In Proc. of 14th Kommunikation in Verteilten Systemen 2005 (KiVS05), Kaiserslautern, Germany, February 2005.

A Honeypot Architecture for Detecting and Analyzing Unknown Network Attacks

P. Diebold, A. Hess, G. Schäfer

Telecommunication Networks Group, Technische Universität Berlin, Germany,
dakkonbb@cs.tu-berlin.de, [hess, schaefer]@tkn.tu-berlin.de

Abstract. In this paper, we propose a honeypot architecture for detecting and analyzing unknown network attacks. The main focus of our approach lies in improving the "significance" of recorded events and network traffic that need to be analyzed by a human network security operator in order to identify a new attacking pattern. Our architecture aims to achieve this goal by combining three main components: 1. a packet filter that suppresses all known attacking packets, 2. a proxy host that performs session-individual logging of network traffic, and 3. a honeypot host that executes actual network services to be potentially attacked from the Internet in a carefully supervised environment and that reports back to the proxy host upon the detection of suspicious behavior. Experiences with our first prototype of this concept show that it is relatively easy to specify suspicious behavior and that traffic belonging to an attack can be successfully identified and marked.

1 Introduction

Recent experiences with attacks in the Internet and especially the tremendous increase in the propagation speed of self-distributing attacks clearly show that the problem of exploiting vulnerabilities of hosts connected to the Internet can not be countered appropriately with an approach that is only aiming to defend from attacks against end systems by fixing security holes when patches become available. In order to overcome this situation, various researchers are working on network based intrusion prevention (examples of existing open-source IPS are Snort-Inline [4], Hogwash [12], IBAN [5] and FIDRAN [7,8]).

However, in order to realize an efficient intrusion prevention system (IPS), relatively detailed knowledge about potential attacking patterns is needed. Most of the systems in use today, therefore, work with a set of so-called *attack signatures*, that describe attacking patterns in sufficient detail for identifying ongoing attacks automatically. However, the specification of such signatures usually needs to be done by experienced network security analysts by either monitoring an existing network and extracting the relevant information as new attacks are launched and get detected, or by directly analyzing new attacking tools, worms, etc. as they become available.

In order to support this task, the use of so-called *honeypots* has been proposed in recent years. One common definition of this term is: *"a honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource"* [16].

As stated in this definition, a honeypot is a system that is built and set up in order to be hacked. Honeypots can be used as intrusion detection facility (burglar alarm), defense- or response mechanism. Apart from this, honeypots can be deployed in order to consume the resources of the attacker or distract him from the valuable targets and slow him down, so that he wastes his time on the honeypot instead of attacking production systems.

The fundamental principle of the honeypot idea is that every connection (even an attempt) or scan, respectively, which is destined to a honeypot can be considered suspicious. A honeypot is not a production system and consequently, nobody has reasons to contact it. According to this, the amount of traffic that is sent to a honeypot is assumed to be manageable and of high significance to intrusion detection. However, as we experienced in an experimental setup (see below), traffic volume received by a honeypot still lies in a rather high range, so that a more intelligent way than just analyzing all received traffic is required.

We therefore developed a honeypot architecture that allows to obtain more significant traffic logs of suspicious behavior by combining filtering of already known attacks with session-individual traffic logging and marking of suspicious sessions based on evidence gained on an actual honeypot host. The remainder of this paper is organized as follows: in session 2 we give some background information on honeypots, review related work and report on our experience with traffic volume received by a honeypot in an experimental setting. Section 3 describes our concept, and in section 4, we describe our prototype implementation of this concept and discuss first results. In the final section 5, we draw some conclusions and give an outlook to future work.

2 Honeypots

Honeypots are often categorized by their level of interaction [16]. So-called *low interaction honeypots* are defined as simulated services, i.e. anything from an open port to a fully-simulated network service. Most of the low interaction honeypots use simple script-based languages to describe the honeypots reactions to attacker inputs. Low interaction honeypots are easy to set up and because of the limited capabilities they are quite secure. The drawbacks are that they are easy to detect for attackers, because the service's reactions are not implemented completely. The information gained is limited as well, because no real vulnerabilities can be distinguished from attack attempts. Its use is restricted to the logging of automated attacks and intrusion detection.

