# Using Attack Injection to Discover New Vulnerabilities

Nuno Neves, João Antunes,
Miguel Correia, Paulo Veríssimo
Fac. de Ciências da Univ. de Lisboa
{nuno,jantunes,mpc,pjv}@di.fc.ul.pt

Rui Neves
Instituto Superior Técnico da
Univ. Técnica de Lisboa
rui.neves@tagus.ist.utl.pt

## Abstract

*Due to our increasing reliance on computer systems, security incidents and their causes are important problems that need to be addressed. To contribute to this objective, the paper describes a new tool for the discovery of security vulnerabilities on network connected servers. The AJECT tool uses a specification of the server's communication protocol to automatically generate a large number of attacks accordingly to some predefined test classes. Then, while it performs these attacks through the network, it monitors the behavior of the server both from a client perspective and inside the target machine. The observation of an incorrect behavior indicates a successful attack and the potential existence of a vulnerability. To demonstrate the usefulness of this approach, a considerable number of experiments were carried out with several IMAP servers. The results show that AJECT can discover several kinds of vulnerabilities, including a previously unknown vulnerability.*

## 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. Software evolution has provided us with applications with ever improving functionality. These enhancements however are achieved in most cases with larger and more complex projects, which require the coordination of several teams of people. Third party software is frequently utilized to speedup programming tasks, even tough in many cases it is poorly documented and supported. In the background, the ever present tradeoff between time to market and thorough testing, puts pressure on the quality of the software. Consequently, these and other factors have brought us to the current situation, where it is extremely difficult to find applications without software bugs. Experience has shown that some of these bugs result in security vulnerabilities that can be exploited by malicious adversaries.

The existence of a vulnerability *per se* does not cause a security hazard, and in fact many times they can remain dormant for many years. An intrusion is only materialized when the right attack is discovered and applied to exploit that vulnerability. After an intrusion, the system might or might not fail, depending on the kind of capabilities it possesses to deal with errors introduced by the adversary. Sometimes the intrusion can be tolerated [32], but in the majority of the current systems, it leads almost immediately to the violation of one or more security properties (e.g., confidentiality or availability).

Several tactics can be employed to improve the dependability of a system with respect to malicious faults [1]. Of course, intrusions would never arise if all vulnerabilities could be eliminated. 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 and/or other similar problems. Intrusion prevention (e.g., vulnerability removal) has been advocated because it reduces the power of the attacker [32]. 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).

This paper describes a tool called *AJECT – Attack inJECtion Tool* that can be used for vulnerability detection and removal. AJECT simulates the behavior of an adversary by injecting attacks against a target system. Then, it observes the execution of the system 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 a flaw, one can employ traditional debugging techniques to examine the application code and running environment, to find out the origin of the vulnerability and allow its subsequent

This work was partially supported by the EC through project IST-2004-27513 (CRUTIAL), and by the FCT through projects POSC/EIA/61643/2004 (AJECT) and the Large-Scale Informatic Systems Laboratory (LASIGE).

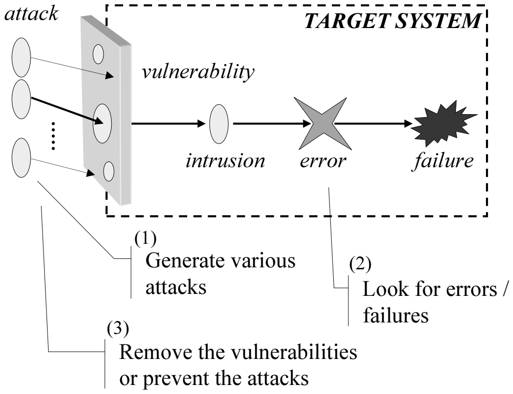

**Figure 1. Using the composite fault model to demonstrate the vulnerability discovery methodology.**

elimination.

The current version of AJECT mainly targets network server applications, although it can also be utilized with most local daemons. We chose servers because, from a security perspective, they are probably the most relevant components that need protection. They constitute the primary contact points of a network facility – an external hacker normally 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. The tool does not need the source code of the server to perform the attacks, i.e., it treats the server as black box. However, in order to be able to generate intelligent attacks, AJECT has to obtain a specification of the protocol implemented by the target server.

To demonstrate the usefulness of our approach, we have conducted a number of experiments with several IMAP servers. The main objective 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 AJECT could be used to detect many IMAP vulnerabilities (this was true for all flawed IMAP servers that we managed to get). Moreover, AJECT was also able to discover a new vulnerability that was previously unknown to the security community.

