IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661 Volume 2, Issue 4 (July-Aug. 2012), PP 17-21
www.iosrjournals.org

Discovery of Vulnerabilites in Network Servers Using Aject

D.Udaya Kumar[1], N.Madhavi[2]
[1,2]Dept. of IT, Sree Vidyanikethan Engg College, India.

Abstract: *The Vulnerability problem is discovered by presenting an attack injection methodology in software components. The Attack injection methodology, implemented 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. The attack injection methodology is 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.*

Keywords: *Attack Injection, Fault Injection, Software Vulnerabilities, Software Engineering, Testing*.

I. Introduction

Through the years our society has become increasingly reliant on the pervasiveness of the Internet to perform every day activities (e.g., tax payments, shopping, and entertainment). Therefore, any disruption that prevents users from utilizing certain services can have a significant negative impact both in the general population and providers. In the Internet, how-ever, any service is susceptible to attacks. In this paper we describe a novel methodology towards the systematic detection of vulnerabilities in network servers, including subtle types of faults that lead to the depletion of some local resource. Resource-exhaustion vulnerabilities are difficult to find because 1) they might be triggered exclusively in very special conditions, and 2) the resource leaks may only be perceived after many activations. For this reason, specific techniques have to be developed to search for these problems, which in spite of being active can till remain concealed. Our approach was implemented in AJECT, a tool that automatically generates a large number of test cases (or attacks) based on a specification of the communication protocol utilized by the target server.

Next, it directs the attacks to the network interface of the target system while gathering detailed resource usage information. A post-processing analysis on the collected data is performed to build accurate resource usage projections and to find vulnerabilities. The attacks that triggered the vulnerabilities and the respective monitoring data can be provided to the developers to assist debugging. 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.

II. Vulnerabilities Definition

Vulnerabilities are the primitive faults within a system—in particular, design or configuration faults—that can be introduced during the system's development or operation. For example, as a step in an overall plan of attack, an attacker might introduce vulnerabilities in the form of malware. Attacks are malicious faults that attempt to exploit one or more vulnerabilities. An attack that successfully exploits vulnerability results in an intrusion, which is normally characterized by an erroneous system state (for example, a system file with unwarranted access permissions for the attacker). If nothing is done to handle these errors, a security failure will probably occur. Attacks often assume the form of inconsistent interactions with different legitimate participants in order to confuse them. 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. And current best practice indicates that it is best answered by performing regular vulnerability assessments to identify the known vulnerabilities in a network before hackers find them. Some environments may require a weekly (or even daily) test frequency due to the value of the resources being protected and the increasing complexity of networks and the speed at which vulnerabilities can now be exploited. Take into consideration that network complexity and connectivity continually increases. The number of vulnerabilities being discovered daily and the speed at which exploits can launch malicious code has grown.

III. Software Vulnerabilities

Software vulnerabilities commonly found in computer programs and discuss their distinguishing features. The purpose of this section is to serve as an introduction for readers who might be familiar with static analysis, but are new to the field of vulnerability research.

IV. Buffer Overflows

One of the oldest known vulnerabilities is the buffer overflow, famously used by the Morris Internet Worm in 1988. The classic buffer overflow is a result of misuse of string manipulation functions in the standard C library. The C language has no native string data type and uses arrays of characters to represent strings. When reading strings from the user or concatenating existing strings, it is the programmer's responsibility to ensure that all character arrays are large enough to accommodate the length of the strings. Unfortunately, programmers often use functions such as strcpy() and strcat() without verifying their safety. This may lead to user data overwriting memory past the end of an array. One example of a buffer overflow resulting from misusing strcpy() is given below:

```
char dst[256];
char* s = read_string ();
strcpy (dst, s);
```

The string s is read from the user on line 2 and can be of arbitrarily length. The strcpy() function copies it into the dst buffer. If the length of the user string is greater than 256, the strcpy() function will write data past then end of the dst[ ] array. If the array is located on the stack, a buffer overflow can be used to overwrite a function return address and execute code specified by the attacker [2]. A heap based overflow can be exploited by overwriting the internal structures used by the system memory allocator and tricking it into overwriting critical system data. However, the mechanics of exploiting software vulnerabilities are outside the scope of this work and will not be discussed further. For the purposes of vulnerability detection, it is sufficient to assume that any bug can be exploited by a determined attacker.

V. Format String Bugs

Format string bugs belong to a relatively recent class of attacks. The first published reports of format string bugs appeared in 2000, followed by the rapid discovery of similar vulnerabilities in most high-profile software projects. The format string bug is a relatively simple vulnerability. It arises when data received from an attacker is passed as a format string argument to one of the output formatting functions in the standard C library, commonly known as the printf family of functions.

