IDS/A: An Interface between Intrusion Detection System and Application

Andrew Hutchison and Marc Welz

Data Network Architectures Laboratory
Department of Computer Science
University of Cape Town
Rondebosch, 7701
Republic of South Africa
{hutch,mwelz}@cs.uct.ac.za

Abstract. We describe a number of problems which may reduce the effectiveness of a conventional network intrusion detection system. These problems are the result of the IDS having to second-guess the components or applications it is protecting. We propose a bi-directional interface between IDS and application. Applications use this interface to describe their state and submit their actions for approval to the IDS. The IDS thus receives information first-hand and is able to block suspect actions immediately. Apart from eliminating some of the above-mentioned problems, the interface may also make it possible to extend the uses of intrusion detection systems to such tasks as gradual, continuous authentication and automated least privilege enforcement.

Keywords: Intrusion Detection, Auditing, Logging.

1 Introduction

Those who deploy and administer intrusion detection systems take the view that it is next to impossible to build and maintain real-world systems which are completely secure.

Intrusion detection systems are thus intended to discover agents who attempt or have managed to bypass those security mechanisms (systems as well as policies) which in an ideal system would have been infallible. See [5, 7] for an introduction and [14] for a list of intrusion detection systems.

Conventional intrusion detection systems operate by examining audit logs or, more frequently, monitoring interactions between system components at a suitable interface. Popular interfaces include the operating system call interface or the network interface.

Using information intercepted at an existing interface for the purpose of detecting intruders has a number of advantages — the approach is:

– Cost effective: Since the interface already exists, the components to be monitored do not need to be modified, reducing the cost of deploying an IDS. Often the interfaces provide debugging or profiling facilities which can immediately be used

for data gathering purposes. For example operating systems may provide means to intercept system calls (eg Unix `ptrace`), while the promiscuous mode of ethernet devices is suitable for intercepting network traffic.

– Unobtrusive: Intercepting traffic at an interface can often be done passively and without performance loss. The classic example is that of a network sniffer — the sniffer need not generate any traffic of its own nor should it degrade performance of the rest of the network. Not all interfaces can be monitored this cheaply and unobtrusively, for example intercepting systems calls can degrade system performance and may be visible to applications. Usually however, it is possible to monitor an interface without providing an intruder with clues that he is being monitored.
– Easily understood: Interfaces tend to be better defined than the internals of the components. The amount of information exchanged between components (inter component interaction) also tends to be less than the amount of information handled inside a given component (intra component activity). This reduces the workload on the monitor at the interface.

Unfortunately monitoring existing interfaces also has a number of disadvantages:

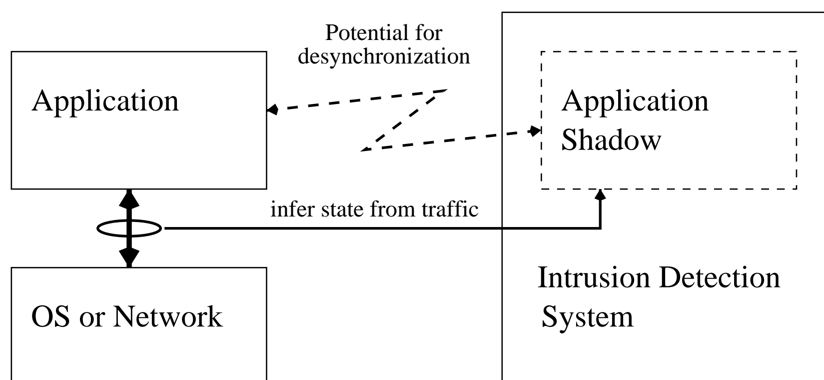

Fig. 1. The problem of maintaining a synchronized shadow of the internal state of an application by monitoring its traffic.

– Interfaces may provide for stateful interactions between components. These may be maintained for extended periods of time. For example, hosts communicating via TCP/IP have to allocate resources for each connection, while the UNIX system call interface uses file descriptors to encode state. An IDS needs to allocate duplicate resources to shadow this state — it has to remain synchronized with the components being monitored (see Fig 1). This can be a difficult task, especially in heterogenous environments where different implementations may vary subtly. In such cases the interface handling subsystem of the IDS may have to be significantly more complex to cover all permutations.

