An immune approach for anomaly detection from network traffic at the application level

Diego Souza Campos
Department of Computing and Statistics
Federal University of Mato Grosso do Sul
Campo Grande, MS, Brazil

Paulo Lício de Geus
Computing Institute
State University of Campinas
Campinas, SP, Brazil

Fabrício Sérgio de Paula
Ciência da Computação
State University of Mato Grosso do Sul
Dourados, MS, Brazil
Phone: +55 (67) 3411-9051
Fax: +55 (67) 3411-9004
E-mail: fabricio.paula@gmail.com

Abstract

Buffer overflow attacks are one of the most important attack classes because they enable an attacker to remotely execute arbitrary code at the target host. These attacks generaly disturb the network traffic at the application level by delivering anomalous data to the target applications. This work presents a prototype to detect anomalous network traffic containing executable code at the application level. This prototype is inspired by the negative selection process performed by T lymphocytes in the human immune system. A case study with the DNS protocol demonstrates that this can be a very promising approach.

Keywords: intrusion detection; anomaly detection; immune system; negative selection.

1 – Introduction

Computer systems and communication over the Internet are nowadays essential, aggregating a high value, both in the social and economic senses. In this way, it is necessary to secure these systems, in order to prevent malicious actions from disturbing the normal utilization of the global communication environment.

However, some computer security issues have been overlooked by software developers and consumers, making possible the arising of situations in which security vulnerabilities might be exploited by malicious individuals, with severe consequences. Considering this situation, several researches have been carried out in order to increase the security of computer systems, employing technologies like firewalls, cryptography, vulnerability analysis and intrusion detection.

Among several attack classes against computer systems, remote buffer overflow attacks are distinguished by their persistence and significance (Cowan et al., 2000), (Larochelle et al., 2001), (Northcutt et al., 2001). In these attacks, someone can send executable code to an application running on the target remote machine. The executable code will travel encapsulated into network packets at the application level, bypassing mechanisms like packet filters. In this way, the executable code can be viewed as anomaly data in the application level protocols, especially to the applications that work basically with text-based data (e.g., DNS resolution requests, FTP commands, SMTP, IRC and others).

This work presents a prototype to achieve anomaly detection in situations such as explained previously, analyzing the application level data captured from network traffic. To do this, we applied a technique inspired on the human immune system. As such, this work makes an attempt to contribute with solutions to improve the security of typical organizations.

This article is structured as follows. Section 2 presents a general view of computer immunology and works on computer security. Section 3 presents the technique to achieve anomaly detection. Section 4 shows a prototype that works at the application level data captured from the network. The tests and experimental results are shown in Section 5 and Section 6 makes conclusions about this work.

2 – The immune system and security

A good security level can be obtained by adopting models that are closer to the conditions of current computer networks, where there is a hostile, failure-prone environment. The human immune system is a good reference model, considering that it can protect the body against a large number of attacks, therefore several features that are desirable to a computer system.

In (de Paula, 2004) are discussed in detail several aspects of the human immune system, including its structural organization, operation of the immune response, the relationship between the immune system and computer security, and its principles that can be applied to security systems. This work, however, will focus only on this part of the biological system: the negative selection process performed by T lymphocytes (a kind of specialized immune cell).

2.1 – Negative selection process

The immune system defends the body against illnesses and infections. This defense is enabled by the capacity of recognizing the cells and foreign molecules and then of eliminating

these substances from the body. To do this, the immune system must perform tasks of pattern recognition to distinguish between the cells and substances of the body and the potentially malefic, foreign substances. Therefore, the detection problem of the immune system consists of discriminating *self* (body substances) from *nonself* (foreign substances) (Somayaji et al., 1997). This discrimination is performed by the immune system through a process called negative selection.

The negative selection process makes possible the attainment of a set of detectors that can perform the identification of nonself. In this process the detectors are submitted to two phases:

- Training or learning phase: new detectors are randomly generated and submitted to a large repertory of body cells. In this phase, if a detector matches some self substance, it is discarded and a new detector is generated. After this training phase, the detectors become mature and ready for detection;
- Detection phase: when several mature detectors match some substance in the body, there is a nonself identification and an immune response will be initiated.

In this way, the immune system decreases the risk of mature detectors matching self and thus makes possible the identification of foreign substances to which the system was not exposed before.

