Detection and Prediction of Resource-Exhaustion Vulnerabilities *

João Antunes Nuno Ferreira Neves Paulo Verissimo
University of Lisboa, Faculty of Sciences, LASIGE – Portugal

Abstract

Systems connected to the Internet are highly susceptible to denial-of-service attacks that can compromise service availability, causing damage to customers and providers. Due to errors in the design or coding phases, particular client-server interactions can be made to consume much more resources than necessary easing the success of this kind of attack. To address this issue we propose a new methodology for the detection and identification of local resource-exhaustion vulnerabilities. The methodology also gives a prediction on the necessary effort to exploit a specific vulnerability, useful to support decisions regarding the configuration of a system, in order to sustain a certain attack magnitude. The methodology was implemented in a tool called PREDATOR that is able to automatically generate malicious traffic and to perform post-processing analysis to build accurate resource usage projections on a given target server. The validity of the approach was demonstrated with several synthetic programs and well-known DNS servers.

1. Introduction

Through the years our society has become increasingly reliant on the pervasiveness of the Internet to perform everyday activities (e.g., tax payments, shopping, 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, however, any service is susceptible to denial-of-service (DoS) attacks. These attacks affect the system's ability to provide the service with the expected quality or they simply bring the service operation to a halt. A recent study about this type of attacks has recorded for instance an average of 5,213 SYN floods per day [30]. The same study also reported that a daily average of 63,912 active bot-infected computers were observed over a period of six months (totaling more than 6 million distinct computers), which in many cases are employed in coordinated DoS attacks.

*This work was partially supported by EC through project IST-2004-27513 (CRUTIAL) and NoE IST-4-026764-NOE (RESIST), by FCT through the Multiannual and the CMU-Portugal Programmes.

Generically, a DoS can be performed in two ways. One method consists in overwhelming the target system or its network connection with an excessive load, by leveraging the utilization of a number of resources (CPU and bandwidth) which surpass the target capabilities. In this case, either some network component starts dropping (good and bad) packets or the target system spends most of its time processing the erroneous requests, preventing valid clients from getting responses within an acceptable delay. In the second approach, the DoS can be carried out by exploiting some known vulnerability in the system through the transmission of well-crafted malicious data. Basically, the attackers take advantage of some dormant fault in the target (i.e., a design flaw, a software bug, or a configuration problem) that can be activated by an unusual network interaction, causing for example a crash. Usually, this type of attacks can be accomplished in a simple and effective way, requiring very little resources.

In this paper we describe a novel methodology towards the systematic detection of DoS 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 still remain concealed.

Our approach was implemented in PREDATOR, 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. On the other hand, the resource projections give a forecast on the amount of effort necessary for an attacker to deplete the server's resources. This information can also be used by administrators to assign critical levels to vulnerabilities, and to assist on decisions regarding hardware upgrades

to sustain stronger attacks, at least until a patch is available.

The methodology was experimentally evaluated with synthetic leak servers, i.e., programs that contained various kinds of resource leaks, and with seven public-domain DNS servers. The results revealed that the proposed methodology is quite suitable not only to discover remotely exploitable vulnerabilities that lead to the server crash, but also with the profiling of different resource usages. In particular, they demonstrated the usefulness of the tool by disclosing two resource-exhaustion vulnerabilities in an active DNS server.

2. Attack Injection Projection Methodology

2.1. Resource-exhaustion vulnerabilities

An effective way to carry out a DoS attack consists in exploiting some known vulnerability in a target system, hereafter 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 a 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 resource-exhaustion vulnerabilities. We define resource-exhaustion vulnerabilities as follows:

Def: A *resource-exhaustion vulnerability* is a specific type of fault that causes the consumption or allocation of some resource in an undefined or unnecessary way, or the failure to release it when no longer needed, eventually causing its depletion.

As this definition indicates there are two classes of problems that can lead to resource-exhaustion. 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 (also known as software aging [32]). 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.

Most DoS attacks aim at exploiting one or a combination of the two classes of problems above. TCP SYN attacks, for instance, cause the server to create half-open connections until it fills up the maximum number of available connections [8]. Other attacks, such as memory leak attacks, cause DoS by continuously forcing the server to execute a particular task that uses some chunk of memory that is not freed upon completion [16].

Keep in mind however that these vulnerabilities can have other impact besides DoS, such as data corruption or some other exceptional illegal state. Consider the following scenario where a server with very little disk space does not properly verify the error status of a write call. If the write call terminates abruptly due to space shortage, the data stored in the disk is left in an inconsistent state.

2.2. 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. Only a continuous and careful monitoring can perceive the overall tendency in which the resource depletion is developing into.

The *attack injection projection methodology (AIP)* 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 performed 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. 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. We use regression analysis on the collected data to produce a statistical model of the actual resource consumption. Linear regression, however, also imposes constrains on the minimum number of repeated injections. If p coefficients have to be estimated, then $n \geq p+1$ data samples are necessary to determine the regression, and therefore at least n injections have to be performed. Consequently, the minimum number of repeated injections is defined by both the monitoring capabilities and the type of regression analysis.

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 DoS 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 hardware configuration.