- External factors may influence the characteristics of the interface and application. An IDS may need to keep track of these factors, since resourceful intruders may find ways of exploiting the altered characteristics of the interface.
  For example, an IDS in an IP network may have to be aware of the network topology. Otherwise it may be possible to desynchronize the IDS with the hosts it is guarding. An example of such a condition may occur when an IDS is placed in front of hosts behind one or more gateways — an intruder can send a TCP packet to close connection with a TTL counter one short of the target host. This packet never reaches the host, but the IDS which shadows the connection is mislead into closing its shadow. See [11] for a detailed description of this and other methods which may be used by intruders to evade detection in a TCP/IP network.
- Some interfaces need to operate at high speeds: Throughput has to be maximized and latency minimized. Passive monitors are at a disadvantage in such environments since they have only limited resources (time and memory) at their disposal to detect suspicious activity — they are unable to delay a message while it is being examined. Intrusion detection system monitoring network interfaces are particularly susceptible to such problems, since these often have to monitor traffic generated by several hosts and switched by specialized hardware, while the IDS is typically executing only on a general purpose processor.
- Interfaces can be layered one on top of the other with an opaque mapping from one to the other. An IDS which monitors the lower one may not be able to derive very much information about the activities occurring at higher levels and vice versa. An example would be that of a database which maps an SQL interface to a UNIX system call interface. An IDS which monitors system calls is unlikely to detect an unauthorized query.
- Finally the most serious difficulty is that interfaces are being secured cryptographically. These measures are explicitly designed to prevent third parties from intercepting traffic which is exactly how most intrusion detection systems operate. Cryptographically protected interfaces are likely to become more common. Currently encrypted interfaces are more or less limited to network interfaces, but it appears likely that such systems as digital content managers, smart card resident applications or agents participating in electronic commerce are likely to extend cryptographic protection to other interface types.

2 Example

In order to illustrate the points in the previous section we will present a short example of how the difficulties of monitoring an interface can be exploited by an attacker. In the example the attacker uses a vulnerability in a CGI script to display the UNIX password file.

As example we have chosen the `phf` CGI script. While the `phf` vulnerability is somewhat dated, it remains good representative.

The request submitted by the attacker to the web server looks as follows:

```
GET /cgi-bin/phf/?Qalias=x%0a/bin/cat%20/etc/passwd
```

A number of network intrusion detection systems contain a signature for the `phf` vulnerability and should thus detect this attack. In order to evade detection an attacker could make the following changes:

Fragment the request : The intruder could split the request over several small packets. Intrusion detections systems which do not reassemble fragments (ie do not attempt to shadow state) will not detect this modified attack.
Intrusion detection systems which do reassemble packets might possibly drop the initial fragment if the attacker were to generate large amounts of background traffic or wait some time before sending the second fragment (state holding attacks).

Desynchronize the IDS with target : If the TCP/IP implementation of the IDS is incomplete, it may be possible to send intermediate fragments with corrupted checksums or invalid sequence numbers.

```
GET /cgi-bin/[test-env HTTP/1.0^M^JUser-Agent: ]phf
/?Qalias=x%0a/bin/cat%20/etc/passwd
```

In the above example the attacker distributes the request over three packets, where the middle packet (boxed) is intended to be discarded by the target (eg because of a bad checksum), but which changes the request seen by the IDS.
Even if the TCP/IP implementation of the IDS is of a good quality, it may be possible to desynchronize the IDS: Assume that the IDS is 3 hops in front of the web server and 6 hops away from the attacker. In this case the the attacker could establish the TCP/IP connection and send send a FIN packet with a TTL of 8. The FIN packet is seen by the IDS, and the IDS closes its shadow, but the packet never reaches the web server since its time to live counter expires before it reaches its final destination. [11] provides a comprehensive description of this and other evasion techniques.

