Vulnerability Discovery with Attack Injection

João Antunes, *Student Member*, *IEEE*, Nuno Neves, *Member*, *IEEE*, Miguel Correia, *Member*, *IEEE*, Paulo Verissimo, *Fellow*, *IEEE*, and Rui Neves

Abstract—The increasing reliance put on networked computer systems demands higher levels of dependability. This is even more relevant as new threats and forms of attack are constantly being revealed, compromising the security of systems. This paper addresses this problem by presenting an attack injection methodology for the automatic discovery of vulnerabilities in software components. The proposed methodology, implemented in AJECT, follows an approach similar to hackers and security analysts to discover vulnerabilities in network-connected servers. AJECT uses a specification of the server's communication protocol and predefined test case generation algorithms to automatically create a large number of attacks. Then, while it injects these attacks through the network, it monitors the execution of the server in the target system and the responses returned to the clients. The observation of an unexpected behavior suggests the presence of a vulnerability that was triggered by some particular attack (or group of attacks). This attack can then be used to reproduce the anomaly and to assist the removal of the error. To assess the usefulness of this approach, several attack injection campaigns were performed with 16 publicly available POP and IMAP servers. The results show that AJECT could effectively be used to locate vulnerabilities, even on well-known servers tested throughout the years.

Index Terms—Testing and debugging, software engineering, test design, testing tools, experimental evaluation, fault injection, attack injection.

1 Introduction

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 advancements in software development have provided us with an increasing number of useful applications with an ever-improving functionality. These enhancements, however, are achieved in most cases with larger and more complex projects, which require the coordination of several teams. Third party software, such as COTS components, is frequently utilized to speed up development, even though in many cases it is poorly documented and supported. In the background, the ever-present trade-off between thorough testing and time to deployment affects the quality of the software. These factors, allied to the current development and testing methodologies, have proven to be inadequate and insufficient to construct dependable software. Everyday, new vulnerabilities are found in what was previously believed to be secure applications, unlocking new risks and security hazards that can be exploited by malicious adversaries.

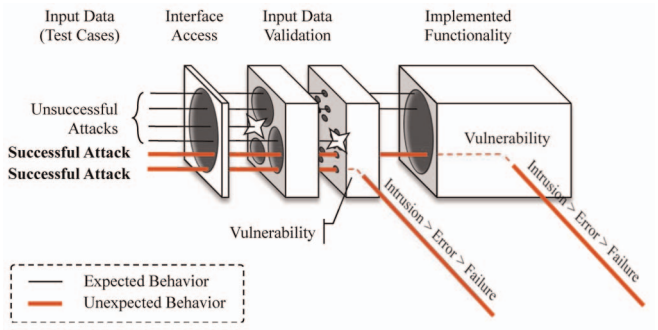
The paper describes an attack injection methodology that can be used for vulnerability detection and removal. It mimics the behavior of an adversary by injecting attacks against a target system while inspecting its execution to determine if any of the attacks has caused a failure. The observation of some abnormal behavior indicates that an attack was successful in triggering an existing flaw. After the identification of the problem, traditional debugging techniques can be employed, for instance, by examining the application's control flow while processing the offending attacks, to locate the origin of the vulnerability and to proceed with its elimination.

The methodology was implemented in a tool called AJECT. The tool was designed to look for vulnerabilities in network server applications, although it can also be utilized with local *daemons*. We chose servers because, from a security perspective, they are probably the most relevant components that need protection because they constitute the primary contact points of a network facility. AJECT does not need the source code of the server to perform the attacks, i.e., it treats the server as a black box. However, in order to be able to generate intelligent attacks, AJECT has to obtain a specification of the protocol utilized in the communication with the server.

To demonstrate the usefulness of our approach, we have conducted 58 attack injection experiments with 16 e-mail servers running POP and IMAP services. The main objective was to investigate if AJECT could automatically discover previously unknown vulnerabilities in fully developed and up-to-date server applications. Although the number and type of target applications was not exhaustive, they are nevertheless a representative sample of the universe of the network servers. Our evaluation confirmed that AJECT could find different classes of vulnerabilities in five of the servers, and assist the developers in their removal by providing the test cases, that is, the attack/vulnerability/intrusion syndromes. These experiments also lead to other interesting conclusions. For instance, we confirmed the expectation that complex protocols are much more prone to vulnerabilities than simpler ones since all detected vulnerabilities were related to the IMAP protocol.

- *J. Antunes, N. Neves, M. Correia, and P. Verissimo are with the Faculty of Sciences, University of Lisboa, Bloco C6, Piso 3, Campo Grande, 1749-016 Lisboa, Portugal. E-mail: {jantunes, nuno, mpc, pjv}@di.fc.ul.pt.*
- *R. Neves is with the Instituto de Telecomunicações, Instituto Superior Tecnico, Technical University of Lisbon, Torre Norte 9-Andar, Av. Rovisco Pais, 1, 1049-001 Lisboa, Portugal. E-mail: rui.neves@tagus.ist.utl.pt.*

Manuscript received 14 Oct. 2008; revised 2 Nov. 2009; accepted 18 Nov. 2009; published online 11 Dec. 2009.
Recommended for acceptance by K. Goseva and K. Kanoun.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-2008-10-0339.
Digital Object Identifier no. 10.1109/TSE.2009.91.



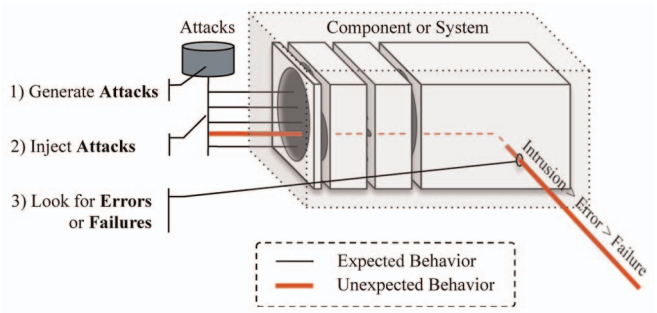


Fig. 2. The attack injection methodology.

Additionally, based on the 16 e-mail servers, we found that closed source applications appear to have a higher predisposition to contain vulnerabilities (none of the open source servers was found vulnerable whereas 42 percent of the closed source servers had problems).

2 Using Attacks to Find Vulnerabilities

Vulnerabilities are usually caused by subtle anomalies that only emerge in such unusual circumstances that were not even contemplated in test design. They tend to elude the traditional software testing methods, mainly because conventional test cases do not cover all of the obscure and unexpected usage scenarios. Hence, vulnerabilities are typically found either by accident or by attackers or special tiger teams (also called penetration testers) who perform thorough security audits. The typical process of manually searching for new vulnerabilities is often slow and tedious. Specifically, the source code must be carefully scrutinized for security flaws or the application has to be exhaustively experimented with several kinds of input (e.g., unusual and random data, or more elaborate input based on previously known exploits) looking for problems during its execution.

Fig. 1 shows a model of a component with existing vulnerabilities. Boxes in the figure represent the different modules or software layers that compose the component, with the holes symbolizing access being allowed (as intended by the developers or inadvertently through some vulnerability). Lines depict the interaction between the various layers. The same rationale can be applied recursively to any abstraction level of a component, from the smallest subcomponent to more complex and larger systems, so we will use the terms component and system interchangeably.

The external access to the component is provided through a known *Interface Access*, which receives the input arriving, for instance, in network packets or disk files, and eventually returns some output. Whether the component is a simple function that performs a specific task or a complex system, its intended functionality is, or should be, protected by *Input Data Validation* layers. These additional layers of control logic are supposed to regulate the interaction with the component, allowing it to execute the service specification only when the appropriate circumstances are present (e.g., if the client messages are in compliance with the protocol specification or if the procedure parameters are within some bounds). In order to achieve this goal, these layers are responsible for the parsing and validation of the arriving data. The purpose of a component is defined by its *Implemented Functionality*. This last layer corresponds to the implementation of the service specification of the component, i.e., it is the sequence of instructions that controls its behavior to accomplish some well-defined objective, such as responding to client requests according to some standard network protocol.

By accessing the interface, an adversary may persistently look for vulnerabilities by stressing the component with unusual forms of interaction, such as sending wrong message types or opening malformed files. These *attacks* are malicious interaction faults against the component's interface [1]. A dependable system should continue to operate correctly, even in the presence of these faults, i.e., it should keep executing in accordance with the service specification. However, if one of these attacks causes an abnormal behavior of the component, it suggests the presence of a *vulnerability* somewhere on the execution path of its processing logic.

