AngeL: a tool to disarm computer systems

Danilo Bruschi, Emilia Rosti
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico 39, 20135 Milano – Italy
E-mail: bruschi, rose@dsi.unimi.it

ABSTRACT
In this paper we present a tool designed to intercept attacks at the host where they are launched so as to block them before they reach their targets. The tool works both for attacks targeted on the local host and on hosts connected to the network. In the current implementation it can detect and block more than 70 attacks as reported in the literature.

The tool is based on the idea of improving the overall security of the Internet by connecting disarmed systems, i.e., hosts that cannot launch attacks against other hosts. Such a strategy was presented in [4]. Here we present an extended version of the tool that has been engineered to consider a wide variety of attacks and to run on various releases of the Linux kernel and the experience learned in building such a tool. A protection mechanism of the tool itself that prevents its removal is also implemented. Experimental results of the impact of the tool on system performance show that the overhead introduced by the tool is negligible from the user's perspective, thus it is not expected to be a hindrance to the successful deployment of the tool.

Keywords
Computer and network security, defense, offense, disarm, attack, monitor

1. INTRODUCTION
In [4] a new approach to computer security, and system protection in particular, was proposed based on the following remarks. In a networked environment, any host can suddenly, possibly involuntarily, become an attacker, that is, a threat for the entire community (usually as a consequence of a compromise it was the victim of). This is particularly true of unattended hosts connected on the network, such as those of non-professional users. Intruders' life would be more difficult if they could not exploit such hosts to attack their targets. Furthermore, there are attacks, such as IP spoofing, that can be more easily and more successfully blocked at the source than at their target, even if sophisticated heuristics are adopted. By defining an adequate characterization of such attacks in terms of their signatures, as in signature based intrusion detection, it could be possible to detect and block them before they leave their origination point.

Moving from the above observations, the authors suggested an alternative approach to system security that builds on "harmless components". Reducing the threat of virtually any network host turning into a source of attack should be a parallel thread to the classical protection oriented one. Recent DDoS attacks have shown that the mere size of an attack, i.e., the number of attacking hosts, is a critical factor in computer security incidents, possibly even more than the "quality" of the attack itself. In a network where no, or just a few, hosts are a threat, global security results from individual harmlessness. Preventing systems from doing any harm, i.e., disarming the systems by turning off their offending capabilities, is a way to improve security. Offending capabilities should be turned off both at host level, so as to prevent local exploitation of the host, i.e., compromising the host, and at network level, so as to prevent an offensive use of the host against other machines. Since it reverses the perspective of intrusion detection, the authors' approach is suggested to be called "ex-trusion" detection and response, as it aims at detecting and acting against outgoing attacks rather than incoming ones [13].

A tool that intercepts all network packets and drops those that it recognizes as typical of a set of attacks it "knows", as they are generated on the machine where the tool is installed, would limit the offensive capabilities of the networked computer running such a tool. Attackers that were to seize that host and use it as a base for their attacks would be significantly limited. Furthermore, such an approach would be interesting for organizations such as large universities or corporations that would not like to see their reputation damaged by being the source of an attack. Moving from protection of one's reputation to liability, from a legal point of view, a tool that disarms a computer could protect the owner of that computer from liability in case the machine were subverted and attacks successfully launched from it[1]. In fact, the system expected to be running such

[1]This could be particularly interesting in countries, such as Italy, where the law holds the owner of a computer liable for whatever action is taken from that computer, regardless of it being compromised or not, unless the owner can show that all "reasonable preventive measures" were taken to protect

a tool is the (personal) computer of non-professional users, with little if any maintenance, especially for what concerns security, interested in a transparent solution that allow to connect a safe machine on the network. Potential limitations of some system functionalities deriving from the adoption of the tool, i.e., the impossibility to run some network tests, are expected to have a negligible impact on the average user who is either not able or not allowed to take any advantage of them.

