnerability scanners even provide good recommendations for remedial actions, such as patches or temporary configurations. These are the same sort of tools that have been used by malicious hackers in order to penetrate and exploit in such systems.

They have a database of well-known vulnerabilities, which should be updated periodically, and a set of attacks that allows their detection. The

analysis of a system is usually performed in three steps: first, the scanner interacts with the target to obtain information about its execution environment (e.g., operating system, available services, open ports, etc); then, this information is correlated with the data stored in the database, to determine which vulnerabilities have previously been observed in this type of system; later, the scanner performs the corresponding attacks and presents statistics about which ones were successful. Even though these tools are extremely useful to improve the security of live systems, they have the limitation that they are unable to uncover new vulnerabilities.

2.5.1 Generations

Nowadays, VATs are much more sophisticated, accurate, and informative than those used in the past. Three generations in the evolution of VATs can be outlined: the *first generation* of vulnerability scanners were designated as a set of hacker's tools, like the utilities and mechanisms that were delivered with standard operating systems to perform functions such as audit, identification and authentication, process isolation, and memory protection. Then, they grew in popularity and were used in the security auditing world, but some important features were missing. Some well-known vulnerability scanners include Satan (Farmer & Venema, 1995) and nmap (Fyodor, 1997). With the *second generation*, vulnerability scanners were much more complex and modular tools, allowing the integration of a knowledge base into their databases of vulnerabilities, i.e., administrators could now learn how to remove/patch found vulnerabilities. Some of these VATs could actually keep some history about the previously found and patched vulnerabilities. Also, their database could be automatically updated from the server, in a growing frequency basis. Some of the most

common products were Internet Scanner (Internet Security Systems Inc., 2006), Retina Network Security Scanner (eEye Digital Security, 2006), and Nessus Vulnerability Scanner (Tenable Network Security, 2006a).

The *third generation* faces the uprising distributed architecture of the real enterprise world. Now these tools possess remote software agents capable of scanning distant and heterogeneous networks. This last generation allow VATs to scale to a much more realistic assessment approach — the entire network. Examples of this type of VAT are QualysGuard Enterprise (Qualys Inc., 2006) and FoundStone Enterprise (McAfee, Inc., 2006).

2.5.2 Host- and Network-based Scanners

There are two main technologies for vulnerability scanners: host-based and network-based. *Host-based* scanners require that an agent software be installed on each host. This type of scanner tools can provide rich security information such as by checking user access logs. Once deployed, they have limited impact on network traffic. However, cost can add up when deploying agents across many desktops and servers. In addition, due to the individual nature of the vulnerability assessment, deployment can be time consuming. *Network-based* tools are available as software appliances and managed services, and can give a quick look at what weaknesses hackers or worms can exploit. Though they do not require any software agent, they may need a dedicated computer to run the scanner. They also require careful planning to avoid conflicts with other security systems, such as firewalls or ACLs. These type of tools can be intrusive by generating considerable network traffic.

Usually, vulnerability scanners have good coverage, but before a vulnerability can be detected it must first be inserted in its database. They

are also fully automated, allowing scheduled vulnerability scans to be executed without human assistance and across entire networks without physical access to their different hosts. moreover, they provide centralized management capabilities and give the auditor total control of the scanning process in one single (but not fixed) location.

New vulnerability scanners tend to be less intrusive, to the extent of even existing *passive* scanners, like the Tenable Passive Vulnerability Scanner (Tenable Network Security, 2006b). These type of VATs continuously monitor the network for known vulnerabilities in a non-intrusive way. They passively listen to network traffic, identifying operating systems and services by fingerprinting packets and cross-referencing them with port numbers.

Chapter 3

Attack Injection Architecture

The ultimate goal of this work is to contribute to the development and deployment of more secure computer systems, which can then be utilized in hostile environments such as the Internet. In particular, one would like to prevent malicious attacks from exploiting vulnerabilities, which could cause intrusions and lead to the failure of the system.

Throughout the years, many tools were aimed at helping the development teams to produce more reliable software, such as compilers that do extensive code checking, debuggers, testing libraries, and memory function wrappers. However, applications are still released with errors, and the systems do fail, even when they are utilized to perform mission-critical functions. Therefore, a complementary approach, closer to the adversary course of action, was taken, i.e., to actually attack the system.

The solution proposed in this thesis is based on locating vulnerabilities through attack injection. This chapter will present the methodology, as well as the designed architecture, its main components, and how they interact with each other.

3.1 Using Attacks to Find Vulnerabilities

Every system's design and implementation should comply with a set of functional and/or non-functional properties, i.e., a service specification that describes its correct operation. The system is said to be *correct* if its service specification is not violated. But how much can one trust in that correctness? A system should give some guarantees that it will not fail. This measure is given by the system's *dependability*, which is the ability of a computing system to deliver service that can be justifiably trusted (Powell & Stroud, 2002). Dependability aims at preventing the failure of the sys-

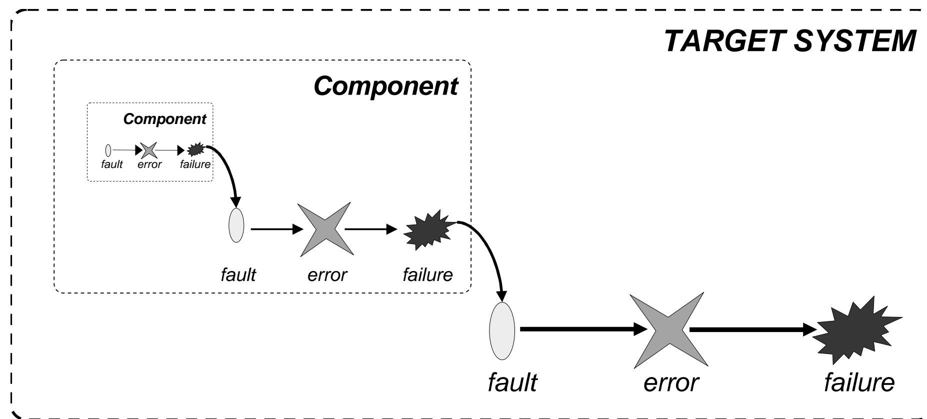


Figure 3.1: Fault, error, and failure.

tem, hence, in order to construct more dependable and reliable systems, one must know how the system and its components can remain correct, i.e., *how* and *why* do systems fail.

The fault model presented in Figure 3.1 tries to explain how systems fail by following the sequence *fault* → *error* → *failure*. For instance, the purpose of a file system is to store and manage data records. These records, organized in computer files, are physically located in a storage device. The service specification of the file system defines, among other things, that any "read" operation will reflect the last "write" operation. Therefore, a record should always return the value of the last "write". A failure occurs when this specification is violated. To understand how these failures happen, one must study the process that explains how they appear. The cause, which is called the *fault*, can have an internal or external origin. For example, an electrical discharge (fault) can change the bits of some record, located in a specific disk sector. This fault can remain unnoticed, or dormant, until the record is read. The fault's manifestation is called an *error*, which in our example is the corrupted record. The *failure* of the system is the external observable effect of the error. If the file system lacks some sort

of detection or correction mechanism for this type of error (e.g., checksums or redundancy), the record will return an incorrect value. This behavior clearly violates the service specification of the file system, or in another words, it represents the failure of the system.

However, the file system can be regarded as a component of a larger system, such as an operating system (OS). Therefore, from the OS point of view, the failure of the file system is seen as the fault of a component. So, as the figure shows, the fault–error–failure sequence is also a recursive model. If the afected disk record holds paging data, such as a swap file, the file system failure (OS fault) will result in a virtual memory error. In turn, this error could freeze the entire system, leading to the failure of the OS.

However, the fault's presence (or even the error's) will not necessarily produce a failure. The system's correct operation can be maintained despite the presence of faults. If the system is supplied with fault detection or tolerance mechanisms, a greater dependability can be achieved.

Nevertheless, the type of faults that can arise are not reduced to accidental or arbitrary faults, like a physical defect or an electrical discharge. Faults can be much more complex and appear with higher probability. Intentional malicious faults are a good example. A potential intruder can leverage the failure probability by conducting a series of targeted attacks that can lead to the violation of the service specification. This type of faults do not follow any probabilistic distribution, nor any behavior pattern.

The AVI (attack, vulnerability, intrusion) composite fault model, introduced in (Powell & Stroud, 2002; Veríssimo et al., 2000), helps us understand the mechanisms of failure due to several classes of malicious faults (see Figure 3.2). It is a specialization of the fault–error–failure sequence

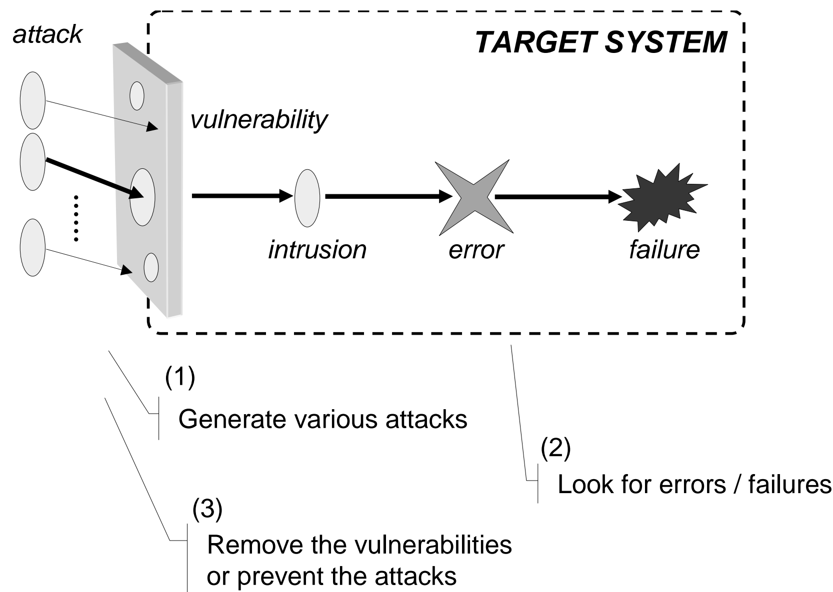


Figure 3.2: Composite fault model (attack, vulnerability, and intrusion).

applied to malicious faults — it limits the fault space of interest to the composition (*attack* + *vulnerability*) → *intrusion*. Let us analyze these fault classes. *Attacks* are malicious external activities, originating from outside the target system boundaries, that intentionally attempt to violate one or more security properties of the system — we can have an outsider or insider user of our network (e.g., an hacker or an administrator) trying to access sensitive information stored in a server. *Vulnerabilities* are usually created during the development phase of the system (e.g., a coding bug allowing a buffer overflow), or during operation (e.g., files with root setuid in UNIX). These faults can be inserted accidentally or deliberately, and with or without malicious intent. An attack that successfully activates a vulnerability causes an *intrusion*. This further step towards failure is normally succeeded by the production of an erroneous state in the system (e.g., a root shell or a new account with root privileges), and if nothing is done to process the error, a failure will follow.

The methodology utilized in the construction of AJECT emulates the behavior of an external adversary attempting to cause a failure in the target system. However, the goal of the attacks is not to exploit the system but rather to detect vulnerabilities that might permit that exploitation. The tool first generates a large number of attacks which it directs against the interface of the target (step 1, in Figure 3.2). A majority of these attacks are expected to be deflected by the validation mechanisms implemented in the interface, but a few of them might be able to succeed in exploiting a vulnerability and causing an intrusion. Some conditions contribute to increase the success probability of the attack. For example, a correct understanding of the interaction protocol used by the target eases the creation of more efficient attacks (e.g., it reduces the randomness of the tests); and a good knowledge about what type of vulnerabilities appear more frequently also helps to prioritize the attacks.

While the attacks are being carried out, AJECT monitors how the state of the system is evolving, looking for errors or failures (step 2). Whenever one these problems is observed, it indicates that a new vulnerability has potentially been discovered. Depending on the collected evidence, it can indicate, with more or less certainty, that a vulnerability exists. For instance, there is a high confidence if the system crashes during (or after) the attack — this attack at least compromises the availability of the system. On the other hand, if what is observed is an abnormal creation of a large file, though it might not be a vulnerability — possibly related to a denial of service — it still needs to be further investigated.