Vulnerabilities are faults caused by design, configuration, or implementation mistakes, susceptible to being exploited by an attack to perform some unintended and usually illegal activity. The component, failing to properly process the offending attack, enables the attacker to access the component in a way unpredicted by the designers or developers, causing an *intrusion*. This further step toward failure is normally succeeded by the production of an *erroneous* state in the system (e.g., a root shell). Consequently, if nothing is done to handle the error (e.g., prevent the execution of commands in the root shell), the system will *fail*.

3 The Attack Injection Methodology

The attack injection methodology adapts and extends classical fault injection techniques to look for security vulnerabilities. The methodology can be a useful asset in increasing the dependability of computer systems because it addresses the discovery of this elusive class of faults. An attack injection tool implementing the methodology mimics the behavior of an external adversary that systematically attacks a component, hereafter referred to as the *target system*, while monitoring its behavior. An illustration of the main actions that need to be performed by such a tool is represented in Fig. 2.

First, several attacks are generated in order to fully evaluate the target system's intended functionality (step 1). In order to get a higher level of confidence about the

absence of vulnerabilities, the attacks have to be exhaustive and should look for as many classes of flaws as possible. It is expected that the majority of the attacks are deflected by the input data validation mechanisms, but others will be allowed to proceed further along the execution path, testing deeper into the component. Each attack is a single test case that exercises some part of the target system, and the quality of these tests determines the coverage of the detectable vulnerabilities.

Ideally, one would like to build test cases that would not only exercise all reachable computer instructions but also try them with every possible instance of input. This goal, however, is unfeasible for most systems due to the amount of effort necessary to generate the various combinations of input data and then to execute them. The effort can be decreased by resorting to the analysis of the source code, and by manually creating good test cases. This approach requires a great deal of experience and acuteness from the test designers, and even then, some vulnerabilities can be missed altogether. In addition, source code might be unavailable because it is common practice to reuse general purpose components developed by third parties. To overcome these limitations and to automate the process of discovering vulnerabilities, we propose generating a large number of test cases from a specification of the component's interface.

The tool should then carry out the attacks (step 2) while monitoring how the state of the component is evolving, looking for any unexpected behavior (step 3). Depending on its monitoring capabilities, the tool could examine the target system's outputs, its allocated system resources, or even the last system calls it executed. Whenever an error or failure is observed, it indicates that a new vulnerability has potentially been discovered. For instance, a vulnerability is likely to exist in the target system if it crashes during (or after) the injection of an attack—this attack at least compromises the availability of the system. Likewise, if what is observed is the abnormal creation of a large file, this can eventually lead to disk exhaustion and subsequent denial-of-service, so it should be further investigated.

The collected evidence provides useful information about the location of the vulnerability and supports its subsequent removal. System calls and the component responses, along with the offending attack, can identify the protocol state and the execution path to find the flaw more accurately. If locating and removing the vulnerability is unfeasible or a more immediate action is required, for instance, if the target system is a COTS component or a fundamental business-related application, the attack description could be used to take preventive actions, such as adding new firewall rules or IDS filters. By blocking similar attacks, the vulnerability can no longer be exploited, thus improving the system's dependability.

4 The Attack Injection Tool

The attack injection methodology can be applied to any type of component that we wish to search for vulnerabilities. Several implementations of this methodology could be created to evaluate different types of targets, from simple COTS components to entire systems. In this study, we decided to focus on network servers because, from a security point of view, this is probably the most interesting class of target systems. First, these large and often complex applications are designed to sustain long periods of uninterrupted operation and are usually accessible through the Internet. Second, an intrusion in a network server usually has a significant impact since a corruption on the server may compromise the security of all clients (e.g., if the adversary gets a root shell). Consequently, network servers are a highly coveted target by malicious hackers.

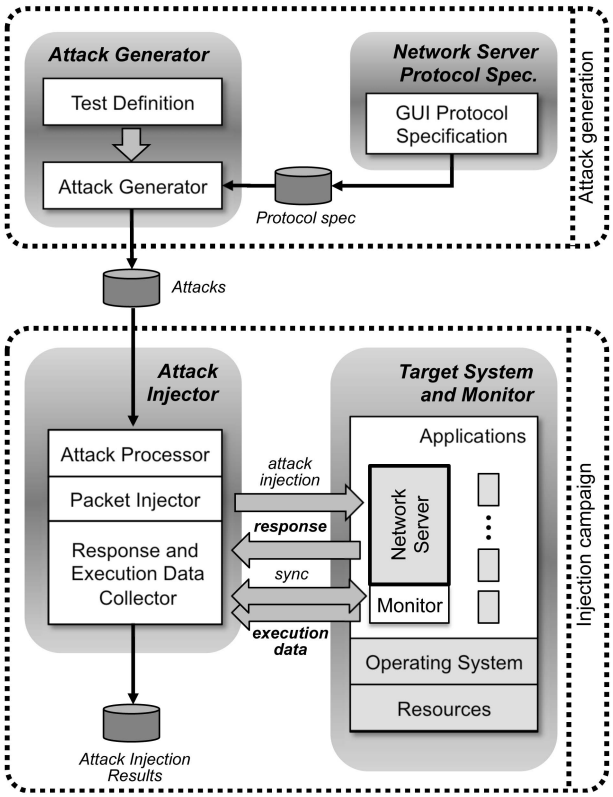


Fig. 3. The architecture of the AJECT tool.

The **A**ttack in**JEC**tion **T**ool (AJECT) is a vulnerability detection tool that implements the proposed methodology. Its architecture and main components can be seen in Fig. 3. The architecture was developed to achieve automatic injection of attacks independently of the target server's implementation. Furthermore, it was built to be flexible regarding the classes of vulnerabilities that can be discovered and the method used to monitor the target system. Therefore, AJECT's implementation provides a framework to create and evaluate the impact of different test case generation algorithms (i.e., by supplying various *Test Definitions*) and other monitoring approaches (i.e., by implementing custom *Monitors*).

The *Target System* is the entire software and hardware components that comprise the target application and its execution environment, including the operating system, the software libraries, and the system resources. The *Network Server* is typically a service that can be queried remotely from client programs (e.g., a mail or FTP server). The target application uses a well-known protocol to communicate with the clients, and these clients can carry out attacks by transmitting erroneous packets. If the packets are not correctly processed, the target can suffer various kinds of

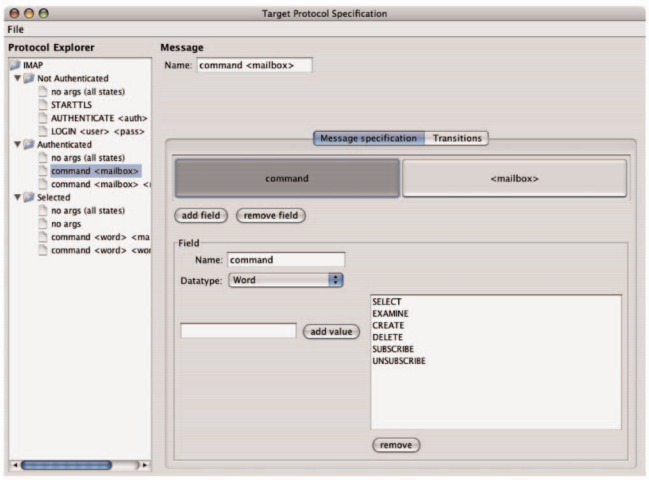


Fig. 4. Screenshot of the AJECT protocol specification.

errors with distinct consequences, ranging, for instance, from a slowdown to a crash.

The *Network Server Protocol Specification* is a graphical user interface component that supports the specification of the communication protocol used by the server. This specification is utilized by the *Attack Generator* to produce a large number of test cases. The *Attack Injector* is responsible for the actual execution of the attacks by transmitting malicious packets to the server. It also receives the responses returned by the target and the remote execution profile collected by the *Monitor*. Some analysis on the information acquired during the attack is also performed (e.g., such as known fatal signals or connection error) to determine if a vulnerability was exposed.

The overall attack injection process is carried in two separate phases: the attack generation phase, performed once per communication protocol, and the injection campaign, executed once per target system.

4.1 Attack Generation Phase

The purpose of *attack generation* is to create a series of attacks that can be injected in the target system. The design of the tool does not require the source code of the server to be available to devise the attacks. This allows AJECT to support a larger number of target systems, such as commercial servers. Instead, the tool employs a specification of the communication protocol of the server, which, in practice, characterizes the server's external interface to the clients. Therefore, by exploring the input space defined by the protocol, it is possible to exercise much of the intended functionality of the target, i.e., the parts of the code that are executed when processing the clients' requests. In contrast to the source code, which is often inaccessible, communication protocols tend to be reasonably well documented, at least for standard servers (e.g., the Internet protocols produced by IETF). Consequently, even if the information about a server is scarce, it is still possible to create good test cases as long as there is some knowledge about the communication protocol.