These functions produce output as specified by directives in the format string. Some of these directives allow for writing to a memory location specified in the format string. If the format string is under the control of the attacker, the printf() function can be used to write data to an arbitrary memory location. An attacker can use this to modify the control flow of a vulnerable program and execute code of his or her choice. The following example illustrates this bug:

```
char *s = read_string ();
printf (s);
```

The string s is read from the user and passed as a format string to printf(). An attacker can use format specifier such as "%s" and "%d" to direct printf() to access memory at an arbitrary location. The correct way to use printf() in the above code would be printf("%s", s), using a static format string.

VI. Integer Overflows

A third kind of software vulnerability is the integer overflow. These vulnerabilities are harder to exploit than buffer overflows and format string bugs, but despite this they have been discovered in OpenSSH, Internet Explorer and the Linux kernel. There are two kinds of integer issues: sign conversion bugs and arithmetic overflows. Sign conversion bugs occur when a signed integer is converted to an unsigned integer. On most

modern hardware a small negative number, when converted to an unsigned integer, will become a very large positive number. Consider the following C code:

```
char buf[10];
int n = read_int ();
if (n < sizeof(buf))
memcpy (buf, src, n);
```

On line 2 we read an integer n from the user. On line 3 we check if n is smaller than the size of a buffer and if it is, we copy n bytes into the buffer. If n is a small negative number, it will pass the check. The memcpy() function expects an unsigned integer as its third parameter, so n will be implicitly converted to an unsigned integer and become a large positive number. This leads to the program copying too much data into a buffer of insufficient size and is exploitable in similar fashion to a buffer overflow. Arithmetic overflows occur when a value larger than the maximum integer size is stored in an integer variable. The main cause of arithmetic overflows is addition or multiplication of integers that come from an untrusted source. If the result is used to allocate memory or as an array index, an attack can overwrite data in the program. Consider the following example of a vulnerable program:

```
int n = read_int ();
int *a = malloc (n * 4);
a[1] = read_int ();
```

On line 1 we read an integer n from the user. On line 2 we allocate an array of n integers. To calculate the size of the array, we multiply n by 4. If the attacker chooses the value of n to be 0x40000000, the result of the multiplication will overflow the maximum integer size of the system and will become 0. The malloc() function will allocate 0 bytes, but the program will believe that it has allocated enough memory for 0x40000000 array elements. The array access operation on line 3 will overwrite data past the end of the allocated memory block.

VII. Attack Injection Projection Methodology

An effective way to carry out a attack consists in exploiting some known vulnerability in a target system, here- after also referred as network server. In other words, the adversary resorts to a malicious interaction to activate the vulnerability and as a result, the server suffers an immediate crash or some sort of service degradation. In the presence of a fatal crash, the fault usually brings the server into an erroneous state that cannot be handled, abruptly ending the execution. Example vulnerabilities that usually produce such behavior are the well-known "buffer overflows" and "divide by zero". On the other hand, in the case of service degradation, faults are more subtle but they can also lead to a complete halt if they occur at a higher rate.

These faults appear due to vulnerabilities. There are two classes of problems that can lead to vulnerabilities. The first one is related to a bad design or an inefficient implementation of the server, forcing it to spend more resources then required. As a consequence, the overloading of the system can be accomplished with much less effort, when compared with an efficient design or implementation. For example, a component with poor resource management or slow algorithms can reserve large chunks of memory that are only partially utilized or waste valuable CPU cycles. The second problem is associated with resource leakages. Here, the resource is indeed necessary but the server fails to make it available after use. Examples are a component that neglects to close a file descriptor or to free some memory, or a log file that grows indefinitely due to some error condition. These flaws are particularly important in long running servers, since the cumulative resource consumption can build up through time. On a malicious setting this is even more serious because the particular situation that causes the resource leakage can be continuously forced by the adversary.

VIII. Searching For Vulnerabilities

Typical solutions for vulnerability discovery, such as scanners or fuzzers, inject faults (i.e., malicious attacks) in the target system and look for an abnormal outcome that indicates some sort of problem [26, 13, 31]. This approach is usually confined to locate vulnerabilities that produce quite visible effects, normally a crash. More subtle results, like those associated with resource-exhaustion vulnerabilities, are much more difficult to observe. The resource loss cannot be detected just by looking at a single snapshot of its utilization.