After the discovery of a new vulnerability, there are several alternatives to deal with it, depending on the current stage of the development of the system (step 3). If the system is, for instance, in development, it is

best to provide detailed information about the attack and the error/failure, so that a decision can be made about which corrective action should be taken (e.g., repair a software bug). On the other hand, if the tests are performed when the system is in live operation, then besides giving information about the problem, other actions might be worthwhile taking, such as automatically change the execution environment to remove the attack (e.g., by modifying some firewall rules) or shutdown the system until the administrator decides what to do.

In order to get a higher level of confidence about the absence of vulnerabilities in the system, i.e., its dependability, the attacks should be exhaustive and should exercise an extensive number of different classes of vulnerabilities. Still, one should also know that for complex systems it is infeasible to experiment all possible attack patterns, and therefore it is possible that some vulnerabilities remain undisclosed. Nevertheless, AJECT can be an important contributor for the construction of more secure systems because it mimics the malicious activities carried out by many hackers, allowing the discovery and subsequent removal of vulnerabilities before a real attempt is performed to compromise the system.

3.2 AJECT's Design

There are four basic entities in the architecture of AJECT, the *Target System*, the *Target Protocol Specification*, the *Attack Injector* and the *Monitor* (see Figure 3.3). The first entity corresponds to the system one would like to test, while the last three are the main components of AJECT.

The Target System (TS) is composed by the target application and its execution environment, which includes the operating system, middleware

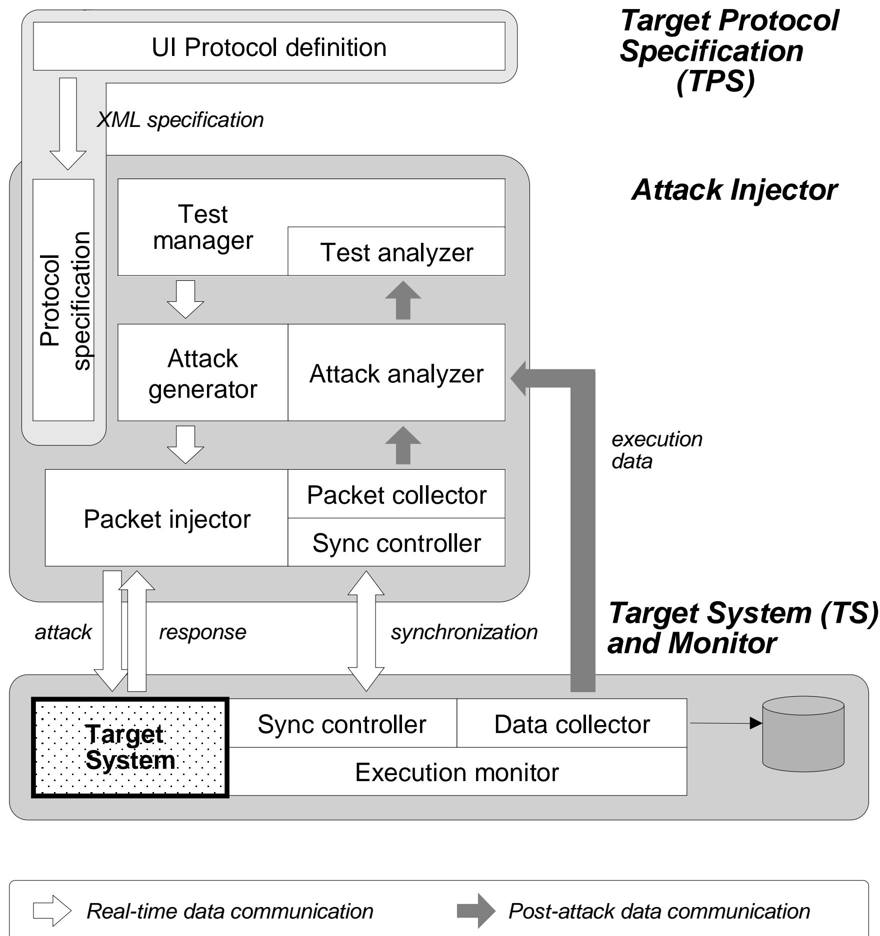


Figure 3.3: The architecture of the AJECT tool.

libraries, and hardware configuration. The target application typically provides some service that can be invoked remotely from client programs (e.g., a mail or FTP server). In addition, it can also be a local daemon supporting a given task for the operating system. In both cases, the target application uses a well-known communication protocol to exchange requests and responses with the clients. It is through this target protocol that these clients can carry out attacks by transmitting malicious packets, i.e., carefully crafted requests with the intent of exploiting a particular vulnerability. If the packets are not correctly processed, the target can suffer

various kinds of errors with distinct consequences, ranging, for instance, from a slow down to a crash.

The Target Protocol Specification (TPS) provides the necessary knowledge to generate target protocol messages. This component offers a graphical interface for the formal definition of the communication protocol used by the target application, which results in an Extensible Markup Language (XML) file, and the ability to import different protocol specifications.

The Attack Injector, or simply the Injector, imports the XML specification through the *protocol specification* module, providing AJECT with the ability to create valid target protocol messages. The main purpose of the Injector is the generation and execution of the attacks, and the reception of the responses returned by the target. It also does some analysis on the information acquired during the attack to determine if a vulnerability was exposed.

The last component in the AJECT's architecture is the Monitor. The main objective of the Monitor is to observe and gather data about the target's execution, which requires a careful synchronization with the Injector.

The architecture was defined to achieve two main purposes, the automatic injection of attacks and the data collection for analysis. However, its design was done in such a way that there is a clear separation between the implementation of these two goals. On one hand, in order to obtain extensive information about the execution, a proximity relation between AJECT and the target is required. Therefore, the Monitor needs to run in the same machine as the target application, where it can use low level operating system functions to gather, for example, statistics about the CPU or memory usage. On the other hand, the injection of attacks can usually be performed from a different machine. In fact, this is a desirable situation

since it is convenient to maintain the target as independent as possible from the Injector, so that interference is kept to the minimum.

3.3 Test, Attack and Packet Hierarchy

The injection of an attack is related to the type of test one wants to perform and materialized through the actual transmission of the (malicious) packets. Therefore, the attack concept is relatively vague and can be quite generic. For instance, an attack could correspond to something as generic as the creation of requests that violate the syntax of the target's protocol messages, or as specific as a special request that contains a particular username and a secret password.

In order to clarify these semantical differences for the test and attack concepts, three levels of abstraction were defined. Therefore, the process of creating an attack can be seen at three levels. The first and most generic level defines a few general *test* classes. Each test will provide an attack generation algorithm. The test will be systematically instantiated resulting in specific *attacks* (the second level). In the last and lowest level, an attack is implemented through the physical transmission of its corresponding *packets* in the network, i.e., the bytes of the messages that correspond to the entire protocol interaction (attack).

As an example, consider one of the tests currently supported in AJECT, a syntax test (see Section 4.5 for more details). This test validates the infringement of the format of the packets used in the target protocol, and looks for processing errors in the number and order of the packets' fields. Even for a straightforward protocol with a few different messages, it is quite easy to generate a reasonable number of distinct attacks, i.e., to cre-

ate several different instances for that test. The low-level representation of an attack is the finite stream of bytes, i.e., the TCP or UDP packets that correspond to the protocol messages of the attack.

3.4 TPS Component

The TPS component is used to create formal specifications of the communication protocol utilized by the target. This specification is essential for two reasons: First, AJECT needs to be capable of bringing the application from one initial state to any other of its possible states. The reason for this is because certain protocol messages are exclusive to particular situations, such as specific requests that are only valid *after* successfully transiting to an authenticated state. Second, the syntax of the messages must be well-known because many non-trivial attacks can only be created if this information is available. Another reason for having a TPS component is to simplify the use of the tool — instead of having to code a special module for each new target protocol, one only needs to produce different high-level specifications (XML files).

Currently, the specification can be done with a graphical interface that allows the definition of a state and flow graph of the protocol. For each state it is possible to identify which messages can be sent and their syntax. The output of the *UI protocol definition* module is an XML file that formally describes the target protocol, which is then imported by the Injector. During the attack generation, the protocol specification module provides the Injector with the essential knowledge to create valid protocol messages for the Attack Generator and to correctly transitate between the different protocol states.

3.5 Injector Component

The Injector is decomposed into three groups of modules, each one corresponding to a different abstraction level of the attack generation hierarchy (see Figure 3.3). At each of the three levels, test, attack, and packet, there is a module whose function is related to the construction of the attacks and another for the collection and analysis of the responses. In more detail,

Test level The *test manager* controls the whole injection process. It receives a protocol specification and a description of a test and then it calls the *attack generator* to initiate a new attack. The *test analyzer* saves and examines various information about the attacks, in order to determine effectiveness of a test to discover vulnerabilities.

Attack level The actual creation of the attacks is the responsibility of the *attack generator*. The *attack analyzer* collects and studies the data related to the target's behavior under a particular attack. It obtains data mainly from two sources: the responses returned by the target after the transmission of the malicious packets, and the execution and resource usage data gathered by the Monitor.

Packet level The *packet injector* connects to the target application and sends the packets defined by the attack generator. Currently, it can transmit messages using either the TCP or UDP transport protocols. The main task of the *packet collector* is the storage of the network data (e.g., attack injection packets and received responses).

3.6 Monitor Component

Although the Monitor appears to be a simple component, this is a fundamental entity that hides some complex aspects. On one side, this component is in charge of setting up all testing environment in the TS: it needs to start up the target application, perform all configuration actions, initiate the monitoring activities, and in the end, free all utilized resources (e.g., processes, memory, disk space). The whole system is re-set after each experiment to guarantee that there are no interferences among the attacks, which simplifies the identification of the attack that caused the problems, and consequently, the discovery of the vulnerability.

On the other side, the Monitor observes the execution of the target while the attack is being carried out. This task is highly dependent on the mechanisms available in the local operating system (e.g., the ability to catch signals, such as memory segmentation errors, in Linux). It is expected that the type of vulnerabilities AJECT is able to diagnose, is related to the type and detail of the collected information.

The monitor is composed by the following modules: the *execution monitor*, which coordinates the various tasks of each experiment and traces the target application's execution; the *data collector*, responsible for the storage of the monitoring data and its transmission back to the Injector; and the *sync controller*, that coordinates both injection and monitoring to determine the beginning and ending of each experiment, and also to ensure that each attack is done under identical conditions.

The execution monitor module can allow AJECT to closely observe the target's flow of control (e.g., by intercepting and logging any software exceptions) to detect if the TS goes to any erroneous software state. Therefore, there are many interesting operational characteristics desirable for

monitoring, such as the reception of a *Segmentation Fault* exception (crash) or something more subtle like an unusual set of OS signals. In the same way, the supervision of the allocation of the system resources, during the target's execution, can also be helpful to detect abnormal behavior activity, which may be indicative of the presence of a vulnerability. AJECT correlates the target's behavior information with its resource usage, such as the memory used by the process or the clock cycles spent by the CPU.

As previously mentioned, all target's execution supervision must be carefully synchronized with the Injector, so that each attack injection is properly monitored and the same initial test conditions are guaranteed throughout all the experiments. The synchronization between Injector and Monitor is performed through a simple synchronization protocol by the sync controller modules. The next chapter (Chapter 4) provides more details about the implementation of such modules and components.

3.7 Test and Attack Analyzer Components

After all the experiments are performed (i.e., every attack injection of each type of test), AJECT must be able to detect the presence of vulnerabilities by resorting to the analysis of the target's behavior. For each action there is a reaction, so for each attack injection there is a target's reaction. This response effect is observed and recorded by the Monitor so that the attack analyzer and test analyzer modules can use this information to perceive any abnormal behavior. These analyzer modules examine the attack injection experiment results by observing network data from the respective attack and response messages. This information is then correlated with the one provided by the execution monitor module (i.e., target's execution

and resource usage data).

AJECT can then assert about the presence of a vulnerability in a specific protocol command (e.g., IMAP SEARCH command) by looking to the target's execution (e.g., detecting a SIGSEGV signal), resource usage (e.g., resource allocation starvation), or protocol responses (e.g., a message giving access authorization to a forbidden file), during that particular attack injection.

Chapter 4

Implementation of the Tool

The modular design of the architecture of AJECT provides the tool with a strong independence at various levels. The architecture is not platform specific, so it can be implemented in any operating system (OS) or hardware. Moreover, since the Injector and the Monitor are independent, new test classes or new monitoring capabilities can be added without interfering with one another.