AJECT offers a graphical user interface tool, called *Network Server Protocol Specification*, to carry out the specification of the communication protocol (see Fig. 4). The tool operator can describe the protocol states and messages, and identify the data types and acceptable ranges of values of each field of a message. Messages are divided into two kinds: messages that request the execution of some specific operation (not changing the state of the protocol) and transition messages that make the protocol jump from one state to another (e.g., a login message). AJECT uses this information to explore the entire protocol state space by creating test cases with innocuous transition messages preceding the attack message. This procedure is exhaustive because all states are eventually tested with every operation that is defined for each state.

Attack generation is dictated by the particular test case generation algorithm, which receives as input the specification of the protocol. The algorithm should aim to maximize the protocol space coverage in order to exercise most of the server's functionality, leading to better code coverage. However, one should keep in mind that full protocol (or code) coverage does not guarantee complete and comprehensive defect discovery. Some vulnerabilities may remain dormant because the defective portion of code was not executed with the required conditions (e.g., a message field data was not large enough to overflow a local buffer) or the erroneous code was just never reached (e.g., the protocol documentation did not contemplate some nonstandard or legacy feature). This problem is, in general, unsolvable for nontrivial servers, and therefore, our objective should be to locate as many flaws as a remote attacker, within the time constraints available for testing.

In the following sections, we provide some details about the set of currently implemented algorithms, which was developed to accommodate several classes of common errors, such as delimiter, syntax, and field-value errors [2], plus a variation of the latter to look for vulnerabilities related to privileged access violations.

4.1.1 Delimiter Test Definition

This specific type of test creates messages with illegal or missing delimiters of a field. For example, on text-based protocols, each field is delimited by a *space* character and, usually at the end of the messages, there are *carriage return* and *line feed* characters. The attack generation algorithm cycles through all message specifications of the protocol and generates copies of each message with small variations of their delimiters, such as messages without one of the delimiters or messages with some delimiters replaced by a predefined set of illegal delimiter characters. Note that, with the exception of the delimiters, all generated messages will contain only legal data taken from the specification. Moreover, one can use any custom data to experiment with as illegal delimiters.

For the sake of simplicity, imagine a simple plain text protocol with a single message: a login command with the respective username and password parameters. One could test double quotes as illegal delimiters (though we should probably specify a larger set of illegal delimiters in a real-life example). This test definition would generate various login messages with a valid username and password but either without delimiters (e.g., test cases with no return character after the command and spaces between the parameters) or with illegal delimiters (e.g., test cases with quotes instead of the space character).

4.1.2 Syntax Test Definition

This kind of test generates attacks that infringe on the syntax of the protocol. The currently implemented syntax violations consist on the addition, elimination, or reordering of each field of a correct message. Note that, as with the previous algorithm, the field specifications are kept unchanged, i.e., they only hold valid values. Like all other test definitions, after generating new message specifications (i.e., variations from the original ones), each specification will result in several test cases, each one trying a different combination of possible field data.

As an example, consider a message containing two different fields (e.g., a command with one parameter) represented as [A] [B]. Below are depicted some of the variations of the original message specification from which test cases are going to be created:

- [A] (removed field [B]),
- [B] [B] (duplicated field [B]), and
- [B] [A] (swapped fields).

4.1.3 Value Test Definition

This test determines if the server can cope with messages with bad data. For this purpose, a mechanism is used to derive illegal data from the message specification, in particular, from each field's specified legal data. Ideally, one would like to experiment with all possible illegal values; however, this proves to be unfeasible when dealing with a large number of messages and fields with arbitrary textual content. To overcome such an impossibility, a heuristic method was conceived to reduce the number values that have to be tried (see Pseudocode 1). The algorithm has the following structure: All states and message types of the protocol are traversed, maximizing the protocol space; then each test case is generated based on one message type. This algorithm differs from the others because it systematically populates each field with wrong values, instead of only resorting to the legal values.

```
TestValue
  Input: Protocol ← specification of the network protocol used
         by the server
  Output: Attacks

  S ← set of all states of the Protocol specification
  foreach State s ∈ S do
    M ← set of message specifications of s
    Transition_s ← set of ordered network packets necessary
          to reach s
    P ← φ

    foreach MessageSpecification m ∈ M do
      foreach FieldSpecification f ∈ m do
        if f is type Numbers then
          f' ← all boundary values, plus some intermediary
               illegal values, from f specification
        elseif f is type Words then
          f' ← combin. of predefined malicious tokens

        m' ← copy of m replacing f with f'
        P ← P ∪ {set of all network packets based on m'
            specification}

    foreach attack_packet ∈ P do
      attack ← Transition_s ∪ {attack_packet}
      Attacks ← Attacks ∪ {attack}

  return Attacks
```

Pseudocode 1. Algorithm for the Value Test generation.

In the case of a field that holds a number, deriving illegal values is rather simple because they correspond to complementary values, i.e., the numbers that do not belong to the legal data set. Additionally, to decrease the number of values that are tried, and therefore, to optimize the injection process, this attack generation algorithm only employs boundary values plus a subset of the complementary values. The current implementation of the illegal data generation uses two parameters to select the complementary values, an *illegal coverage ratio* and a *random coverage ratio*. The illegal coverage ratio chooses equally distant values based on the total range of illegal values. On the other hand, the random coverage ratio selects the illegal values arbitrarily, from the available set of illegal numbers. For instance, if there are only 100 illegal numbers, from 0 to 99, a 10 percent illegal coverage ratio would add numbers 5, 15, 25, etc., whereas the same ratio of random coverage would select 10 random numbers from 0 to 99. This simple procedure allows most illegal data to be evenly distributed while still keeping a reduced set of test cases.

Creating illegal words, however, is a much more complex problem because there are an infinite number of character combinations, making such an exhaustive approach impossible. Our objective was to design a method to derive potentially desirable illegal words, i.e., words that are usually seen in exploits, such as large strings or strange characters (see Pseudocode 2, which is called in the underlined line of Pseudocode 1). Basically, this method produces illegal words by combining several tokens taken from two special input files. One file holds malicious tokens or known expressions, collected from the exploit community, previously defined by the operator of the tool (see Fig. 7). AJECT expands the special keyword `$(PAYLOAD)` with each line taken from another file with payload data. This payload file could be populated with already generated random data, long strings, strange characters, known usernames, and so on. The resulting data combinations from both files are used to define the illegal word fields.

```
generateIllegalWords
  Input: Words ← specification of the field
  Input: Tokens ← predefined list of malicious tokens, e.g.,
         taken from hacker exploits
  Input: Payload ← predefined list of special tokens to fill in
         the malicious tokens
  Input: max_combinations ← maximum number of token
         combinations
  Output: IllegalWords

  // step 1: expand list of tokens
  foreach t ∈ Tokens do
  if t includes keyword $(PAYLOAD) then
    foreach p ∈ Payoad do
      t' ← copy of t replacing $(PAYLOAD) with p
      Tokens ← Tokens ∪ {t'}
    Tokens ← Tokens \ {t}

  // step 2: generate and append all k−combinations of tokens
  k ← 1
  while k ≤ max_combinations
    k−combinations ← (Tokens k) // combinations of k elements
          from Tokens
    IllegalWords ← IllegalWords ∪ k−combinations
    k ← k + 1

  return IllegalWords
```

Pseudocode 2. Algorithm for the generation of malicious strings.

As an example, here are some attacks that were generated by this method, and that were used to detect known IMAP vulnerabilities ("$<A \times 10>$" means that character "A" is repeated 10 times) [3]:

- A01 AUTHENTICATE $<A \times 1296>$;
- $<\%s \times 10>$; and
- A01 LIST INBOX $<\%s \times 10>$.

4.1.4 Privileged Access Violation Test Definition

This algorithm produces tests to determine if the remote server allows unauthorized accesses to its resources. The existing implementation is a specialization of the Value Test Definition, employing only very specific tokens related to resource usage, such as known usernames or path names to existing files. The resulting attacks consist of messages related to privileged operations, which evaluate the protection mechanisms executed by the network server. If the server is authorizing any of these operations, such as disclosing private information, granting access to a known file, or writing to a specific disk location that should not have been allowed, it reveals that the server holds some access violation vulnerability. Further investigation on the collected data, including the server replies and its configuration, throws light into the cause of the problem, whether it is due to misconfiguration, bad design, or some software bug.