A prototype of the tool that was able to detect a small number of network attacks, based on attack signatures, was implemented to verify the viability of the proposed approach [3]. Since the prototype implementation was successful, we describe here the full scale version of the tool, which has been extended to include also local attacks, not considered in [3]. A more extensive and sophisticated knowledge base of signature attacks, including local ones, is bundled in the tool that is able to identify more than 70 attacks, both at network level and at host level. A mechanism to harden the tool protection itself so as to make its removal more difficult has been added. A preliminary set of experimental results of the impact of the tool on system performance show that the overhead introduced by the tool is negligible from the user's perspective. Therefore, the issue of performance is not expected to be a hindrance to the successful deployment of the tool. Other factors that may hinder the success of the tool, such as the economic one both as for the deployment, maintenance and update of such a tool, or the technical one, such as the impossibility of using mobile IP or the difficulty to tell a harmful behavior from a legitimate one, thus taking punitive measures against possibly innocent users, were investigated in [4].

The current implementation of the tool, called AngeL, is based on the most recent version of the Linux kernel (hence the capital L in the name). It is implemented as a loadable kernel module comprising two distinct modules. The host based module is the new one and handles local attacks, i.e., attacks performed by an authorized account against the host where the tool has been installed in order to gain higher privileges. The network based module is a refined version of the one present in the prototype and handles attacks aimed at other hosts on the network.

This paper is organized as follows. Section 2 discusses related work. In Section 3 we describe the module handling attacks targeted on the local host and in Section 4 the module handling attacks targeted on networked hosts. Techniques adopted to make the tool "tamper-resistant" are discussed in Section 5. Preliminary experimental results on the use of this module by a small community of user are reported in Section 6. The are very encouraging and indicate that the current stable version of such tool should be considered as a basic component in the design of security architectures.

2. RELATED WORK

In this section we compare the proposed tool with existing solutions that exhibit a certain degree of similarity and discuss the differences. A wealth of literature exists on Intrusion Detection and prevention, nothing exists on "extrusion" detection, except for [4] and the present paper. However, similar approaches as the one adopted in the host module have been investigated in the literature. In [14], an intrusion prevention/detection system based on system call pattern analysis is described. A specification language based on regular expressions for events is defined that allows to characterize a program based on the normal/abnormal sequence of system calls it makes, taking into consideration also their arguments. At run time, an interception mechanism captures each system call and efficiently matches the current pattern against the one defined, possibly taking actions against the process if necessary. The system presented in [14] is more general than AngeL host module, where only a critical fraction of system calls is considered. Like AngeL, it shows that system protection by system call interception and analysis is a viable and efficient way to enforce system security, as the overall overhead on process execution time is never greater than 5%. It could be interesting to investigate how well the pattern matching algorithm proposed in [14] could perform on the signature analysis in AngeL network module.

The Generic Software Wrappers is another system based on system call interception [10] by means of wrappers. It includes a wrapper definition language that allows to define generic wrappers for all possible system calls and a wrapper support subsystem implemented as a loadable kernel module, like AngeL which is also password protected. Unlike AngeL, where the host module is activated upon each instance of the selected system calls, a wrapper is activated when the activation criteria defined by the user for that wrapper are verified. Although this may gain in terms of efficiency and lower overhead, it may reduce the effectiveness of the system itself if the activation criteria are not comprehensive enough to consider all relevant cases. Another key difference is that GSW is meant for systems with a sensible administrator who would be in charge of installing it and defining the wrappers needed. AngeL is meant to be installed automatically as part of the operating system with minimum if any knowledge of its presence by the user.

A different approach to prevent attacks aimed at increasing privileges by sending a piece of code to be executed on the victim system is presented in [8]. The source code, be it C or shell code, is examined and the presence of typical attack code features are identified. A neural network is trained to perform the analysis with fairly good results in terms of false alarm rate. It can then be applied to scan all the download traffic of a system in order to detect remote attack codes before they are installed or executed on the system.

The STAT methodology [15] based on defining attack scenarios that abstract from the system specific details of attack signatures could be an interesting alternative to the plain attack signatures used in AngeL. Note that both STAT and AngeL follow the misuse approach to intrusion detection, although the high level description of attack scenarios in STAT allows to represent only those step in an intrusion that are critical for the effectiveness of the attack. AngeL uses, on the contrary, low level attack signatures for efficiency and simplicity reasons. However, abstracting away from specific details allows to identify variations of attacks that may otherwise go unnoticed, thus reducing the impact of updating

the machine.

the signature base of the system.