There is, however, one restriction between the Monitor and the TS. The latter component is the system where the application server lies, and though its implementation details are irrelevant, its underlying OS is of utmost importance for the Monitor. The Monitor's dependence on the TS is such, that both Monitor and target application are required to run on the same machine. In fact, due to implementation details, related to the OS support, the Monitor is actually responsible for starting the execution of the target application.

AJECT is currently implemented in two different programming languages. The Monitor was written in C++, because of the low-level operations it provides, while the rest of the tool was implemented in Java. C++ features specific OS low-level functionality, crucial for monitoring the target process. In spite of the fact that C++ is an object-oriented and a high-level programming language, it lacks an important feature that Java possesses — a Java program can run similarly, and without modification, on any Java virtual machine. One should be able to write a program once and run it anywhere. This allows the Injector and the TPS to run on virtually any platform. The present chapter will give a more thorough insight on the implementation of AJECT.

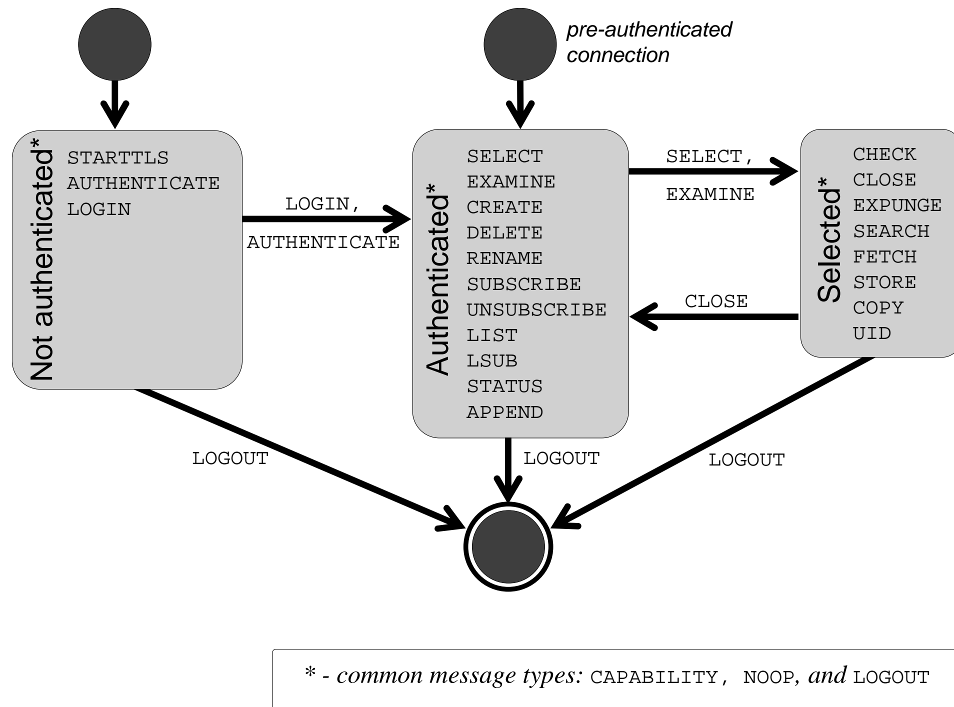


Figure 4.1: Example for the IMAP protocol finite state machine.

4.1 TPS Component

All attack generation and injection is totally independent of the intrinsics of the target protocol. The TPS component is responsible for the understanding of the protocol utilized by the target application. Without it, AJECT would not be able to create the protocol messages that will constitute an attack. Moreover, it is necessary for transiting between the different protocol states, from which the attacks are launched.

The target protocol can be regarded as a formal language, produced by a formal grammar or by a deterministic finite state machine. For example, the IMAP protocol can be described as a three-state deterministic finite automaton, as displayed in Figure 4.1. The boxes represent the IMAP protocol states, with its possible message types (i.e., IMAP commands):

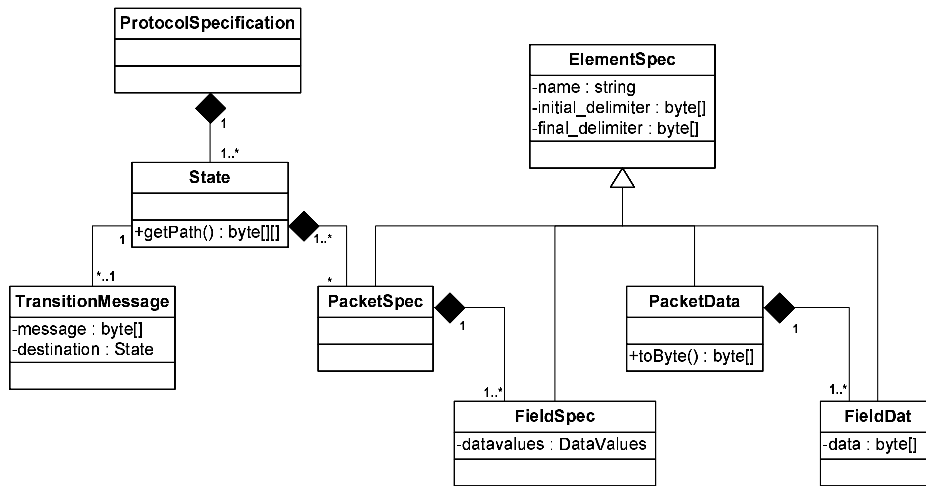


Figure 4.2: UML class diagram for the ProtocolSpecification class.

not authenticated, authenticated, and selected states. Some commands, if successfully executed, will trigger a state transition of the protocol (as depicted by the arrows).

Therefore, in AJECT, each protocol can be depicted as a set of states, which in turn are composed by two types of messages: those that execute some actions in the current state and those for transiting to other states. This composition was mapped onto the implementation of the protocol specification module. Figure 4.2 presents the UML class diagram for the Java class that implements this module, the `ProtocolSpecification` class. The figure shows the composition of the `ProtocolSpecification` class in `State` classes, which in turn, are composed by `PacketSpec` and `TransitionMessage` classes.

As for the target protocol definition, it can be specified either by coding a Java class that extends the `ProtocolSpecification` class or by using a special user interface application[1]. In the latter case, the definition of

[1] This user interface was developed by Ana Cotrim as part of her Curso de Especialização Profissionalizante em Engenharia Informática. For this reason the thesis will only provide a generic description of this component. More details are available in Ana's project report.

the target protocol results in an XML file, which is imported by AJECT's protocol specification module. This module supports the test management and the actual generation of the attacks.

As mentioned earlier, this protocol specification module has two goals: the first one is to create valid protocol messages, and the second is to help AJECT to move between the different protocol states, so that they all can be experimented. For the first goal, the class `PacketSpec` was created. This class encompasses the formal specifications for creating the packets and its fields, along with the description of the type of data. As for the second goal, the representation of the transition message types was reduced to only one message. The reason for this simplification is that all valid message types (i.e., protocol commands), transition-related or not, are represented by a `PacketSpec` class. So, when AJECT needs to change the state of the protocol, only one `TransitionMessage` is necessary. Another simplification is that this `TransitionMessage` class does not require any knowledge about the type of data or information about how to construct a packet. Basically, it is an array of bytes of the actual packet that will change the protocol state.

The `PacketSpec` class holds all the information required for the construction of the packet, represented by `PacketData` class. Essentially, a `PacketData` is a set of `FieldData` elements, i.e., a concatenation of byte-arrays, whose construction is dictated by the `DataValues` class (see Figure 4.3). This figure shows the UML class diagram for the definition of the different types of data. The `DataValues` class details the type of data accommodated in each given field. More specifically, each `FieldSpec` is determined by its valid and invalid `DataType` classes. This class is responsible for defining the type of data (both valid and invalid) the field

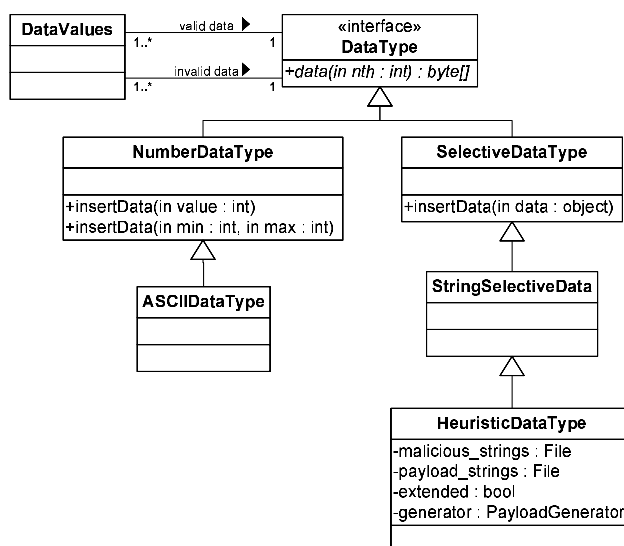


Figure 4.3: UML class diagram for the `DataValues` and `DataType` classes.

holds. For instance, text-based protocols (e.g., POP, IMAP, HTTP, etc.) are composed of a finite set of commands (i.e., valid data), which can be defined by the `StringSelectiveData`. For the erroneous commands (i.e., invalid data), they can be defined by the `ASCIIData` class, which is a set of random bytes, defined by ASCII characters — one can define a string of characters within a given ASCII range, which basically represents strings of any value.

Additionally, these packet-related classes specify more information, such as the delimiters, which can also be useful for classes of tests that attack the format of the packets.

4.2 Injector Component

This injector component implements the attack injection and analysis. The first step in the injection of the attacks is to create them. The protocol specification module imports the the necessary information, to construct

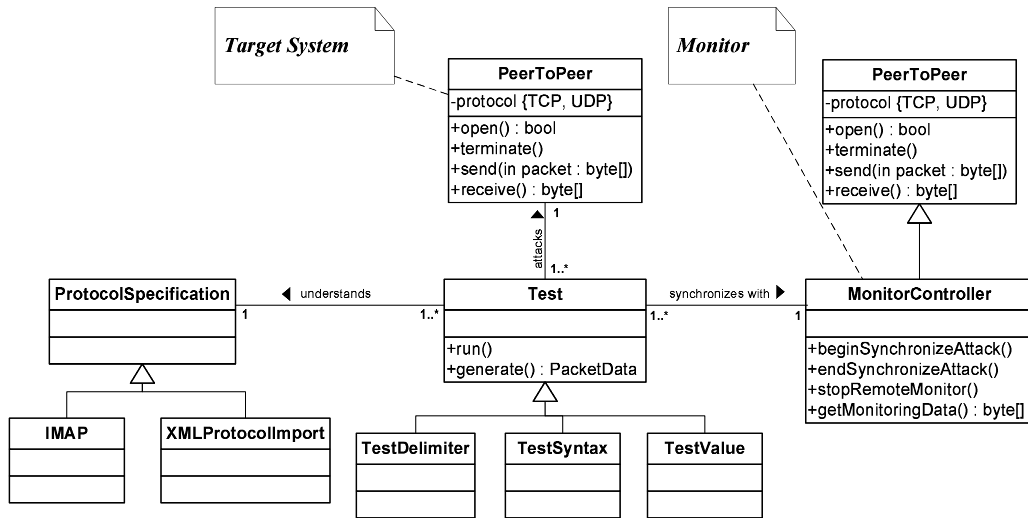


Figure 4.4: UML class diagram of the Injector component.

protocol packets, from the TPS component. This knowledge of the target protocol is crucial for the creation of the attacks.

The attack injection is done at the three distinct levels of abstraction: test, attack, and packet. This subdivision of the attack concept is important in order to understand its composition in the different modules. Each test will *attack* several aspects of the TS, such as his ability to cope with invalid data. The UML classes, and their relationships, of the Injector are presented in Figure 4.4. The figure shows that the central `Test` class relates to three other classes, namely:

- `ProtocolSpecification` — a class that defines the target's communication protocol, as was previously described in Subsection 4.1;

- `PeerToPeer` — a class for interacting with the TS, i.e., to inject the attacks. This class has point-to-point network facilities, such as for opening/closing sockets (UDP or TCP), and sending/receiving data.

- `MonitorController` — extends `PeerToPeer` class in order to im-

plement the communication with the Monitor, through a custom synchronization protocol. The synchronization process will be explained in more detail later in Subsection 4.4.