To keep the number of attacks manageable, a special focus must be taken on the selection of the tokens. Examples of good malicious tokens are directory path names (relative and absolute), well-known filenames, existing usernames, and so on, which can also be combined with payload tokens such as "/", "../", or "". The created test cases stress the server with illegal requests, such as reading the contents of the "../../../etc/passwd" file, accessing other user's data or just writing to an arbitrary disk location. Here are a few examples of generated IMAP attacks that were able to trigger some vulnerabilities in previous experiments [3]:

- A01 CREATE /$<A \times 10>$;
- A01 SELECT ./../../other-user/inbox; and
- A01 SELECT "{localhost/user=\"}."

These four test case generation algorithms provide a good starting point to evaluate the attack injection methodology and assess its usefulness; in the future, other algorithms could easily be added to cover more kinds of problems. The first two test definitions can be very useful in locating earlier implementation errors on the network servers. However, in our experiments, neither of them was able to produce test cases that discovered vulnerabilities in fully developed and tested servers. This is explained by the simplicity of the generated attacks that can be immediately blocked by the initial parsing and validation mechanisms of the server. On the other hand, the value and the privileged access violation test definitions, while focusing on trying different malicious values in each field, resulted in more complex attacks that allowed the detection of several vulnerabilities.

4.2 Injection Campaign Phase

The *injection campaign phase* corresponds to the actual process of executing the previously generated test cases, i.e., the injection of the attacks in the target system (see AJECT's architecture in Fig. 3). The *Attack Injector*, or simply the injector, carries out each attack sequentially. It decomposes each test case in its corresponding network packets, such as the transition messages to change the protocol state and the actual attack message, and sends them to the network interface of the server.

During the attack injection campaign, both the server's responses and its execution data are collected by the injector. The injector resorts to a *Monitor* component, typically located in the actual target system, to inspect the execution of the server (e.g., UNIX signals, Windows exceptions, allocated resources, etc.). The monitor can provide a more accurate identification of unexpected behavior. However, its implementation requires 1) access to the low-level process and resource management functions of the target system and 2) synchronization between the injector and the monitor for each test case execution.

Though such a monitoring component is desirable and very useful, it is not absolutely necessary, and in some circumstances, it might even be impractical. Some target systems, for instance, cannot be easily tampered with or modified to execute an extra application, such as in embedded or proprietary systems. In other cases, it might be just too difficult to develop a monitor component for a particular architecture or operating system, or for a specific server due to some special feature (such as acting as a proxy to other servers). In order to circumvent potential monitoring restrictions and to support as many target systems as possible, three alternative monitoring components were developed:

- **Deep Monitor** can trace the server's main process as well as its children in great detail; the current version is written in C++ for UNIX-based platforms, and it can intercept UNIX signals, including any abnormal signals such as SIGSEGV faults, while maintaining a record of the server resource utilization.
- **Shallow Monitor** can be used in virtually any platform; however, since it does not access the native features of the underlying operating system, it can only monitor the program's termination code; nevertheless, this code is useful because it still allows the detection of fatal crashes or other abnormal behavior; the current version of this monitor was written in Java.
- **Remote Monitor** executes in the injector machine, and therefore, it has no in-depth execution monitoring capabilities; although it complies with the monitor component interface, i.e., it synchronizes with the injector, it has to *infer* the state of the server based on the network connection (e.g., server replies, connections errors, etc.); this monitoring solution is the simplest and fastest approach when using an unknown or inaccessible target system.

A monitor can also help to control the execution of the server, such as to restart the program when there is a hang. The deep and shallow monitors reinitiate the target system after transmitting to the injector the data collected during each attack. Restarting the target application has the advantage of clearing the effects of previous attacks, and thus, making the execution of the test cases independent of

one another. This also simplifies the identification of the test case that triggered an abnormal behavior since it always corresponds to the last injected attack. For certain vulnerabilities, however, this behavior might be a disadvantage. Some programming flaws might not be activated by a single attack, but only by stressing the server with a series of attacks. By continuously injecting different attacks without restarting the target application, one is accumulating their successive effects and thus provoking software aging. These kinds of problems are captured with the remote monitor because it does not rejuvenate the server between injections.

The deep monitor is our current most complex and complete monitor. It can observe the target application's flow of control, while keeping a record of the amount of allocated system resources. The tracing capabilities are achieved with the PTRACE family functions to intercept any system calls and signals received by the server process (e.g., a SIGSEGV signal, indicating a memory access violation, is usually related to buffer overflow vulnerabilities). The LibGTop[1] library functions are utilized to fetch resource usage information, such as the memory allocated by the process (e.g., the total number of memory pages or the number of nonswapped pages) and the accumulated CPU time in user- and kernel-mode, at some specific system calls. Since the deep monitor is OS-dependent, it can only be used in UNIX-based systems.

The shallow monitor is much simpler but platform independent. Though it lacks any real in-depth monitoring capabilities, it can still control the server (i.e., restart the process) and collect the return status code after the completion of every test case. At this moment, both implementations of deep and shallow monitors have a limitation—they cannot monitor background processes (also known as UNIX *daemons* or Windows services) as they immediately detach themselves from the main process. However, this is not a major problem if applications are tested during development since they are usually built as regular processes for debugging purposes.

The remote monitor infers the server's behavior through passive and external observation. It resides in the injector machine and collects information about the network connection between the injector and the server. After every test case execution, the monitor closes and reopens the connection with the server, signaling the server's execution as failed if some communication error arises.

5 Experimental Framework

This section provides the details about the laboratory conditions in which the experimental evaluation took place. It includes a description of the network server protocols that were specified and tried with AJECT and the testbed configuration.

5.1 Network Server Protocols

The experimental evaluation was designed to assess the advantages of using attack injection to discover vulnerabilities in real applications. For that purpose, we chose fully developed and commonly used standard protocols, POP3 [4] and IMAP4Rev1 [5], instead of obscure or immature protocols that are typically implemented in applications with reduced levels of testing and utilization. Therefore, finding bugs in our targets will usually be hard because applications have gone through several revisions, where all vulnerabilities had an opportunity to be removed. Additionally, the selected protocols are not overly complex, leading to much simpler and less error-prone implementations.

1. A public library, available at http://directory.fsf.org/libs/LibGTop.html.

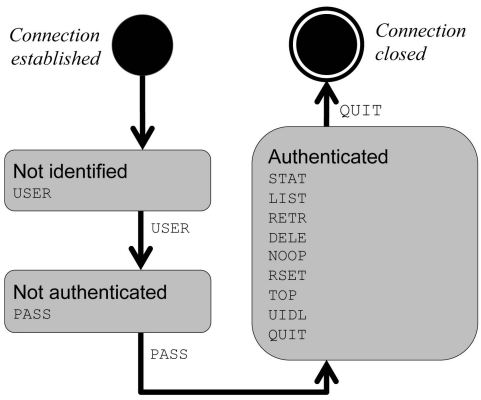


Fig. 5. Finite state machine of the protocol POP3.

5.1.1 POP Protocol

The Post Office Protocol (POP) is a widely used protocol for e-mail retrieval. It was designed to allow users without a permanent connection to remotely view and manipulate messages. POP3 servers listen on port number 110 for incoming connections, and use a reliable data stream (TCP) to ensure the transfer of commands, responses, and message data.

The POP standard describes three session states throughout the execution: the authorization, the transaction, and the update states. However, given the interaction messages between the client and the server, we have defined a different state diagram that could more accurately represent the state transitions. Fig. 5 shows the finite state machine used in the specification of the POP protocol in AJECT. The client initiates the connection with the server in the *not identified* state. The actual authorization process is composed of two separate steps (transition messages to nonfinal states are in bold): First, the client must supply its username (USER command) to the server, changing the protocol to the *not authenticated* state, and then it provides the corresponding password (PASS command), which, if correct, brings the protocol into the *authenticated* state, where the client can perform all e-mail message access and retrieval transactions. When the client is finished, it issues the QUIT command allowing the server to perform various housekeeping functions, such as removing all messages marked for deletion, before closing the connection.