## 2. Using Attacks to Find Vulnerabilities

The AVI (attack, vulnerability, intrusion) composite fault model introduced in [1, 31] helps us to understand the mechanisms of failure due to several classes of malicious faults (see Figure 1). It is a specialization of the well-known sequence of *fault* → *error* → *failure* 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[1] 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., a 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 introduced accidentally or deliberately, 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 new account with root privileges), and if nothing is done to process the error, a failure will happen.

The methodology utilized in the construction of AJECT emulates the behavior of an external adversary attempting to cause a failure in the target system. The tool first generates a large number of attacks which it directs against the interface of the target (step 1, in Figure 1). A majority of these attacks are expected to be deflected by the validation mechanisms implemented in the interface, but a few of them will be able to succeed in exploiting a vulnerability and will cause an intrusion. Some conditions help to augment the probability of success of the attack, for example: a correct understanding of the interaction protocol utilized by the target facilitates the creation of more efficient attacks (e.g., reduces the number of random tests); and a good knowledge about what type of vulnerabilities appear more frequently also helps to prioritize the attacks.

While attacks are being carried out, AJECT also monitors how the state of 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 the abnormal creation of a large file, though it might not be a vulnerability – related to a possible 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 a implementation stage, it is best to provide detailed information about the attack and 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 the operational stage, then besides giving information about the problem, other actions might be worth-

[1] Outside the target system boundaries.

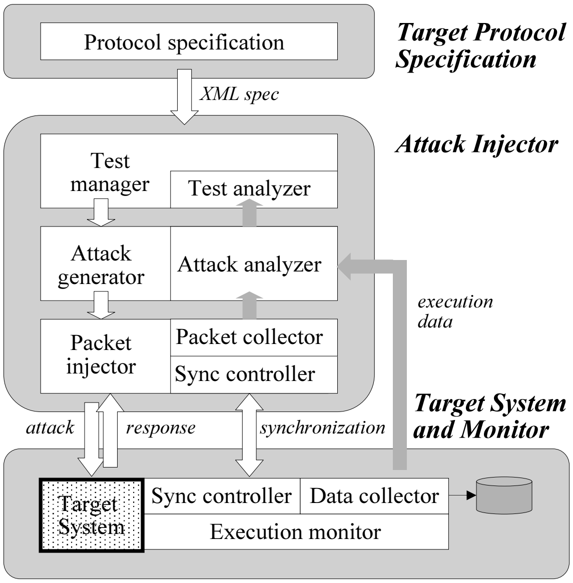

**Figure 2. The architecture of the AJECT tool.**

while 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 the current version, AJECT only provides information about the attack and observed errors/failures.

In order to get a higher level of confidence about the absence of vulnerabilities in the system, 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 try all possible attack patterns, and therefore it is possible that some vulnerabilities will remain undisclosed. Nevertheless, we feel that 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. AJECT Tool

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 2). The first entity corresponds to the system we want to test and the last three are the main components of AJECT.

The Target System is composed by the target application and its execution environment, which includes the operating system, middleware libraries and hardware configuration. The target application is typically 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 of the operating system. In both cases, the target application uses a well-known protocol to communicate with the clients, and these clients can carry out attacks by transmitting malicious packets. 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 component provides a graphical interface for the specification of the communication protocol used by the target application. The Attack Injector, or simply the Injector, is responsible for the generation and execution of the attacks, and for receiving 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 main objective of the Monitor is to observe and gather data about the target system 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 necessary. Therefore, the Monitor needs to run in the same machine as the target, where it can use the low level operating system functions to get, for example, statistics about the CPU and 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 a minimal level.

### 3.1. 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 (malicious) packets. Therefore, the attack concept is relatively vague and can be quite generic. For instance, an attack could correspond to something as general 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 secret username and password.

In AJECT, the process of creating an attack can be seen at three levels. The first and most generic level defines the a general *test* classes. Each test will then be systematically instantiated resulting in specific *attacks* for that particular test (the second level). In the last level, an attack is implemented through the transmission of its corresponding *packets*.

As an example, consider one of the tests currently supported in AJECT, a syntax test (see Section 4.2 for more details). This test validates the format of the packets utilized by 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 packets, it is quite simple to generate a reasonable number of distinct attacks, i.e., to create several instances of the test. For example, just imagine a packet with three fields that have to appear in a given order, and an attack corresponds to the re-ordering of these fields.