2.2 – Self/nonself discrimination in a computer system

The problem of protecting computer systems from intrusions can be viewed, much like in the immune system, as the self/nonself discrimination (Somayaji et al., 1997). In a computer system the self definition can be made by observing memory utilization in a host, collective network behavior, network traffic in a router, ingoing and outgoing TCP/IP packets of a specific host, sequence of instructions and even some user behavior. Also the self definition must be tolerant to legitimate changes (Somayaji et al., 1997), including file editing, installation of new applications, addition of new users, changes on the user behavior, and diverse system administration activities.

By applying this ideia, Forrest, Allen, Perelson e Cherukuri presented an algorithm for the probabilistic detection of modifications on protected data (Forrest et al., 1994). This algorithm, which is based on the negative selection, works as follows:

- Generation of detectors: the detectors are strings with the same length. These strings are randomly generated, and each one will later be selected as a mature detector if it does not match the protected data while the training phase takes place;
- Information monitoring: the protected data is compared with the detectors. The activation (matching) of a detector signals that an alteration was made on the analyzed data.

In this algorithm, each string which constitutes a detector is defined as a sequence of bits. The detection has effect through the matching of *r*-contiguous bits between the analyzed data and the detectors, for some predefined value of *r*. Therefore, the algorithm searches for *r* contiguous positions when the bits are the same, in a pair of strings. This algorithm was applied to search for file modification through code insertion, byte mutation and computer virus contamination. Experimental results showed efficiency and further research was carried out as a consequence (D'haeseleer et al., 1996).

The negative selection process and a variation of the algorithm described in this section constitutes the base of the prototype which will be presented in Section 4.

3 – Buffer overflow attacks

Buffer overflows are one of the most "famous" methods used by attackers in order to obtain high privileges and non-authorized access in a computer system. This method consists in supplying a large volume of data to an application, trespassing the predefined limits of data arrays. In this way, it is possible to inject arbitrary executable code, altering the legitimate line of execution of the attacked process.

The most common technique in buffer overflow attacks is based on rewriting, at the process stack, the return address of a procedure. In this case, the attacker sends data containing two parts: a malicious executable code and a new return address, which will point to this code. Consequently, when the actual procedure in execution ends the inserted code is executed and the intrusion takes place. A large number of notifications emmited by the CERT/CC (*Computer Emergency Response Team/Coordination Center*) are related to this attack category.

Technically, a buffer overflow is possible due to problems with programs which do not validate the input data. This security flaw happens mainly in programs implemented with a programming language that does not provide automatic limit checking, such as the C language, where basic functions like *scanf()*, *gets()*, *strcpy()*, among others, are frequently used. The exploitation of these flaws can lead to severe damages, for example, such as the first large

security incident of the Internet – the *Internet Worm*, in 1988 – which, among other flaws, exploited the program *fingerd* with this technique.

The main methods applied to detect buffer overflow attacks are presented as follows:

- StackGuard (Cowan et al., 2000) are a set of compiler extensions which performs additional checkings in the return address stored in the stack. In this way, StackGuard can prevent only buffer overflow attacks which manipulate return addresses, but not other pointers. Point-Guard (Cowan et al., 2000) is a generalization of StackGuard in such a way to protect other code pointers. In (Larochelle et al., 2001) are presented some strategies to report flaws in the use and manipulation of arrays through the analysis of the program's source code. The main drawback of these approaches is that they make it necessary to recompile all applications to be used. Perhaps by this reason they are not so largely used;
- In (Xu et al., 2002) and (Suh et al., 2002) are presented hardware mechanisms to prevent attacks. The strategy employed by (Xu et al., 2002) works with a copy of the return addresses in order to discard rewritten adressses and as such is not a general solution to the problem. In (Suh et al., 2002) a mechanism is responsible for detecting unexpected deviations in the control flow of a program or the insertion of executable code. Therefore, this method cannot detect attacks which change only adjacent data near the exploited buffer;
- In (Toth and Kruegel, 2002) is shown a very promising strategy, used to detect executable code in the network traffic. It is presumed that packets can carry some data sequences that can be viewd as a portion of executable code. When a sufficient large portion of "code" is identified, a possible attack is detected. Initial tests showed that this strategy can be free of false-positives. However, this technique is very dependent of the architecture which will be the target of the attack;
- (Kruegel et al., 2002) describes a detection method based on the behavior of the network traffic, considering each service in a separate way. Although this idea was not developed specifically for buffer overflow attacks it can be used to detect them, by analyzing the size of requests to network services. However, this choice suffers from the false-positive problem because the use of statistical measures slowly accommodates the changes in the behavior of new accesses;

Some other approaches were developed in search for solutions to the specific buffer overflow problem and related attacks. These approaches exploit techniques that range from

compilation to addition of dedicated hardware. However, no definitive solution was encountered yet.