All interactions between the client and the server are in the form of text strings that end with *Carriage Return* and *Line Feed* (CRLF) characters. Client messages are case-insensitive commands, with three or four letters long, followed by the respective parameters. Server replies are

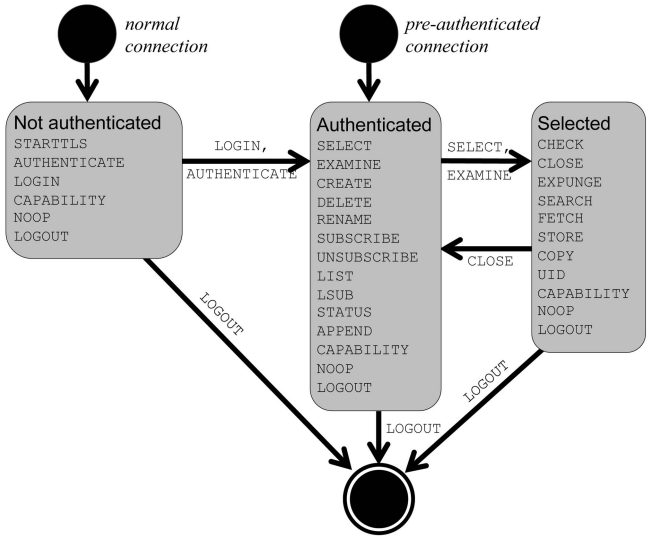


Fig. 6. Finite state machine of the protocol IMAP.

prefixed with +OK or -ERR, indicating a successful or unsuccessful command completion.

5.1.2 IMAP Protocol

The Internet Message Access Protocol (IMAP) is a popular method for accessing electronic mail and news messages maintained on a remote server. This protocol is specially designed for users that need to view e-mail 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 (also known as mailboxes) in a way that is functionally equivalent to a local file system.

The client and server programs communicate through a reliable data stream (TCP) and the server listens for incoming connections on port 143. Once a connection is established, it goes into one of the four states, as depicted in the finite state machine in Fig. 6. Normally, the protocol starts in the *not authenticated* state, where most operations are forbidden. If the client is able to provide acceptable authentication credentials (i.e., a single message with a valid username and password), the connection goes to the *authenticated* state. Here, the client can choose a mailbox, which causes the connection to go to the *selected* state, allowing it to execute commands that manipulate the mailbox's messages. The connection is terminated when the client logs out from the server.

All interactions between the client and the server are in the form of strings that end with a CRLF. Depending on the type of the client command, the server response contains zero or more lines with data, ending with one of the following completion results: OK (indicating success), NO (indicating failure), or BAD (indicating a protocol or syntax error). To simplify the matching between the requests and the responses, both the client commands and the respective server completion result are prefixed with the same distinct alphanumeric tag (e.g., A01, A02, etc.).

IMAP is a much more complex protocol than POP, offering a wider functionality to its clients. It provides an extensive number of operations, which include: creation, deletion, and renaming of mailboxes; checking for new messages; permanently removing messages; server-based RFC-2822 and MIME parsing and searching; and selective fetching of message attributes and texts. Although some of the protocol commands are very simple (e.g., composed by a single field), most of them are much more intricate than in POP and require various parameters.

5.2 Experimental Environment

One of the features of AJECT that we wished to test was its interoperability, i.e., its ability to support different target systems, independently of the hardware, operating system, or software implementations. Therefore, it was necessary to utilize a flexible testbed to ensure that the distinct requirements of the target applications could be accommodated.

The testbed consisted of three PCs with Intel Pentium 4 at 2.80 GHz and 512 MB of main memory, and running Windows XP SP2. Two of the PCs acted as target systems and the remaining one as the injector. The target PCs also executed a VMware virtual machine software on top of the native OS, configured with 256 MB of RAM. A total of 16 virtual machine (VM) configurations were set up, split among the two PCs, each installed with its own operating system (either Windows XP SP2 or Ubuntu 6.10) and respective network server (one of the 16 tested e-mail servers). A Shallow Monitor was installed in every Windows VM, whereas the Ubuntu VMs featured both the Deep Monitor and Shallow Monitor. The Remote Monitor was executed remotely from the Injector machine, so no additional installation was required on the VMs.

AJECT carried out the injection campaigns on the remaining PCs and collected the monitoring data from the monitor components. The protocol specification and the attacks were also previously generated in this Injector machine.

6 Experimental Results

6.1 Target Network Servers

The network servers were carefully selected from an extended list of e-mail programs supporting POP3 and IMAP4Rev1. All servers were up-to-date applications, fully patched to the latest stable version and with no known vulnerabilities. All server applications had gone through many revisions and are currently supported by an active development team. Since we wanted to experiment with targets with distinct characteristics, we chose servers running in different operating systems (Windows versus Linux) and programmed under opposing philosophies regarding the availability of the source code (closed source versus open source).

Table 1 lists the e-mail servers used in the experiments. Each server was subject to more than one attack injection campaign. For instance, since all programs supported both POP and IMAP, they were tested at least twice, once with each set of specific protocol attacks. If the servers run in Windows and Linux, then they were analyzed in both operating systems. Finally, to evaluate the advantages and penalties of the different monitoring strategies, each server was individually experimented with every supported monitor. This information is summarized in the last

TABLE 1
Target POP and IMAP E-Mail Servers

E-mail Servers (POP3/IMAP4)	OS	Version	Build Date	Monitor
602LAN Suite *(602 Software)*	W	5.0.08.0403	4/8/2008	R
Citadel*	U	7.32	2/17/2008	R
dovecot*	U	1.1.rc3	3/9/2008	D/S/R
Hexamail Server Corporate	U/W	3.1.0.002	-	R
hMailServer	W	4.4.1	3/9/2008	R
IMail Server *(Ipswitch)*	W	2006,23	12/5/2007	R
Kerio MailServer *(Kerio Tech.)*	U/W	6.5.1	5/12/2008	R
Mailtraq *(Fastraq)*	W	2.12.1.2364	5/8/2008	R
Mdaemon *(Alt-N Technologies)*	W	9.6.5	6/19/2007	R
Merak Mail Server *(IceWarp)*	U/W	9.1.0	9/17/2007	R
NoticeWare Email Server NG	W	4.6.2	4/3/2008	R
Softalk Mail Server Corp.	W	8.5.1.431	10/30/2007	R
SurgeMail Mail Server *(NetWin)*	U/W	3.9e	4/10/2008	R
uw-imap* *(Univ. of Washington)*	U	2007b	6/4/2008	D/S/R
WinGate Email Server *(Qbik)*	W	6.2.2	7/12/2008	R
xmail*	U/W	1.25	1/3/2008	D/S/R

* - Open source; U - Unix/Linux; W - Windows
D - Deep monitor; S - Shallow Monitor; R - Remote Monitor

```
$(PAYLOAD)
aject@$(PAYLOAD)
$(PAYLOAD)@aject
<aject<$(PAYLOAD)>
<$(PAYLOAD)<aject>
"$(PAYLOAD)"aject"
./private/passwd
D:\home\jantunes\AJECT\private\passwd
```

(a)

```
$(PAYLOAD)
aject@$(PAYLOAD)
$(PAYLOAD)@aject
<aject<$(PAYLOAD)>
<$(PAYLOAD)<aject>
"$(PAYLOAD)"aject"
./private/passwd
D:\home\jantunes\AJECT\private\passwd
(\$(PAYLOAD))
"{localhost/user=\"$(PAYLOAD)}"
(FLAGS BODY[$(PAYLOAD) (DATE FROM)])
(FLAGS $(PAYLOAD))
```

(b)

Fig. 7. File with malicious tokens for both protocols. (a) POP protocol. (b) IMAP protocol.

column of Table 1, which shows the monitor components employed with each e-mail server. The remote monitor, for instance, was used with all target systems because it probes network servers remotely. Since the deep monitor was implemented to test open source Linux servers, it was only utilized with servers developed for this operating system (except for Citadel, which did not provide a regular process execution). Finally, the shallow monitor was used in the experiments with any e-mail server that could be executed as a command line foreground process. Overall, each protocol was used in the 29 possible combinations of server/OS/monitor tuples, resulting in 58 independent attack injection campaigns.

6.2 Test Configuration

A limited number of experiments were carried out initially to get a first understanding of how effective were the attack generation algorithms. These preliminary results showed that delimiter and syntax test algorithms create test cases too simplistic to discover vulnerabilities in mature network servers. The attacks produced from these methods are not representative of the universe of attacks created by hackers or penetration testers, and are only useful during the initial software development phases. As explained previously, the privileged access violation test algorithm is actually a specialization of the value test algorithm, requiring very specific malicious tokens and a more careful and time-consuming analysis of the results. For this reason, and to maximize the automation of the several injection campaigns, we decided to center our efforts on testing the networks servers with the attacks produced with the value test algorithm. However, should the focus be on a specific server (e.g., to debug a particular network server), more time and work should be invested on experimenting larger sets of attacks, generated with different algorithms.