## 3.2. The Components of the Tool

**3.2.1. Target Protocol Specification (TPS) Component.** TPS is used to create a specification 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 states (or at least to a majority of them) because certain messages can only be sent in particular situations (e.g., a command $C$ is only allowed to proceed after an authentication). 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 server protocol, one only needs to produce a high level specification.

Currently, the specification is 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 TPS component is a XML description of the protocol, which is then imported by the Injector.

**3.2.2. Attack Injector Component.** The Injector is decomposed into three groups of modules, each one corresponding to a level of the attack generation hierarchy (see Figure 2). In every level there is a module whose function is related to the construction of the attacks and another module for the collection and analysis of the responses. In more detail,

**Test level** The *test manager* controls the whole process of attack injection. 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, to determine the effectiveness of a test to discover vulnerabilities.

**Attack level** The actual creation of new 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 protocols. The main task of the *packet collector* is the storage of the network data (i.e., attack injection packets and received responses).

**3.2.3. Monitor Component.** Although the Monitor appears to be a simple component, it is a fundamental entity, and it hides some complex aspects. On one side, this component is in charge of setting up all testing environment in the target system: it needs to start up the target application, perform all configuration actions, initiate the monitoring activities, and in the end, to free all utilized resources (e.g., processes, memory, disk space). We chose to reset the whole system after each experiment to guarantee that there are no interferences among the attacks. On the other side, the Monitor observes the execution of the the target while the attack is being carried out. This task is highly dependent on the mechanisms that are available in the local operating system (e.g., the ability to catch signals).

The monitor is composed by the modules: the *execution module*, which coordinates the various tasks of each experiment and traces the target execution; the *data collector*, responsible for monitoring data storage and its transmission back to the Injector; and the *sync controller* that determines the beginning and ending of each experiment.

As an example, on a UNIX machine, the current version of the Monitor collects mainly two types of data. First, it observes the target's flow of control by intercepting and logging any software exceptions. To achieve this goal, the PTRACE family functions were utilized to intercept any signals that the target receives (e.g., SIGSEGV). Second, the supervision of the system resources allocated during the target execution is helpful to detect abnormal behavior which may be indicative of a vulnerability. 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), and user-mode and kernel-mode CPU time accumulated by the process. This resource monitoring data is obtained with the LibGTop[2] library functions.

**3.2.4. Test and Attack Analyzer.** After the execution of the experiments, AJECT must be able to detect the presence of vulnerabilities by resorting to the analysis of the target's behavior. For each action there's a reaction, so for each attack injection there's the target's reaction. The Test and Attack analyzer modules examine an attack injection experiment result by observing the network data of the respective attack and response messages, and by correlating this in-

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

| Any State | (S1) Not Authenticated |
|---|---|
| CAPABILITY | STARTTLS |
| NOOP | AUTHENTICATE <auth mechanism> |
| LOGOUT | LOGIN <username> <password> |

| (S2) Authenticated | (S3) Selected |
|---|---|
| SELECT <mbox> | CHECK |
| EXAMINE <mbox> | CLOSE |
| CREATE <mbox> | EXPUNGE |
| DELETE <mbox> | SEARCH [charset spec] <criteria...> |
| RENAME <mbox> <new name> | FETCH <seq set> <msg data \| macro> |
| SUBSCRIBE <mbox> | STORE <seq set> <msg data> <value> |
| UNSUBSCRIBE <mbox> | COPY <seq set> <mbox> |
| LIST <reference> <mbox [wildcards]> | UID <COPY \| FETCH \|... > <args> |
| LSUB <reference> <mbox [wildcards]> | |
| STATUS <mbox> <status data items...> | |
| APPEND <mbox> [flag list] [date] <msg literal> | |

**Table 1. Commands tested in each IMAP state.**

formation 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 a particular attack injection.

## 4. 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.

### 4.1. IMAP Protocol

The Internet Message Access Protocol (IMAP) is a popular method for accessing electronic mail and news messages maintained on a remote server [11]. 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 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 four states. 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, which causes the connection to go to the *selected* state. In the selected state it is possible to execute the commands that 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 action occurs (e.g., server shutdown).

All interactions between the client and server are in the form of strings that end with a CRLF. 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 as provided in the client command.

