ISSN:0975-9646

Protocol Based Attack Injection Framework to Fault Diagnosis in Server Applications

Kalagotla satish kumar, L.R.Krishna kotapati

M.Tech, Software Engineering, Gitam Institute of Technology,
Gitam University, Vishakapatnam, India.

***Abstract* – Today our increasing reliance on network connected computer systems, security incidents and their causes are important problems that need to be addressed. New threats and forms of attacks are constantly being revealed by adversaries, to compromise the secured server applications. This paper describes an attack injection methodology is implemented in AJECT tool, that can be used for susceptibility detection and removal. The AJECT tool uses a specification of the server's communication protocol to automatically generate a large number of attacks by using a predefined test case generation algorithm. AJECT not only injecting the attacks through the network to server and it monitors the behavior of the server both from a client perspective and inside the target machine. The observation of an unexpected behavior indicates a successful attack and the potential existence of a flaw. Experimental results show that AJECT can discover several kinds of errors, in today secured application.**

***Keywords*- Testing and debugging, Fault injection attack injection, test design**.

I. INTRODUCTION

Now a days our everyday life activities has increased over the years, as more and more tasks are accomplished with computer systems 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. In application development process we are using third party software, such as COTS components, is to speed up development, even though in many cases it is poorly documented and supported. Because of scarily documented and tested 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.

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).

Recently several techniques 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. 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 vulnerability 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 in-JECtion Tool that can be used for vulnerability detection and removal. AJECT simulates the behavior of an adver-sary by injecting attacks against a target system. Then, it observes the execution of the target 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 elimination.

AJECT 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 12 attack injection experiments with 6 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 systems. Our evaluation confirmed that AJECT could find different classes of vulnerabilities in five of the servers, and assist the developers in their removal.

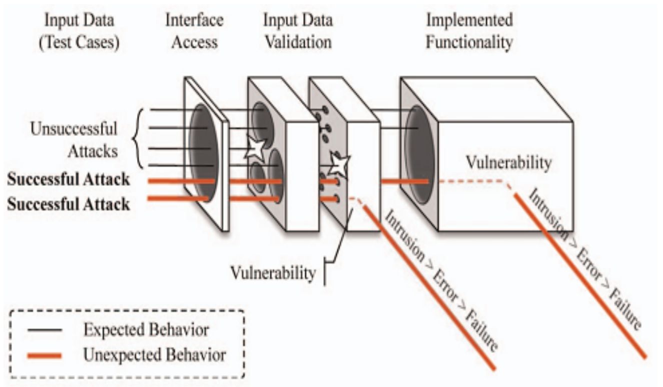


Figure 1 . Existed Attacking methodology

2. INTRUSION DETECTION BY ATTACKS

The AVI(attack, vulnerability, intrusion)composite fault model introduced in [1] helps us to understand the mechanisms of failure due to several classes of malicious faults (seeFigure1). 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

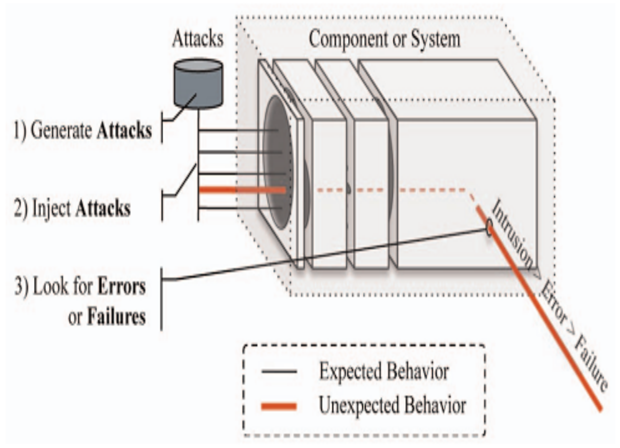


Figure 2 . Proposed Attack Injection Methodology

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 vulnerability somewhere on the execution path of its processing logic.

3. THE ATTACK INJECTION METHODOLOGY

There are four basic entities in the architecture of AJECT, the Target System, the Target Protocol Specification, the Attack Injection the Monitoring system (seeFigure3). 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 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, approximate 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.

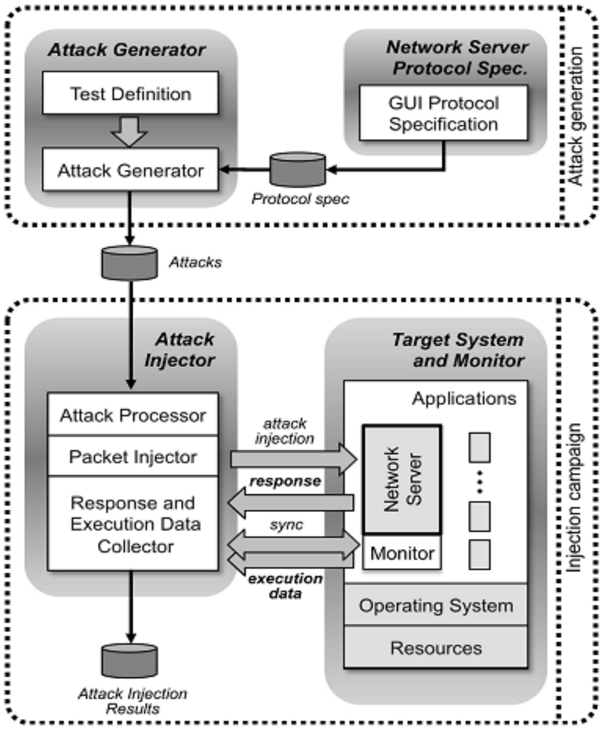


Figure 3. AJECT tool Architecture.