4 – Prototype for anomaly detection based on the immune system

This section presents a prototype of an intrusion detection system (IDS) inspired by the negative selection process of the immune system. In this way, the prototype performs anomaly detection and can detect buffer overflows, code injection and other attacks that remotely exploit programs which do not correctly validate the input data. The prototype was developed to inspect the network traffic, at the application level protocols. It is important to emphasize that some applications can send and receive diverse data and code (e.g., FTP data connection), but in several cases the applications work with text-based commands (e.g., *queries* DNS and FTP control connection) and the attackers commonly use these channels to execute arbitrary codes and commands.

The detection system was developed using the C language with the Debian GNU/Linux 3.1 operating system, kernel version 2.4.27 and the GCC compiler version 3.3.3.-7. The hardware used during the implementation and the main tests was an AMD Athlon XP 2600+, 1.91 GHz, 256MB DDR RAM, 40GB Ultra-DMA IDE 7200 RPM hard-disk. The network traffic was handled through the use of the Libpcap library version 0.8.3-3.

The implementation was done considering the DNS protocol as a case study due to its large utilization and frequency that this protocol is exploited in application attacks through buffer overflows.

4.1 – Libpcap library

The Libpcap a system-independent interface for packet capture at the user space. It allows for efficient capture through the use of BPF (Berkeley Packet Filter) filtering. This library was developed by Van Jacobson, Craig Leres and Steven McCanne, at the Lawrence Berkeley National Laboratory, University of California, Berkeley. The Libpcap is applied in several well-known projects, such as TCPDump, Snort, Arpwatch, Fragrouter, among others. This simple, yet powerful library is very suitable to work with the TCP/IP stack protocol, enabling a very flexible and easy handling of network traffic at the application end.

The library provides an API with diverse functions and predefined types to enable the capture, processing, storing and recovering of packets. The capture can be done from any network interface (online) or even from a file (offline) in a standard format.

4.2 – Prototype structure

The strategy employed in the prototype, the negative selection, is presented in Figure 1. The section above the separation line shows the detector generation phase, while the section below the line shows the network traffic monitoring phase.

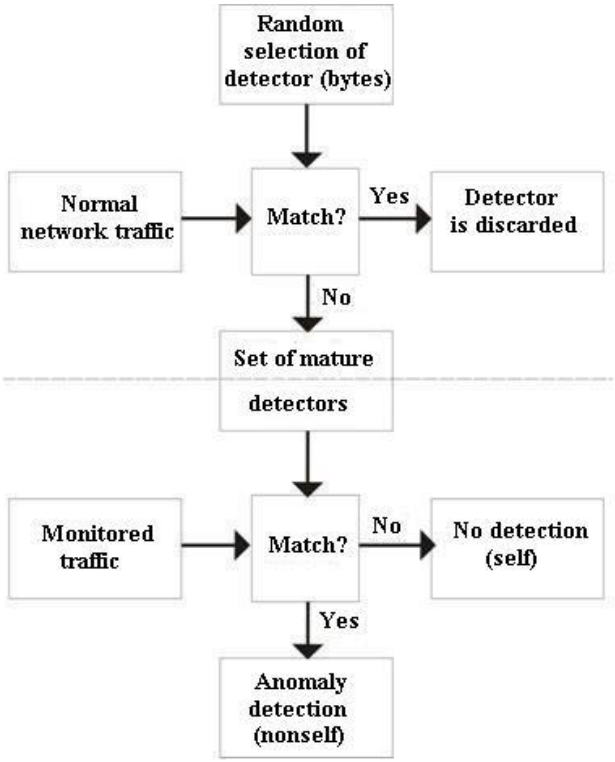

Figure 1. Negative selection employed in the prototype

The detectors are created through the random selection of bytes from the /bin/ls program file and, therefore, each detector can hold only one ASCII character. While the training phase happens, these detectors are compared with normal network traffic at the application level protocol (free from attacks) in the system. If a detector matches any normal network traffic (i.e., the detector byte is encountered in the traffic), then this detector is discarded and a new detector is created. At the end of this process, the set of mature detectors holds only bytes that are improbable to appear under normal traffic conditions. The set of mature detectors is used to analyze the network traffic at the monitoring phase. When diverse detectors consecutively match a portion of network traffic, it is identified as an anomaly. These phases are detailed in the next sections.