2.3. Modeling resource usage

The AIP methodology produces statistical models representing the use of *each* resource for *each* test case. Since complex models typically require more computation time, there is usually a tradeoff between the accuracy of the model and the number of tests that can be performed within a reasonable time frame. In order to maximize tests, and increase confidence on the correctness of the server, the model has to be relatively simple to allow a rapid calculation. Among the different mathematical models that were considered, we opted for linear regression because it gave excellent results for the kind of analysis and vulnerabilities we were focusing.

Least-square analysis is employed to compute the parameters of the linear regression fitting. In its simplest form, linear regression estimates one parameter plus the constant intercept, which results in a straight line. This idea can be generalized to higher degree polynomials by estimating p parameters. For example, with p = 2 the projection is a quadratic function (i.e., a parabola), and with p = 3 a cubic function. The number of parameters is also related to the number of inflections, or "curves", that the polynomial has. A linear regression with p parameters will result in a polynomial of degree p with $p-1$ inflections. A cubic function, for instance, will be useful to represent data that follows a cubic polynomial pattern, i.e., that has two inflections.

Therefore, it is important to study the nature and pattern of the data itself in order to determine the best polynomial that fits it. First, consider that it is the same attack that is injected multiple times, which results in the execution of the same task over and over again. Also note that the monitoring mechanism measures the total amount of resources spent by the server throughout n injections. Consequently, the resource usage data is the *accumulated value* since the beginning of the injections of that attack (and not per injection). This translates into a *nondecreasing monotonic* resource utilization (e.g., CPU time never decreases, memory consumption is constant or increasing).

The following two propositions result from an understanding of the data, and help us determine the degree of the linear regression polynomial:

P1: Intuitively, since the resource usage data is nondecreasing, it should be suitably represented by straight line or

a degree-two polynomial, i.e., by a curve with zero or one inflection. Therefore, the number of parameters to be estimated can be $p \leq 2$, plus the constant intercept.

P2: Additionally, if resource waste is increasing (e.g., a memory leak), it does not necessarily grow at a constant rate. For example, some additional overhead may increase the resource consumption even further. This means that a straight line may not be enough to accurately represent the loss of the resource. Therefore, in some cases a polynomial with at least one inflection ($p \geq 2$) may be required.

From these propositions we conclude that $p = 2$ estimated parameters are necessary and sufficient to correctly model the consumption of a resource due to the repeated execution of the same attack (i.e., $y = ax^2 + bx + c$).

3. The PREDATOR Tool

PREDATOR, which stands for *PREDicting ATtacks On Resources*, is a fully automated black box testing tool that implements the AIP methodology to search for vulnerabilities. Unlike other ordinary fault injection tools, PREDATOR not only produces the test cases and injects them in the target, but also computes resource usage profiles for every attack and monitored resource.

The main features that characterize PREDATOR are: Firstly, *thorough resource and process monitoring*, making it capable of automatically detecting small resource usage variations, such as CPU cycles, number of child processes or threads, memory, disk, or open files. For this reason it can discover various kinds of bugs, like deadlocks or memory/disk leaks, that can be exploited to produce a service disruption; Secondly, *generation resource usage projections*, through a post-processing analysis of the collected data. This allows the discovery of attacks that can compromise the availability of the system, as well as the most dangerous protocol interactions; Thirdly, *test case prioritization*, achieved by two-phase attack injection campaigns. Only a minimum number of injections are performed in the first exploratory phase, and then just the most promising attacks are evaluated in the exploitive phase.

3.1. Architecture

The architecture of the PREDATOR tool is presented in Figure 1. The overall operation of the tool can be divided in the attack generation and the two injection campaigns (exploratory and exploitive).

The attack generation is performed off-line and only once for each target communication protocol. The attacks can be used in all test campaigns against target systems that share the same application protocol (e.g., DNS, FTP). The format

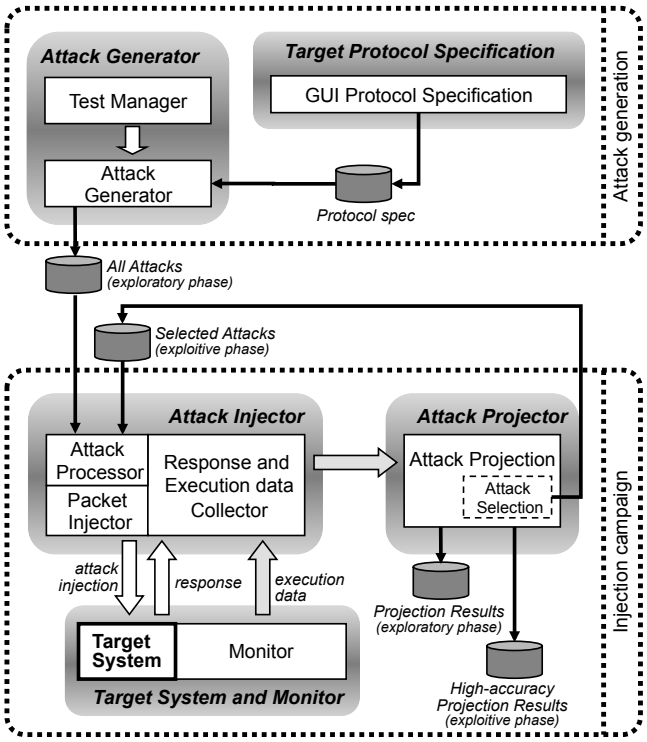

Figure 1. PREDATOR's architecture.