The Attack inJECtion Tool (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 errors with distinct consequences. 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.

A. Test Attacking 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 bequite 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. 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.

B. AJECT Component Phases

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.

C. Attack Generation Component

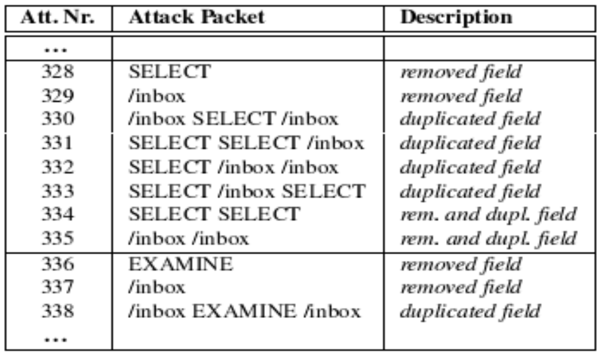
The purpose of attack generation is to create a series of attacks that can be injected in the target system. The design of the tool does not require the source code of the server to be available to devise the attacks. This allows AJECT to support a larger number of target systems, such as commercial 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 protocol send 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 the reis 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. 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 generator will generates various types of test definitions to find the vulnerabilities in server applications.

Delimiter Test Definition is 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. For example this test definition would generate various login messages with a valid username and password but either with or without delimiters.

Syntax Test Definition is type 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. 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).
-



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*
...		

Figure .4 Example of predefined test cases.

D. 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 3). 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, at 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 so fastest 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).

E. 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.

F.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 information 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 by looking to the targets execution, 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.

G. Attack Generation algorithm

Attack generation algorithm can be used to generate the various types of attacks based on server application protocol specification .The given below algorithm will generates various types of attacks by embedding the predefined malicious tokens and illegal values in the testcases. 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 legalvalues.

```
TestValue
  Input: $Protocol$ $\leftarrow$ specification of the network protocol used
         by the server
  Output: $Attacks$

  $S$ $\leftarrow$ set of all states of the $Protocol$ specification
  foreach State $s \in S$ do
    $M$ $\leftarrow$ set of message specifications of $s$
    $Transition_s$ $\leftarrow$ set of ordered network packets necessary
         to reach $s$
    $P$ $\leftarrow$ $\phi$

    foreach MessageSpecification $m \in M$ do
      foreach FieldSpecification $f \in m$ do
        if $f$ is type Numbers then
          $f'$ $\leftarrow$ all boundary values, plus some intermediary
               illegal values, from $f$ specification
        elseif $f$ is type Words then
          $f'$ $\leftarrow$ $\underline{\text{combin. of predefined malicious tokens}}$

        $m'$ $\leftarrow$ copy of $m$ replacing $f$ with $f'$
        $P$ $\leftarrow$ $P$ $\cup$ {set of all network packets based on $m'$
             specification}

    foreach $attack\_packet \in P$ do
      $attack$ $\leftarrow$ $Transition_s$ $\cup$ $\{attack\_packet\}$
      $Attacks$ $\leftarrow$ $Attacks$ $\cup$ $\{attack\}$

  return $Attacks$
```

IV. 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 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.

A. 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 vulnerable versions were no longer available in the official websites.

Table1 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.

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	S2	A01 CREATE <A×244>
	BO	any*	<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	S2	A01 SELECT <A×1296>
	ID	S2	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 .1. Attacks generated by AJCET on IMAP Serves

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 1 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 1 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 Mail utils, there were already more than 200 attacks that similarly crashed the application. 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 test cases.

Some times 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 1 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 E-Mail Server application were tested, including the most recent one, and all of them were vulnerable to this attack.

V. CONCLUSIONS

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 in correctly, 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 over flows, format strings, and information disclosure bugs).In addition, AJECT was able to discover a new buffer overflow vulnerability.

VI. 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. Ver´issimo, 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, June2004.

[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, June1989.

[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 DSN2002,pages F–9–14, June2002.

[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. Christ mansson and R. Chillarege. Generation of an error set that emulates software faults. In Proc. of the Int. Symp. On Fault-Tolerant Computing, pages304–313, June1996.

[9] C.Cowan, S.Beattie, J.Johansen, and P.Wagle. Point guard: Protecting pointers from buffer overflow vulnerabilities. In Proc.ofthe12th 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. Stack Guard: Automatic adaptive detection and prevention of buffer-overflow attacks. In Proc. of the 7th USENIX Security Conference, pages63–78, Jan.1998.

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

[12] J.Duraes˜ 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,June2003.

[13] J. Duraes˜ and H. Madeira. A methodology for the automate 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, June1990.

[15] Found Stone Inc. Found Stone 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.