So-called *high interaction honeypots* — we emphasize here that we do not make a distinction between medium and high interaction honeypots — are real services which are usually executed in a secured environment. The attacker communicates with an actual service implementation but theoretically he does not achieve total system control in case of a successful attack if the honeypot is well designed. One advantage of this approach is that a high interaction honeypot is difficult to unmask. Actually, the attacker should not be able realize that he interacts with a honeypot because a real service is used. Of course, the honeypot system can be only as secure as its sandbox subsystem — in case that it comprises such functionality.

Another important factor for the usefulness of a honeypot is accessibility. Obviously, it is only useful if it gets attacked. Moreover, in order to gather the intrusion related information, logging and sniffing techniques must be integrated into a honeypot architecture. Accordingly, the question arises how to realize a honeypot which resembles a vulnerable system, but which can not be exploited, and which beyond this possesses intrusion detection capabilities, in order to inform the administrator about the occurrence of an attack.

2.1 Related Work

Honeyd [15] is a low-to-medium-interaction honeypot system. Installed on a Unix system it listens on the network interface card (NIC) for incoming ARP requests. If an ARP request is detected *Honeyd* initiates an ARP request itself. If no response to the own ARP request is given and a rule for the requested IP exists in the configuration file, *Honeyd* overtakes the IP and starts pre-configured services on the specified ports. The honeypot comes with some shell scripts that emulate services (e.g. a WWW server). *Honeyd* is able to emulate the behavior of most common IP-stack implementations (Windows, Linux, etc.) that can be detected by the tools *nmap* [6] and *x-scan* [18] by using the same rule base as the scanners. *Honeyd* is open source and has been successfully used on the Cebit 2003 by the heise publisher (see article [2]).

Bait'n'Switch [10] is a honeypot response mechanism that redirects the attacker from valuable targets to a honeypot system. *Bait'n'Switch* is realized as a *Snort* [4] inline extension. Whenever a successful attack is detected, the IDS drops the packets of the first attack and all further traffic from the host that initiated the attack is rerouted to a dedicated honeypot host. This process is hidden from the attacker so that he does not realize that he is not communicating with the original target anymore. The attacker's further interaction with the honeypot can later be analyzed and the production system is protected from the attacker's further actions. The system reacts on attacks that are described in an IDS signature database and can therefore only react on previously known attacks.

The *Intrusion Trap System* [17] is an improved version of *Bait'n'Switch*. In case that *Snort* detects an attack, the *Intrusion Trap System* is able to directly redirect the attack to a honeypot host. This way, even the traffic of the first attack can be handled by a honeypot and the attacker is unable not notice that he is not communicating with the production system. However, the approach has the same limitation to known attacks as *Bait'n'Switch*.

Honeycomb [11] is realized as a *Honeyd* extension. It is based on the idea that any traffic directed to the honeypot can be considered an attack. *Honeycomb* automatically generates *Snort* and *Bro* [13] signatures for all incoming traffic. New signatures are created if a similar pattern does not yet exist. Existing signatures are updated whenever similar traffic has been detected, so the quality of the signatures is increased with each similar attack session. Signatures can be updated to match mutations of existing attacks. For each mutation a more generic description for the signature is generated, so that the original attack and the mutation are both matched. This way the signature base is kept small. The mechanism creates signatures for all traffic directed to the honeypot. Unfortunately the attacks are not verified to be successful in any way. Therefore, it suffers of false positives if any non-attack traffic is directed to the honeypot like e.g. the IPX protocol. A computer connected to the Internet especially on a dial-up connection is addressed even by non attack traffic. Whenever a search engine tries to mirror the host or a peer to peer program tries to connect, a signature is generated. Signatures must be checked manually afterwards whether they were created for an attack or for something else. An approach to verify the attack patterns is desirable. The signature generation mechanism could be used to create IDS signatures if an appropriate attack traffic is identified and directed to the system.

Finally, the idea of so-called *honeynets* [14] exist. Instead of simulating a single vulnerable host, a honeynet tends to simulate a complete network by the deployment of a set of honeypots.