The `Test` class has two main methods: the function *generate()*, responsible for creating an attack, which will be iteratively called inside the second main method; and the function *run()*. A pseudocode of this latter function is depicted in Pseudocode 4.1. The `ProtocolSpecification` provides the necessary knowledge for the creation of the packets, such as the protocol states and its respective messages. The algorithm for the *run()* function starts by cycling through all states and picking each message at a time, which will then be used to generate attacks based on its formal specification. For instance, in a value test, the *generate()* function will create new packets from the original specification, with different data, raging from valid to invalid values. For each of these attacks, follows a three-step injection process. First, in the *pre-injection* phase, the Injector and Monitor are synchronized for the subsequent attack injection. Additionally, a network connection with the TS is opened and tested. Then, follows the *injection* itself, where the attack packet is sent and logged along with its reply. Third, the injection process is concluded with another synchronization between the Injector and the Monitor, so that the latter can release and terminate all resources used by its monitoring and the target process. This will reset the environmental test conditions, ensuring that all attacks are performed under the same conditions. The target's network connection is also closed. Since this is not a passive vulnerability detection method, the same initial test conditions must be guaranteed. This *post-injection* step is required so that the attack injections and monitor observations are not tainted or adulterated by some previous attacks.

```
run()
.   AttackData ← vector for monitoring data
.
.   /*
.    * For each possible state of the Target's Protocol,
.    * and for each valid message in this state ...
.    */
.   foreach State ∈ ProtocolSpecification do
.   .   foreach PacketSpec ∈ State do
.   .   .
.   .   .   // initializes an internal list of attacks for this particular message
.   .   .    initialize (PacketSpec)
.   .   .
.   .   .   /**
.   .   .    * Generate attack (packet with data) for injecting
.   .   .    */
.   .   .   while (PacketData ← generate()) ≠ φ ) do
.   .   .   .
.   .   .   .   // Pre−injection: prepares Monitor and TargetSystem for injection
.   .   .   .   MonitorController.beginSynchronizeAttack()
.   .   .   .   TargetSystem.open()
.   .   .   .   TargetSystem.ping()
.   .   .   .
.   .   .   .   // Attack injection and logs data
.   .   .   .   Messages ← ProtocolSpecification.getPath(State)
.   .   .   .   TargetSystem.send(∀ m ∈ Messages) // go to state
.   .   .   .   TargetSystem.send(PacketData) // attack
.   .   .   .   reply ← TargetSystem.receive() // reply
.   .   .   .   AttackData ← (PacketData, reply) ⋃ AttackData
.   .   .   .
.   .   .   .   // Post−injection: prepares Monitor and TargetSystem
.   .   .   .   // for next injection, and logs data
.   .   .   .   MonitorController.endSynchronizeAttack()
.   .   .   .   MonitoringData ← MonitorController.getMonitoringData()
.   .   .   .   AttackData ← MonitoringData ⋃ AttackData
.   .   .   .   TargetSystem.terminate()
.   .   .
.   .
.
```

Pseudocode 4.1: Function run() of the Test class.

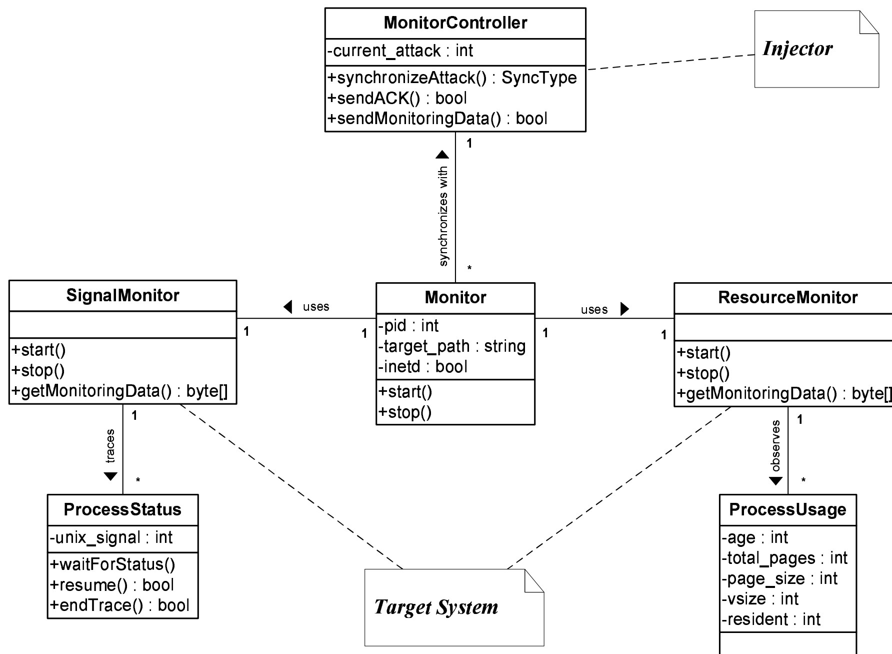


Figure 4.5: UML class diagram of the Monitor component.

4.3 Monitor Component

Another component of major importance of the attack injection tool is the Monitor. This component has three main roles:

- management and observation of the target's process execution;
- synchronization with the Injector component;
- clean up and reset of the experimental conditions.

Figure 4.5 depicts the UML classes used in this component. The `Monitor` class is the main and starting monitoring object. It is directly responsible for launching and terminating the target's process, through the methods *start()* and *stop()*, respectively. The execution monitor and data collector modules, presented in the architecture, are implemented by the `SignalMonitor` and `ResourceMonitor` classes. They represent the

monitoring programming objects for tracing the process' execution (e.g., UNIX signals) and resource usage (e.g., CPU, memory). Analogously to the Injector component, the Monitor also has a `PeerToPeer` type of class responsible for the communication between the Monitor and the Injector. This `MonitorController` class extends the `PeerToPeer` class, implementing a simple synchronization protocol. All target's execution supervision must be carefully synchronized with the Injector so that each attack injection is properly monitored and the same initial test conditions guaranteed. This synchronization will allow the Monitor to launch the target process and initiate the monitoring facilities, prior the attacks. The synchronization protocol is also used to signalize the end of the attack, allowing the Monitor to *kill* the target process and terminate all monitoring activities.

Since the Monitor needs to be in active communication with the Injector and, in simultaneous, to observe the TS, it executes more than one task in parallel. To accomplish these tasks, a multithreaded Monitor was conceived with three concurrent threads:

- a synchronization thread — for listening and responding to the Injector;
- a signal monitoring thread — for tracing and logging UNIX signals the target process receives;
- and a resource usage monitoring thread — for continuously observing and recording the target process' resource usage.

A detailed description explaining how the threads are managed within the Monitor component is presented in Pseudocode 4.2. The pseudo code presents three main functions: *sync_function()*, *monitor_signals_function()*,

```
sync_function()
.    while (true) do
.    .    switch (MonitorController.synchronizeAttack())
.    .    case SYNC_START:
.    .    .    thread_signal(SIGNAL_START_MONITOR)
.    .    case SYNC_END:
.    .    .    thread_signal(SIGNAL_STOP_MONITOR)
.    .    case SYNC_EXIT_MONITOR:
.    .    .    thread_signal(SIGNAL_EXIT_MONITOR)
.    .
.

monitor_signals_function()
.    Monitor.start() // launches target process
.    SignalMonitor.start() // traces signals received by the target process
.    .
.

monitor_resource_function()
.    ResourceMonitor.start() // traces resources used by the target process
.

main()
.    thread1 ← synchronization thread
.    thread2 ← signal monitoring thread
.    thread3 ← resource monitoring thread
.    quit ← false
.
.    thread1.run(sync_function)
.
.    while (quit = false) do
.    .    switch (thread_wait())
.    .    case SIGNAL_START_MONITOR: // starts monitoring
.    .    .    thread2.run(monitor_signals_function)
.    .    .    thread3.run(monitor_resource_function)
.    .    .    MonitorController.sendACK()
.    .
.    .    case SIGNAL_STOP_MONITOR: // stops monitoring
.    .    .    Monitor.stop()
.    .    .    thread2.terminate()
.    .    .    thread3.terminate()
.    .    .    data ← Monitor.getMonitoringData()
.    .    .    MonitorController.sendMonitoringData(data)
.    .
.    .    case SIGNAL_EXIT_MONITOR:
.    .    .    quit ← true
.    .
.
```

Pseudocode 4.2: Monitor component

and *monitor_resource_function()*. Each thread is charged with the execution of one of these functions. The synchronization thread (*thread1*) is launched only once, and it lasts for the entire execution of the Monitor. This thread is responsible for signalizing the Monitor's main thread after receiving synchronization messages from the Injector. These signals are handled in a signal handling cycle, where they are used to coordinate the various tasks of the Monitor. For instance, if the Injector sends a message of the type `SYNC_START`, the synchronization thread will emit a `SIGNAL_START_MONITOR` signal, which will direct the Monitor to: instruct both monitoring threads (*thread2* and *thread3*) to execute their respective thread functions, and to reply with an acknowledge synchronization message. The monitoring threads will then observe and record the target's process execution. After the injection of the attack, the Injector will send a `SYNC_END` message to the Monitor, which will be received by the synchronization thread. A `SIGNAL_STOP_MONITOR` signal is then received in the Monitor's handling cycle commanding the termination of both monitoring threads. In the end, all monitoring data retrieved during the attack is sent to the Injector. There is also a special synchronization message to instruct the Monitor to stop its execution and exit.

4.3.1 Signal Monitoring

A potential vulnerability is found if an abnormal behavior is detected on the TS. The underlying OS offers some monitoring facilities that can be used to notice these irregularities. For instance, on UNIX machines there are OS functions for tracing the execution of a particular process, such as the PTRACE family functions used by some debuggers like GDB (GNU Fountation, 2006). These functions control the execution of a process by

tracing the signals it receives. Upon a received signal, the tracing process (i.e., the signal monitoring thread[2]) intercepts the signal and interrupts the traced process (i.e., target's application). The signal is then logged and the target's process is instructed to continue the execution. This will passively trace the execution of the target application, without interfering with its normal behavior. All signals are intercepted and recorded for posterior analysis. Unusual signals, such as a *Segmentation Fault*, are a very good indicator of the presence of a fault.

The current implementation of the Monitor is able to detect standard POSIX signals, present in any UNIX-based machine, or derivatives, such as Linux or BSD. In order to use this monitoring method in other OSes, (e.g., Microsoft Windows), one needs to adapt it to the specific ways these OSes signalize their exceptional software states (e.g., signals, exceptions, etc.).

4.3.2 Resource Monitoring

The supervision of the system resources allocated during the target execution can be helpful to detect abnormal behavior which may be indicative of the presence of a vulnerability. For instance, if an application has suddenly allocated much more memory, it can be indicative of an erroneous state of memory starvation, or if the process is consuming a too great number of CPU cycles, it indicates a potential resource interlock.

AJECT correlates the target's behavior information with its resource usage, such as the memory used by the process (e.g., total number of allocated memory pages, number of pages of virtual memory, number of non-swapped pages), user-mode and kernel-mode CPU time accumulated by

[2]A thread is also considered a lightweight process, or LWP.

the process.This resource monitoring data is obtained through the LibG-Top[3] library and logged for later analysis.

4.4 Synchronization Protocol

To successfully diagnose vulnerabilities through attack injection, the tool must not only generate the actual attacks, but also observe its effects. Though it might seem a simple task, each attack must be carefully synchronized with its respective monitoring. Among the various tasks that result from this synchronization, there is:

- the execution of the target application prior to the attack (launched from the Monitor);
- the continuous supervision of the target (e.g., signal tracing, CPU usage, total number of allocated memory pages, etc.);
- the termination of the target application after the attack.