of messages, their fields, and data types of the protocol are defined through a graphical interface (GUI Protocol Specification). Additionally, the protocol states and transition messages are also identified. The GUI produces a protocol specification file which is later used by the test case generation algorithm to construct the attacks. The algorithm (similar to [18], implemented in the Test Manager and Attack Generator) creates variations of the protocol messages that test the target system's ability to cope with some erroneous attribute, such as an illegal parameter or out-of-order messages. All protocol states are tried, so each attack is also composed by the transition messages required to take the protocol to the state where the attack is supposed to inject the fault.

Both injection campaigns share exactly the same operation, although as pointed out earlier, just a few selected test cases are used in the exploitive phase. The Attack Processor decomposes each attack in its components, i.e., the state transition messages and the attack message itself. The Packet Injector sends the transition messages to the Target System, and once the protocol is in the designated state, it repeatedly injects the attack message several times. For example, 256 injections in the exploratory phase (due to the granularity of the memory monitoring) and 1024 injections in the exploitive phase (which empirically provided accurate projections for most situations). There is also an implicit synchronization between the Attack Injector and the Monitor (not represented in the figure). The Monitor launches a fresh copy of the Target System process (i.e., the server ap-

plication) before initiating a new test case, and terminates its execution once the injection campaign is done.

The Target System resource usage data acquired by the Monitor after every response, and the contents of the responses themselves, are gathered by the Response and Execution Data Collector. The Attack Projector uses linear regression on this data to build the statistical profiles of the resource usage. In the exploratory phase, a list with the most dangerous attacks, i.e., those with higher estimated coefficients, is continuously updated. At the end, PREDATOR outputs the resource usage projections for all attacks.

3.2. Implementation issues

PREDATOR resorts to third-party libraries to implement the monitor. The PTRACE facility is employed to intercept signals or system calls made by any of the server's processes (i.e., the main process, any forked children, and threads). Received signals are logged and the usage data is obtained at a few specific resource-related systems calls (e.g., memory utilization is probed after a memory allocation or deallocation call). This mode of operation passively traces the execution of the server, without much interference with its normal behavior. We have tried to reduce the overhead to a minimum by only updating the usage data at the relevant system calls. In some extreme situations, however, the monitoring activities can create some delays because of the constant pause, probe for data, and resume cycle. We plan to address this issue in future versions of PREDATOR.

The monitor maintains and regularly updates a global table with the resource usage data. The following local resources are watched:

- *total number of processes*, including forked children and threads of the target server. PTRACE signal interceptions are used to track new process ids (PIDs);
- *memory pages*, given by the number of resident set pages minus the shared pages, are obtained through the LibGTop library [5];
- *file descriptors*, such as those identifying opened disk files or network sockets, are kept in a updated list of file descriptors. LSOF [1] calls are used to keep track of the open files;
- *disk usage*, specified by the number of bytes written to disk. This value is obtained by parsing the LSOF's output for the files in use, and recording their size throughout the execution;
- *CPU cycles*, which corresponds to the work performed by the processor by all server's processes, is obtained by performance hardware counters. The linux kernel had to be patched to associate to each process a private set of *virtual* hardware counters [21]. PREDATOR controls and accesses these counters through the PAPI library [12];
- *wall time*, measured as the elapsed time from the beginning of the main process execution, is computed by simple `gettimeofday()` calls. This resource is monitored mainly to compare its value with the number of CPU cycles. Large CPU and wall time discrepancies normally indicate a non-active wait, which suggests the presence of some timeout or deadlock.

4. Experimental Evaluation

The main purpose of the evaluation was to validate the AIP methodology and to demonstrate its usefulness by detecting resource exhaustion vulnerabilities in DNS servers. The experiments were carried out in two Intel Pentium Dual-core 2.8Ghz PCs, with 512MB of main memory. One PC was running the Injector, while the other corresponded to the Target System (also running the Monitor component). Each target network server was installed in a separated cloned partition of a basic Ubuntu Linux Distribution, with approximately 360MB of free disk space.

The target servers were chosen to be the representative for the different types of resource leaks (synthetic leak servers) and for the real world code (DNS servers). The synthetic leak servers provided a simple, yet controllable experimental approach for the methodology validation. As for the DNS servers, their development and testing have evolved and stabilized throughout the years. They are small (few lines of code) and sustain continuous execution and testing, making them a challenging target.