The IMAP protocol provides a 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 for efficiency. Table 1 represents 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.

### 4.2. Predefined Tests

The Injector component currently implements three different kinds of tests, which are used to automatically generate a large number of attacks. The tool however was developed to support tests in a generic way, which means that other tests will possibly be added in the future to cover more classes of attacks.

| Att. Nr. | Attack Packet | Description |
|---|---|---|
| ... | | |
| 328 | SELECT | *removed field* |
| 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* |
| 337 | /inbox | *removed field* |
| 338 | /inbox EXAMINE /inbox | *duplicated field* |
| ... | | |

**Table 2. Syntax test attacks sample.**

**4.2.1. Syntax Test.** This kind of test generates attacks/packets that infringe the syntax specification of the protocol as provided by the TPS. Example of attack generations that constitute syntax violations consist on the addition or elimination of each field of a correct message, or permutations of all its fields.

This test regards a packet as a sequence of fields, each one with a certain number of bits. The type of data stored in a field or its content is considered irrelevant. The implementation of this test generates all possible combinations within certain parameters, such as maximum number of added/removed fields, producing a large number of attacks.

Table 2 shows a subset of the attacks that are automatically generated using this test. In the example, these attacks are packet variations of the SELECT and EXAMINE commands. The field contents are kept unchanged, but they are removed or duplicated in some cases.

**4.2.2. Value Test.** The TPS component also defines the type and range of valid values of the fields of the 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: first, a set of all valid packet specifications for a particular state is obtained; second, a packet with correct fields is generated for each specification; third, each field of every packet is mutated with a number of malicious values (each mutation corresponds to a distint attack). Since there are several fields in a packet, and a field can take many different values, this procedure can produce a large number of attacks. With the objective of keeping this number manageable, not all illegal values are experimented, but only a subset such as frontier values or large strings.

For example, consider a packet with two integer fields, the first is always 1 and the second can take values between 0 and 1000. An attack on the second field would utilize values that are almost valid (e.g., the boundary values -1, 0, 1000, 1001) or very invalid (e.g., large negative/positive integers). On the IMAP protocol, several of the command parameters are strings. The construction of malicious strings is reasonably complex because it can easily lead to an explosion on the number of attacks[3]. Therefore, a procedure based on heuristics was employed in their generation: First, (i) a set of random tokens (fixed sized strings) is obtained; (ii) a set of malicious tokens is pre-selected (e.g., "%c"); (iii) a set of joining tokens is chosen (e.g., "\", space or none); Then, a large number of strings with different sizes is obtained from the combination of one or more types of tokens. These strings are later used to replace valid values, hence testing the target's robustness in coping with this type of malicious input.

**4.2.3. Information Disclosure Test.** This type of test tries to get secret (or private) information that is stored in the target machine. The information is usually saved in the disk or in the memory of the server, and it can correspond, for instance, to the passwords kept in a configuration file or the in memory resident environment variables.

The current implementation of the test focus mainly in obtaining data stored in files. First, it determines which fields of a packet are utilized by the server to name files (e.g., argument with value "X" is used to select the file "./../dir/X"). To accomplish this task, either the field is tagged as a name during protocol specification or a conservative approach has to be taken, where all fields are considered as an eventual name. Then, the test generates a large number of path names to well 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 disclosing some confidential information.

For example, consider the file "/etc/passwd" that contains the usernames (and possibly the encrypted passwords) on a UNIX machine. Some of the names that could be tried in the attacks are: ["./../etc/passwd"]; ["./../../etc/passwd"]; ["./../../../etc/passwd"]. On the IMAP protocol, there are a few commands that use arguments to name a file – for example, the *mbox* on a EXAMINE command.

## 4.3. Testbed

The experiments used several IMAP applications that were developed for different operating system flavors. 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., Fedora and Suse) and the other in

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

Windows (e.g., XP and 2000). The third PC run the Injector components, collected the statistics, and performed the analysis of the results. This configuration of the testbed allows for the parallel execution of two injection experiments (if needed, more target PCs can be easily added to increase the concurrency of the system).

# 5. Experimental Results

The current section presents an evaluation of the vulnerability discovery capabilities of AJECT. This study carried out 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, to find out all IMAP vulnerabilities that were disclosed in the current year. The vulnerable products were then obtained (at least most of them) 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 approach that we considered following was to spend all our resources testing a small group of IMAP servers (one or two), trying to discover a new set of vulnerabilities. We decided not to use this strategy because it would probably not allow us to fulfill all goals. For example, the second goal would be difficult to achieve, since we would be testing code produced by the same developers, and these tend to make similar mistakes. Nevertheless, during the experiments, we were able to discover a new vulnerability that, as far as we can tell, was previously unknown by the security community.