4.3 – Training phase

A set containing training data is necessary to begin the training phase. This training data is composed by a file which contains network traffic free from attacks that is related with the application protocol under analysis. This set comprehends the self data, used to discard detectors which can produce false-positives. Therefore, this must be a very representative set of the normal traffic. Because the attacks are launched from the attacker to the target host, this set contains only packets received by the target host.

After randomly selecting the set of ASCII detectors from the /bin/ls file they are trained through negative selection. Each detector has associated a value which comprehends its *training level*. The training level for new detectors is zero. At the begin of the training, each detector is compared with each byte of the training data packets. If a detector is not encountered in a packet, then its training level increments by one, otherwise it is discarded and replaced by a new detector. The training ends when all detectors reach the *good training level*, which is defined by a constant threshold value. Figure 2 illustrates this training phase.

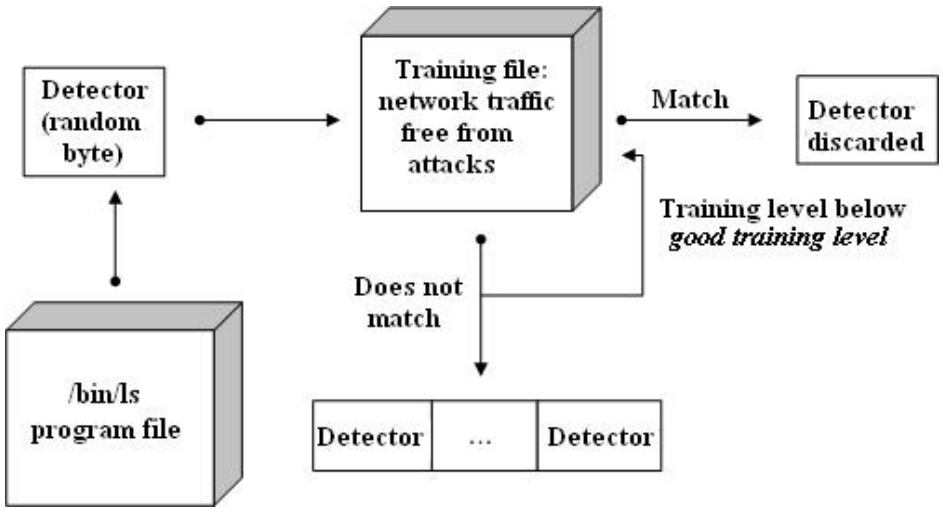

Figure 2. Creation of detectors and the training phase

4.4 – Monitoring phase

Once the detectors are trained, the monitoring phase can be performed by capturing network traffic from a network interface in real time or from a file containing traffic previously captured in the TCPDump format.

As with the training phase, each TCP/IP packet is gathered through Libpcap and decoded up to the application layer. The application layer data are temporarily stored in an array of bytes to be compared with the mature detectors. After this, the content of this array is sequentially analyzed with the detectors.

When the amount of contiguous bytes of this array which matches the detectors reaches a threshold, called *detection level*, an anomaly is identified. Figure 3 illustrates how the monitoring phase works.

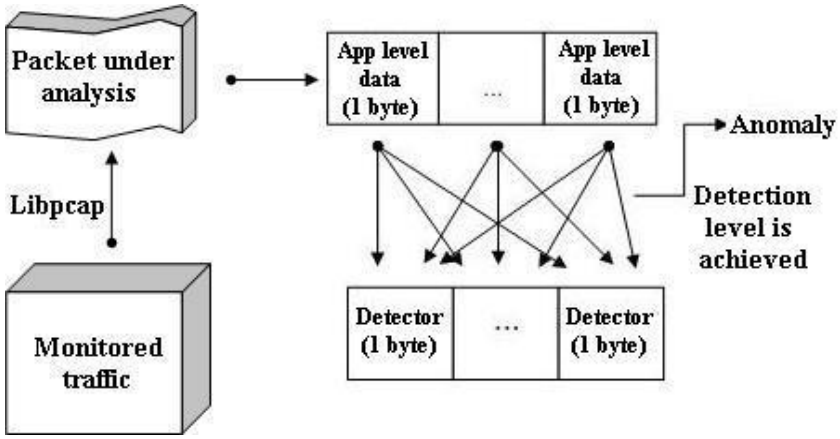

Figure 3. Monitoring phase and anomaly detection

5 – Tests and experimental results

During the tests, three fundamental variables were considered for the process of training and monitoring:

- The number of detectors: amount of detectors used in both training and detection;
- Good training level: amount of packets which are used for training each detector. Mature detectors must reach the good training level, or else be discarded and replaced;
- Detection level: number of contiguous bytes of an incoming packet which must be reached for an anomaly to be identified.