Due to its behavior, i.e., it is always invoked upon certain conditions, such as specific system call execution and network packet transmission, AngeL is often compared to a reference monitor [1, 12]. However, the similarity is more from a functional point of view, since AngeL cannot be proved to work correctly since it relies upon heuristics, it is not small since it is bundled to the attack signature set, it is not complete as the signature set needs regular updates, it is as tamper-proof as possible. Under this respect, AngeL behavior is closer to that of a personal firewall, although by being a kernel patch, AngeL offers stronger resistance to tampering, transparency to the user who is not required to define any security level or turn on/off any security feature explicitly, but no protection against incoming attacks since AngeL's aim is to prevent outgoing attacks.

3. THE MODULE FOR HOST TARGETED ATTACKS

Attacks targeted on the local host are performed by authorized internal users in order to augment their privileges on the system where they have an account, or by external users that have gained access to a local account by guessing the password or breaking the authentication procedure in order to gain root privileges. As reported by the CSI-FBI report [7], this is the most popular type of attack used to compromise computer systems and more than 50% of the computer security attacks in the USA were performed by internals.

In order to build the new module to handle local attacks, which was not present in the original prototype, we analyzed different kinds of host based attacks (see Appendix 1) and concluded that they can be classified into two different species: attacks aimed at gaining higher privileges, in the large majority based on the buffer overflow technique, and attacks whose scope is to consume a resource of the local host, i.e., the X-server or main memory, so as to achieve a denial of service. Attacks such as those aimed at having a process misbehave by passing it bad data are not considered as semantic analysis of process arguments is beyond the scope of this paper. We now illustrate each of the two species in details.

3.1 Higher Privilege Attacks

The most common attack used to gain higher privileges on a system is the buffer overflow. Therefore, we examine it here as the first case we implemented in the host targeted attack module in AngeL. Furthermore, in the current release, we only consider the buffer overflow attacks where a program receives on the command line, as the argument of an option, a string containing the binary code of a program executing the `execve("/bin/sh")` system call, or equivalent ones, instead of a regular input parameter. We will refer to such a code as shell code in the rest of the paper. Some examples of shell codes we have considered are given below.

Other types of buffer overflows where the shell code is injected as input data at run-time are not considered in the current release of the tool and will be included in the future ones. Because the vast majority of local buffer overflows use the other type of buffer overflow illustrated before, we

```
/*
 * setregid and generic shell code
 * for slirp[v1.0.10(RELEASE)]
 */
\xeb\x29\x5e\x31\xc0\xb0\x2e\x31\xdb\xb3\x0c\xcd\x80
\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b
\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89
\xd8\x40\xcd\x80\xe8\xd2\xff\xff\xff\x2f\x62\x69\x6e
\x2f\x73\x68

/*
 * spawns a shell from a program executing chroot()
 */
\xeb\x4f\x31\xc0\x31\xc9\x5e\x88\x46\x07\xb0\x27\x8d
\x5e\x05\xfe\xc5\xb1\xed\xcd\x80\x31\xc0\x8d\x5e\x05
\xb0\x3d\xcd\x80\x31\xc0\xbb\xd2\xd1\xd0\xff\xf7\xdb
\x31\xc9\xb1\x10\x56\x01\xce\x89\x1e\x83\xc6\x03\xe0
\xf9\x5e\xb0\x3d\x8d\x5e\x10\xcd\x80\x31\xc0\x89\x76
\x08\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56
\x0c\xcd\x80\xe8\xac\xff\xff\xff/bin/sh

/*
 * Alephone's shell code for system("/bin/sh");
 */
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89
\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff
\xff\xff/bin/sh

/*
 * This shellcode exploits cxterm5.1-pl1.
 * It works on RH5.2-RH6.0, Slackware 3.6.
 * Shellcode is injected via the DISPLAY variable
 */
\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89
\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c
\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff
\xff\xff/bin/sh
```

Figure 1: Examples of shell codes.

started with the most popular type and decided to leave the less popular one for future releases of AngeL.