## 5.1. Applications Under Test

To set up the experiments, we have searched for IMAP flaws in two of the most well-known 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. From this search it was possible to find 27 reports of security problems related to IMAP products during the year 2005. 7 of these reports, however, only provided minimal information, which did not allow us to completely understand the vulnerability. Therefore, we decided to exclude these 7 reports and pursue our investigation with the remaining 20.

| ID | Application | OS | Date | Vuln. ID |
|---|---|---|---|---|
| A1 | MailEnable Professional 1.54 and Enterprise Edition 1.04 | Win | Apr | 1014/5, 2278 |
| A2 | GNU Mailutils 0.6 | Lin | May | 1523 |
| A3 | E-POST Inc. SPA-PRO Mail @Solomon 4.0 4 | Win | Jun | BT13838/9 |
| A4 | Novell NetMail 3.52 B | W/L | Jun | 1756/7/8 |
| A5 | TrueNorth eMailServer Corporate Edition 5.2.2 | Win | Jun | BT14065 |
| A6 | Alt-N MDaemon 8.0 3 | Win | Jul | BT14315/7 |
| A7 | GNU Mailutils 0.6.1 | Lin | Sep | 2878 |
| A8 | University of Washington Imap 2004f | Lin | Oct | 2933 |
| A9 | Floosietek FTGate 4.4 | Win | Nov | BT15449 |
| A10 | Qualcomm Eudora WorldMail Server 3.0 | Win | Nov | 3189 |
| A11 | MailEnable Professional 1.6 and Enterprise Edition 1.1 | Win | Nov | BT15492/4 |
| A12 | MailEnable Professional 1.7 and Enterprise Edition 1.1 | Win | Nov | BT15556 |

**Table 3. Applications with vulnerabilities.**

From the analysis of the reports, it was possible to identify 9 IMAP products with vulnerabilities (see Table 3). In a few cases, more than one version of the same application had problems. 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 associated reports. If a four digit number NNNN is present, then the identifier is CVE-2005-NNNN. In some situations there was no CVE number assigned to the vulnerability, and in those cases the bugtraq identifier is provided (e.g., BT13838). For applications with multiple reports, it was used a condensed representation – for example, 1014/5 corresponds to CVE-2005-1014 and CVE-2005-1015.

There were two other products identified in the reports – the Ipswitch Collaboration Suite/IMail 8.13 and the Up-IMAPProxy 1.2.4. For the 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, the exploits were incapable of exploring the described flaw, and therefore, we decided to disregard these products from further evaluation (either the reports were false or the obtained applications had already been fixed).

## 5.2. Vulnerability Assessment

After the identification of the products with flaws, it was necessary to obtain as many applications (with the right versions) as possible. We had one main difficulty while attempting to accomplish this objective – in some cases the

| ID | Vuln Type | State | Potential Attack |
|---|---|---|---|
| A3 | ID | S2 | A01 SELECT ./../../<OTHER-U>/inbox |
| A4 | BO | any | <A×2596> |

**a) Potentially detected vulnerabilities.**

| ID | Vuln Type | State | First Successful Attack |
|---|---|---|---|
| A1 | BO | S2 | A01 AUTHENTICATE <A×1296> |
|  | BO | S2 | A01 SELECT <A×1296> |
| A2 | FS | any | <%s×10> |
| A5 | FS | S2 | A01 LIST <A×10> <%s×10> |
| A6 | BO<br>BO | S2<br>any* | A01 CREATE <A×244><br><A×1260> |
| A7 | FS | S3 | A01 SEARCH TOPIC <%s×10> |
| A8 | BO | S2 | A01 SELECT ”{localhost/user=\”}” |
| A9 | BO | S2 | A01 EXAMINE <A×300> |
| A10 | ID | S2 | A01 SELECT ./../../<OTHER-U>/inbox |
| A11 | BO<br>ID | S2<br>S2 | A01 SELECT <A×1296><br>A01 CREATE /<A×10> |
| A12 | DoS | S2 | A01 RENAME <A×10> <A×10> |

**b) Detected previously known vulnerabilities.**

| Application | Vuln Type | State | First Successful Attack |
|---|---|---|---|
| TrueNorth eMail-Server Corporate Edition 5.3.4 | BO | S3 | A01 SEARCH <A×560> |

**c) New vulnerability discovered with AJECT.**

**Table 4. Attacks generated by AJECT to detect IMAP vulnerabilities (* using CRAM-MD5 auth scheme).**

vulnerable versions were no longer available in the official web sites. This was specially true for commercial products, where whenever a new patched version is produced, the older ones are removed. A web search had to be carried out to get the older versions, however, in two occasions without success.