The tests were performed with a dataset collected in the Administration and Security Laboratory at the Computing Institute of the State University of Campinas. This dataset contains DNS queries captured during 43 days, reaching 26,2 MB of data. This set is free from attacks, which was verified through the *Snort* (Roesch, 2006) tool. Disjunct sets of queries were used for the training and monitoring phases. It was used a network traffic captured during an

intrusion attempt against the *named* daemon (DNS server), obtained through execution of a buffer overflow exploit. The results shown here were obtained as an average of 20 trials.

Table 1 presents the results of anomaly detection by varying the number of detectors. The good training level and the detection level assume fixed values of 100 and 5, respectively. Figure 4 illustrates the mains aspects of Table 1, representing the number of detectors versus false positives and false negatives (in percentage) for an easier understanding.

Number of detectors	False positives	False negatives	Replaced detectors	Training time	Detection time
10	0%	85%	7	0.005s	0.900s
20	0%	50%	14	0.007s	1.330s
30	0%	35%	18	0.008s	1.680s
40	0%	35%	29	0.005s	2.020s
50	0%	25%	36	0.007s	2.380s
60	0%	25%	44	0.006s	2.740s
70	0%	10%	51	0.008s	3.070s
80	0%	5%	60	0.010s	3.420s
90	0%	0%	66	0.012s	3.780s
100	0%	0%	73	0.010s	4.130s

Table 1. Training and detection results varying the number of detectors

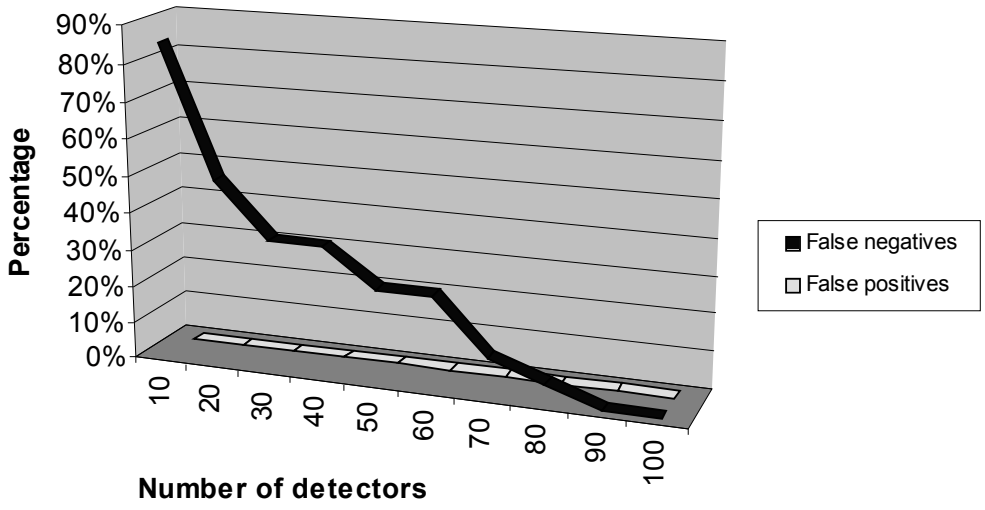

Figure 4. Number of detectors versus false positives and false negatives

By adopting only 10 detectors there is a high level of false negatives (85%), i.e., attacks that were not were identified. This happens because there are few detectors when compared with the detection level (5). Then, it is necessary that 5 contiguous bytes of the attack packet be some of the only 10 mature detectors. With 30 detectors the number of false negatives shows a

steep fall, reaching 35%, however still far from an acceptable percentage. With 90 detectors the anomaly detection was very precise, without false positives or false negatives.

As can be noted, there is no false positives, due to the good training level adopted. Although there are few detectors, they are well trained (this issue is discussed after the next table) and as such do not match the legitimate packets. The number of replaced detectors during the training phase is greater when the number of detectors is increased, since there are more detectors which can match the self packets. The training time has a slight variation possibly because the replaced detectors will initiate their training later, delaying the training process. The detection time is also greater when the number of detectors is increased because there are more detectors to be compared with the self packets.

Table 2 presents the results by varying the good training level. The number of detectors and the detection level assume fixed values of 100 and 5, respectively. Figure 5 illustrates the good training level versus false positive and false negative levels (in percentage).