Exploit translations : It is possible to rewrite the request, both by taking advantage of encoding schemes and the file system which stores the script. For example, both the strings `/cgi-bin/./phf` and `/cgi-bin/%70%68%66` are equivalent to `/cgi-bin/phf`. An effective intercepting IDS needs to test for all permutations.

Use a fast attack : Instead of displaying the password file with `cat`, the attacker could mail the password file to a throwaway account. This type of attack may only require a single packet to succeed — an IDS which responds to such attacks by killing the connection or adding a firewall rule may very well be too late.

Access `phf` over an encrypted channel : Instead of accessing the script via HTTP, the intruder could use HTTPS, the SSL enabled version of the protocol. Since a NIDS won't have access to the plaintext, it will be unable to detect this attack variant at all.

Some of the problems outlined above are the result of incomplete or poorly implemented network intrusion detection systems. These problems are being corrected — NIDS exist which provide solid TCP/IP implementations and which understand higher-level protocols. Reportedly systems are being developed which even take the network topology and operating system variations into account [12].

However, the long term trend is toward systems with ever higher bandwidth and lower latency which suggests that conventional IDS (and especially NIDS) will remain vulnerable to fast attacks or attempts to exhaust IDS resources.

Cryptographically secured interfaces present the greatest challenges to conventional intrusion detection systems. Without changes an IDS monitoring an encrypted interface is at best reduced to performing traffic analysis. Conceivably key escrow systems could be used to allow the IDS access to the plain text, but this would make the IDS itself a highly attractive target for attackers and reduce user confidence in integrity of the cryptographic channel.

3 IDS/A Overview

In light of the above difficulties and trends, we have decided to design a dedicated interface between IDS and application[1]. In our system the IDS no longer attempts to intercept communications between components, but instead communicates directly with the components it is supposed to protect.

In this regard our interface resembles a logging or auditing interface.

Like the interface of the conventional Unix `syslog` or `XDAS` [8], our interface is visible to the application programmer as a set of library calls. Calls made to this library are preprocessed in the application context before being transformed into a representation suitable for transmission to the IDS. Messages received from the IDS are processed in a similar manner. See Fig. 2.

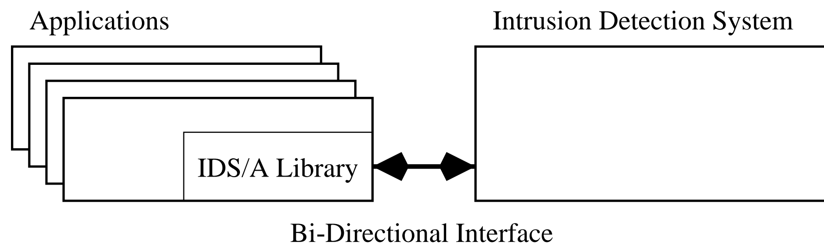

Fig. 2. IDS/A Interface Overview

However, our interface also differs from a normal logging interface in a number of ways:

Unlike a logging interface which merely records past events, our system is a bi-directional interface designed to have applications submit pending events or actions to the IDS for approval. The IDS has thus an opportunity to block suspect events before they occur. In other words the interface makes it possible for the IDS to function as reference monitor as well as logger.

[1] We use the word application in the wider sense. NFS servers and ftp daemons are included in our definition of application.

Our system also differs from conventional auditing subsystems in another way: The interpretation of conventional audit records usually requires domain specific information. Our interface provides a fallback mechanism: In addition to domain specific information, the application can indicate the cost of allowing an event to system availability, confidentiality and integrity. This domain-independent information may be used by an IDS, even if it does not have access to domain knowledge, to decide if a request should be granted. For example, if an IDS were to detect anomalous activity it might enter a paranoid mode where it would only grant requests which carry a low risk of breaching confidentiality, while high risk events would be blocked at the cost of decreasing availability.

In this paper we neither concern ourselves with the transport mechanism between IDS and application nor with the IDS itself, except for mentioning that our current prototype uses Unix domain sockets as transport to a small rule-based IDS which runs under Linux.