The synchronization is accomplished through a simple protocol, composed by four types of messages (see Figure 4.6 for their format representation):

- `SYNC_START` — message sent from the Injector, signalizing that an attack injection is about to begin;
- `SYNC_END` — message sent from the Injector, signalizing that an attack injection has ended;
- `SYNC_ACK` — message sent from the Monitor to acknowledge a received synchronization message;

[3] `http://directory.fsf.org/libs/LibGTop.html`

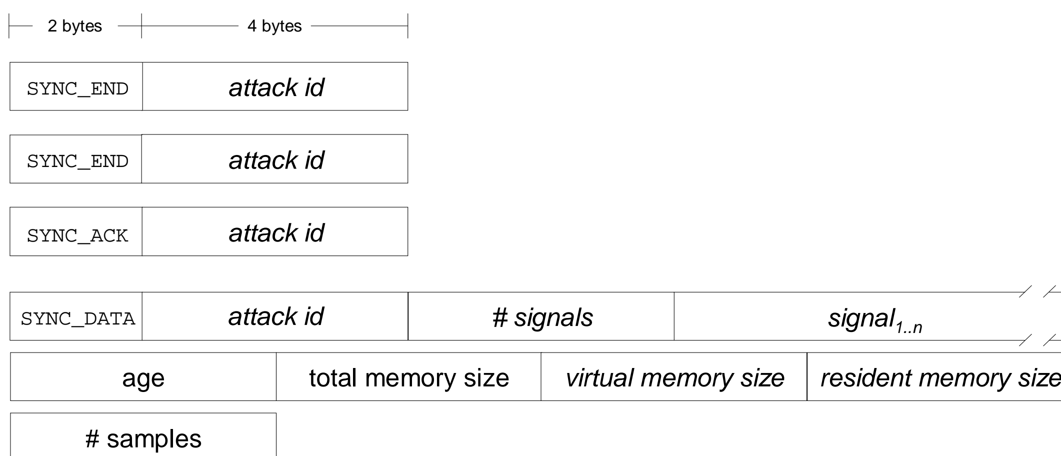


Figure 4.6: Synchronization protocol messages.

- SYNC_DATA — message sent from the Monitor, similar to the previous message, but appended with monitoring data.

There exists also a fifth message instructing the Monitor to terminate its execution and exit. This message is sent by the Injector when all tests have finished.

The whole injection and monitoring of the attacks follows a three-step process attained by both the Injector and the Monitor (see Figure 4.7 and also Pseudocode 4.1 and 4.2). The figure depicts the interactions between the Injector and the Monitor, along with their internal actions and synchronization primitives. These primitives are implemented by the MonitorController class, such as the *beginSynchronizeAttack()* and the *endSynchronizeAttack()* methods on the Injector side, and the *synchronizeAttack()*, the *sendACK()*, and the *sendMonitoringData()* methods on the Monitor side.

The *pre-attack* injection/monitoring starts when the Injector generates an attack and sends a SYNC_START message to the Monitor. In turn, the Monitor replies with a SYNC_ACK after launching the target's process and starting the monitoring threads. The Injector then knows that the TS is

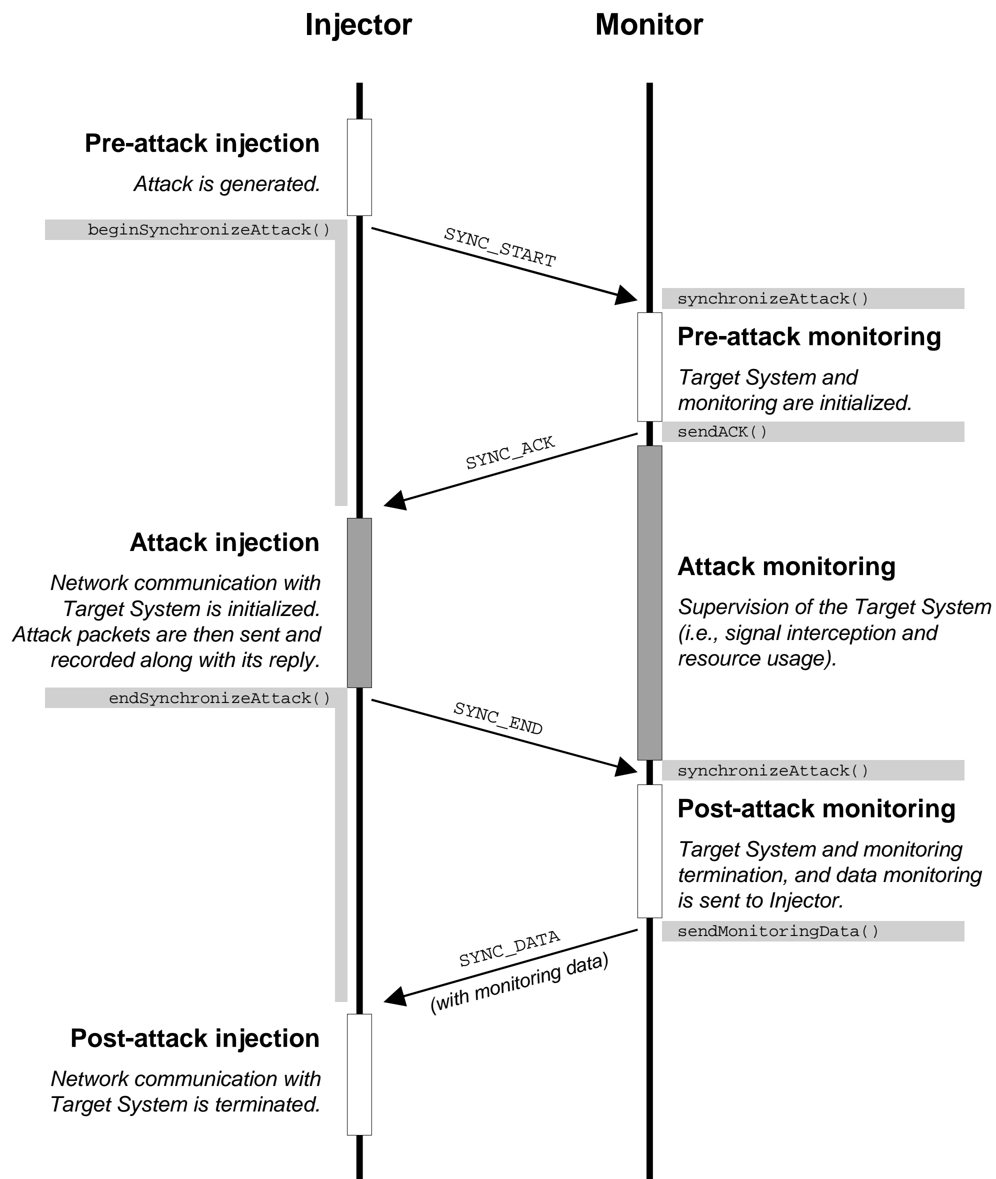


Figure 4.7: Synchronization protocol between the Injector and the Monitor.

prepared for the attacks and under the Monitor's supervision.

The second step, the actual *attack* injection/monitoring, begins with the communication initialization between the Injector and the TS. The Injector then proceeds by transmitting the target protocol messages. First, a set of messages for transiting to a specific protocol state (`TransitionMessage`),

and then, the actual attack packets (`PacketData` created from `Packet-Spec`) generated by the `Test` class. All communication between the TS and the Injector, as well as the target's process behavior (captured by the Monitor), are recorded for posterior analysis.

Finally, after receiving the attack's reply from the TS (or after a timeout without any answer), AJECT reaches the *post-attack* injection/monitoring step. The Injector sends a `SYNC_END` synchronization message to the Monitor and terminates all communication with the TS. The Monitor will then *kill* the target's process, and terminate all monitoring and logging activity for that attack. The Monitor will also gather all monitoring data for the attack and send it to the Injector in a `SYNC_END` message.

4.5 Attack Tests

The tests, executed by the Injector, are implemented by subclasses of the `Test` class, as depicted in Figure 4.4. The figure shows three different test classes, `TestDelimiter`, `TestSyntax`, and `TestValue`. However, other different classes can be created, covering more classes of attacks, thus increasing the tool's capability to discover more vulnerabilities.

The current test classes verify if the TS is able to cope with different kinds of protocol errors, namely:

- protocol messages with invalid, or missing delimiter characters;
- out of order, missing, or additional message fields;
- protocol messages with several kinds of invalid data (e.g., large or frontier values) or potentially dangerous data (e.g., information disclosure requests).

Each type of test generates a large number of attacks from the TPS. Pseudocode 4.1 shows how the `Test` class retrieves the necessary target's protocol knowledge to generate legal protocol messages. To each `State` class corresponds a set of valid message specifications (`PacketSpec`). These specifications are a set of rules to create valid messages, i.e., packets with data that constitute an attack.

The Injector component currently implements four different classes of tests: a delimiter test, a syntax test, a value test, and a privileged access violation test. The tool was also developed to support tests in a generic way, which means that more tests can easily be added to cover more kinds of attacks.

4.5.1 Delimiter Test

Usually, applications are thoroughly tested for its normal and expected functionality, disregarding its robustness in dealing with malformed messages. This specific type of test plays with the *initial_delimiter* and *final_-delimiter* fields of the `ElementSpec` class (see Figure 4.2). These fields represent the delimiters of a particular field or packet. For example, the IMAP protocol messages end with a *carriage return* and *line feed* characters, while each field is delimited by *space* characters.

The current implementation of AJECT swaps and deletes the *initial_-delimiter* and *final_delimiter* fields of the `ElementSpec` objects. The generated packets consist of malformed protocol messages (i.e., with invalid or missing delimiters) but with valid data.

4.5.2 Syntax Test

This kind of test generates attacks/packets that infringe the syntax specification of the protocol as provided by the TPS. Example syntax violations consist on the addition, elimination, or re-ordeting of certain fields of a correct message.

This test regards a packet as a sequence of fields, each one occupying a certain number of bits. The type of data stored in a field is considered irrelevant, therefore, a 32-bit integer is deemed equivalent to any other type of data, such as 400 characters string. The main information required by the test is the size of every field, which is usually either predefined (e.g., it always occupies 4 bytes) or determined with some special control character (e.g., the space character serves as field terminator).

As an example, consider a message containing three different fields, which is represented as [A] [B] [C]. A few of the automatically generated attack packets that could be produced are:

- [A] [B];
- [A] [C];
- [A] [A] [B] [C];
- [A] [B] [A] [C];
- [A] [B] [C] [A];

As one can see, the fields remain unchanged, it is their place in the message that changes, being removed or duplicated elsewhere.

4.5.3 Value Test

The TPS component also defines the type and validity of the data of the target protocol messages. This test class verifies if the target is able to cope with packets containing erroneous values.

An attack is generated in the following manner: each protocol message is used to generate several attack packets based on that message; also, each field of this protocol message is iteratively chosen to be the *invalid field*; all the remaining fields will hold legal data, while the invalid field is filled with malicious data. Since there are several fields in each packet, and a field can take many different malicious values, this procedure can produce a large number of attacks. With the objective of keeping this number manageable, only a subset of the invalid data, hopefully representative of the whole set, is experimented.

As an example, consider a packet with two integer fields. The first field is always set to 1, while the second field can take values between 0 and 1000. The first generated attacks would exercise different invalid values for the first field (e.g., -1, 0, and 1), while maintaining a legal value for the second value (e.g., 500). When all invalid data iterations are exhausted, the second field is chosen to be the invalid field. Then, several attacks are generated with the value 1 for the first field, and boundary and illegal values for the second (e.g., -1, 0, 1000, 1001, -100000, 100000). As one could see, this test experiments different types of invalid values: almost valid (i.e., boundary values) and very invalid (i.e., large negative/positive integers).

However, several protocols are string-based, such as the IMAP protocol, used in the experiments presented in Chapter 5. Creating attacks for this type of protocols can be achieved by generating strings from different character combinations. The construction of these malicious strings is rea-

sonably complex because it can easily lead to a combinatory explosion[4].

Also, most of these character combinations are deemed equivalent in regarding to the main characteristics that distinguish them in the program's control path. So, an heuristic-based procedure was employed for the generation of the malicious data: first, a set of random tokens (fixed sized strings of random characters) is obtained; then, a set of specified malicious tokens (e.g., “%c”, “%x”, “"”) and of joining tokens (e.g., “\”, space or none) is chosen. A large number of strings is obtained from the combination of one or more types of these different tokens. The result are strings with most of the characteristics that are usually found in hacker's exploits, such as large strings or strings with strange characters (e.g., format string specifiers).

These strings are later used in the invalid fields, hence testing the target's robustness in coping with this type of malicious input.

4.5.4 Privileged Access Violation Test

This type of test tries to induce the server in granting access to some privileged operation, such as getting secret (or private) data from the TS, or even modifying it. These privileged operations are always associated with data, such as files or directories. Such actions may involve reading some well-known file, or writing to a particular directory. The success of such protocol requests indicate the incorrect action of the server, and thus the presence of a vulnerability.