4.1. AIP validation

The AIP methodology computes a model representing the resource usage until exhaustion, and therefore, is able to predict, within a small and acceptable error, future resource utilization. To verify and validate this hypothesis several synthetic programs were developed with distinct types of leaks. All programs were based on a simple TCP echo server – it has only one kind of interaction, where the client sends a "hello" message, and the server returns it back. Every time the server receives a message, it creates some sort of resource waste. Table 1 presents the fundamental characteristics of various synthetic leak servers. It shows the identifier of the server, the type of leak, the number of attack injections until terminating the experiment (5000) or until the machine stopped responding, and the kind of resource that was exhausted. Since the size of the leak could be decisive to the conclusions, in some cases variations of the same server were used. Two CPU leak servers were configured to execute additional instructions, a cumulative sum and a cumulative multiplication (B_1 and B_2); a server which created

Server ID	Leak type	No. of injections	Type of exhaustion
A	no leak	5,000	predefined no. of injections
B_1	CPU leak (add)	5,000	predefined no. of injections
B_2	CPU leak (mult)	5,000	predefined no. of injections
C	fork leak	4,888	memory exhaustion
D_1	pthread leak (stack size 16KB)	2,659	memory exhaustion
D_2	pthread leak (stack size 8MB)	383	memory exhaustion
E_1	memory leak (malloc 4B)	3,652,789	memory exhaustion
E_2	memory leak (malloc 30KB)	86,482	memory exhaustion
F	file open leak	1,019	open file limit
G	socket leak	1,019	open file limit
H	disk leak (write 30 KB)	13,042	disk exhaustion

Table 1. Synthetic leak servers with resource leaks.

another process per interaction (C); similarly, two servers which wasted a thread, with a stack of 16KB and 8MB, respectively (D_1 and D_2); two memory leak servers with 4B and 30KB (E_1 and E_2), a disk leak server with 30KB (H) ; and two servers that did not close a file or socket descriptor per interaction (F and G).

Minimum number of injections As stated in Section 2, the minimum number of repeated injections for each attack must meet two requirements: allow the calculation of linear regression projections, and guarantee the detection of any resource usage variation. Since we need potentially to estimate p = 2 parameters, three independent data points are required, i.e., at least three injections have to be done.

On the other hand, monitoring mechanisms can have distinct levels of granularity. For instance, in the current version of PREDATOR, all resources have a fine-grain type of monitoring with the exception of memory – the tool can only assess the number of memory pages assigned to the process. This affects the minimum number of injections because memory changes are only perceived when the process requests an additional page. The glibc `malloc()` implementation acquires at least 16 bytes on a 32-bit system (4 bytes for the preceding size field + 4 bytes for trailing size field + at least 8 bytes for the user block[1]). Therefore, no matter how many bytes a program requests, `malloc()` will allocate a block of at least 16 bytes. If a memory page is 4096 bytes long, in the worse case of a 1B leak, there must be 256 attack injections to force a new page request. One however should notice that other memory management implementations, which do not waste so much memory with control data, may require a larger number of injections.

Table 2 shows the linear regression projections for resources disk space (denoted by $\hat{y}_d$) and memory pages (denoted by $\hat{y}_m$) of two synthetic leak servers. The first of them had a 4B disk leak and the other wasted 4B of memory per interaction. The calculated models with 256 injections estimated parameters that reflected the increase in resource consumption (non-zero coefficients in the x variable), therefore, they identify the vulnerabilities. The table also demonstrates the impact of the granularity of monitoring. Since the mechanism measuring disk usage is capable of detecting variations of a single byte, three injections (n = 3) were enough to completely determine the model for the first server. However, for the memory case it was necessary to inject 256 times to reflect in the model the growing memory consumption, i.e., the 0.0001 coefficient.

[1]The minimum of 8 bytes for the user block is imposed because when the chunk is freed, the memory manager must store two free list pointers (double-linked list) in this space.

	Disk leak (4B)	memory leak (4B)
$n = 3$	$\hat{y}_d = \mathbf{4x}$	$\hat{y}_m = 114$
$n = 256$	$\hat{y}_d = \mathbf{4x}$	$\hat{y}_m = \mathbf{0.0001x} + 113.99$

Table 2. Projections for a disk and memory leak created from n injections.

Resource usage projection To validate the linear regression model, PREDATOR generated resource usage projections for the various synthetic leak servers, which were calculated with the measurements made on first 1024 attack injections. Then, the Injector continued to attack the server until it stopped, so that real data could be obtained about the behavior of the resources as the server became exhausted. Finally, the real data was compared with the predictions.

The resource usage projections with p = 2 estimated parameters are presented in Table 3. We tried to use polynomials with degree higher than 2, but in all cases the additional coefficients were always zero. As it is possible to observe, in most cases resource consumption was either constant (sometimes zero) or had a constant growth (x^2 coefficient was zero). Highlighted in bold are the coefficients that reflect the curved-shaped lines. These are the most dangerous vulnerabilities because the consumption increase is "accelerating".