We envisage that once the interface has been established it will be possible to have multiple implementations providing the same IDS/A interface to applications, but that these implementations will use different transports and IDS types. For example, one implementation might use a network transport to a centralized misuse detector, another a local transport to an anomaly detector. These differences would be transparent to applications, and in cases where applications are dynamically linked to the IDS/A client library, it should be possible to substitute implementations without having to modify the applications binary at all.

4 IDS/A API

Our API consists of a number of library calls which can be grouped in to the following categories:

Session Management: These function calls are used to set up and tear down the interface between application and IDS. The initialization function provides the IDS with the name of the service or application, as well as a credential[2].

Logging: The logging functions form the core of our system.

Our logging or reporting functions record requests for permission to process certain events or perform certain actions — as part of the return code the IDS indicates if it is permissible to perform the task. This differs from conventional logging functions which record events or actions after they have occurred and whose return code merely indicates whether it was possible to record the log entry.

Our logging functions accept several parameters. The most important ones are described below:

- An event identifier or *name* which distinguishes between different events generated by the same applications. Identifiers are assigned by the application designer. For example, each command received by an ftp server (eg `USER`, `PASS`, `STOR`) might have a different identifier.

[2] Our prototype ignores the credential, instead it queries the operating system directly to establish the owner of the process.

- A *namespace or scheme* which permits the application designer to aggregate related events. For example, this field may be used in an ftp server to indicate that it shares a number of events with a related web server.
- A set of *cost or risk ratings* which the application designer may use to quantify the potential risks of a given action in a domain independent manner. An action can be rated according to its impact on system or data availability, integrity and confidentiality. For example, a request to drop a database table would pose a higher risk to data availability and integrity than a request to display its content. However, displaying its content would carry a greater risk of breaching confidentiality than deleting it.
- A *flag* indicating how the application intends to handle an IDS return code which would disallow the request. This provides a mechanism for the application designer to indicate circumstances under which a given event can or should not be disallowed but should be logged.
- A *textual description* or comment intended for humans monitoring the IDS.
- A number of internal fields which are set inside the library (and verified by the IDS) and not intended to be modified by the caller. These include time, process owner and process identifier.
- A number of fields to provide *additional, event specific information*. These fields are represented as a typed attribute value pairs. For example, a database server may use these fields to log the name of a table to be created, or a login program may provide fields which specify the username and terminal used for a login attempt.

Note that for the sake of convenience the parameters can be collected into a composite structure before being passed to the logging function. Also note that some of the individual fields described above can be composites.

Debugging and Profiling: These functions can be used to examine the internals of our library and track down errors.

Miscellaneous: The IDS/A library provides a number of utility functions. For example, there exist functions to handle the creation, updating and deallocation of structures to hold the parameters which are submitted to the logging functions.

5 Example

In this simplified example we show how a toy web server might make use of the IDS/A library to report the execution of CGI scripts. Error checks have been omitted.

```
1 IDSA_CONNECTION *c;
2 ISDA_EVENT *e;
3 char *script,*hostname;
4
5 /* retrieve the request */
6 script=httpd_parse_cgi();
7 hostname=httpd_remote_name();
```

```
 8
 9 /* initialize the IDSA interface */
10 c=idsa_open("toy-httpd",NULL,IDSA_F_FAILOPEN);
11 e=idsa_event();
12
13 idsa_name(e,"execute-cgi");
14
15 idsa_add_scan(e,"client-name",IDSA_T_HOST,hostname);
16 idsa_add_scan(e,"script-name",IDSA_T_FILE,script);
17
18 idsa_risks(e,1,IDSA_V_UNKNOWN,IDSA_V_HIGH,IDSA_V_HIGH);
19
20 /* if IDS permits run CGI */
21 if(idsa_log(c,e)==IDSA_L_OK){
22   http_run_cgi(p);
23 }
```

A brief description of the above C fragment follows:

Lines `6` and `7` are functions provided by the web server which retrieve the name of the script and host making the request.

Line `10` opens a connection to the IDS. The application reports itself as `toy-httpd` with a null credential. In the event that the IDS is unreachable, the all events should be allowed (this behaviour can be changed at runtime by the IDS).

