SPAM
State Profiling and Analyzing Module
An approach to DoS attack detection

Petros Efstathopoulos and **Vassilis Pappas**
{pefstath, vpappas}@cs.ucla.edu

1. Introduction

Denial of service Attacks have become one of the most common ways to attack a system. Numerous recent incidents have proved that even carefully designed systems (e.g. large scale web servers) can be brought down by a DoS attack. This is mainly because today, there are no wide spread and well established mechanisms to detect DoS attacks and give the servers appropriate notice. Many ongoing research projects aim to deal with the DoS attack detection. The goal of this project was to develop a mechanism for detecting a certain category of DoS attacks.

One way to categorize DoS attacks is based on the kind of resources the attackers try to abuse. We can say that there are to kinds of attacks:

- ***Busy attacks***, which are targeted to renewable resources (like CPU cycles) and try to bring down the system by not allowing legitimate usage of these resources
- ***Claim-and-hold attacks***, which are targeted to non-renewable resources (like memory and disk space) and try to allocate and hold large portions of limited resources, thus bringing down the system due to resource (e.g. memory) starvation.

SPAM addresses the problem of Busy attacks. The method we used to attack the problem is based on real-time in-kernel process monitoring, used to gather data about a process' behavior, which are later on inspected by a user space application that is responsible for analyzing data and detecting possible DoS attacks.

2. SPAM Design

2.1 Assumptions

The design of SPAM is based on a set of assumptions about processes' structure. Most server processes can be seen as a repeated pattern o CPU intensive, computational work surrounded by two or more system calls. The general pattern that is repeated all over server processes is assumed to be the following:

- Execute initial system call (e.g. read data)
- Execute other possibly needed system calls (or none)

- Process request (CPU intensive)
- Execute other possibly needed system calls (or none)
- Execute final system call (e.g. write/send results)

A more formal way of describing this structure is by using a grammar:

S → aS
S → ε
a → dPd
d → Cd
d → ε

Where C is "system call" and P is "processing"

The equivalent regular expression is: $\mathbf{S = (C^{+}PC^{+})^{*}}$

Most of the popular internet services follow this scheme. Web servers receive (read) requests, read necessary local data (if needed), process the requests (CPU intensive part) and finally write results to the appropriate place (e.g. files, sockets etc). The same pattern can be also observed in the rest of the popular services (SMTP, DNS, POP, IMAP, FTP etc). This is the main reason that led us to believe that this assumption is reasonable and realistic.

Our second assumption is based on process model presented. We assume that Busy attacks exploit weaknesses of the computation code ("processing" part of the server code). During a busy attack, the attacker is trying to provide malicious input to the server, so as to consume almost all the CPU cycles. As we noted earlier, the exploitable CPU intensive part of request processing code is located between system calls.

The previous assumptions show that system calls can be seen as predefined checkpoints inside the code, since they almost always surround the vulnerable portions of the process' code. One can monitor the process' execution by tracing the system calls and measuring interesting properties like time intervals between specific system calls and the rate each of the system calls is requested.

2.2 Design Goals of SPAM

Based on the observation that system calls can be seen as predefined checkpoints that surround the vulnerable portions of a process' code, SPAM tries to monitor provide Busy attack detection by applying two-phase monitoring of the process. During phase 1, the process is monitored and data are gathered in order two build a state machine describing the process' execution under normal circumstances. Phase 2 is using the state machine built on phase 1 to detect suspicious behavior.

Phase 1 : Building a state machine for the server process

When monitoring the process, SPAM constructs a list of all the system calls that the process requests. A unique identifier is assigned to each of the system calls. This identifier is calculated from the value of the instruction pointer and thus signifies the position of the call inside the process. If a system call is inside a function (e.g. a *`read()`* call inside function *`getdata()`*) that is invoked from the main program (or maybe another function and so on), then the calculated identifier is unique for every different invocation of the function (*`getdata()`*) inside the process' code. This achieved by recursive inspection of the stack, in order to gather all the return addresses (instruction pointers) that would uniquely identify the system calls position in the code. A simple a example of this procedure is shown on Figure 1.

```
int main()
{
   while(!done)
   {
      getdata();
      read(fd, buf, 256);
      getdata();
   }
}

void getdata()
{
     read(fd, buf, 256);
}
```

Figure 1

In this example all three `read()` calls will have different IDs. For each invocation of the `getdata()` function, SPAM recursively traces the stack until it reaches `main()`. Since the two invocations have a different return address in main, the calculated IDs will be different. The same goes for the `read()` call inside `main()`, only this time calculating the unique ID is simpler since there are no nested calls.
We should note on each iteration of the loop, the same IDs are calculated for each of the three calls, since their position in the process' code doesn't change.

After assigning a unique ID to each monitored system call, we can build a state machine that describes the process' normal execution. Each uniquely identified system call will be a different state (node) of the state machine. An arc connecting two states (transition from one state two another) assumes that computation possibly took place between those two system calls. The value of the weight on the arc signifies the average/expected time it takes to move from one state to another. Also, the rate at which each state occurs is recorded. When the state machine is built, we have an overview of the process' execution, the average time it takes to process requests (based on the average time to move from one state to another) and the average rate at which at which each part of the process is executed.