The central idea of honeypots is, that any traffic directed to the honeypot, is considered an attack. The state of the art shows that one the one hand the gathering of attack related information currently requires the manual analysis of the log files and on the other hand the approach to generate signatures that match any incoming traffic leads to false positives that have to be eliminated manually. Furthermore, honeypot approaches that only react to known signatures do not gather any information valuable for identifying new signatures. Summarizing, the existing approaches do not provide sufficient support for the (semi-)automatic detection and analysis of unknown attacks.

2.2 Field-Test

As described in section 2.1 the open-source low-interaction honeypot *Honeyd* is able to simulate several host identities and in addition, it offers scripts that simulate network services. We configured *Honeyd* such that it represents a typical Linux installation (with one IP-address) which runs a WWW- and a SSH-server. For four days we connected the setup to the Internet and we logged each connection. The corresponding log file grew to a size of $18 MByte$. A consecutive evaluation of the log file showed that a big part of the conncections were caused by known attacks (see table 1).

Table 1. *Honeyd* field-test results

Event	No of occurrences
Nimda	8871
CodeRed	2155
CodeRed II (3 versions)	2626
MyDoom	1369
W32/Welchia.D	1674
Attempts to access the IIS-samples	645
Attempts to get '/etc/passwd'	168
Attempts to execute cmd.exe	123245

Apart from the attacks enlisted in table 1, many entries were enregistered that were caused by browser connections or by “talkative” protocols like NetBIOS, IPX, etc. The conducted experiment clearly shows that even in case of a honeypot with an “unadvertised” IP-address assigned to it, the amount of logged (attacking) traffic is immense. Since the focus of our work on honeypots is towards the detection and analysis of unknown network security attacks, further measures to increasing the significance of logged traffic are required. In the next section, we describe our approach to this problem.

3 Concept

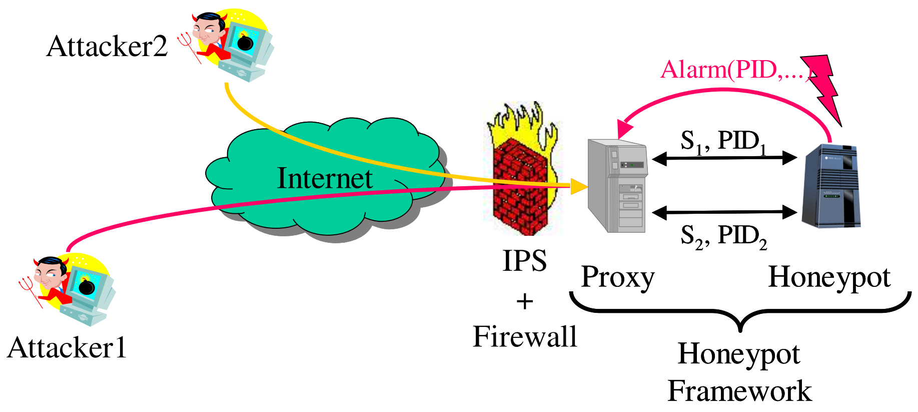

Fig. 1. An example scenario

The goal of our approach is the design and realization of a generic high interaction honeypot framework that allows to (semi-)automatically identify application layer based attacks (e.g. buffer overflows, format string attacks, etc.). Figure 1 depicts an example scenario that includes two attackers, a firewall / intrusion prevention system (IPS) and our honeypot framework. The purpose of the IPS / firewall is to filter the incoming traffic for known attacks. The honeypot framework itself consists of a proxy and a honeypot host. The proxy host is responsible for the session-individual logging of the network traffic that was sent to the honeypot. Furthermore, in case of a detected attack the proxy provides a mechanism to replay a specific previously logged session. An advantage of the bipartite approach is that the honeypot and the log files are kept on separate hosts so that in case of a complete system takeover of the honeypot host by an attacker the log files remain save. Besides this, the replay mechanism that is integrated in the proxy can be used to analyze a discovered attack in detail, as well as to test if other system configurations are just as vulnerable as the honeypot service to the attack.