Line `11` sets up an event structure, and sets up reasonable defaults for a number of parameters, hence these fields are not set up subsequently.

Line `13` sets the name of this event (`execute-cgi`).

Line `15` inserts the name of the host making the request into the event structure, while line 16 provides the file name of the CGI script to be executed.

Line `18` indicate that this action is a high risk operation: It could result in breach of confidentiality or failure of system integrity.

Finally line `21` submits the request to the IDS for approval. If the request is granted, line `22` runs the CGI program.

An attacker who attempts to misuse a vulnerable CGI script in this system has significantly fewer chances of avoiding detection: Desynchronization attacks are not an option since there exists a direct interface between application and IDS. Requesting the script via an encrypted channel such as HTTPS will also not hide the attacker. Attempts to overload the web server in the hope of having the IDS fall behind in processing are also likely to be unsuccessful since the application waits for the IDS to return a result. Fast attacks can be prevented since the IDS is consulted before the script is executed. Even an IDS which lacks domain knowledge might block the request if it is anomalous as it is tagged as having a high risk to confidentiality and integrity.

6 Analysis

Philosophically it is appealing to get the designers and builders of applications involved in the business of detecting intruders. Often application programmers are considered to be the source of security problems. But since they seldom deliberately set out to create insecure systems, we feel that it may be worthwhile to make them part of the solution. We think that this approach offers a number of potential advantages:

- The designer of an application should have a good notion of what features are important and hence worth reporting. Furthermore applications may provide their own models which may differ from the one used by the underlying components. For example, a database server operates on tables and relations, while the underlying operating system works with disk blocks. Since log messages describe the application in terms of their model, the application designer is effectively enlisted to perform the task of feature selection on behalf of the intrusion detection system. Feature selection can improve the performance of anomaly detection system significantly [10].
- Communicating directly with an application eliminates the resources required to mirror application state. For example, it is no longer necessary for an IDS monitoring a web server to allocate resources to shadow the TCP/IP connections.
- The potential to desynchronize the IDS from the application is eliminated, since the application reports events directly to the IDS. This also reduces the long-term maintenance costs of the IDS since its no longer necessary to keep up with different application revisions.
- Our approach remains feasible in cryptographically secured environments, since an IDS in our system does not have to intercept traffic intended for third parties. Securing the IDS/A interface itself is a straightforward task which could make use of conventional cryptographic techniques in cases where the transport mechanism is deemed to be unreliable.
- Since the IDS is able to communicate directly with the application it becomes possible to disable features selectively and cleanly. This is in contrast to current approaches which rely on rather coarse methods such as killing processes or terminating network connections.
- Our library makes it easy to select between a fail open or fail closed policy[3]. A fail closed policy can be enforced by having the logging function prohibit all events when the IDS is unavailable, while a fail open policy will have all logging calls succeed. Conventional intrusion detection systems which passively intercept traffic are inherently fail open, restricting site policies.
- Our approach deteriorates gracefully under high load. If the IDS becomes overloaded, it may react by either throttling an application with delayed responses, or by instructing the library in the client context to drop unimportant messages — in other words it is possible to perform preselection inside the library calls. This is efficient since it happens in the context of the application, but remains transparent to

[3] An example of a fail open policy is when an electric door opens during a power failure, whereas a fail closed policy would keep the door closed. As in the physical world it is not always clear which policy should be selected for a given application.

the application programmer. It is even conceivable that a future version our library might allow the IDS to upload intrusion detection code into the library to distribute its load across applications and improve robustness. In contrast, conventional passive traffic monitors have no way of throttling the traffic at an interface and usually discard arbitrary messages if overloaded.

Apart from the direct advantages listed above, involving the application designer has also indirect benefits. The benefits are social — by making the IDS facilities available to the application programmer (as part of the system infrastructure), he is made aware of the security implications of his work.

In cases where an application designer uses the IDS/A interface to provide a sound description of system activity and associated risks or costs, the task of those implementing and maintaining an IDS could be simplified greatly — data acquisition is made easier, and with a good risk description it becomes possible to rely on the (possibly superior) domain knowledge of the application designer instead of duplicating his effort in understanding the behaviour of the application.