The technique adopted by AngeL to handle buffer overflow attacks is based on a wrapper for the `execve()` system call, since buffer overflows are based on the execution of such a system call. Thus, in any system where Angel has been installed, a call to `execve()` is intercepted by the tool and its parameters carefully analyzed in order to verify if malicious code is hidden in them. AngeL first examines the execution environment of the new process that should execute after completing the `execve()` system call is analyzed. The values of environment variables[2] such as $HOME or $TERM, are checked to see if they contain an executable shell code or a suspicious character, e.g., "/". Such a step is implemented as an exhaustive search over all the environment variables for the set of shell codes collected in the tool knowledge base. If this check yields a negative result, i.e., none of the environment variables hides any of the shell codes known to the tool, AngeL starts the analysis of the properties of the program whose execution is invoked via `execve()`. The main properties of such a program are gathered by calling the `stat()` system call, which provides the information regarding the privileges of the new process, i.e., whether it will be setuid or setgid to root. If this is the case, our analysis continues by checking the parameters that will be passed to the program to see whether there is some known shell code. If this check too yields a negative result, the new process is finally spawned, otherwise the executing program is terminated and a message logged for the superuser. Note that in this case, a drastic action is taken, which may damage the innocent user whose process is being exploited. Less drastic actions could be taken, such as preventing the execution of the offending code while letting the original program proceed. Refinement such as this will be taken into consideration for future releases of AngeL. The same applies in case of hostile packets being intercepted by the network module.

The `execve()` wrapper execution is fairly overhead prone, as it will appear in Section 6, because the search for shell codes is lengthy and computationally heavy. Some improvements are possible by implementing optimized string matching algorithms, which is part of future releases.

3.2 Local DoS

If the malicious user is unable to perform an attack aimed at gaining higher privileges, or he/she is simply not interested in gaining control of the host and a nuisance action is sufficient, a simple way to damage the system is to reduce its availability to the point of making it useless or extremely slow, by consuming one or more of its resources. The easier technique to achieve such a goal is to launch a local DoS attack. The most popular attacks of this type are the fork bombing, aimed at exhausting the resource number of processes in the system, and the malloc bombing, aimed at consuming the dynamic memory area (e.g., the heap) of a process. In order to block these attacks, we adopted the following strategy. The `fork()`, `vfork()`, `clone()` system calls for the fork bombing attack and the `brk()` system call for the malloc bombing attack are protected by wrappers.

[2]The use of environment variables to insert the shell code in the vulnerable program has appeared in some cases of buffer overflow exploits.

When any of the mentioned system calls is executed, the corresponding wrapper is executed instead, which verifies the current use of the resource by the calling process together with the rate of use in the last time interval. If one of these parameters is greater than a threshold value the potentially hogging process is terminated. The critical factor in this strategy is obviously the identification of the correct threshold values and time interval. The time interval is set to one second, which strikes a good balance between too short an interval, which would not allow to observe a trend, and a too long one, which would on the contrary emphasize even small differences.

After an extensive tuning phase, we have defined a set of threshold values, which are the default values. For this reason, the default values implement a fairly restricted environment. In order to make the tool as transparent as possible to non-professional users or users with limited system administration knowledge, the threshold values of the parameters cannot be modified. However, since the tool could also be installed on systems with experienced administrators who could be willing to tune the parameters to their configurations, they can still change the parameters by modifying the tool source code.

The system admits at most 100 processes in execution per user via the compiler directive

```
#define MAX_FORKS_PER_USER 100
```

with a maximum forking rate of 50 processes per second via the compiler directive

```
#define MAX_FORKS_PER_SECOND 50.
```

As for the memory allocation, the system accepts up to 500.000 malloc requests per second via the compiler directive

```
#define MAX_BRK_PER_JIFFIE 5000
```

where JIFFIE is one hundredth second for a total maximum memory allocation of 20MB, via the directive

```
#define MAX_BRK_DIMENSION 20000000.
```

As a special case of local DoS, we illustrate the local Xserver attack. In this case, the Xserver is forced to behave as a CPU hog when it receives a control packet that specifies a negative value for the XC-QUERY-SECURITY-1 parameter at a 20 byte offset from the interested field. The X server takes such a value and starts decrementing it until it reaches 0, without checking the initial value. Thus, by setting XC-QUERY-SECURITY-1 to a negative value, e.g., -1, the whole range of first the negative and then the positive 64 bit integers is spanned before the variable reaches 0. Because the variable is a long, i.e., a double word integer, the operation takes some time (in the order of minutes) during which the system does not respond to any signal whatsoever.