Fig. 7 shows the contents of the malicious tokens file used in the value test algorithm. The ability to generate good illegal data was of the utmost importance, i.e., values correct enough to pass through the parsing and input validation mechanisms, but sufficiently erroneous to trigger existing vulnerabilities. Therefore, picking good malicious tokens and payload data was essential. Known usernames and hostnames ("aject") were defined in these tokens, as well as path names to sensitive files (e.g., "./private/passwd"). The special keyword "$(PAYLOAD)" was expanded into various words: 256 random characters, 1000 A's, 5000 A's, a sequence of format string characters (i.e., "%n%p%s%x%n%p%s%x"), a string with many ASCII nonprintable characters, and two strings with several relative pathnames (i.e., "../../../" and "./././").

AJECT's attack generation created one attack file for each protocol, containing the attack messages and the transition messages (to inject the malicious message in the correct protocol state). For the sake of fairness in the experiments, only the mandatory protocol functionality (i.e., no protocol extensions) was tested. Based on the 13 message specifications of the POP protocol, the algorithm created a 90 MB attack file with 35,700 test cases, whereas for IMAP, which has a larger and more complex set of messages, there were 313,076 attacks in a 400 MB file. Given the size of the attack files, AJECT's injector only reads and processes one attack at a time, thus keeping a small memory usage.

The attacks to the POP and IMAP protocols were generated only once, taking a negligible amount of time—from a few seconds to a couple of minutes. On the other hand, injecting the attacks proved to be a more time-demanding task. The actual time required for each injection experiment depended on the protocol (i.e., the number of attacks), the e-mail server (e.g., the time to reply or to timeout), and the type of monitor (e.g., the overhead of an additional software component constantly intercepting system calls and restarting the server application, as opposed to the unobtrusive remote monitor). Overall, the POP injection campaigns took between 9 and 30 hours to complete, whereas the IMAP protocol experiments could last between 20 and 200 hours.

TABLE 2
E-mail Servers with Newly Discovered Vulnerabilities

Vulnerable Servers	Version	Corrected	Attacks / Observable Behavior
hMailServer (IMAP)	4.4.1	4.4.2 beta	>20k CREATE and RENAME messages
			Server becomes unresponsive until it crashes
NoticeWare (IMAP)	4.6.2	5.1	>40 A01 LOGIN Ax5000 password
			Server crashes
Softalk (IMAP)	8.5.1.431	8.6 beta 1	>3k A01 APPEND messages
			Server crashes after going low on memory
SurgeMail (IMAP)	3.9e	3.9g2	A01 APPEND Ax5000 (UIDNEXT MESSAGES)
			Server crashes
WinGate (IMAP)	6.2.2	-	A01 LIST Ax1000 *
			Server denies all subsequent connections: "NO access denied"

6.3 Vulnerability Assessment

Each experiment involved the injector in one machine and the target system in another (i.e., a VM image configured with either Windows or Linux, one of the 16 target network servers with one of the supported monitor components). The monitor component was used to trace and record the behavior of the e-mail server in a log file for later analysis. If a server crashed or graciously terminated the execution, for instance, the fatal signal and/or the return error code was automatically logged (depending on the type of the monitor). The server was then restarted (automatically, or manually in case of the remote monitor) and the injection campaign resumed for the next attack. At the end of the experiment, we analyzed the output results, looking for any unexpected behavior. The log file presents in each line the result of a test case, making it easier to visually compare different attacks and to perceive divergent behavior. Any suspicion of an abnormal execution, i.e., any line in the log file with dissimilar monitoring data, such as an unusual set of system calls, a large resource usage, or a bad return error code, was further investigated. AJECT allows us to replay the last or latter attacks in order to reproduce the anomaly, and thus confirm the existence of a vulnerability. The offending attacks were then provided as test cases to the developers to debug the server application.

However, in these experiments, we focused on the most automated log analysis, and only the most evident abnormal behaviors, related with OS signals, exceptions, and resource usages, were looked for.

AJECT found vulnerabilities in five e-mail servers that could eventually be exploited by malicious hackers. Table 2 presents a summary of the problems, including the attacks that triggered the vulnerabilities, along with a brief explanation of the unexpected behavior as seen by the monitor (see last column). All vulnerabilities were detected by some fatal condition in the server, such as a crash or a denial-of-service. Two servers, hMailServer and Softalk, showed signs of service degradation before finally crashing, suggesting that their vulnerabilities are related to bad resource management (e.g., a memory leak or an inefficient algorithm). In every case, the developers were contacted with details about the newly discovered vulnerabilities, such as the attacks that triggered them, so that they could reproduce and correct the problem in the following software release (third column of Table 2). Since the servers were all commercial applications, we could not perform source code inspection to further investigate the cause of the anomalies, but had to rely on the details disclosed by the developers instead.

The first vulnerabilities were discovered in the IMAP service provided by the hMailServer. This Windows server gave signs of early service degradation when running with the remote monitor. The telltale sign was a three-minute delay, with 100 percent peeks of CPU usage, when processing the attacks with the LIST command. Since the remote monitor does not rejuvenate the system, the server was responding to the request with an extensive list of mailboxes created from the previous attacks. Further investigation on this issue showed that the server eventually crashed when processing over 20,000 different CREATE and RENAME messages continuously. After helping the developers to pinpoint and correct the exact cause of the problem, they informed us that there was in fact a stack buffer overflow vulnerability, triggered when creating several deep-depth mailboxes (e.g., "./././././././ mailbox"). Additionally, there was a second problem, which could also be remotely exploited to create a denial-of-service—an inefficient parsing algorithm that was very slow on large buffers caused the high CPU usage.

NoticeWare server also ceased to provide its service when the attacks were injected. In the experiment with the remote monitor, the server was successively crashing after a series of different attacks. Then, after manually restarting the server and resuming the experiment, the server eventually failed again. The attacks were all related to the LOGIN command, for example, crashing after processing over 40 LOGIN messages with long arguments. This indicates that some problem was present in the parsing routine of that command, and the number and nature of the attacks hinted that some resource exhaustion vulnerability existed. However, no details were disclosed about the vulnerability by the developers (even though they showed great interest in using our tool themselves), and therefore, we could not confirm this conclusion.

Softalk was another Windows IMAP server with clear service degradation problems, probably caused by some resource exhaustion vulnerability. The IMAP server was consuming too much memory while processing several APPEND requests, until it eventually depleted all resources and crashed. This particular issue was aggravated by the fact that the attacker did not need to be authenticated in order to exploit the vulnerability. The overall number of

attacks until resource depletion obviously depended on the amount of available memory and on the actual hardware configuration. Nevertheless, it did not require the attacker much effort to cause the denial-of-service. The developers were contacted with details to reproduce the anomaly and eventually corrected their software, but they revealed no details about the vulnerability.

Both versions of the SurgeMail IMAP server, running on Windows and Linux, were also found vulnerable. However, in contrast to the other three cases, only a single message was needed to trigger the vulnerability. An authenticated attacker could crash the server by sending one APPEND command with a very large parameter. Even though the presence of the large parameter suggests a buffer overflow vulnerability, the developers indicated that the problem was related to the execution of some older debugging function that was left in the production code—this function intentionally crashed the server when facing some unexpected conditions.

WinGate could also be exploited remotely by an adversary. In spite of the simplicity of the attack, oddly enough it was quite difficult to reproduce the anomaly with the developers. The denial-of-service vulnerability that was found stopped the IMAP service by denying further client connections. The attack was quite trivial and consisted of a single LIST command with two parameters: several A's and an asterisk. After processing the command, the server never crashed but replied to all new connections with the following message: "NO access denied" refusing further access to new clients. This problem affected all new connections, independently of the IP source address, so this was not a blacklist defense mechanism. However, in the few communications with the developers, they were unable to reproduce the issue. Given this apparent difficulty, we repeated the same experiment in a separate Windows XP machine with a fresh server install (instead of the cloned VM image), still obtaining the same DoS result. Over the course of the experiments, we kept the developers up-to-date with our observations, but we received no further contact from them.

There was a sixth IMAP server, the 602LAN Suite, that revealed some intermittent problems, but we were unable to deterministically reproduce them. Occasionally, the IMAP server ceased its operation after the first 100,000 attacks, sometimes it took over 200,000 attacks, but most of the time, it did not crash at all. Therefore, even though we suspect that there is some flaw in the target system, we did not have the time or the tools (e.g., source code) to either confirm or repudiate the existence of a vulnerability in the server.