Unfortunately our system also has a number of disadvantages:

- There is a significant upfront cost: Modifying existing applications requires time and effort. In cases where a software vendor is not prepared to make the changes and source is not available, it becomes impossible to use our interface[4].
- Our system relies on the application to report events accurately and in sufficient detail. If the application is unable to perform this task, the IDS becomes ineffective.
- Conventional passive traffic monitors can be made entirely invisible. The IDS/A interface is visible to the application and may thus attract the attention of an attacker who may attempt to overload or mislead the IDS.
  This may not be as significant a problem since it is also possible to overload or mislead a passive monitor, even if it is completely invisible. Furthermore, it should be possible to regulate access to the IDS/A interface more tightly than to other interfaces since normal users have no business accessing it. In this regard the IDS/A interface is similar to the separate signalling channels which exist in public telephone networks and which have been mooted for the internet [13].
  We have used such an approach in our prototype — its interface is only available via a Unix domain socket and thus not accessible from the network. So, for example, if our IDS guards an http server, then a remote attacker first needs to break into the host by other means before being able to access the IDS/A interface[5]. This differs from a conventional NIDS intercepting traffic destined for the web server — it is immediately accessible to the attacker who may have legitimate reasons for accessing the network and may use this as a pretext for manufacturing traffic to desynchronize the NIDS.
  Finally, we have designed our interface in such a way that the applications initiate contact with the IDS and not vice versa. This makes it more difficult for an intruder

[4] Hence our choice of Linux as the platform for our prototype implementation — it is an environment where most applications programmers provide source code.

[5] But once the attacker has broken into the host, attacking the web server usually becomes uninteresting.

to gain control of an application — the application is only vulnerable to a spoofed IDS at startup. In contrast the components of systems such as SNMP are always vulnerable to imposters.

7 Related Work

Our approach can be viewed in a number of different ways.

It can be thought of as being an extension of conventional loggers such as `syslogd` or `XDAS` [8]. In these systems the data flow is unidirectional, messages are emitted by the application and captured by the logger, whereas our system is bi-directional — our logger (actually IDS) is also able to send messages to the application. Another difference between our system and a classical system logger is that the messages of the latter are primarily intended to be a permanent record to be examined by humans, while our messages are intended to machine readable. Suggestions to make logs machine readable can be found in [3, 1].

Another view of our system is that of an optional, adaptive and extensible reference monitor. Applications voluntarily register with our reference monitor (actually IDS) and provide their own list of subjects, objects and operations most suited to their model. The reference monitor examines the access matrix for access patterns which are either anomalous or indicative of misuse and alerts the application which voluntarily blocks the problematic operation. Viewed from this perspective our system can be seen as a generalization of the tcpwrapper [16] suite.

Finally our interface can be related to the protocols which are being developed to handle the communications between intrusion detection modules.

A number of these protocols are intended to allow intrusion detection components of different vendors to interoperate, and provide support for locating, authenticating and managing IDS components in large networks, as well as propagating alerts and attack descriptions. These include the IDGW [6] and the CIDF [15] group.

Other groups have been developing their own protocols as part of their distributed intrusion detection systems. An example of the latter is AAFID [2].

In general these systems acquire information by examining audit trails or by intercepting traffic in the conventional manner. However, it is conceivable that some could make use of the IDS/A interface by providing their own implementation of the IDS/A client library which would use their protocols to communicate with their IDS implementation.

8 Future Work

The IDS/A interface can be thought of as making the services of an IDS available to the application programmer — thus the IDS not only becomes part of the security or network infrastructure as suggested in [17] but part of the tools available to any programmer. This means that the IDS can be applied to a wider number of tasks, including the following:

8.1 Progressive Authentication

Normally the authentication phase is an all or nothing event. If authentication succeeds, the user is granted full access, otherwise no access is granted.

The IDS/A interface makes an alternative possible — an IDS using an anomaly detector might compare the current behaviour of a user to his stored profile, and use the closeness of the match between current and past behaviour to adjust the access rights of the user accordingly.