Table 4 presents a summary of the attacks generated by AJECT that successfully activated the software bugs. Each line contains our internal application identifier (ID, also see Table 3), the type of vulnerability (where BO is a heap or stack Buffer Overflow; ID is an Information Disclosure; FS is a Format String; DoS is a Denial of Service [26]), the IMAP state in which the attack was successful (also see Table 1), and the attack itself. In order to keep the description of the attacks small, we had to use a condensed form of command representation where: <*A*×*N*> means letter 'A' repeated *N* times; and <*OTHER-U*> corresponds to another existing username.

For the two applications that we were unable to get, it was necessary to employ a different approach in the tests. The Injector was used to generate and carry out the attacks against a dummy IMAP server. Basically, this server only stored the contents of the received packets and returned simple responses. The packets were then later analyzed to determine if one of the attacks could activate the reported vulnerability. In Table 4 a) are presented the results of these experiments, and in both cases an attack was generated that could supposedly explore the vulnerabilities.

The vulnerabilities actually detected with AJECT are presented in Table 4 b). From the table it is possible to conclude that AJECT is capable of detecting several kinds of bugs, ranging from buffer overflows to information disclosure. Since we had a limited time for testing, and since we wanted to evaluate a large number of applications, we had to interrupt the tests as soon as a vulnerability was discovered so only the first successful attack is presented. In the few cases where experiments were run for a longer period, we noticed that 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 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. Therefore, in order to find out exactly if attacks are equivalent, one would need to have access to the source code of the applications (something impossible to obtain for a majority of the products). Consequently, we decided to take a conservative approach, where all attacks were deemed equivalent except in the situations where they correspond without any doubt to different vulnerabilities.

During the course of our experiments, we were able to discover a previously unknown vulnerability (see Table 4 c)). The attack sends a large string in a SEARCH command that causes a crash in the server. This indicates that the bug is a boundary condition verification error, which probably 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.

## 6. Related Work

The paper describes a methodology and a tool for the discovery of vulnerabilities on services provided by network or local daemons, through the injection of attacks (i.e, malicious faults). This work has been influenced by several research areas, namely:

**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 [3, 20]. Traditionally, fault injection has been utilized to emulate several kinds of hardware faults, ranging from transient memory corruptions to permanent stuck-at faults. Three main techniques have been developed to inject these faults: hardware-based tools resort to additional hardware to actually introduce the faults in the system, in most cases through pin-level injection, but also through radiation and electromagnetic interference [4, 17, 24]; simulation models with different levels of abstraction, e.g., device and network, have been employed by simulation-based tools to study the behavior of systems, starting from the early stages of design [16, 22]; software-based tools insert errors in the various parts of a running system by executing specific fault injection code [23, 29, 18, 7]. The emulation of other types of faults has also been accomplished with fault injection techniques, for example, software and operator faults [8, 12, 6]. 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 [25, 2].

**Vulnerability Scanners** are 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: COPS [14], FoundStone Enterprise [15], Internet Scanner [21], Nessus [28], and QualysGuard [27]. 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; 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 systems in production, they have the limitation that they are unable to uncover unknown vulnerabilities.

**Static vulnerability analyzers** look for potential vulnerabilities in the source code of the applications. Typically, these tools examine the source code for dangerous patterns that are usually associated with buffer overflows, and then they provide a listing of their locations [34, 33, 19]. 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 [13]. Static analysis has also been applied to other kinds of vulnerabilities, such as race conditions during the access of (temporary) files [5]. 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 skip some of the existing vulnerabilities.

**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 damages caused by their exploitation. Most of these techniques were developed to protect systems from buffer overflows. A few examples are: StackGuard [10], Stack Shield [30], and PointGuard [9] are compiler-based tools that determine at run-time if a buffer overflow is about to occur, and stop the program execution before it can happen. A recent study compares the effectiveness of some of these techniques, showing that they are useful only to prevent a subset of the attacks [35].