6.4 Additional Remarks

Several interesting conclusions also arise just by looking into the characteristics of the vulnerable servers. These conclusions would, however, benefit from further confirmation with larger samples of representative servers. First, all discovered vulnerabilities were related to the IMAP protocol, which seems to indicate that more complex protocols lead to more error-prone implementations, as one would expect. A rough analysis between the POP and IMAP protocols, in terms of number of states and messages types, indicates that POP is a much simpler protocol. Therefore, implementations of the IMAP protocol usually require more lines of code, and consequently, more sophisticated tests, which makes the debugging operations a much harder process, increasing the chances of some defect being missed.

Second, all vulnerabilities were detected in closed source commercial servers. In fact, 42 percent of the tested closed source servers had confirmed vulnerabilities, which is a quite significant number. In part, this result could be explained because the number of open source servers was significantly lower than the closed source servers, accounting for only 25 percent of all target systems. Naturally, a more complete experimental evaluation with a larger and more balanced sample of network servers could clarify this particular aspect. Nevertheless, we conjecture that the real reason why open source servers have fewer flaws is probably related to the larger, and usually more active, community behind these servers (testing, utilizing, and sometimes even developing them). Moreover, only the mandatory protocol functionality was specified in the attack generation. Therefore, any extra functionality or complexity potentially present in some commercial servers was not an issue, since all testing was restricted to the same set of common features.

Another interesting result is related to the monitoring approach required for the detection of the vulnerabilities. Unfortunately, none of the servers that was found vulnerable with the remote monitor was supported by other monitoring approaches. However, based on the attacks and the observable behavior, one can infer that the deep and shallow monitors could also detect the vulnerabilities discovered in the SurgeMail and WinGate servers. In order to detect the vulnerabilities in hMailServer, NoticeWare, and Softalk servers, these monitors would need to be modified to stop restarting the servers between test cases. These vulnerabilities could only be discovered with software aging, via the continuous execution of the server process—just replaying the last offending attack could not reproduce the abnormal behavior, which actually required the reinjection of a larger subset of the most recent attacks.

These results also shed some light on the process of conducting an injection campaign. As mentioned before, not restarting the target application between injections has proven to be advantageous in detecting some vulnerabilities that require software aging. However, since each attack potentially changes the state of the target application, it might be possible that two attacks cancel each other out, thus invalidating the effects of an unpredictable number of test cases. Therefore, the order in which the attacks are carried out becomes important, if not determinant. Injecting the same attacks in a different order might yield different results, making this approach interesting and worthy of further study. Whenever possible, both methods should be used: first, by exhausting all protocol messages individually, and then, by continuously reinjecting all attacks without restarting the server, thus emulating the effects of software aging.

The results presented here show that AJECT can be very useful in discovering vulnerabilities even in fully developed and tested software. Actually, even though developers were not forthcoming in disclosing the details of the

vulnerabilities, which is understandable because all servers were commercial applications, most of them showed great interest in using AJECT as an automated tool for vulnerability discovery. The attack injection methodology, sustained by the implementation of AJECT, could be an important asset in constructing more dependable systems and in enforcing the security of the existing ones.

7 Related Work

This paper describes a methodology and a tool for the discovery of vulnerabilities on network servers through the injection of attacks (i.e., malicious faults). This work has been influenced by several research areas, as given below.

7.1 Fault Injection

This 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 [6], [7]. Traditionally, fault injection has been utilized to emulate several kinds of hardware and software faults, ranging from transient memory corruptions to permanent stuck-at faults. Xception [8] and FTAPE [9] are examples of tools that can inject hardware or software faults in a target system under evaluation. The emulation of other types of faults has also been accomplished with fault injection techniques, for example, software and operator faults [10], [11]. 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 [12], [13]. However, due to the relative simplicity of the mimicked faults, it is difficult to apply these tools to more complex faults, like security vulnerabilities of network servers. AJECT injects more complex faults, which try to emulate the behavior of an attacker while probing the interface of a server.

7.2 Fuzzers

Fuzzers deal with this intractability by injecting random samples as input to the software components. These are much less methodical than classic fault injection tools. For instance, Fuzz [14], inspired by the noisy dial-up lines that sometimes scrambled command line characters and crashed an application, generates large sequences of random characters to be used as testing parameters for command-line programs. Many programs failed to process the illegal arguments and crashed, revealing dangerous flaws like buffer overflows. 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 [15]. Throughout the years, fuzzers have evolved into less random and more intelligent vulnerability detectors [16], [17], [18]. Fuzzers do not require any preconception about the system's behavior, and thus, can find odd oversights and defects that manual testing often fails to locate. However, fuzzers only exercise a random sample of the system behavior, and frequently, the test cases are either too simplistic or specialized to be reused on different target systems. Fuzzers also lack thorough monitoring mechanisms, such as the different monitoring approaches present in AJECT that support the detection of many kinds of vulnerabilities, raging from fatal crashes to resource exhaustion. AJECT is independent of the target application and implemented protocol since it allows the specification of diverse protocols from which the attacks will be generated.

7.3 Vulnerability Scanners

These are the tools whose purpose is the discovery of vulnerabilities in computer systems (in most cases, network-connected machines). Several examples of these tools have been described in the literature, and currently, there are some commercial products: Nessus [19], SAINT [20], and QualysGuard [21]. 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., type of operating system, available services, 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; finally, 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 systems in production, they have the limitation that they are unable to uncover unknown vulnerabilities.

7.4 Static Vulnerability Analyzers

These analyzers look for potential vulnerabilities in the code (i.e., source code, assembly, or binary) of the applications. This is a different approach from attack injection in which analyzers inspect the source code for dangerous patterns, usually associated with buffer overflows, and provide a listing of their locations [22], [23]. Next, the programmer only needs to go through the parts of the code for which there are warnings, to determine if an actual problem exists. More recently, this idea has been extended to the analysis of binary code [24]. Static analysis has also been applied to other kinds of vulnerabilities, such as race conditions during the access of (temporary) files [25]. In the past, a few experiments with these tools have been reported in the literature showing them as quite effective for locating programming problems. These tools, however, have the limitation of producing many false warnings and miss some of the existing vulnerabilities.

7.5 Runtime Prevention Mechanisms

These mechanisms change the runtime 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 damages caused by their exploitation. Most of these techniques were developed to protect systems from buffer overflows, and few examples are: StackGuard [26] and PointGuard [27]. These tools are compiler-based and determine at runtime if a buffer overflow occurred, and stop the program execution before the attack code is executed. A recent study compares the effectiveness of some of these techniques, showing that they are useful only to prevent a subset of the attacks [28]. Other approaches provide the OS with some sort of memory page

protection. For example, Windows XP SP 2 introduced Data Execution Prevention (DEP), which prevents any application from execution code from a nonexecutable region, limiting some forms of code injection attacks [29]. PaX provides program memory randomization to thwart attacks that require pointer prediction [30]. Another approach, Libsafe, intercepts library calls to prevent attackers from overwriting the return address and hijacking the control flow of the application [31].

7.6 Software Rejuvenation

Other techniques address the problem of bad resource management, whose effects accumulate over time in what is called software aging [32]. Software rejuvenation is meant to mitigate the effects of the phenomenon and impact of the software aging [33]. While not being concerned with the actual cause of the aging effects (e.g., a memory leak or an unreleased file lock), software rejuvenation is very successful in proactively removing the effects of software aging by restarting or rebooting the system or part of it. However, after the system is rejuvenated, the vulnerabilities that caused or sped up the aging effects are typically not removed, and thus, the problem will eventually arise again.

8 Conclusion

The paper presents a methodology and a tool for the discovery of vulnerabilities in server applications, which is based on the behavior of malicious adversaries. AJECT injects several attacks against a target network server, while observing its execution. This monitoring information is later analyzed to determine if the server executed correctly, or on the other hand, if it exhibited any suspicious behavior suggesting the presence of a vulnerability.

Our evaluation confirmed that AJECT could detect different classes of vulnerabilities in e-mail servers and assist the developers in their removal by providing the required test cases. The 16 servers chosen for the experiments were fully patched and up-to-date applications and most of them had gone through many revisions, making them challenging targets. In any case, AJECT successfully discovered vulnerabilities in five servers, which corresponded to 42 percent of all tested commercial applications. Even though few details are available about the vulnerabilities since they were found in closed source programs, it was possible to infer that three of the flaws were related to resource management.

Acknowledgments

This work was partially supported by the FCT through the Multiannual Funding and Project POSC/EIA/61643/2004 (AJECT), and by the CMU-Portugal Programs.