The information disclosure test follows the same steps of the previous class of test, but with a different configuration. In this test, the number of

[4]Just think that a string with 10 characters can have 26^{10} different combinations, even if we limit ourselves to the a..z characters.

random tokens is set to a minimum, while the set of malicious tokens and of joining tokens is carefully chosen. Good malicious tokens are directory path names, well-known filename, and existing usernames. These tokens are then automatically combined with the previously chosen joining tokens, such as ".", "..", or "/". This combinations generate a large number of path names to known files, which it uses during the attack generation. If a response provides valid data for one of the malicious requests, then the server is probably performing some illegal action, such as disclosing some confidential information.

For example, consider the file "/etc/passwd" that contains the usernames and encrypted passwords on a Linux machine. Some of the names that could be tried in the attacks are: ["./../etc/passwd"]; ["./../../etc/passwd"]; ["./../../../etc/passwd"].

Chapter 5

Evaluation

The previous chapters showed the attack injection methodology for the detection of vulnerabilities and its design and implementation materialized in AJECT. This chapter will present the validation of such methodology and implementation. First, a description of the basic foundation for the experiments is provided, such as the communication protocol, the hardware and software specifications, the different tests, etc. Then, the chapter presents and analyses the experimental results achieved with AJECT using several the IMAP servers.

5.1 Experimental Framework

This section gives a brief overview of the IMAP communication protocol that is utilized by the servers under test. It also describes the classes of attacks that were tried by the injector, and provides some information about the testbed.

5.1.1 IMAP Protocol

The Internet Message Access Protocol (IMAP) is a popular method for accessing electronic mail and news messages maintained on a remote server (Crispin, 2003). This protocol is specially designed for users that need to view email messages from different computers since all management tasks are executed remotely without the need to transfer the messages back and forth between these computers and the server. A client program can manipulate remote message folders (mailboxes) in a way that is functionally equivalent to local folders. The IMAP protocol provides a extensive number of operations, which include: creation, deletion and renaming of mailboxes; checking for new messages; permanently removing messages;

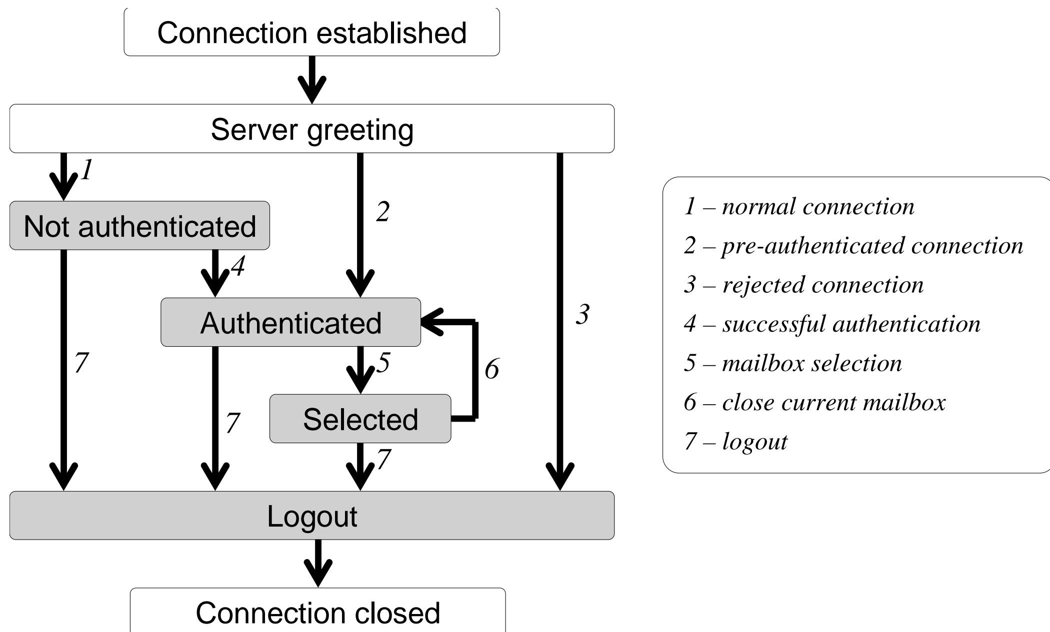


Figure 5.1: IMAP state and flow diagram.

server-based RFC-2822 and MIME messages format parsing and searching; and selective fetching of message attributes and text for efficiency.

The client and server programs communicate through a reliable data stream (typically TCP) and the server listens for incoming connections on port 143. Once a connection is established, it goes into one of four states (see Figure 5.1). Normally, it starts in the *not authenticated* state, where most operations are forbidden. If the client is able to provide acceptable authentication credentials the connection goes to the *authenticated* state. Here, the client can choose a mailbox, hence transiting to the *selected* state, and execute the commands that will manipulate the messages. The connection goes to the *logout* state when the client indicates that it no longer wants to access the messages (by issuing a LOGOUT command) or when some exceptional unilateral action occurs (e.g., server shutdown).

All interactions exchanged between the client and server are in the

form of strings that end with a CRLF (carriage return and line feed characters). The client initiates an operation by sending a command, which is prefixed with a distinct tag (e.g., a string A01, A02, etc). Depending on the type of command, the server response contains zero or more lines with data and status information, and ends with one of following completion results: OK (indicating success), NO (indicating failure), or BAD (indicating a protocol or syntax error). To simplify the matching between requests and responses, the completion result line is started with the same distinct tag provided in the client command.

5.1.2 Testbed and Implementation Issues

The experiments used several IMAP applications that were developed for different operating systems. Therefore, it was necessary to utilize a flexible testbed to ensure that the distinct requirements about the running environment could be accommodated. The testbed consisted of three PCs with Intel Pentium 4 at 2.80GHz and 512 MBytes of main memory. Two of the PCs corresponded to target systems, and each contained the IMAP applications and a Monitor. One of the machines could be booted in a few Linux flavors (e.g., Ubuntu, Fedora, and Suse) and the other on Windows (e.g., XP and 2000). The third PC ran the Injector components, collected the statistics, and performed the analysis of the results. This testbed configuration allowed for the parallel execution of two injection experiments (if needed, more PCs with target systems could easily be added, increasing the concurrency of the system).

At this moment, two Monitor components have been developed in C++, one for Linux and another for Windows. The Linux version implements all functionally that has been previously described, namely it

collects a variety of execution data about the target (e.g., UNIX signals, resource usage) and synchronizes with the Injector. The Windows version is in an early stage of development, and it only provides basic functionality. Currently the Injector is capable of generating a large number of attacks for different test classes (e.g., syntax test, value test, privileged access violation test), and performs some level of analysis on the acquired execution data. The Java language was used in the implementation of the Injector to ensure that portability issues would not arise.

5.2 Experimental Results

The current section presents an evaluation of the vulnerability discovery capabilities of AJECT. This study executed several experiments to accomplish three main objectives: One goal was to confirm that AJECT is capable of catching a significant number of vulnerabilities automatically; A second goal was to demonstrate that different classes of vulnerabilities could be located with the tool, by taking advantage of the implemented tests; A third goal was to illustrate the generic nature of the tool, by showing that it can support attack injections on distinct IMAP server applications.

To achieve these objectives, we used AJECT to expose several vulnerabilities that were reported in the past in some IMAP products. Basically, the most well-known bug tracking sites were searched for IMAP vulnerabilities that were disclosed in 2005. The available vulnerable products were then obtained and installed in the testbed. The experiments consisted in using AJECT to attack these products, to determine if the tool could detect the flaws.

Another possible approach was to spend all our resources testing a

small group of IMAP servers (one or two), trying to discover a new set of vulnerabilities. The experimental strategy presented in this thesis did not follow this approach because it would probably not allow to fulfill all experimental objectives. By lowering the number of different applications, and consequently of different development teams, the window of different classes of vulnerabilities would necessarily diminish. The same developers tend to make similar mistakes, so a larger spectrum of applications will probably contain different types of vulnerabilities.

Also, during the injection campaigns, AJECT was able to discover a new vulnerability, previously unknown to the security community.

5.2.1 Applications Under Test

To set up the experiments, vulnerability tracking sites — the *BugTraq* archive of `www.securityfocus.com`, and the *Common Vulnerabilities and Exposures* (CVE) database at `www.cve.mitre.org` — and several other hacker and security sites were searched for IMAP vulnerabilities. From this search it was possible to find 27 reports of security problems related to IMAP products during 2005. 7 of these reports were excluded because they proved themselves useless by not providing any specific information about the vulnerability itself.

From the analysis of the remaining 20 reports, it was possible to identify 9 IMAP products with vulnerabilities. In a few cases, more than one version of the same application had problems. Table 5.1 provides a summary of these applications. For each product version, the table indicates our internal identifier (*ID*), the operating system where it runs (*OS*) and the date of the first report about a vulnerability (*Date*). Sometimes other reports appeared at a later time. Column *Vuln. ID* has the identifiers of the

ID	Application	OS	Date	Vuln. ID
A1	MailEnable Professional 1.54* and Enterprise Edition 1.04*	Win	Apr	CVE-2005-1014/5, CVE-2005-2278
A2	GNU Mailutils 0.6*	Lin	May	CVE-2005-1523
A3	E-POST Inc. SPA-PRO Mail @Solomon 4.0 4*	Win	Jun	BugTraq 13838/9
A4	Novell NetMail 3.52 B*	W/L	Jun	CVE-2005-1756/7/8
A5	TrueNorth eMailServer Corporate Edition 5.2.2*	Win	Jun	BugTraq 14065
A6	Alt-N MDaemon 8.0 3*	Win	Jul	BugTraq 14315/7
A7	GNU Mailutils 0.6.1*	Lin	Sep	CVE-2005-2878
A8	University of Washington Imap 2004f*	Lin	Oct	CVE-2005-2933
A9	Floosietek FTGate 4.4*	Win	Nov	BugTraq 15449
A10	Qualcomm Eudora WorldMail Server 3.0	Win	Nov	CVE-2005-3189
A11	MailEnable Professional 1.6 and Enterprise Edition 1.1	Win	Nov	BugTraq 15492/4
A12	MailEnable Professional 1.7 and Enterprise Edition 1.1	Win	Nov	BugTraq 15556

Table 5.1: Applications with vulnerabilities.

associated reports (i.e., CVE or BugTraq identifiers). For applications with multiple reports, it was used a condensed representation — for example, CVE-2005-1014/5 corresponds to CVE-2005-1014 and CVE-2005-1015.

There were two more products identified in the reports — the Ipswitch Collaboration Suite/IMail 8.13 and the Up-IMAPProxy 1.2.4. For the two products we were able to obtain the allegedly vulnerable versions and the exploits that were distributed by the hacker community. However, for some unknown reason, neither the AJECT tool nor the public available exploits were capable of exploring the described vulnerabilities. Therefore, we decided to disregard these products for further evaluation.

5.2.2 Vulnerability Assessment

After the identification of the flawed products, it was necessary to obtain as many applications (with the right versions) as possible. However, while

attempting to obtain the reported (i.e., vulnerable) versions we met two main difficulties. First, in some cases these older versions were no longer available in the application's maintainers sites. This was specially true for commercial products, where whenever a new or patched version was produced, the older ones were removed. In most cases, where this older versions were not found in the official sites, a more thorough web search (e.g., using P2P networks) was found successful. A second problem was related to the cost of the commercial products. In these cases only the trial versions of the applications were available, which occasionally did not provide the required functionality for the discovery of the vulnerability.

Therefore, in order to assess AJECT, a different approach was employed for the unavailable applications. The Injector was used to generate and carry out the attacks against a dummy IMAP server. This simple server only stored the contents of the malicious packets received from the Injector, and returned simple responses. The packets were later analyzed to determine if one of the attacks could activate the reported vulnerability.

Table 5.2 presents some attacks generated by AJECT that successfully activated the software bugs present in the IMAP servers. Each line contains the internal application identifier (see Table 5.1), the type of bug, the IMAP state in which the attack was successful, and the attack itself. The attack injection campaigns were able to locate different types of bugs, including stack and heap buffer overflows, format strings, and information disclosure (Koziol et al., 2004, also see the next section). Information disclosure flaws may also allow other kind of attacks, specially if they could be explored with different IMAP commands, combined with write permissions. For example, a "CREATE pathname" command would allow the creation of a new file named "pathname".