The honeypot host consist of a honeypot service, namely a real service, and a host intrusion detection system (HIDS). The running service is the bait that attracts worms respectively hackers whereas the HIDS supervises the honeypot service. The realized detection mechanism is generic and allows the detection of attacks on the basis of system-call signatures. The reasoning behind this approach is that the main part of current attacks exploit a vulnerability that is specific for a software (e.g. a buffer overflow vulnerability of a WWW-server). In case of a successful attack, the hacker will sooner or later exploit its newly gained authorizations which most often results in an observable system change. An example for this would be an attacker that tries to open a new network socket in order to download further hacking utilities. Or another popular example is a worm that starts on each infected system an email relaying server which it uses for its further spreading. On the system-call level both examples can easily be monitored. The interface between honeypot service and HIDS is generic such that is possible to exchange or add a honeypot service in an easy and flexible manner.

As depicted in figure 1, it might occur that a honeypot service is running two or more process entities (parent + children processes) simultaneously. For example, if the honeypot system was running a WWW-server as a honeypot service, the server parent process would create a child process for each HTTP-session.

3.1 The Honeypot Host

Figure 2 depicts the internal software architecture of the honeypot host, which consists of the honeypot service, the HIDS-manager (hidsmgr), the honeypot monitor (monitor) and the host intrusion detection system (HIDS). Honeypot service, monitor and HIDS-manager are running in user-space, whereas the HIDS is located in the kernel space.

Generally, most common operating systems make a distinct differentiation between application and operating system. Each time a user-space process requires an operating system service (e.g. the opening of a network socket) the service must send the proper system call to the kernel. The kernel checks the request of the process and then it decides whether to fulfill it or not. By inserting a HIDS into kernel-space and by redirecting the system-calls to the HIDS, it is possible to extend the functionality of the kernel. In our case, it is possible to monitor on the basis of system-call level what a process in user-space does. In addition, it is possible to introduce more detailed decision criteria in the kernel to determine whether the desired action is allowed or not (a similar mechanism has also been proposed for performing access control on active networking nodes; see also [9]).

The general method of system call interception is depicted in figure 3 and shows the interception of the *socket* system call. The user process uses the *socket()* command to create a socket for network communication. A process must execute a system call to gain access to the operating system services. Normally, this is done by wrapper functions which are part of standard libraries. The wrapper function puts the variables to be submitted into the correct order, and then executes the proper system call. At the entry point into the kernel, the kernel uses a table — the so-called system call table — for the forwarding of the incoming system calls to the corresponding functions. By changing the destination of a pointer inside the system call table, we can redirect a defined system call to another function, in our case to the HIDS. That checks if the service is authorized to use a specific operating system service. If this test is passed, the HIDS then calls the standard kernel function belonging to the system call.