References

[1] P. Verissimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch, "Intrusion-Tolerant Middleware: The Road to Automatic Security," *IEEE Security and Privacy*, vol. 4, no. 4, pp. 54-62, July/Aug. 1996.

[2] B. Beizer, *Software Testing Techniques*, second ed. Van Nostrand Reinhold, 1990.

[3] N. Neves, J. Antunes, M. Correia, P. Verissimo, and R. Neves, "Using Attack Injection to Discover New Vulnerabilities," *Proc. Int'l Conf. Dependable Systems and Networks*, June 2006.

[4] J. Myers and M. Rose, "Post Office Protocol—Version 3," RFC 1939 (Standard), updated by RFCs 1957, 2449, http://www.ietf.org/rfc/rfc1939.txt, May 1996.

[5] M. Crispin, "Internet Message Access Protocol—Version 4rev1," Internet Eng. Task Force, RFC 3501, Mar. 2003.

[6] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 913-923, Aug. 1993.

[7] M.-C. Hsueh and T.K. Tsai, "Fault Injection Techniques and Tools," *Computer*, vol. 30, no. 4, pp. 75-82, Apr. 1997.

[8] J. Carreira, H. Madeira, and J.G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *Proc. Int'l Working Conf. Dependable Computing for Critical Applications*, pp. 135-149, http://citeseer.ist.psu.edu/54044.html; http://dsg.dei.uc.pt/Papers/dcca95.ps.Z, Jan. 1995.

[9] T.K. Tsai and R.K. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool," *Proc. Int'l Conf. Modeling Techniques and Tools for Computer Performance Evaluation*, pp. 26-40, http://portal.acm.org/citation.cfm?id=746851&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618, Sept. 1995.

[10] J. Christmansson and R. Chillarege, "Generation of an Error Set That Emulates Software Faults," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 304-313, June 1996.

[11] J. Durães and H. Madeira, "Definition of Software Fault Emulation Operators: A Field Data Study," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 105-114, June 2003.

[12] P. Koopman and J. DeVale, "Comparing the Robustness of POSIX Operating Systems," *Proc. Int'l Symp. Fault-Tolerant Computing*, pp. 30-37, June 1999.

[13] M. Mendonça and N. Neves, "Robustness Testing of the Windows DDK," *Proc. Int'l Conf. Dependable Systems and Networks*, pp. 554-564, June 2007.

[14] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Comm. ACM*, vol. 33, no. 12, pp. 32-44, 1990.

[15] P. Oehlert, "Violating Assumptions with Fuzzing," *IEEE Security and Privacy*, vol. 3, no. 2, pp. 58-62, http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1423963, Mar./Apr. 2005.

[16] Univ. of Oulu, "PROTOS—Security Testing of Protocol Implementations," http://www.ee.oulu.fi/research/ouspg/protos/, 1999-2003.

[17] M. Sutton, "FileFuzz," http://labs.idefense.com/labs-software.php?show=3, Sept. 2005.

[18] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[19] Tenable Network Security, "Nessus Vulnerability Scanner," http://www.nessus.org, 2008.

[20] Saint Corp., "SAINT Network Vulnerability Scanner," http://www.saintcorporation.com, 2008.

[21] Qualys, Inc., "QualysGuard Enterprise," http://www.qualys.com, 2008.

[22] D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," *Proc. Network and Distributed System Security Symp.*, Feb. 2000.

[23] E. Haugh and M. Bishop, "Testing C Programs for Buffer Overflow Vulnerabilities," *Proc. Symp. Networked and Distributed System Security*, pp. 123-130, Feb. 2003.

[24] J. Durães and H. Madeira, "A Methodology for the Automated Identification of Buffer Overflow Vulnerabilities in Executable Software without Source-Code," *Proc. Second Latin-Am. Symp. Dependable Computing*, Oct. 2005.

[25] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems*, vol. 9, no. 2, pp. 131-152, Spring 1996.

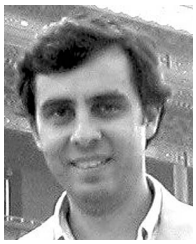
[26] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks," *Proc. USENIX Security Conf.*, pp. 63-78, https://db.usenix.org/publications/library/proceedings/sec98/cowan.html, Jan. 1998.

[27] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities," *Proc. USENIX Security Symp.*, pp. 91-104, http://www.usenix.org/publications/library/proceedings/sec03/tech/cowan.html, Aug. 2003.

[28] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," *Proc. Network and Distributed System Security Symp.,* pp. 149-162, Feb. 2003.
[29] Microsoft, Corp., "A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003," http://support.microsoft.com/kb/875352, Sept. 2006.
[30] "PaX," http://pax.grsecurity.net/, 2009.
[31] T. Tsai and N. Singh, "Libsafe 2.0: Detection of Format String Vulnerability Exploits," white paper, Avaya Labs, 2001.
[32] S. Garg, A.V. Moorsel, K. Vaidyanathan, and K.S. Trivedi, "A Methodology for Detection and Estimation of Software Aging," *Proc. Int'l Symp. Software Reliability Eng.,* p. 283, 1998.
[33] K. Vaidyanathan and K.S. Trivedi, "A Comprehensive Model for Software Rejuvenation," *IEEE Trans. Dependable and Secure Computing,* vol. 2, no. 2, pp. 124-137, Apr.-June 2005.

João Antunes received the MSc degree in informatics from the University of Lisboa in Portugal in 2006. He is currently working toward the PhD degree in attack injection. He has been a computer science researcher at Navigators Research Group of LASIGE, Portugal, since 2004. He has been involved in national and international research projects in computer and network security, including AJECT, CRUTIAL, and the European Network of Excellence ReSIST. He is the author of several research papers. He is a student member of the IEEE.

Nuno Neves received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1998. He is an associate professor in the Department of Informatics at the Faculty of Sciences at the University of Lisboa. His main research interests include secure and dependable distributed and parallel systems. He is on the editorial board of the InderScience *International Journal of Critical Computer-Based Systems*. In recent years, he has been involved in several security-related European projects, such as CRUTIAL, MAFTIA, and RESIST, and coordinated the national projects DIVERSE, RITAS, AJECT, and COPE. His work has been recognized on several occasions, for example, with the IBM Scientific Prize 2004 and the William C. Carter Award at IEEE FTCS 1998. He has more than 65 international publications in journals and conferences. He is a member of the IEEE. More information about his research can be found at http://www.di.fc.ul.pt/nuno.



Miguel Correia is an assistant professor in the Department of Informatics on the Faculty of Sciences at the University of Lisboa, and an adjunct faculty member of the Carnegie Mellon Information Networking Institute. He is a member of the LASIGE Research Unit and the Navigators Research Team. He has been involved in several international and national research projects related to intrusion tolerance and security, including the MAFTIA and CRUTIAL EC-IST Projects, and the ReSIST NoE. He is currently the coordinator and an instructor of the joint Carnegie Mellon University and the University of Lisboa MSc in Information Security. He has more than 50 publications in international journals, conferences, and workshops. His main research interests include intrusion tolerance, security, distributed systems, and distributed algorithms. He is a member of the IEEE. More information about his research can be found at http://www.di.fc.ul.pt/mpc.

Paulo Verissimo is a professor in the Department of Informatics (DI) at the Faculty of Sciences at the University of Lisboa (http://www.di.fc.ul.pt/~pjv), and the director of LASIGE (http://lasige.di.fc.ul.pt). He is an associate editor of the Elsevier *International Journal on Critical Infrastructure Protection,* and a past associate editor of the *IEEE Transactions on Dependable and Secure Computing.* He was a member of the European Security and Dependability Advisory Board. He is the past chair of the IEEE Technical Committee on Fault-Tolerant Computing and of the Steering Committee of the DSN conference. He leads the Navigators Research Group of LASIGE. His research interests include architecture, middleware, and protocols for distributed, pervasive, and embedded systems, in the facets of real-time adaptability and fault/intrusion tolerance. He is the author of more than 150 refereed publications in international scientific conferences and journals, and coauthor of five books. He is a fellow of the IEEE and the ACM.

Rui Neves received the Eng-diploma and PhD degrees in electrical and computer engineering from the Instituto Superior Técnico at the Technical University of Lisbon, Portugal, in 1993 and 2001, respectively. He has been a professor at the Instituto Superior Técnico since 2005. In 2006, he joined the Instituto de Telecomunicações (IT) as a research associate. His research interests include evolutionary computation applied to the financial markets, sensor networks, embedded systems, and mixed signal integrated circuits. During his research activities, he has collaborated/coordinated several EU and national projects.