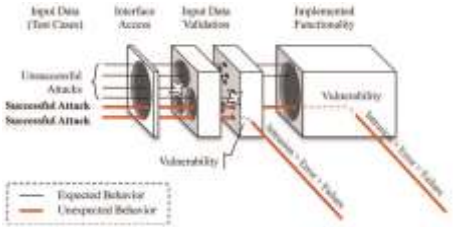

Fig 1:Attack Injection Process

The attack injection projection methodology searches for vulnerabilities through a comprehensive analysis on the system resource utilization of many (potentially malicious) client/server interactions. This is accomplished by injecting attacks into the network interface of the target system while monitoring the server's evolution. The whole procedure is performed in three basic phases:

Test case generation phase Given a specification of the communication protocol utilized by the server, the test case generation creates valid and invalid network interactions (or attacks). When transmitted to the target system, these messages will test its ability to cope with some erroneous data, such as an out-of-bounds value, a missing field, or an incorrect type of message. This phase can employ different attack generation algorithms specialized in detecting specific classes of vulnerabilities. As more knowledge is gained on how to activate more vulnerabilities, additional generation algorithms can be implemented to produce test cases that could trigger those vulnerabilities.

Exploratory phase this phase runs the entire universe of the generated test cases, by carrying out each test case with a fresh copy of the target server. Each attack has to be per-formed several times, as opposed to a single injection used in the traditional fault injection, so that the monitoring data can be gradually collected to build a trend on the resource usage. Ideally, it is desirable to keep a lower number of injections to ensure a feasible exploratory phase. However, depending on the resource and monitoring capabilities, it might be necessary to execute a large number of attacks to guarantee that changes on the resource use can be observed. For example, if the monitoring mechanism can measure exactly how much memory is allocated and released by each attack, then two repeated injections might be sufficient to detect any memory usage variation.

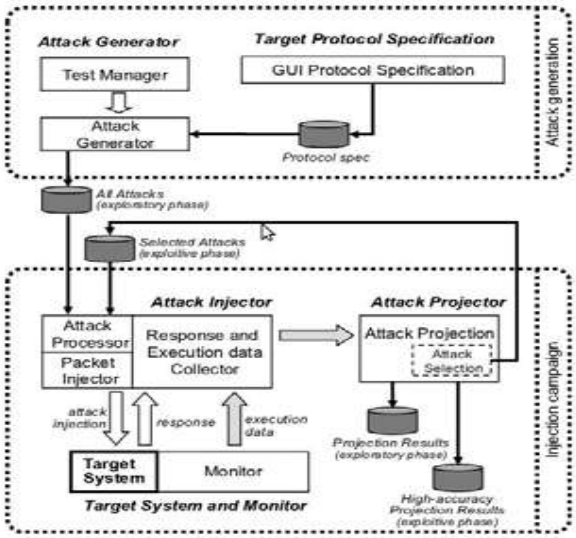

Fig 2: The architecture of the AJECT tool.

On the other hand, if the memory leak is small and the monitoring mechanism works with a page size granularity (e.g., counts the number of pages assigned to the process), then it is necessary to re-inject the attacks as many times as needed to force the allocation of an additional memory page. In any case, enough information must be obtained to construct a usage profile for each considered attack and resource. At the end of this phase there is, for all test cases, a profile for each monitored resource. With this information it is possible to recognize which attacks are more dangerous by looking for the profiles with higher growth rates. Exploitive phase The execution of this last phase is optional, though it should provide more accurate profiles, which supports for better forecasts of the resource utilization and to remove false positives. The second injection campaign is initiated exclusively for the small subset of attacks that showed highest DoS potential.

Essentially, it performs a larger number of repeated injections for these attacks and calculates new and more precise projections. Using the AIP methodology in existing servers has several useful applications. It allows the discovery of vulnerabilities, whether caused by simple bugs, e.g., buffer overflows, or by more subtle faults that also result in the resource-exhaustion. AIP can also contribute to the identification of the root of the problem, by discovering which resources are being depleted and by providing the test cases that activate the vulnerabilities.

Developers can then use this invaluable information to fix the problem on the server. Additionally, the resource usage models can support further analysis since they let us forecast the consumption of resources in various scenarios with distinct attack magnitudes. For example, it is possible to find out: what are the main resource bottlenecks of a system; how many attacks can be sustained before the execution halts, and therefore

estimate the critical level of the attack; compare the robustness of two implementations of the same server given some hard- ware configuration.

IX. 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 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. 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.

References

[1]. P. Verissimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y.Deswarte, R. Stroud, 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]. N. Neves, J. Antunes, M. Correia, P. Verissimo, R. Neves,"Using Attack Injection to Discover New Vulnerabilities," Proc.Int'l Conf. Dependable Systems and Networks, June 2006.

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

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

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

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

[7]. J. Carreira, H. Madeira, 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, [Online] Available : http://citeseer.ist.psu.edu/54044.html; http://dsg.dei. uc.pt/Papers/dcca95.ps.Z, Jan. 1995.