## 7. Conclusion

The paper 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 application while it runs 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.

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

## References

[1] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*. Jan. 2002. http://www.research.ec.org/maftia/deliverables/D21.pdf.

[2] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *Proc. of the Int. Conference on Dependable Systems and Networks*, pages 867–876, June 2004.

[3] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Trans. on Computers*, 42(8):913–923, Aug. 1993.

[4] J. Arlat, Y. Crouzet, and J. Laprie. Fault injection for dependability validation of fault-tolerant computer systems. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 348–355, June 1989.

[5] M. Bishop and M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[6] A. Brown, L. C. Chung, and D. A. Patterson. Including the human factor in dependability benchmarks. In *Workshop on Dependability Benchmarking, in Supplemental Volume of DSN 2002*, pages F–9–14, June 2002.

[7] J. Carreira, H. Madeira, and J. G. Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *IEEE Trans. on Software Engineering*, 24(2):125–136, Feb. 1998.

[8] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 304–313, June 1996.

[9] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proc. of the 12th USENIX Security Symposium*, Aug. 2003.

[10] 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. In *Proc. of the 7th USENIX Security Conference*, pages 63–78, Jan. 1998.

[11] M. Crispin. Internet Message Access Protocol - Version 4rev1. Internet Engineering Task Force, RFC 3501, Mar. 2003.

[12] J. Durães and H. Madeira. Definition of software fault emulation operators: A field data study. In *Proc. of the Int. Conference on Dependable Systems and Networks*, pages 105–114, June 2003.

[13] J. Durães and H. Madeira. A methodology for the automated identification of buffer overflow vulnerabilities in executable software without source-code. In *Proc. of the Second Latin-American Symposium on Dependable Computing*, Oct. 2005.

[14] D. Farmer and E. H. Spafford. The COPS security checker system. In *Proc. of the Summer USENIX Conference*, pages 165–170, June 1990.

[15] FoundStone Inc. FoundStone Enterprise, 2005. http://www.foundstone.com.

[16] K. Goswami, R. Iyer, and L. Young. Depend: A simulation-based environment for system level dependability analysis. *IEEE Trans. on Computers*, 46(1):60–74, Jan. 1997.

[17] O. Gunnetlo, J. Karlsson, and J. Tonn. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 340–347, June 1989.

[18] S. Han, K. G. Shin, and H. A. Rosenberg. Doctor: An integrated software fault-injection environment for distributed real-time systems. In *Proc. of the Int. Computer Performance and Dependability Symposium*, pages 204–213, 1995.

[19] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proc. of the Symposium on Networked and Distributed System Security*, Feb. 2003.

[20] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, Apr. 1997.

[21] Internet Security Systems Inc. Internet Scanner, 2005. http://www.iss.net.

[22] E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, and J. Karlsson. Fault injection into VHDL models: The MEFISTO tool. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably in Dependable Computing Systems*, pages 329–346. Springer-Verlag, 1995.

[23] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A tool for the validation of system dependability properties. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 336–344, June 1992.

[24] J. Karlsson, J. Arlat, and G. Leber. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proc. of the Int. Working Conference on Dependable Computing for Critical Applications*, pages 267–287, Sept. 1995.

[25] P. Koopman and J. DeVale. Comparing the robustness of POSIX operating systems. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 30–37, June 1999.

[26] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, N. Mehta, and R. Hassel. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley Publishing Inc, 2004.

[27] Qualys Inc. QualysGuard Enterprise, 2005. http://www.qualys.com.

[28] Tenable Network Security. Nessus Vulnerability Scanner, 2005. http://www.nessus.org.

[29] T. Tsai and R. Iyer. An approach towards benchmarking of fault-tolerant commercial systems. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, pages 314–323, June 1996.

[30] Vendicator. Stack Shield : A stack smashing technique protection tool for Linux, Jan. 2001. http://www.angelfire.com/sk/stackshield/.

[31] P. Veríssimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the Third IEEE Information Survivability Workshop*, Oct. 2000.

[32] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.

[33] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proc. of the 16th Annual Computer Security Applications Conference*, Dec. 2000.

[34] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proc. of the Network and Distributed System Security Symposium*, Feb. 2000.

[35] J. Wilander and M. Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proc. of the Network and Distributed System Security Symposium*, pages 149–162, Feb. 2003.