Good training level	False positives	False negatives	Replaced detectors	Training time	Detection time
1	0,51%	10%	22	0.014s	3.802s
5	0,20%	10%	30	0.011s	3.960s
10	0,19%	10%	35	0.014s	4.015s
15	0,1%	10%	45	0.016s	4.012s
20	0,06%	5%	51	0.011s	4.080s
25	0,001%	0%	64	0.011s	4.179s
30	0%	0%	68	0.016s	4.173s
35	0%	0%	69	0.013s	4.185s
50	0%	0%	69	0.017s	4.183s
100	0%	0%	76	0.018s	4.212s

Table 2. Training and detection results varying the good training level

A little percentage of false positives is identified when the good training level varies from 1 to 25 packets. However, this level cannot be simply ignored, due to the large number of packets analyzed during the anomaly detection. The problem happens due to the new detectors being compared with an insufficient amount of normal (self) packets during the training phase. When the good training level reaches 30 the false positive problem is eliminated.

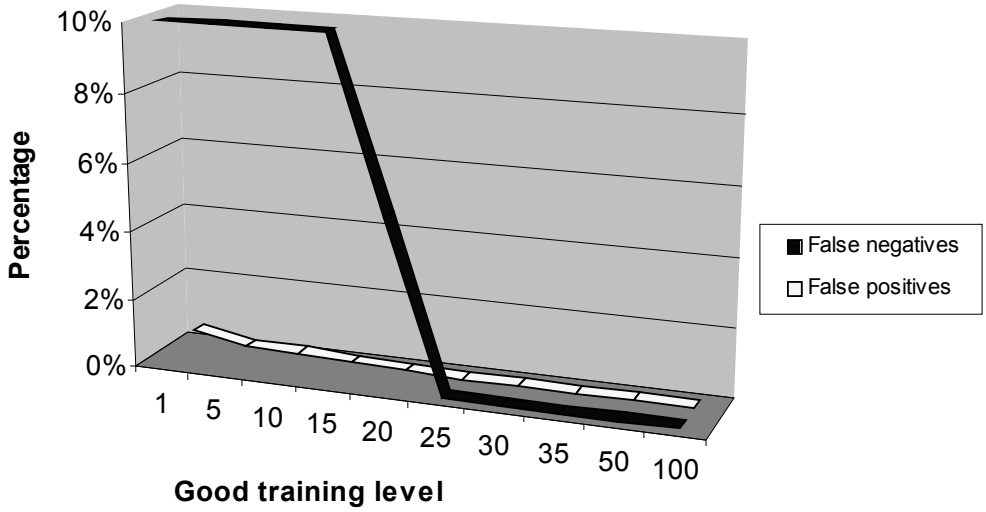

Figure 5. Good training level versus false positives and false negatives

False negatives happen at a larger rate, following a constant factor up to the good training level reaching 15. This also happens because the detectors were exposed to a low amount of self data during training phase, enabling the subsequent misdetection of self instead of detecting nonself. When the good training level reaches 25 packets, the false negative level falls to zero. The number of replaced detectors during training phase is greater when the good training level is increased, since it is more likely for a detector to match self packets when the number of packets is larger. The training time remains basically constant, because the number of detectors are all the same. The detection time also does not change significantly by the same reason. In this work, the good training level is by default 100, using a very low number of self packets at the training phase.

Table 3 presents the results by varying the detection level. The good training level and the number of detectors assume both a fixed value of 100. Figure 6 illustrates the detection level versus false positive and false negative levels (in percentage).

A low, but unacceptable value (due to the number of packets analyzed) level of false positives was identified when the detection level was set to 1. In this case, the anomaly happens if only one byte of the packet matches any detector, which is very likely if the number of detectors is 100. When the detection level reaches 2 and 3 the number of false positives is low. When the detection level reaches 4, the false positive level reaches zero.