4. THE MODULE FOR NETWORK TARGETED ATTACKS

This module is developed to be integrated with the personal firewall capability of Linux, i.e., the `netfilter` tool. With netfilter, a user can customize actions that some filters will apply to the packets of various protocols. Different "hooks," i.e., points where the filters can be inserted in the netfilter skeleton, are defined for each network protocols, for which a user can specify a set of rules he wants the protocol to apply to the packet. AngeL modules are connected to the NF_IP_LOCAL_OUT hook, i.e., the hook that netfilter provides to handle outgoing IP packets, just before they are passed to the data link layer protocols.

In the development of our module, we divided the attacks targeted on network services in two broad categories: attacks that exploit network and transport layer protocols vulnerabilities (such as SynFlood for TCP or SMURF for ICMP [5, 11]), and those that exploit application layer protocols vulnerabilities, such as the PHF attack on HTTP.

4.1 Network and transport layer attacks

The network and transport layer module handles the UDP, TCP, ICMP, and IP protocols. From our perspective, the attacks to such protocols are characterized by the need to remember previous behavior in order to detect a malicious intent, or by the simple packet inspection in order to detect the malicious intent. The former case is the most demanding for the module from a performance point of view as it requires to maintain, and examine, previous states of the protocols and the packets.

Attacks such as IP spoofing, SMURF, LAND [6] are examples of attacks that can be recognized by simple packet header inspection. As an example, in order to detect a packet with a spoofed source address that is about to leave the host it is sufficient to compare the packet IP source address with all the IP addresses of the network interfaces of the host. Similar strategies can be adopted for other attacks of this kind.

Attacks such as SynFlood and XsHoK are examples of attacks that require that the module maintain variables describing the history of the behavior of the host with respect to some critical parameters. As an example, in order to prevent a host to flood a remote Xserver with false requests, a table addressed by the destination IP address x and the user id, uid, is maintained. Each table entry contains the number of connections established by that uid to the Xserver of system x. If such a value exceeds a given threshold value, uid is disabled from opening further connections to the Xserver on x.

4.2 Application layer attacks

The current release of AngeL considers the following application layer protocols: HTTP, FTP, Sendmail and Telnet. The attacks considered for the Telnet and FTP protocols are remote buffer overflows of some implementations of the servers for these services. These attacks are detected and blocked by inspecting the packet payload in outgoing packets to ports 21 and 23, respectively, looking for shell codes such as those described in Section 3.1.

With respect to the HTTP protocol, we started by considering attacks that are performed by forcing the server to execute various commands. These attacks too are detected by payload inspection. They are characterized by the fact that they hide command execution requests in "GET" or "POST" requests. The AngeL module looks for command execution requests in the packets leaving the host for port 80. Based on this strategy, the module blocks the following attacks: PHF hacking, IIS arbitrary command execution, Infosearch arbitrary command execution, Alibaba arbitrary command execution, Amlite Vulnerability and BizDB vulnerability.

5. ON THE DIFFICULTY OF REMOVING ANGEL

The construction of tools such as AngeL always raises controversial issues regarding the possibility to easily bypass them and their update. In particular,

- software modules intended to protect a system can be removed or bypassed in various ways by intruders who have gained control of the system, e.g., by mean of a root compromise, thus making such a protection ineffective; AngeL, being a software module, is not immune to this drawback;
- updating the signature database used by all types of filter modules to detect attacks is particularly difficult and critical, especially if the database is bundled in the code, and periodical update is necessary to maintain the effectiveness of the filter; AngeL's attack signature database is bundled in the code.

Although these issues may seem different, they are related, as we will explain in what follows. One of the implementation choices we faced during the development of AngeL, was between static kernel module and loadable kernel module. In the first case, the module is loaded together with the kernel at boot time as opposed to the second case where the module is loaded after the boot phase completes as part of the executing kernel. In the former case, an intruder who gains superuser privileges on a machine executing AngeL can remove it only by downloading on the machine a copy of the kernel without AngeL and rebooting the machine with the new kernel. This solution provides a good security level, as good as it is possible with software modules, assuming that a reboot operation would not go unnoticed. However, updating the signature database becomes a very serious problem because any database update would require the re-compilation of the kernel, a critical and time consuming operation.

On the other hand, by implementing AngeL as a loadable module, we simplify the update problem significantly. In order to update the signature database we only need to compile the new version, remove the older one using the `rmmod` system utility and replace it with the new module using the `insmod` system utility. Unfortunately, with this solution, it would be very easy for an intruder to remove the module from the kernel.