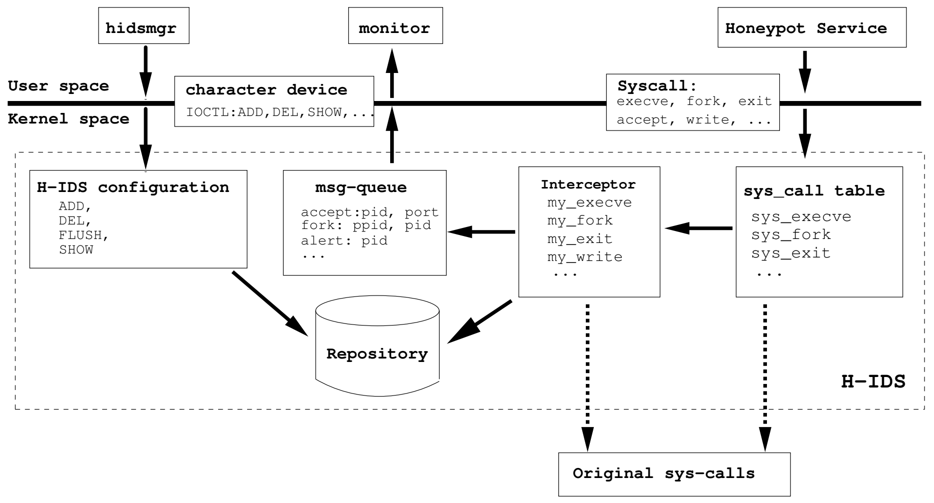

Fig. 2. The honeypot host

The HIDS is realized as a Linux Kernel Module (LKM) and it detects intrusions on system-call-level. Whenever a user space process tries to execute a series of system-calls that matches an attack signature a security alert is raised. Furthermore, the HIDS refuses to forward the last requested system-call to the operating system in order to prevent the honeypot system from being harmed. Consecutively, the HIDS triggers the monitor process to send an alarm message to the logging proxy. The attack signatures are specified inside the repository as a series of system-calls or simply as a black-list of disallowed system-calls. Besides this, it is also possible to configure the HIDS such that the execution of a specific group of applications (e.g. common gateway interface - CGI) is authorized. The user space front-end monitor handles the synchronization of local process ID and remote session ID between honeypot and proxy server.

The HIDS checks for each observed system-call, whether or not it is executed by a process under supervision. The access to the operating-system is either granted or not, depending on the policy. Moreover, if necessary a message is sent to the monitor and then forwarded to the logging component.

Finally, the HIDS-manager can be used to reconfigure the HIDS at runtime. It provides a set of functions which first can be used to add, delete or modify existing attack patterns. Second, the manager also allows to modify the list of services that must be observed by the HIDS.

3.2 The Logging Proxy

The logging proxy listens for connection requests to the honeypot service which originate from a potential hacker. Next, the proxy server acts itself as a client on behalf of the user / attacker and forwards the request (using its own IP address) to the honeypot service. Besides forwarding, the proxy server also creates for each forwarded session

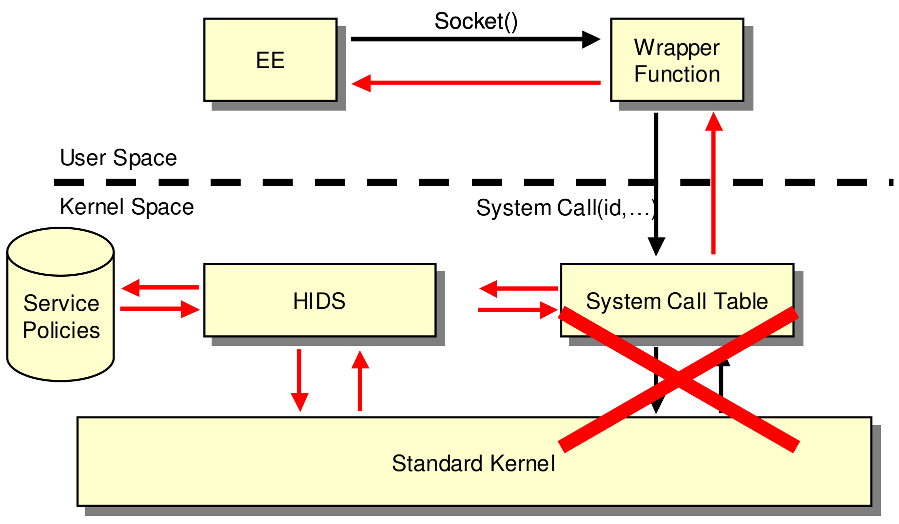

Fig. 3. Interception of a system call

(session ID) an individual log file which in addition, contains the IP-address of the attacker, the connection ports and a timestamp. To the attacker, the proxy server is invisible; all honeypot service requests and returned responses appear to be directly from the proxy host.

The proxy logs the connection data of the honeypot service. A difficulty thereby is to match attack session and PID of the corresponding process on the honeypot host, as one host keeps the logs while the other one detects the attacks. If a new client connection is initiated by an attacker, the proxy sends the new session ID via the control channel to the monitor on the honeypot monitor. This one acknowledges the successful connection of the proxy to the honeypot service by sending a message to the proxy server, which contains the PID of the corresponding child process and the ports of the incoming connection. The ports are used to track which connection and session ID belong together. Furthermore, the proxy server maintains a list of currently established connections to the honeypot service. On a fork of the honeypot service a new PID message is automatically sent by the honeypot service monitor to the proxy server. In case that a process tries to execute a series of an unauthorized system-calls (attack signature), the honeypot monitor triggers an alert and sends a corresponding message to the proxy server. The alert-message contains the PID of the honeypot service process that violated the security policy and the kind of violation. By the means of the alert message the proxy server tags the corresponding session and adds the alert information to the proper log file. In addition, the proxy server is able to stop the ongoing attack session — if specified so in its local security policy.