ID	Vuln. Type	IMAP State	Potential Attack
A3	Access Violation	S2	A01 SELECT ./../../<OTHER-USER>/inbox
A4	Buffer Overflow	any	<A×2596>

a) Potentially detected vulnerabilities

ID	Vuln. Type	IMAP State	First Successful Attack
A1	Buffer Overflow Buffer Overflow	S2 S2	A01 AUTHENTICATE <A×1296> A01 SELECT <A×1296>
A2	Format String	any	<%s×10>
A5	Format String	S2	A01 LIST <A×10> <%s×10>
A6	Buffer Overflow Buffer Overflow	S2 any*	A01 CREATE <A×244> <A×1260>
A7	Format String	S3	A01 SEARCH TOPIC <%s×10>
A8	Buffer Overflow	S2	A01 SELECT "{localhost/user=\"}"
A9	Buffer Overflow	S2	A01 EXAMINE <A×300>
A10	Access Violation	S2	A01 SELECT ./../../<OTHER-USER>/inbox
A11	Buffer Overflow Access Violation	S2 S2	A01 SELECT <A×1296> A01 CREATE /<A×10>
A12	Denial of Service	S2	A01 RENAME <A×10> <A×10>

b) Detected previous known vulnerabilities

Application	Vuln. Type	IMAP State	First Successful Attack
TrueNorth eMailServer Corporate Edition 5.3.4	Buffer Overflow	S3	A01 SEARCH <A×560>

c) New vulnerability discovered with AJECT

Table 5.2: Attacks generated by AJECT to detect IMAP vulnerabilities. (<$A\times N$> A repeated N times; <*OTHER-USER*> OTHER-USER is other existing username; * using CRAM-MD5 auth scheme)

The results of the experiments against the dummy IMAP server are shown in Table 5.2a. These two rows display the generated attacks that, supposedly, could activate the reported vulnerabilities.

The known vulnerabilities detected with AJECT are presented in Table 5.2b. Testing several different applications is very time consuming, since besides the application retrieval, it implicates the software installation and the IMAP server configuration. Moreover, each test could take a significant amount of time to complete. Therefore, we decided to carry out

the injection campaigns only until the discovery of the first vulnerability of each application. The command corresponding to this first successful attack is presented in the last column of the table. In the few cases where experiments were left to run for a longer period, several distinct attacks were able to uncover the same problem. For example, after 24500 injections against the GNU Mailutils, there were already more than 200 attacks that similarly crashed the application.

Sometimes it was difficult to determine if distinct attacks were or not equivalent in terms of discovering the same flaw, specially in the cases where they used different IMAP commands. For example, if a bug is in the implementation of a validation routine that is called by the various commands, then the attacks would be equivalent. On the other hand, if no code was shared then there should be different bugs.

The equivalence of the attacks lies in the equivalence of the executed code instructions. If the attacks trigger the same vulnerability, i.e., the execution of the same piece of code, they are *equivalent*. However, different vulnerabilities are always detected by *non-equivalent attacks*, even the server's behavior is apparently similar. Actually, in order to find out the equivalence of the attacks, one would need to access the source code of the applications (something impossible to obtain for the majority of the products) and to monitor the instructions in real-time. Consequently, a more simplistic approach was taken: all successful attacks are deemed equivalent, except in the situations where the server's behavior or the attacks are obviously distinct, and therefore, correspond to different vulnerabilities.

During the course of our experiments, AJECT was also able to discover a previously unknown vulnerability as shown in Table 5.2c. The attack sends a large string in a SEARCH command that causes a crash in

Any State
CAPABILITY
NOOP
LOGOUT

S1) Non Authenticated
STARTTLS
AUTHENTICATE <auth mechanism>
LOGIN <username> <password>

S2) Authenticated
SELECT <mbox>
EXAMINE <mbox>
CREATE <mbox>
DELETE <mbox>
RENAME <mbox> <new name>
SUBSCRIBE <mbox>
UNSUBSCRIBE <mbox>
LIST <reference> <mbox [wildcards]>
LSUB <reference> <mbox [wildcards]>
STATUS <mbox> <status data items...>
APPEND <mbox> [flag list] [date] <msg literal>

S3) Selected
CHECK
CLOSE
EXPUNGE
SEARCH [charset spec] <criteria...>
FETCH <seq set> <msg data | macro>
STORE <seq set> <msg data> <value>
COPY <seq set> <mbox>
UID <COPY | FETCH |... > <args>

Table 5.3: Commands tested in each IMAP state.

the server. This indicates that the bug is a boundary condition verification error, which corresponds to a buffer overflow. Several versions of the eMailServer application were tested, including the most recent one, and all of them were vulnerable to this attack.

5.2.3 Test Results

In Table 5.3 are represented the commands that were experimented in the various IMAP states. Some of the commands are very simple (e.g., composed by a single field) but others are much more intricate. As expected, the number of malicious packets generated from each command specification is proportional to its complexity.

The remainder of this section will provide some explanations for the attack injection results presented in Table 5.2.

Delimiter test

This class of test first retrieves the delimiters characters from the TPS. Each protocol field was separated by a space character, so this was the field's final delimiter. The initial tag (used in every protocol message) was defined as the packet's initial delimiter. So, in the TPS definition, "A001 " was specified as being the initial delimiter. For the message's final delimiter, the RFC-3501 (Crispin, 2003) specifies the carriage return and line feed characters (or CRLF for short).

Attacks directed at the packet's initial delimiter resulted in the server assuming that the first field was the initial tag, when in fact was the IMAP command. Since the protocol command was being mistaken for the initial tag, these attacks were instantly rejected for *unknown command* reasons.

It was interesting to observe that most IMAP servers did not require the packet's final delimiter to be CRLF, but just CR or *newline*. When omitted, the servers concatenated the messages forming a larger message. This was not in conformance with the goal of the tool, which was to create single, independent, and easily reproducible attacks, instead of a strange conjunction of packets from different attacks.

Att. No.	Attack Packet	Description	
...			
328	SELECT	*removed field*	SELECT
329	/inbox	*removed field*	
330	/inbox SELECT /inbox	*duplicated field*	
331	SELECT SELECT /inbox	*duplicated field*	
332	SELECT /inbox /inbox	*duplicated field*	
333	SELECT /inbox SELECT	*duplicated field*	
334	SELECT SELECT	*rem. and dupl. field*	
335	/inbox /inbox	*rem. and dupl. field*	
336	EXAMINE	*removed field*	EXAMINE
337	/inbox	*removed field*	
338	/inbox EXAMINE /inbox	*duplicated field*	
...			

Table 5.4: Syntax test attacks sample.

The same concatenation behavior happened with the field's final delimiter. This time it would concatenate the fields, still maintaining the packet's integrity. However, no abnormal behavior was detected from any of the attacks generated from this class of test.

Syntax test

Another class of test that did not produce any detected abnormal behavior in the TS was the syntax test. The generated attacks were very simple and were quickly dismissed by the parsing validation mechanisms.

Table 5.4 shows a subset of the generated attacks using this test class. These example attacks are packet variations of the SELECT and EXAMINE commands. The field contents are kept unchanged, but they are removed or duplicated elsewhere.

By infringing the syntax of the protocol in such an obvious way, the attacks were immediately dismissed by the validation routines, so no vulnerabilities were detected by the syntax test.

Value test

This test was very successful in detecting buffer overflow, denial of service, and format string vulnerabilities, because it focused on the generation of malicious data (e.g., long strings or strings with format string specifiers).

The idea behind the attack generation is very simple. As explained earlier, a set of malicious and joining tokens was previously specified. Then, the value test will generate various combinations from this tokens with some random data. The resulting attacks are packets with some invalid fields that explore some characteristics usually found in hacker exploits.

With this class of test, AJECT was able to detect 11 known vulnerabilities: 7 buffer overflow, 1 denial of service, and 3 format string vulnerabilities. A new and previously unknown buffer overflow vulnerability was also detected with this test.

Privileged access violation test

The goal of this test is to generate protocol requests that induce the server into performing some privileged action, without the necessary credentials. Three IMAP servers were found vulnerable to these attacks that tried to get secret (or private) data from the target system, or even to modify it. Such information is usually found in the server's hard-disk or memory, and it can correspond, for instance, to passwords kept in a configuration file or in memory resident environment variables. Hence, this test resorts to some special tokens, such as well-known file, directory, and user names.

On the IMAP protocol there a few arguments of some commands that are used to name a file. For example, the *mbox* on the EXAMINE command refers to a mailbox, which is specified by its file system path. So,

this is a very interesting field for information disclosure vulnerabilities, or more general access violations. Actually, both detected vulnerabilities were related to the mailbox field: an information disclosure vulnerability, and another that granted write access to any directory.

Chapter 6

Conclusion

This thesis presents a tool for the discovery of vulnerabilities in server applications. AJECT simulates the behavior of a malicious adversary by injecting different kinds of attacks against the target server. In parallel, it observes the running application in order to collect various information. This information is later analyzed to determine if the server executed incorrectly, which is a strong indication that a vulnerability exists.

The attack injection methodology and its implementation are accomplished with a modular design and architecture. In fact, AJECT is relatively portable to different systems, since the Injector runs on a Java virtual machine. Currently, any Unix-based server can be fully tested with AJECT. Microsoft Windows-based servers can also be experimented, but with minimal functionality.

Additionally, different kinds of servers can be tested by providing their protocol specifications (i.e., XML files). Actually, by using this specification in the attack generation, the space of test-cases is not restricted by the intrinsics of the specific target protocol. Hence, AJECT can produce many test-cases independently of the target system.

Besides the good results achieved with the current implementation, new classes of test can be easily created and accommodated into AJECT, increasing its vulnerability coverage.

Another important feature present in AJECT is its automatic operation. The tool performs automatic test-case generation (i.e., the creation of the attacks) and injection, while at the same time it launches, terminates, and monitors the target server.

To evaluate the usefulness of the tool, several experiments were conducted with many IMAP products. These experiments indicate that AJECT could be utilized to locate a significant number of distinct types of vulner-

abilities (e.g., buffer overflows, format strings, and information disclosure bugs). In addition, AJECT was able to discover a new buffer overflow vulnerability.

6.1 Future Work

We believe this area still has plenty to offer in terms of research opportunities. We plan to further ameliorate this attack injection methodology and corresponding implementation. As for the the Injector we would like to improve the test-case generation. With the currently available tests, experimenting a single IMAP server can take up to 21 days of continuous testing (taking 10 seconds per attack, for a total of 188400 attacks). However, it was observed that most of the attacks end up being equivalent, so a more efficient attack generation could provide a smaller number of attacks for the same results. Additionally, the efficiency of each attack could also be improved, and we would like to create new test classes that could detect more vulnerabilities.

As for the Monitor there are some aspects that could also be improved. First, the detection and monitoring capabilities could be enhanced — there are potentially other behavior characteristics capable of revealing abnormal software states. And second, we would like to complete the Monitor's porting to other operating systems.

Though AJECT does not require access to the source code of the target application, its integration could add new interesting features. One could better evaluate the injection campaigns by analyzing the source code coverage. This could also help to leverage the quality of the attack generation. Real-time source code monitoring could also be used to find the equiva-

lence of the attacks (and thus aborting similar attacks), or even to pinpoint the exact vulnerability location.

On the other hand, there are also other research areas that could contribute to the vulnerability detection process. Perhaps, by conjugating the advantages of other V&V techniques, such as model-checking or static vulnerability analyzers.

Bibliography

AIDEMARK, J., VINTER, J., FOLKESSON, P., & KARLSSON, J., 2001. GOOFI: Generic Object-Oriented Fault Injection Tool. In: *Proceedings of the International Conference on Dependable Systems and Networks*, pages 83–88.

ALBINET, A., ARLAT, J., & FABRE, J.-C., 2004. Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. In: *Proceedings of the International Conference on Dependable Systems and Networks*, pages 867–876.

ARLAT, J., COSTES, A., CROUZET, Y., LAPRIE, J.-C., & POWELL, D., 1993. Fault Injection and Dependability Evaluation of Fault-Tolerant Systems. In: *IEEE Transactions on Computers*, 42(8):913–923.

ARLAT, J., CROUZET, Y., KARLSSON, J., FOLKESSON, P., FUCHS, E., & LEBER, G. H., 2003. Comparison of Physical and Software-Implemented Fault Injection Techniques. In: *IEEE Transactions on Computers*, 52(9):1115–1133.

ARLAT, J., CROUZET, Y., & LAPRIE, J.-C., 1989. Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems. In: *Pro-*

ceedings of the International Symposium on Fault-Tolerant Computing, pages 348–355.

ASHENDEN, P. J., 1990. The VHDL Cookbook. Technical report, University of Adelaide, South Australia.