Detection level	False positives	False negatives	Replaced detectors	Training time	Detection time
1	7,40%	0%	77	0.012s	4.165s
2	0,07%	0%	75	0.013s	4.166s

3	0,008%	0%	76	0.013s	4.175s
4	0%	0%	74	0.012s	4.177s
5	0%	0%	71	0.013s	4.178s
6	0%	5%	79	0.015s	4.177s
7	0%	10%	74	0.009s	4.178s
8	0%	25%	74	0.012s	4.178s
9	0%	100%	77	0.015s	4.176s
10	0%	100%	75	0.013s	4.178s

Table 3. Training and detection results varying detection level

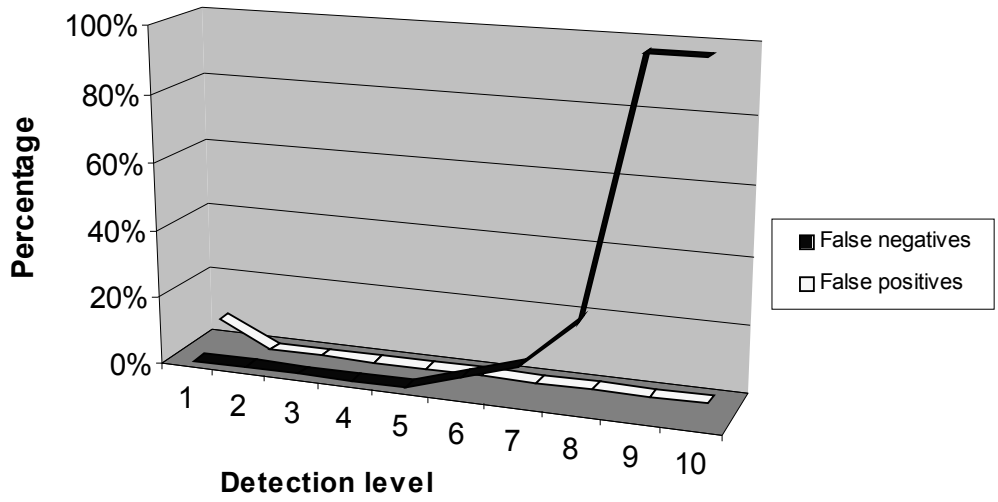

Figure 6. Detection level versus false positives and false negatives

On the other hand, when the detection level reaches 6 the false negatives begin to appear (5%). When the detection level reaches 9 no attack is identified, with 100% of false negatives. This fact is due to the rigorous rule imposed to the anomaly detection: at least 9 contiguous bytes must match the mature detectors, making the detection strongly specific and very improbable for general attacks. As can be noted, a little change in the detection level leads to drastic changes in the false positive and false negative levels. In this way, this parameter must be chosen very carefully to obtain the desired objective. Only at the levels 4 and 5 there is no false positives nor false negatives, achieving a good anomaly detection. As such, in this work the detection level was chosen to be 5.

The number of replaced detectors and the training time do not have any relationship with the detection level, because they act in different phases. The detection time remains constant because the detection level assumes very close values during the tests.

5.1 – Improving the detection

The detection performance can be improved by adopting a simple hash table. The length of this table can be 256 due to the fact that mature detectors can assume only values between 0 and 255. In this way, each byte of the network traffic is used to directly access the hash table. If the accessed position has the value 1 then the byte under analysis matches a detector. The training phase remains the same.

The performance tests were done with a new computer system because the previous system was unavailable: an Ubuntu 6.10 operating system, kernel version 2.6.17 and the same previous version of GCC compiler and Libpcap library. The hardware used during the tests was an Intel Core Duo 1.73GHz, 2GB DDR2 SDRAM, 120GB SATA 5400 RPM hard-disk.

Our previous tests showed that the detection time remains almost always constant. This time will be changed when the number of detector is changed. By considering this, Table 4 presents detection times varying the number of detectors, when no hash table is used and when the hashing is applied. Figure 7 illustrates these results. As can be noted, the detection using a hash table is faster then without the hashing. The hashing remains the detection time constant (about 0.4 seconds) while the detection without a hash table tends to linearly increase with the increasing number of detectors. In this way, the hashing is a good solution to achieve performance during the detection phase.

Number of detectors	Detection time without hash table	Detection with hash table
10	0.762s	0.438s
20	1.134s	0.438s
30	1.379s	0.433s
40	1.693s	0.462s
50	2.007s	0.439s
60	2.319s	0.437s
70	2.736s	0.435s
80	3.050s	0.440s
90	3.353s	0.436s
100	3.672s	0.438s