Process ID Tracking Generally, processes can be identified by their unique process identification number, the PID. As already mentioned, many networking services — that can be used as the honeypot service — create a new process environment for each connection that they accept. Accordingly, a mechanism is required to keep track of the processes that must be observed by the HIDS. Figure 4 depicts our principle of PID tracking. Initially, we open a *shell* which we subsequently add — with the help of the HIDS manager — to the list of processes that are monitored by the HIDS. Next, the chosen honeypot service is started from inside the shell, which automatically assigns it to the group of services that must be observed by the HIDS.

Normally, a new process is created by a networking service through the execution of the *fork()* system-call (other possibility *clone()*). In this case the operating system creates a new process environment and assigns a new process ID to it. Now, in case that an attacker connects to the honeypot service, then the operating system creates the new process by executing the *fork()* command and returns two values. One value, which is the PID of the newly created process, is returned to the parent process, whereas the newly created child process receives a zero as return value. Whenever a process that is member of the list of services that must be observed by the HIDS creates a child process, then the newly created child process is automatically assigned to be a member of the list.

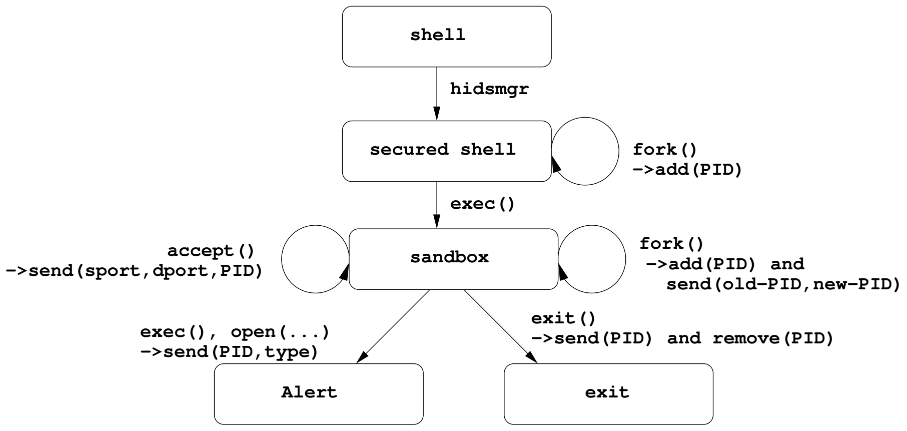

Fig. 4. Process ID (PID) tracking

3.3 Replay Mechanism

The session-specific log-files that are created by the proxy server can be used by the replay tool, that is part of the proxy server, to repeat a selected (attack) session. In case that the honeypot service is stateless and deterministic, the attacker can be replaced by the replay tool and each suspicious log-file can be replayed in sequence. This allows to analyze attacks in detail as well as to test if other systems are equally vulnerable to an attack.

4 Proof Of Concept

We realized a Linux-based prototype of the described honeypot architecture. In order to test its functionality we setup a testbed, which consists of three Pentium III 800 machines running Linux 2.4.24, and which are connected via Ethernet. The middle host — equipped with two network interface cards — is running the proxy server. The remaining hosts are used as attacker and honeypot.

The Dune Web Server [3], which is known to contain several security holes, was chosen as honeypot service. Furthermore, we downloaded the according exploitation tool *xdune* [1] from the Internet. *Xdune* is a tool that can be used to remotely exploit a buffer overflow vulnerability of the *Dune* web server. We started our honeypot system with the *Dune* server as honeypot service. Before starting the *Xdune* tool, we simultaneously started several harmless HTTP sessions in order to check if the honeypot system is capable to automatically identify and tag the attacking session.