The optimal solution would be the one that gives the same security level of a static kernel module with the flexibility of a loadable kernel module. Using some features of Linux, we were able to implement the optimal solution.

AngeL is configured as a loadable kernel module. In the loading phase a password is associated with it, i.e., it is loaded with the instruction

```
insmod angel password = ****.
```

The password is encrypted using MD5 and stored in a kernel area. An AngeL device (/dev/angel) is also created, on which no read operation is defined, only write operation is possible. In order to remove the module, the module password must be written on the AngeL device /dev/angel. The write operation on such a device verifies if the newly written password is equal to the original one, whose MD5 is maintained in kernel memory. If this is the case, a flag is set that enables the module removal, otherwise the only way to remove the module would be to download on the disk a new boot file, and reboot the system as if the module were a static kernel module.

Although a system reboot is an operation which does not usually go unnoticed, it is difficult for a "average" end-user, i.e., a user with little system administration skills, to notice the difference between the malicious boot and the usual boot. Furthermore, in order to make the operation less noticeable, the intruder may wait for a "natural" boot to occur, i.e., wait for the system to reboot because of some unrecoverable problems rather than force a reboot. It depends on the type of system how often such an event is for the intruder to be willing to wait or not. Unfortunately, while we can work on improving the protection measures to prevent our module from being removed from the system, there is not much that can be done to prevent the system from being reinstalled completely. The only viable solution in this case would be a hardware implementation of the module. This is the only way a disarmed system could not be rearmed.

6. EXPERIMENTAL EVALUATION

In this section we describe the results of a set of experiments aimed at investigating the impact of AngeL on system performance. The hardware platform used for the experimental evaluation is a PC with a 133 MHz Pentium and 64 MB RAM running Linux Slackware 7.0, 2.4.2 kernel. We performed two sets of tests, one for each module of the tool. In order to measure the overhead of the local attacks module, we ran some kernel programs that only execute the wrapped system calls. Table 1 shows the average number of fork() and execve() system calls per second that can be issued in the absence of the module (rightmost column), with the module (central column), and with full checks on environment variables (leftmost column). The numbers reported in the table are the average over 20.000 runs. As the very small values of the coefficient of variation indicate, the measurements are fairly stable. As the table shows, checking all the environment variables in case of execve() reduces the maximum throughput in terms of completed system calls per second of about 13%. If only the call parameters are checked, a negligible reduction of 2.5% in the throughput is observed. Unfortunately, with the increasing popularity of format bugs, checking environment variables is becoming more and more important. Although a 12.6% reduction in throughput is not negligible, the execve() system call is not invoked continuously, so the impact is not so serious.

	w/o ANGEL	w/ANGEL	Δ
execve()	82.171 (0.015)	72.983 (0.009)	-11.1%
fork()	31.189 (0.02)	30.286 (0.02)	-2.8%

Table 1: Average number of system calls completed per second for the execve() and fork() system calls without the module (rightmost column) and with the module (central column). In the case of the execve() system call, the leftmost column gives the results with the module checking all environment variables. The value in parentheses is the coefficient of variation.

Table 2 illustrates the results of the measurements taken for the network module, on the native system and with the module, under hostile traffic. The metric in this case is the average number of packets per second the system can send under the various scenarios. The numbers reported in the table are the average over 39.000 packets. In this case too, very small values of the coefficient of variation are obtained. As the table shows, the largest impact is on http traffic, as the worst case is considered for matching all strings to be checked. However, since the outgoing traffic is significantly less than the incoming one, a throughput reduction in the range of 7% to 15% does not affect the performance as perceived by the user.

protocol	w/o ANGEL	w/ANGEL	Δ
http	869.7 (0.009)	739.5 (0.006)	-15%
ftp/lpd/telnet	220.9 (0.01)	203.2 (0.006)	-7.7%
sendmail	1195.5 (0.01)	1110 (0.009)	-7.1%

Table 2: Average number of packets per second for various application layer protocols without and with the module, under different types of traffic. Coefficients of variation are given in parentheses.

7. CONCLUSIONS

In this paper we have described a tool that disarm computers by intercepting hostile traffic carrying attacks at networked hosts and blocking it, and local attacks such as local DoS and buffer overflows. The tool is publicly available under the Gnu Copyleft License. It can be downloaded at the following URL: http://www.laser.dsi.unimi.it/AngeL. The current version was developed under the kernel 2.4.x and runs also for versions 2.2.18 and following.