Server	CPU M cycles	Processes	Memory pages	File descriptors	Disk bytes
A	$\hat{y} = 0.93x - 1.04$	$\hat{y} = 1.00$	$\hat{y} = 106.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
B_1	$\hat{y} = \mathbf{0.01x^2} + 0.35x - 22.89$	$\hat{y} = 1.00$	$\hat{y} = 91.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
B_2	$\hat{y} = \mathbf{0.41x^2} + 13.74x - 522.30$	$\hat{y} = 1.00$	$\hat{y} = 91.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
C	$\hat{y} = 0.22x - 14.05$	$\hat{y} = 1.00x + 1.00$	$\hat{y} = 69.99x + 114.62$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
D_1	$\hat{y} = 0.08x - 0.33$	$\hat{y} = 1.00x + 1.00$	$\hat{y} = \mathbf{2.04x^2} + 132.58x + 124.69$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
D_2	$\hat{y} = 0.12x + 0.03$	$\hat{y} = 1.06x - 2.01$	$\hat{y} = \mathbf{3.03x^2} + 134.90x + 137.42$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
E_1	$\hat{y} = 0.04x + 0.29$	$\hat{y} = 1.00$	$\hat{y} = 103.63$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
E_2	$\hat{y} = 0.04x + 0.32$	$\hat{y} = 1.00$	$\hat{y} = 1.00x + 104.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
F	$\hat{y} = 0.14x + 1.56$	$\hat{y} = 1.00$	$\hat{y} = 1.09x + 110.51$	$\hat{y} = 1.00x + 5.00$	$\hat{y} = 0.00$
G	$\hat{y} = 0.12x + 0.31$	$\hat{y} = 1.00$	$\hat{y} = 94.00$	$\hat{y} = 1.00x + 5.00$	$\hat{y} = 0.00$
H	$\hat{y} = 0.12x + 0.27$	$\hat{y} = 1.00$	$\hat{y} = 112.00$	$\hat{y} = 6.00$	$\hat{y} = 28672.00x$

Table 3. Resource usage projections for the synthetic leak servers (with p = 2).

Another point worth of notice is the implicit correlation between the resources, in particular the CPU and the memory. For instance, servers D_1 and D_2 have a thread leak, i.e., a new thread is created at each request (which is not terminated). The creation of a thread has an impact on three resources: the number of processes/threads, CPU, and memory. Therefore, there is a correlation among these resources for this sort of vulnerability (e.g., the correlation coefficient between the number of processes and memory pages is $R = 0.97$). This reveals the potential of the AIP methodology to better understand the dependencies and relationships between the different resources, which can contribute to the identification of specific vulnerabilities.

The results of the goodness-of-fit of the linear regression projections, for p = 1 and p = 2 parameters, are depicted in Table 4. The table shows two well know statistical measures for each of the major resources: the *adjusted coefficient of determination (R_a^2)* and the *mean squared error (MSE)*. Since we want to evaluate the goodness-of-fit of the projection for the entire data, and not only for the subset used in the linear regression, the coefficient of determination[2] had to be adjusted to the sample size. An R_a^2 of 1.0 indicates that the regression line perfectly fits the data (some values are missing because of the regular resource usage). However, since the entire data is much larger than the 1024 injections used in the regression, the residual standard deviation can sum up to produce negative R_a^2 values. The other measure, MSE, is the expected value of the square of the error. It measures the amount by which the estimator differs from the quantity to be estimated. When comparing two estimators for the same data, the one that gives a smaller MSE is better.

The table clearly shows that in all cases but three, the projections were extremely accurate (R_a^2 of approximately 1 and MSE for p = 1 and p = 2 in the same order of magnitude). The exceptions (highlighted in bold) are the two CPU projections for the CPU leak servers, and the memory projection for the thread leak server (D_1). In all three cases a straight line projection (p = 1) did not correctly represent the actual resource usage data. Figure 2, for instance, shows the memory usage projections and the actual consumption for the D_1 server. These results confirm our initial intuition that for certain types of resource consumption we may need curve-shaped projections. Overall, the experimental results shows that linear regression with p = 2 estimated parameters produces valid and accurate projections for the entire lifetime of the server, i.e., until the exhaustion of the resources.

[2]The coefficient of determination (R^2) is the proportion of variability (defined as the sum of squares) in a data set that is accounted for by a statistical model.

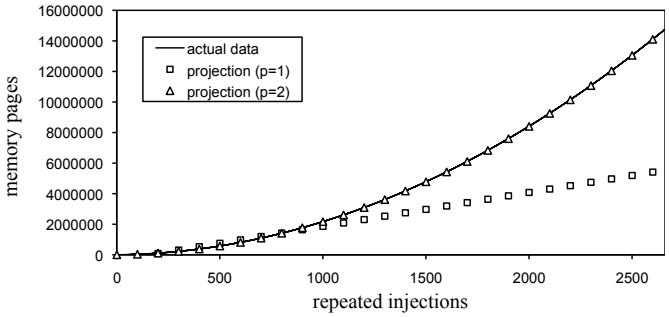

Figure 2. Memory usage projections for the D_1 synthetic server (with thread leak).

4.2. DNS experimental results

In order to test PREDATOR with real applications, we decided to focus on a single protocol to reduce the effort devoted to the production of protocol specifications (which is required for the attack generation), allowing more time to test different servers. The Domain Name System (DNS) is a network component that performs a crucial role in the Internet [14]. It is a hierarchical and distributed service that stores and associates information related to the Internet domain names. DNS employs a query/response stateless protocol. Messages have a large number of fields, which can take a reasonable range of possible values (e.g., 16-bit binary fields or null-delimited strings), and the erroneous combination of these fields can be utilized to perform attacks.