The normal order of events — without the intervention of a HIDS — would be as follows. First *Xdune* searches in a brute-force manner for the correct return address that directs the process execution to the injected code. If successful, the injected shell code opens a listener and executes the command */bin/sh*. With the HIDS running on the honeypot system, the attempts to open a network socket respectively to start a shell triggered an alarm. Furthermore, the system correctly tagged the log-file on the proxy server. Consequently, only the tagged log-file must be analyzed in order to create an attack signature for an intrusion detection / prevention system.

5 Conclusions and Future Work

As can be concluded from our experiment with a public domain honeypot implementation, the common assumption that all traffic destined to a honeypot host represents an attack does not hold in reality. In order to increase the significance of recorded traffic for intrusion detection, we therefore propose a honeypot architecture that combines filtering of known attacks with session-individual logging and evidence gathering on a honeypot host. Our honeypot in fact combines the high interaction honeypot idea with a host based intrusion detection system that is based on supervision of system calls. As most attacks coming in from the Internet sooner or later try to perform a “suspicious” action like starting a shell process, accessing specific files, etc. our approach is able to operate effectively even with rather simple signatures of “suspicious” behavior. By marking the session logs of traffic streams that resulted in suspicious behavior, a significant reduction of traffic to be evaluated is attained, so that forensic analysis of recorded traffic is considerably simplified. In our future work, we plan to improve the ability of our system call supervision functionality to track state (e.g. allowing incremental evidence gathering as compared to simple call sequences), in order to be able to specify more complex behaviors as suspicious. Furthermore, the logging proxy could be augmented with further measures to reduce the amount of input data to forensic analysis by aggregating instances of identically looking attacking sessions.

References

1. Vade 79. Xdune an exploit for the Dune http server. http://downloads.securityfocus.com/-vulnerabilities/exploits/xdune.c, 2003.
2. Andre von Raison and Lukas Grunwald. Wireless Honeypot auf der Cebit, Messe-Trend Mobile Hacking. *iX*, 5:16, 2003.
3. Baris. Dune. http://freshmeat.net/projects/dune/, 1999.
4. Jay Beale, James C. Foster, Jeffrey Posluns, Ryan Russell, and Brian Caswell. *Snort 2.0 Intrusion Detection*. Syngress, 2003.
5. W. La Cholter et al. IBAN: Intrusion Blocker based on Active Networks. In *Proc. of Dance 2002*.
6. Fyodor. The art of port scanning. *Phrack Magazine*, 7, 1997.
7. A. Hess, M. Jung, and G. Schäfer. FIDRAN: A flexible Intrusion Detection and Response Framework for Active Networks. In *Proc. of 8th IEEE Symposium on Computers and Communications (ISCC'2003)*, July 2003.
8. A. Hess and G. Schäfer. ISP-Operated Protection of Home Networks with FIDRAN. In *First IEEE Consumer Communications and Networking Conference (CCNC'2004),*, January 2004.
9. A. Hess and G. Schäfer. Realizing a flexible access control mechanism for active nodes based on active networking technology. In *IEEE International Conference on Communications (ICC 2004)*, Paris, France, June 2004.
10. Alberto Gonzalez Jack Whitsitt. Bait'n'Switch. Technical report, Team Violating. http://baitnswitch.sf.net.
11. C. Kreibich and J. Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.
12. Jason Larsen. Hogwash. http://hogwash.sourceforge.net/docs/overview.html.
13. Vern Paxson. Bro: a system for detecting network intruders in real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.
14. The Honeynet Project. *Konw Your Enemy*. Addison-Wesley, 2002.
15. Niels Provos. Honeyd - A Virtual Honeypot Daemon. In *10th DFN-CERT Workshop*, Hamburg, Germany, Februrary 2003.
16. Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.
17. Miyake Takemori, Rikitake and Nakao. Intrusion trap system: An efficient platform for gathering intrusion related information. Technical report, KDDI R and D Laboratories Inc., 2003.
18. Xfocus Team. X-scan version 3.1 english. http://www.xfocus.org, 2004.