Phase I1 : Use the state machine to monitor a process

After we have built the state machine for a specific process under normal execution, we can monitor the process and compare the data we gather with the expected behavior, as described by the state machine. When transitions from one state to another don't take longer than expected (or do not occur in greater rates than expected), SPAM assumes that the process is behaving properly and the likelihood of its being under attack is very little. If on the other hand, transitions between states occur with higher frequency than expected or take more time than the corresponding arc's weight (plus the standard deviation for that particular transition), SPAM assumes that the process is under attack. We may then take action to resist to the attack (e.g. re-nice the attacked server process).

It is clear that the implementation of SPAM should have two modes of operation: "training mode" (where the state machine is built) and "monitoring mode" (where the state machine is used to monitor the server application).

3. Implementation

The implementation of SPAM consists of a Linux kernel module and a user space application. The kernel module is responsible for tracing the monitored system calls and gathering the data that will be used to build the state machine. The user space application initially uses the data provided by the kernel module to build the state machine. After the training phase is over, the application compares the data gathered by the kernel module with the information from state machine and decides whether the system is under attack or not.

3.1 SPAM Linux kernel module

The kernel module's main responsibility is to monitor certain system calls and record them as "events". This is done by intercepting these system calls and replacing them with custom versions. For each system call new function is created that does the following things:

i) The system call is traced and uniquely identified (based on the value of the instruction pointer and all the other needed information retrieved from the stack)
ii) The event is recorded (system call type, ID, timestamp, duration of the call etc)
iii) The request is redirected to the original system call function which will do the actual serving of the call.

In order to provide an efficient way of communication with the user space application, the SPAM module also creates a virtual character device (`/dev/spam`). The user space application may control the module (issue commands, queries etc) using the *`ioctl`* call mechanism on this device. The application can also get the data gathered by the module by reading the device.

Let's consider an example: process 239 is a web server that needs to be monitored. The administrator will have to issue the `SPAM_MONITOR_PID ioctl` and provide 239 as argument. Then, when the web server issues a `read()` system call, the SPAM module will identify process 239 as one that's being monitored, trace the call and log the event. After that the original `read()` function will be called by the module to service the request. If the administrator wants to collect the data gathered by the module, he/she will have to open `/dev/spam` and read data from there.

The way processes are monitored by the SPAM module does impose significant overhead to the systems performance. The tracing operation can be seen as a number of memory references (tracing the stack) whose number depends on the number of nested calls inside the process. After tracing is complete, the SPAM module allocates a structure to log the event and issues the original system call. We could say that the overhead for each call is trivial.

3.2 User Space Application

The user space application reads the raw system call events either from the */dev/spam* device, or from the ptrace system call interface, and creates a system calls' state machine for each process. Each state is uniquely identified by the system call number and the concatenation of the return addresses of all the function that have been called starting from the main function and including the return address of the system call. For example if we consider the following piece of pseudocode with the hypothetical instruction code addresses at the left:

```
0       int main(){
1               read();
2               function1();
3               write();
4               function1();
5               write();
6       }
7       int function1(){
8               read();
9               while(){
10                      write();
11              }
12              return;
13      }
```

then the unique identifier of the read system call at line 1 will be: *#read+#1*, whereas the identifiers of the read system call at the function1 will be: *#read+#2+#8* and *#read+#4+#8*, for the first and the second call of this function respectively. We must mention that the + sign in the above expressions doesn't represent number addition, but a string concatenation. In order to create unique values of the same size a hash function is applied on each string (at the current implementation the hash function is just an xor of all the bytes of the string).

Given that each system call issued in a program can be uniquely identified, the process state machine is constructed by connecting with arcs the states of the system calls that are adjusted within the source code. Thus an arc of the process state machine represents a transition from one system call to another. In the case of the pseudo-code given before the corresponding state machine is shown in Figure 2.\