Acknowledgments

The authors are grateful to Carla Marceau who excellently presented the paper on their behalf, to Bob Blakley whose note keeping effort was this time even more important, and to Aldo Scaccabarozzi and Paolo Perego, who patiently collected attack exploits and implemented them into AngeL modules.

8. REFERENCES

[1] Anderson J., "Computer security technology planning study," *U.S. Air Force Electronic System Division Technical Report* 73-51, October 1972.

[2] Bandel D. "Linux Security Toolkit," IDG Books, 2000.

[3] Bruschi D., Cavallaro L., Rosti E., "Less harm, less worry or how to improve network security by bounding system offensiveness," *Proceedings of ACSAC '00, 16th Annual Computer Security Application Conference*, New Orleans, pp 188-195, 2000.

[4] Bruschi D., Rosti E., "Disarming offense to facilitate defense," *Proceedings of the New Security Paradigm Workshop 2000*, Ireland, pp 69-75, Sept. 2000.

[5] CERT-CC, "TCP SYN flooding attacks and IP Spoofing attacks," CERT Advisory CA-96.21, http://www.cert.org, 1996-98.

[6] CERT-CC, "IP Denial of service attacks," CERT Advisory CA-97.28, http://www.cert.org, 1997-98.

[7] Computer Security Institute, http://www.gocsi.com/prelea_00321.htm.

[8] Cunningham R., Rieser A., "Detecting source code of attacks that increase privilege," presented at RAID 2000, available at http://www.raid-symposium.org/raid2000/Materials/Abstracts/53/53.pdf

[9] Erlingsson, U., Schneider, F.B., "IRM Enforcement of Java Stack Inspection", *Proceedings of the IEEE Symposium on Security and Privacy*, pp.246-55, May 2000.

[10] Fraser T., Badger L., Feldman M., "Hardening COTS software with generic software wrappers," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[11] Huegen C., "The latest in denial of service attacks: smurfing. Description and information to minimize effects," http://users.quadrunner.com/chuegen/smurf.cgi, last update Feb. 2000.

[12] Lampson B., "Protection," republished in *Proc. of the 5th Princeton Symposium, Operating System Review*, Vol 8, No 1, pp 18-24, Jan. 1974.

[13] McHugh, J., et al., Discussion at NSPW2001, 2001.

[14] Sekar R., Uppuluri P., "Synthesizing fast intrusion prevention/detection systems from high-level specifications," *Proceedings of the Usenix Security Symposium*, pp , 1999.

[15] Vigna G., Eckmann S., Kemmerer R., "The STAT tool suite," *Proceedings of DISCEX 2000*, 2000.

APPENDIX

A. ATTACK REFERENCES

The following list is just a partial list of the attacks handled by the tool based on the cve.mitre.com database. Only the reference number is provided for the sake of space.

1. CVE-2000-0454
2. CVE-2000-1180
3. CVE-1999-0137
4. CVE-2000-0438
5. CAN-1999-0114
6. CVE-2000-0824
7. CVE-2000-0844
8. CVE-1999-0032
9. CVE-1999-0335
10. CAN-2000-0545
11. CAN-2000-0545
12. CVE-2000-0218
13. CAN-1999-0317
14. CAN-1999-0317
15. CAN-1999-0651
16. CVE-1999-0034
17. CVE-1999-0138
18. CVE-2000-0703
19. CVE-1999-0733
20. CVE-2000-0090
21. CAN-1999-0623
22. CAN-2000-0620
23. CVE-1999-0038
24. CVE-1999-0128
25. CVE-1999-0166
26. CVE-1999-0016
27. CVE-1999-0513
28. CVE-1999-0265
29. CVE-2000-0305
30. CVE-1999-0067
31. CVE-2000-0207
32. CAN-2000-0866
33. CAN-1999-0885/0776
34. CVE-2000-0287
35. CVE-2000-0638/639
36. CVE-2000-0810/811
37. CVE-2000-0138
38. CAN-2000-0573
39. CVE-2000-733
40. CAN-2000-0917
41. CVE-2000-733
42. CVE-2000-0567
43. CVE-2000-0352