The experimental validation was conducted with seven known DNS servers: BIND 9.4.2 [10], MaraDNS

Server		CPU		Processes		Memory		Files		Disk	
		R_a^2	MSE	R_a^2	MSE	R_a^2	MSE	R_a^2	MSE	R_a^2	MSE
A	$p=1$	1.00	1.2×10^3	–	0.00	–	0.00	–	0.00	–	0.00
	$p=2$	1.00	1.1×10^3	–	0.00	–	0.00	–	0.00	–	0.00
B_1	$p=1$	**-0.23**	2.6×10^9	–	0.00	–	0.00	–	0.00	–	0.00
	$p=2$	0.99	1.5×10^7	–	0.00	–	0.00	–	0.00	–	0.00
B_2	$p=1$	**-0.27**	1.2×10^{13}	–	0.00	–	0.00	–	0.00	–	0.00
	$p=2$	1.00	2.0×10^8	–	0.00	–	0.00	–	0.00	–	0.00
C	$p=1$	0.99	45.9905	1.00	1.002	1.00	1.8×10^7	–	0.00	–	0.00
	$p=2$	0.96	2.6×10^2	1.00	1.003	1.00	1.8×10^7	–	0.00	–	0.00
D_1	$p=1$	0.98	94.582	1.00	1.001	**0.31**	1.3×10^{13}	–	0.00	–	0.00
	$p=2$	0.91	4.7×10^2	1.00	1.001	1.00	4.2×10^7	–	0.00	–	0.00
D_2	$p=1$	1.00	0.86	1.00	3.8×10^2	0.95	1.1×10^9	–	0.00	–	0.00
	$p=2$	0.99	2.1574	1.00	3.8×10^2	1.00	2.2×10^6	–	0.00	–	0.00
E_1	$p=1$	0.96	3.7×10^7	–	0.00	0.99	8.9×10^4	–	0.00	–	0.00
	$p=2$	0.98	2.2×10^7	–	0.00	0.99	8.9×10^4	–	0.00	–	0.00
E_2	$p=1$	0.99	8.8×10^3	–	0.00	1.00	9.5×10^3	–	0.00	–	0.00
	$p=2$	0.99	9.8×10^3	–	0.00	1.00	9.5×10^3	–	0.00	–	0.00
F	$p=1$	1.00	0.69	–	0.00	1.00	1.2611	1.00	1.002	–	0.00
	$p=2$	0.99	12.6607	–	0.00	1.00	1.2524	1.00	1.003	–	0.00
G	$p=1$	1.00	2.3726	–	0.00	–	0.00	1.00	1.002	–	0.00
	$p=2$	0.97	32.7006	–	0.00	–	0.00	1.00	1.003	–	0.00
H	$p=1$	1.00	1.1951	–	0.00	–	0.00	–	0.00	1.00	8.2×10^8
	$p=2$	1.00	37.0782	–	0.00	–	0.00	–	0.00	1.00	8.2×10^8

Table 4. R_a^2 and MSE for the resource usage projections for the synthetic leak servers.

1.2.12.05 [28], MyDNS 1.1.0 [15], NSD 3.0.6 [20], PowerDNS 2.9.21 [23], Posadis 0.60.6 [22], and rbldnsd 0.996a [27]. All these servers are highly customizable, with several options that could affect the monitoring data gathered during the experiments. To make our tests as reproducible as possible, we chose to run the servers with no (or minimal) changes to the default configuration.

PREDATOR generated a total of 19,104 different attacks from the DNS protocol specification, using a test case generation algorithm that created message variations with illegal data. The exploratory phase repeated the injection of each attack 256 times and selected the best attack for each resource (i.e., the attack that caused higher consumption) to be used in a second injection campaign. In the exploitive phase, each of the selected attacks was injected 1024 times.

The final resource usage projections (from the exploitive phase) are presented in Table 5. Four projections are highlighted in bold, the higher CPU resource projection (BIND), the higher processes resource projection (PowerDNS), and a couple of increasing memory resource projections (MaraDNS and PowerDNS). The CPU increase is expected because as more tasks are executed, more CPU cycles are spent. However, it is interesting to note that the most CPU intensive server happens to be also the most used DNS server in the Internet, BIND. This means that BIND is more susceptible to a CPU exhaustion, i.e., the CPU has no idle times, than the remaining target systems.

PowerDNS increases the total number of processes/threads from 7 to 8, which results in the highlighted processes projection. Further inspection, i.e., by running the exploitive phase with more injections, showed that the

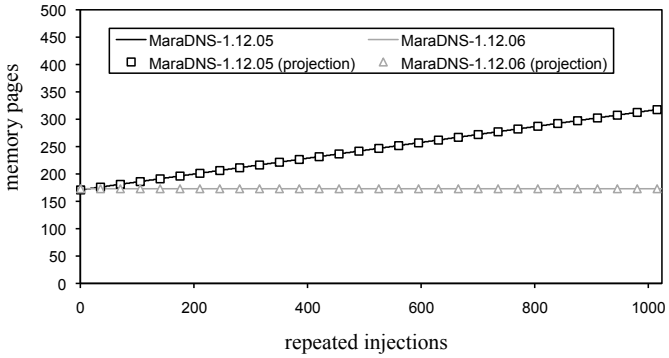

Figure 3. Memory consumption in MaraDNS.