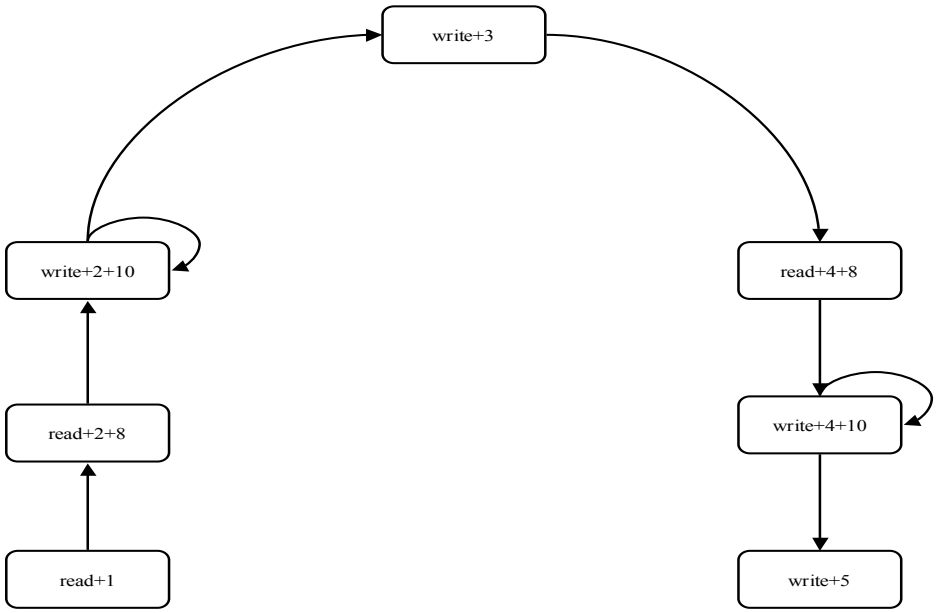

Figure 2

It is worthwhile to mention that that the system calls issued inside the function1 are shown up in the state machine twice given that this function is called in two different places in the main function. The state machine of each process represents all the possible execution paths that it may follow and it can uniquely identify a process. In order to construct the full state machine each program is run multiple times, with all the possible execution variations. Each time a new state machine is created and it may be different than the already constructed machines. The final state machine, that represents all the possible execution paths, is the union of the gathered state machines.

The user space program is also responsible for collecting the statistics that describe a normal execution behavior of the process. In more detail, the gathered statistics are: the delay time between two subsequent system calls, which represents the execution time of the code that resides between the system calls and it is assigned as a weight on each arch, and the access rate, for each node of the state machine, and it is assigned as a weight on the corresponding node. Because both of these parameters depend on the machine's load and the process priority we do not measure them by using real time. Instead we use a normalized time that is constructed by dividing the real time

with the time that the processor assigns for that process. We estimate the assigned time by computing the exponential average of the state's transition rate and deriving it with the following formula:

$T_a=1/ExpAvg(R_i)$, where R_i is the state transition rate, by the time that the i-th system call is issued, and it is computed as:

$R_i=1/(t_i-t_{i-1})$, where t_i is the time that the I-th system call is issued and t_{i-1} is the call time of he previous one.

At the training phase the user space application computes the average and the standard deviation for the states access rates and the states transition delays. It is worthwhile to mention that is important to measure both the rate and the delay because each of them can pinpoint deferent types of attacks. The state transition rate can identify the attacks that exploit a loop in the source code and force the attacked process to run a considerable number of loops, whereas the delay can identify attacks that exploit a part of code that resides between two system calls and it poses a heavy computational load under certain conditions. At the monitoring phase the application program computes the exponential average of the states access rates and the states transition delays, in order to capture deviations only for the recently measured values. If the difference between the average value computed at the training phase and the average value computed at the monitoring phase is larger than the standard deviation, then there is an indication of DoS attack. If the number of those indications, which appeared in the near past, exceeds a certain threshold then an action must be taken in order to respond to the DoS attack. One simple solution is to re-nice the affected process, while other more complicated ones can alter the execution path of the process, by returning error values for the subsequent system calls.

The user space program is invoked with the *./traced* command and takes the following three option: *-n* for training, *-c* for monitoring and *-m* for merging the statistics, gathered in a series of training phases, in one state machine file.

4. Related Work

Qie, Pang and Peterson[1] developed a toolkit , which is applied during the compilation time of the program, in order to provide programmer's support for building DoS attacks resilient software. Their method in concept is similar to our technique as it measures the access rate of certain checkpoints, which are inserted at the source code. The insertion of these checkpoints is a manual procedure and requires from the programmer to have good knowledge of the programs execution path and the programs expected behavior. Our approach does not modify at all the original code, it does not even require the availability of the source code.

A lot of work has been done in the area of the process anomaly and misuse detection [3,4]. The system described in [3] shares many similarities with our system given that they build a model that describes the normal execution of a process by monitoring system calls and they use a kernel module in order to do the tracing. On the

other hand their model is based only on monitoring valid sequences of system call traces, and it cannot identify anomalies due to DoS attacks. Our scheme not only uses a more general model, based on the processes state machine, but it also collects statistics related with the execution time of the particular parts of a program. Thus it is able to detect anomalies due to execution of malicious code or anomalies due to DoS attacks.

5. Conclusion

This project has shown that it is possible to build a system that is able to detect anomalies in the execution of a process that are due to a certain type of DoS attacks. We build a prototype system that consists of a Linux kernel module, used in order to record system call events, and a user space application, used for building the process state machine and gathering the statistics. Our future goal is to build a more stable version of the system and to make an extended evaluation by testing it over a variety of DoS attack cases.

6. References

References

[1] Xiaohu Qie, Ruoming Pang and Larry Peterson "*Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software*" OSDI, 2002.
[2] Daniel P. Bovet and Marco Cesati "*Understanding the Linux Kernel*".
[3] Anil Somayaji and Stephanie Forrest "Automated response using system call delays" 9th USENIX Security Symposium, 2000.
[4] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji and Tomas Longstaff "*A sense self for Unix processes*" IEEE Symposium on Security and Privacy, 1996.