Table 4. Detection times without and with the use of a hash table

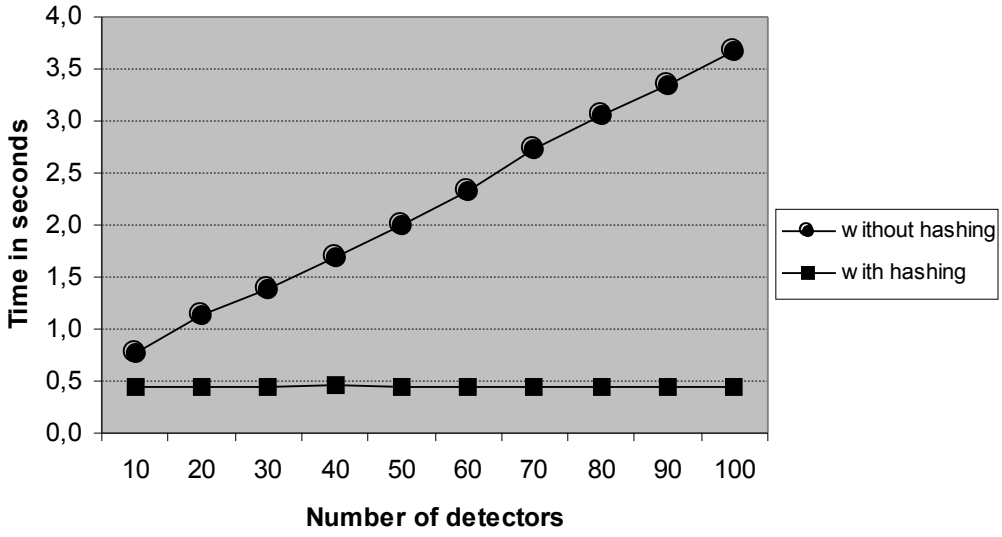

Figure 7. Detection level versus false positives and false negatives

6 – Conclusion

We have presented a prototype inspired by the negative selection of the human immune system to achieve anomaly detection. The detectors are very simple (only isolated bytes) that do not match the normal network traffic. However, experimental results showed that this can be a very promising method to detect attacks that disturbs the normal behavior of text-based application protocols.

Neither dedicated hardware nor recompilation techniques are required by the prototype unlike other methods such as described in Section 3. By working at the packet level, it is also possible not only to detect intrusions, but also to stop attacks before some flaw can be exploited. Although other works were carried out as a consequence of (D'haeseleer et al., 1996) these researches does not examine the network traffic at the application level, but only network and transport level protocols which can be free from attacks (at these levels) during an application level attack.

The main prototype's variables are discussed and after the tests we suggest some values to these variables in such way that the detection can be practical. We also have presented an improvement in the detection phase in order to achieve an acceptable performance during the detection phase.

Future work comprehends an expansion of the prototype to support the test with other application protocols, like HTTP, FTP (control connection), IMAP, POP and others. In this case, each application protocol can have its own set of detectors that will be very specific for

that protocol. It is also planned to study other representation of detectors and new algorithms for both training and detection phases, including methods considering partial matching of detectors.

7 – References

COWAN, S., WAGLE, P., PU, C., BEATTIE, S. and WALPOLE, J. 2000. Buffer overflows: Attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition.

DE PAULA, F. 2004. *Uma arquitetura de segurança computacional inspirada no sistema imunológico*. Doctoral Thesis, Instituto de Computação – Universidade Estadual de Campinas – UNICAMP.

D'HAESELEER, P., FORREST, S. and HELMAN, P. 1996. An immunological approach to change detection: algorithms, analysis and implications. IEEE Symposium on Computer Security and Privacy.

FORREST, S., ALLEN, L., PERELSON, A. and CHERUKURI, R. 1994. Self-nonself discrimination in a computer, IEEE Symposium on Security and Privacy.

KRUEGEL, C., TOTH, T. and KIRDA, E. 2002. A Service specific anomaly detection for network intrusion detection. ACM Symposium on Applied Computing.

LAROCHELLE, D. and EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. USENIX Security Symposium.

NORTHCUTT, S., COOPER, M., FEARNOW, M. and FREDERICK, K. 2001. Intrusion Signatures and Analysis. New Riders Publishing.

ROESCH, M. Snort. Available at http://www.snort.org Accessed on 01/06/2006.

SOMAYAJI, A., HOFMEYR, S. and FORREST, S. 1997. Internetworking Principles of a computer immune system. New Security Paradigms Workshop.

SUH, E., LEE, J. and DEVADAS, S. 2002. Secure program execution via dynamic information flow tracking. Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

TOTH, T. and KRUEGEL, C. 2002. Accurate buffer overflow detection via abstract payload execution. Fifth International Symposium on Recent Advances in Intrusion Detection.

XU, J., KALBARCZYK, Z., PATEL, S. and IYER, R. 2002. Architecture support for defending against buffer overflow attacks. Second Workshop on Evaluating and Architecting System Dependability.