PowerDNS server was in fact limited to 8 processes/threads. The observed raise in the memory consumption projection was also due to the same cause, since the OS allocates memory when starting a new process/thread on behalf of the same application. Therefore, no vulnerability actually existed in both cases as the resource consumption eventually stabilized. This example demonstrates the usefulness of the last phase of the AIP methodology – it allows the user to tradeoff some additional time executing extra injections on a small set of attacks, with a better accuracy of the projections. In some exceptional cases, such as to confirm the attacks that potentially could exploit a vulnerability, we deliberately increased the number of injections (> 1024) to take the server closer to the exhaustion point.

Another DNS server, MaraDNS, also showed a raising memory usage projection. In fact, the memory consumption of MaraDNS is a clear example of a memory leak vulnera-

Server	CPU M cycles	Processes	Memory pages	File descriptors	Disk bytes
BIND-9.4.2	$\hat{y} = \mathbf{0.39x + 25.31}$	$\hat{y} = 1.00$	$\hat{y} = 1251.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
MaraDNS-1.2.12.05	$\hat{y} = 0.18x + 4.85$	$\hat{y} = 1.00$	$\hat{y} = \mathbf{0.14x + 170.85}$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
MyDNS-1.1.0	$\hat{y} = 0.14x + 0.21$	$\hat{y} = 1.00$	$\hat{y} = 494.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
NSD-3.0.7	$\hat{y} = 0.02x + 0.78$	$\hat{y} = 3.00$	$\hat{y} = 534.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
PowerDNS-2.9.21	$\hat{y} = 0.19x - 19.61$	$\hat{y} = \mathbf{0.01x + 7.04}$	$\hat{y} = \mathbf{2.40x + 4983.49}$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
Posadis-0.60.6	$\hat{y} = 0.29x - 0.02$	$\hat{y} = 2.00$	$\hat{y} = 812.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$
rbldnsd-0.996a	$\hat{y} = 0.02x + 1.50$	$\hat{y} = 1.00$	$\hat{y} = 175.00$	$\hat{y} = 0.00$	$\hat{y} = 0.00$

Table 5. Resource usage projections for the DNS servers.

bility. But the memory exhaustion was not restricted to the selected attack. Several of the generated attacks caused the same abnormal behavior, which allowed us to identify the relevant message fields that triggered the vulnerability. Any attack requesting a reverse lookup, or a non-Internet class records, made memory usage grow. A closer look at the server's code path of execution, showed that the server stops processing these queries once they are detected because they are not currently supported. However, the respective parsing function fails to free a couple of previously allocated variables. Successively injecting any of these two kinds of attacks caused the server to constantly allocate more memory, eventually requesting a new memory page. Both resource-exhaustion vulnerabilities could be exploited remotely to halt the server. They were deemed by the MaraDNS developers as fairly serious, and were credited to PREDATOR [3]. Figure 3 compares the projections for memory consumption of the vulnerable (1.12.05) and corrected (1.12.06) versions of the server.

5. Related Work

This work has been influenced by several research areas. MESSALINE [4], Xception [7], or FTAPE [29], are examples of tools that can inject hardware or software faults in a target system under evaluation. By forcing and reproducing the occurrence of such irregular and unusual events, they can evaluate the target system's ability to cope with them. However, due to the relative simplicity of the mimicked faults, it is difficult to apply these tools to more complex faults, like security vulnerabilities of network servers.

Fuzzers deal with this intractability by injecting random samples as inputs to the tested software components. Originally, they were used against UNIX commands. For instance, Fuzz [13] generates large sequences of random characters which were used as parameters for command-line programs. Many programs failed to process the illegal arguments and crashed, indicating flaws like buffer overflow. Throughout the years, fuzzers have evolved into more intelligent, and less random, vulnerability detectors [31, 6, 24]). However, in some cases they have become too specialized, and they still lack more thorough monitoring mechanisms.

Robustness testing studies the exception handling effectiveness of the more complex software systems [11, 2]. Automated robustness testing tools probe OS APIs (e.g., operating system calls, device driver interfaces, or other software modules) to see how effective the target system is at handling the presence of erroneous input conditions.

There is also a field of study dedicated to the injection of *malicious* faults. Attack injection [18, 19] is used to generate and inject a large number of attacks in a target system while monitoring its behavior. The attacks can be directed at any interface of the target system, e.g., network device, file system, command-line parameters, and are based on the interface specification. AJECT attacks the network interface of target servers to discover vulnerabilities. It is composed of an injector and a monitor. The injector uses a specification of the network protocol (e.g., IMAP) to generate a broad spectrum of attacks. Then, it sends these messages to the target system and collects its replies. The generation is accomplished accordingly to a specific attack creation algorithm, which complies to some predefined test class, namely syntax test or value test. A monitor component closely observes the injection process and traces the server's execution (e.g., UNIX signals) and some basic resource usage (e.g., number of allocated memory pages and the time spent by the CPU). A vulnerability is detected upon the observation of an *unusual* server behavior, such as the reception of SIGSEGV signal. PREDATOR differs from AJECT in three ways: the considerable extension of the monitoring capabilities; the introduction of a new structured attack injection process, the AIP methodology, which required a complete rewrite with two injection campaign phases and a post-processing analysis; and finally, the capability to provide an estimate of the resource usage prediction.