BETOUIN, P. P., 2006. BSS (Bluetooth Stack Smasher). Http://www.secuobs.com/news/05022006-bluetooth10.shtml.

BIEGE, T., 2005. Radius Fuzzer. Http://www.suse.de/ thomas/index.html.

BISHOP, M. & DILGER, M., 1996. Checking for Race Conditions in File Accesses. In: *Computing Systems*, 9(2):131–152.

BROWN, A., CHUNG, L. C., & PATTERSON, D. A., 2002. Including the Human Factor in Dependability Benchmarks. In: *Workshop on Dependability Benchmarking, in Supplemental Volume of DSN 2002*, pages F–9–14.

BUSH, W. R., PINCUS, J. D., & SIELAFF, D. J., 2000. A Static Analyzer for Finding Dynamic Programming Errors. In: *Software – Practice & Experience*, 30(7):775–802.

CARREIRA, J., MADEIRA, H., & SILVA, J. G., 1995. Xception: Software Fault Injection and Monitoring in Processor Functional Units. In: *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 135–149.

CARREIRA, J., MADEIRA, H., & SILVA, J. G., 1998. Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers. In: *IEEE Transactions on Software Engineering*, 24(2):125–136.

CERT COORDINATION CENTER, 2006. Statistics 1988-2006. Http://www.cert.org/stats/.

CHEN, H., DEAN, D., & WAGNER, D., 2004. Model Checking One Million Lines of C Code. In: *Proceedings of the Network and Distributed System Security Symposium*, pages 171–185.

CHEN, H. & WAGNER, D., 2002. MOPS: an Infrastructure for Examining Security Properties of Software. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 235–244.

CHESS, B. & MCGRAW, G., 2004. Static Analysis for Security. In: *IEEE Security & Privacy*, 2(6):76–79.

CHOI, G. S. & IYER, R. K., 1992. FOCUS: An Experimental Environment for Fault Sensitivity Analysis. In: *IEEE Transactions on Computers*, 41(12):1515–1526.

CHRISTMANSSON, J. & CHILLAREGE, R., 1996. Generation of an Error Set that Emulates Software Faults. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 304–313.

CLARK, J. A. & PRADHAN, D. K., 1993. REACT: A Synthesis & Evaluation Tool for Fault-tolerant Multiprocessor Architectures. In: *Reliability and Maintainability Symposium*, pages 428–435.

CLARKE, E. M., GRUMBERG, O., & LONG, D. E., 1994. Model Checking and Abstraction. In: *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542.

CLARKE, E. M., GRUMBERG, O., & PELED, D. A., 2000. *Model Checking*. The MIT Press.

COWAN, C., BEATTIE, S., JOHANSEN, J., & WAGLE, P., 2003. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In: *Proceedings of the USENIX Security Symposium*, pages 91–104.

COWAN, C., MCNAMEE, D., BLACK, A., PU, C., WALPOLE, J., KRASIC, C., WAGLE, P., ZHANG, Q., & MARLET, R., 1997. A Toolkit for Specializing Production Operating System Code. Technical Report CSE-97-004, Oregon Graduate Institute of Science and Technology.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., & HINTON, H., 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In: *Proceedings of the USENIX Security Conference*, pages 63–78.

CRISPIN, M., 2003. Internet Message Access Protocol - Version 4rev1. Internet Engineering Task Force, RFC 3501.

CROUZET, Y. & DECOUTY, B., 1982. Measurements of Fault Detection Mechanisms Efficiency: Results. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 373–376.

DURÃES, J. & MADEIRA, H., 2003. Definition of Software Fault Emulation Operators: A Field Data Study. In: *Proceedings of the International Conference on Dependable Systems and Networks*, pages 105–114.

DURÃES, J. & MADEIRA, H., 2005. A Methodology for the Automated Identification of Buffer Overflow Vulnerabilities in Executable Software Without Source-Code. In: C. A. Maziero, J. G. Silva, A. M. S. Andrade, & F. M. de Assis Silva, editors, *Proceedings of the 2nd Latin-American Symposium on Dependable Computing*, vol. 3747 of *Lecture Notes in Computer Science*, pages 20–34.

eEYE DIGITAL SECURITY, 2006. Retina Network Security Scanner. Http://www.eeye.com.

ELSMAN, M., FOSTER, J. S., & AIKEN, E., 1999. Carillon – a System to Find Y2K Problems in C Programs. Http://bane.cs.berkeley.edu/carillon.

EVANS, D., GUTTAG, J., HORNING, J., & TAN, Y. M., 1994. LCLint: A Tool for Using Specifications to Check Code. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96.

FARMER, D. & VENEMA, W., 1995. SATAN – Security Administrator Tool for Analyzing Networks. Http://www.porcupine.org/satan/.

FOSTER, J. S., FÄHNDRICH, M., & AIKEN, A., 1999. A Theory of Type Qualifiers. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203.

FYODOR, 1997. Nmap Security Scanner. Http://insecure.org/nmap/.

GNU FOUNTATION, 2006. GDB. Http://www.gnu.org/software/gdb/.

GOSWAMI, K. K. & IYER, R. K., 1990. DEPEND: A Design Environment for Prediction and Evaluation of System Dependability. In: *Proceedings of the Digital Avionics Systems Conference*, pages 87–92.

GOSWAMI, K. K., IYER, R. K., & YOUNG, L. T., 1997. DEPEND: A Simulation-Based Environment for System Level Dependability Analysis. In: *IEEE Transactions on Computers*, 46(1):60–74.

GREENE, A., 2005. SPIKEfile. Http://labs.idefense.com/labs-software.php?show=14.

GUNNEFLO, U., KARLSSON, J., & TORIN, J., 1989. Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 340–347.

HAUGH, E. & BISHOP, M., 2003. Testing C Programs for Buffer Overflow Vulnerabilities. In: *Proceedings of the Symposium on Networked and Distributed System Security*, pages 123–130.

HEIN, A. & GOSWAMI, K., 1995. Combined Performance and Dependability Evaluation with Conjoint Simulation. In: *Proceedings of the European Simulation Symposium*, pages 26–28.

HSUEH, M.-C. & TSAI, T. K., 1997. Fault Injection Techniques and Tools. In: *IEEE Computer*, 30(4):75–82.

INTERNET SECURITY SYSTEMS INC., 2006. Internet Scanner. Http://www.iss.net.

JENN, E., ARLAT, J., RIMÉN, M., OHLSSON, J., & KARLSSON, J., 1994. Fault Injection into VHDL Models: The MEFISTO Tool. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 66–75.

KARLSSON, J., ARLAT, J., & LEBER, G., 1995. Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture. In: *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 267–287.

KARLSSON, J., LIDEN, P., DAHLGREN, P., JOHANSSON, R., & GUNNEFLO, U., 1994. Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms. In: *IEEE Micro*, 14(1):8–11, 13–23.

KOOPMAN, P. & DEVALE, J., 1999. Comparing the Robustness of POSIX Operating Systems. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 30–37.

KOZIOL, J., LITCHFIELD, D., AITEL, D., ANLEY, C., EREN, S., MEHTA, N., & HASSELL, R., 2004. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. John Wiley & Sons.

KUMAR, S., KLENKE, R., AYLOR, J. H., W., B., JOHNSON, WILLIAMS, R. D., & WAXMAN, R., 1994. ADEPT: A Unified System Level Modeling Design Environment. In: *Proceedings of the Rapid Prototyping of Application Specific Signal Processors Conference*, pages 114–123.

LALA, J. H., 1983. Fault Detection, Isolation and Reconfiguration in FTMP: Methods and Experimental Results. In: *AIAA/IEEE Digital Avionics Systems Conference*, pages 21.3.1–21.3.9.

LAROCHELLE, D. & EVANS, D., 2001. Statically Detecting Likely Buffer Overflow Vulnerabilities. In: *Proceedings of the USENIX Security Symposium*, pages 177–190.

MADEIRA, H., RELA, M. Z., MOREIRA, F., & SILVA, J. G., 1994. RIFLE: A General Purpose Pin-level Fault Injector. In: *European Dependable Computing Conference*, vol. 852 of *Lecture Notes in Computer Science*, pages 199–216.

MARTÍNEZ, R. J., GIL, P. J., MARTÍN, G., PÉREZ, C., & SERRANO, J. J., 1999. Experimental Validation of High-Speed Fault-Tolerant Systems using Physical Fault Injection. In: B. Weinstock, Charles & J. Rushby, editors, *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, pages 249–268.

MCAFEE, INC., 2006. FoundStone Enterprise. Http://www.foundstone.com.

MILLER, B. P., FREDRIKSEN, L., & SO, B., 1990. An Empirical Study of the Reliability of UNIX Utilities. In: *Communications of the ACM*, 33(12):32–44.

MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., & DILL, D. L., 2002. CMC: A Pragmatic Approach to Model Checking Real Code. In: *Proceedings of the Operating System Design and Implementation*, pages 75–88.

OEHLERT, P., 2005. Violating Assumptions with Fuzzing. In: *IEEE Security and Privacy*, 03(2):58–62.

POWELL, D. & STROUD, R., editors, 2002. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*. Http://www.research.ec.org/maftia/deliverables/D21.pdf.

QUALYS INC., 2006. QualysGuard Enterprise. Http://www.qualys.com.

SCHWETMAN, H., 1986. CSIM: a C-Based Process-Oriented Simulation Language. In: *Proceedings of the Conference on Winter simulation*, pages 387–396.

SHANKAR, U., TALWAR, K., FOSTER, J. S., & WAGNER, D., 2001. Detecting Format String Vulnerabilities with Type Qualifiers. In: *Proceedings of the USENIX Security Symposium*, pages 201–220.

SIEH, V., TSCHÄCHE, O., & BALBACH, F., 1997. VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions. In:

Proceedings of the International Symposium on Fault-Tolerant Computing, pages 32–36.

SUTTON, M., 2005. FileFuzz. Http://labs.idefense.com/labs-software.php?show=3.

TENABLE NETWORK SECURITY, 2006a. Nessus Vulnerability Scanner. Http://www.nessus.org.

TENABLE NETWORK SECURITY, 2006b. Tenable Passive Vulnerability Scanner. Http://www.tenablesecurity.com.

TSAI, T. & IYER, R., 1996. An approach towards Benchmarking of Fault-Tolerant Commercial Systems. In: *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 314–323.

TSAI, T. K. & IYER, R. K., 1995. Measuring Fault Tolerance with the FTAPE Fault Injection Tool. In: *International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, vol. 977 of *Lecture Notes in Computer Science*, pages 26–40.

UNIVERSITY OF OULU, 1999–2003. PROTOS – Security Testing of Protocol Implementations. Http://www.ee.oulu.fi/research/ouspg/protos/.

VENDICATOR, 2000. Stack Shield : A Stack Smashing Technique Protection Tool for Linux. Http://www.angelfire.com/sk/stackshield/.

VERÍSSIMO, P., NEVES, N. F., & CORREIA, M., 2003. Intrusion-Tolerant Architectures: Concepts and Design. In: R. Lemos, C. Gacek, & A. Romanovsky, editors, *Architecting Dependable Systems*, vol. 2677 of *Lecture Notes in Computer Science*, pages 3–36.

VERÍSSIMO, P., NEVES, N. F., & CORREIA, M., 2000. The Middleware Architecture of MAFTIA: A Blueprint. In: *Proceedings of the Third IEEE Information Survivability Workshop*.

VIEGA, J., BLOCH, J. T., KOHNO, Y., & MCGRAW, G., 2000. ITS4: A Static Vulnerability Scanner for C and C++ Code. In: *Proceedings of the Computer Security Applications Conference*, page 257.

VISSER, W., HAVELUND, K., BRAT, G. P., & PARK, S., 2000. Model Checking Programs. In: *Proceedings of the Automated Software Engineering*, pages 3–12.

WAGNER, D., FOSTER, J. S., BREWER, E. A., & AIKEN, A., 2000. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In: *Proceedings of the Network and Distributed System Security Symposium*, pages 3–17.

WILANDER, J. & KAMKAR, M., 2003. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In: *Proceedings of the Network and Distributed System Security Symposium*, pages 149–162.

YANG, J., TWOHEY, P., ENGLER, D. R., & MUSUVATHI, M., 2004. Using Model Checking to Find Serious File System Errors. In: *Proceedings of the Operating System Design and Implementation*, pages 273–288.