Another area relevant to our research focuses on resource usage monitoring. Most of the work in this area is done mainly on performance monitoring and memory leaks. Supporting the former, various timing facilities are available in Linux systems, such as /proc/stat, getrusage, getpinfo, or even more portable solutions such as LibGTop [5]. However, given the small time granularity of many operating system activities [9] or due to processor fluctuations [33] the measured time may yield incorrect or inaccurate values. In order to improve the timing resolution, hardware performance counters present in modern processors can be utilized.

Besides the CPU time, memory is also an important resource. Memory leak detectors such as Valgrind [17] or

memprof [25] trace the memory allocation and de-allocation during the program execution. However, it is up to the developer to provide the different execution paths, or test cases, in order to attain a reasonable coverage.

6. Conclusions

The paper presents a new methodology for the detection of resource-exhaustion vulnerabilities. This methodology can be applied to any network server, as long as a specification of its protocol is provided. It was implemented in a fully automated black box testing tool called PREDATOR. The tool not only produces test cases and injects them in the server, but also computes resource usage profiles which predict the utilization of every monitored resource for all tests. The attacks that trigger vulnerabilities can be identified by its resource usage projections, showing an unexpected resource consumption.

The methodology was experimentally validated with synthetic servers, which showed that it is quite suitable to profile different kinds of resource leaks. As for real applications, we decided to focus on seven well-known public-domain DNS servers. Despite the fact that these servers have been extensively tested throughout the years, we still found new vulnerabilities, which we believe is an important demonstration of the added value of our tool and methodology.

References

[1] V. A. Abell. lsof – LiSt Open Files, 2007. http://people.freebsd.org/ abe/.

[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. Conf. on Dependable Systems and Networks*, June 2004.

[3] J. Antunes. MaraDNS multiple remote denial of service vulnerabilities. In *Bugtraq Mailing List*, 2007. http://www.securityfocus.com/bid/24337.

[4] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, 1989.

[5] M. Baulig and D. Kacar. LibGTop – library that provides system information, 2007. http://directory.fsf.org/libs/LibGTop.html.

[6] T. Biege. Radius fuzzer, Sept. 2005. http://www.suse.de/ thomas/index.html.

[7] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. In *Proc. of the Int. Working Conf. on Dependable Computing for Critical Applications*, Jan. 1995.

[8] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987 (Informational), Aug. 2007.

[9] S. Hines, B. Wyat, and J. M. Chang. Increasing timing resolution for processes and threads in linux. Unpublished.

[10] Internet Systems Consortium, Inc. BIND – Berkeley Internet Name Domain, 2007. http://www.isc.org/sw/bind.

[11] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. on Software Engineering*, 2000.

[12] K. London, S. Moore, P. Mucci, K. Seymour, and R. Luczak. The PAPI cross-platform interface to hardware performance counters. In *Department of Defense Users' Group Conference Proceedings*, June 2001.

[13] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 1990.

[14] P. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.

[15] D. Moore. MyDNS, 2006. http://mydns.bboy.net.

[16] M. Murphy. eServ memory leak enables denial of service attacks. In *Bugtraq Mailing List*, 2003. http://www.securityfocus.com/archive/1/321306.

[17] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of the Programming Language Design and Implementation*, June 2007.

[18] N. Neves, J. Antunes, M. Correia, P. Veríssimo, and R. Neves. Using attack injection to discover new vulnerabilities. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, June 2006.

[19] N. F. Neves. Locating file processing vulnerabilities. *Fast abstract in Supplement of the Int. Conf. on Dependable Systems and Networks*, June 2006.

[20] NLnet Labs. NSD – Name Server Daemon, 2007. http://www.nlnetlabs.nl/nsd.

[21] M. Pettersson. Linux performance-monitoring counters driver, 2002. http://www.csd.uu.se/ mikpe/linux/perfctr.

[22] Posadis. Posadis, 2004. http://posadis.sourceforge.net.

[23] PowerDNS. PowerDNS, 2007. http://www.powerdns.com.

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

[25] O. Taylor. MemProf: Profiling and leak detection, 1999-2007. htp://www.gnome.org/projects/memprof.

[26] Tenable Network Security. Nessus vulnerability scanner, 2006. http://www.nessus.org.

[27] M. Tokarev. rbldnsd, 2006. http://posadis.sourceforge.net.

[28] S. Trenholme. MaraDNS – a security-aware DNS server, 2007. http://www.maradns.org.

[29] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Int. Conf. on Modeling Techniques and Tools for Computer Performance Evaluation*, LNCS. Sept. 1995.

[30] D. Turner, S. Entwisle, M. Denesiuk, M. Fossi, J. Blackbird, D. McKinney, R. Bowes, N. Sullivan, P. Coogan, C. Wueest, O. Whitehouse, and Z. Ramzan. Symantec internet security threat report. Technical Report Volume XI, Symantec, Mar. 2007.

[31] University of Oulu. PROTOS – security testing of protocol implementations, 1999–2003. http://www.ee.oulu.fi/research/ouspg/protos/.

[32] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. on Dependable and Secure Computing*, 2005.

[33] A. Wiebalck, T. M. Steinbeck, and V. Lindenstruth. Fluctuating processors - recognizing and resolving cpu load in the kernel. *Linux Magazine*, June 2003.