So, for example, an attacker using the mail client of the CEO from Alaska after hours would not fit the profile and might be granted minimal right to read mail, but would be unable to delete messages. If it had turned out that the CEO had been on a fishing trip to Alaska, then this approach would have let essential mail through.

This topic is closely related to the field of intrusion tolerance as described in [4].

8.2 Automatic Least Privilege Enforcement

Using the IDS/A interface it would be possible for an IDS to enforce a "if you don't use it you lose it" policy, whereby a user would use the rights to a particular privilege if it had not been exercised. For example, a complex word processor might slowly leak advanced features when used by a naive user. At some stage the naive user might lose the capability to write self-replicating macros. The naive user would not be aware of this loss, but might appreciate the sudden decline in macro virus infections.

We have described this and the previous application of an application integrated with a profiling module in [9].

8.3 Weak Program Verification

While it is usually too expensive to perform a formal verification of an application, it is often possible to make certain statements about a correct program. For example, a certain ftp server should never make the transition from the state `wait-for-username` to `user-authenticated` without first entering the `wait-for-password` state. If such a set of assertions is provided to an IDS using the IDS/A interface, then the IDS could be used to verify the correctness of the program and prevent it from entering an incorrect state.

9 Conclusion

In this paper we have presented a simple and direct interface between IDS and application. This interface allows the IDS to monitor an application directly and block suspicious actions immediately.

Our approach has a number of advantages over intrusion detection systems which monitor applications indirectly and attempt to second-guess their state. We have described these advantages, along with the disadvantages of our approach. Fortunately, with the ascendancy of open source software, one of the most significant factors hindering the development of our method, the inability to modify applications, is falling away.

We have tried not to be prescriptive and have only specified the C API, leaving the lower transport layers as well as the IDS itself to those implementing the system. We have also deliberately tried to keep the IDS/A interface simple, since we believe that large, complex systems are more likely to fall to attackers.

Despite the simplicity of the IDS/A interface, it offers a number of interesting research opportunities which we are pursuing.

References

1. J. Abela, T. Debeaupuis, and E. Guttman. Universal format for logger messages. http://www.hsc.fr/gulp/, 1997.
2. J. Balasubramaniyan, J. O. Garcia-Fernandez, E. H. Spafford, and Zamboni D. An architecture for intrusion detection using autonomous agents. Technical report, COAST Laboratory, June 1998.
3. M. Bishop. A standard audit trail format. In *National Information Systems Security Conference*, pages 136–145, October 1995.
4. C. Cowan and C. Pu. Death, taxes, and imperfect software: Surviving the inevitable. In *ACM New Security Paradigms Workshop*, pages 54–70, September 1998.
5. D. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2), February 1987.
6. M. Erlinger, S. Staniford-Chen, et al. IETF intrusion detection working group. http://www.ietf.org/html.charters/idwg-charter.html, 1999.
7. R. Graham. Network intrusion detection systems frequently asked questions. http://www.robertgraham.com/pubs/network-intrusion-detection.txt, 1998.
8. The Open Group. *Distributed Audit Service (XDAS) Base*. The Open Group, 1997.
9. A. Hutchison and M. Welz. Incremental security in open, untrusted networks. In *Future Trends in Distributed Computer Systems*, pages 151–154, November 1999.
10. W. Lee, S. J. Stolfo, and K. Mok. Mining audit data to build intrusion detection models. In *International Conference on Knowledge Discovery and Data Mining*, September 1998.
11. T. H. Ptacek and T. N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, 1998.
12. M. Roesch. IDS list. http://msgs.SecurePoint.com/cgi-bin/get/ids-0003/7/2/2/2/1.html, March 2000.
13. B. Schneier. Crypto-Gram newsletter. http://www.counterpane.com/crypto-gram-0002.html, 2000.
14. M. Sobirey. Intrusion detection systems page. http://www-rnks.informatik.tu-cottbus.de/ sobirey/ids.html, 1995.
15. S. Staniford-Chen, Tung B., et al. Common intrusion detection framework page. http://www.gidos.org/, 1998.
16. W. Venema. TCP wrapper, network monitoring, access control and booby traps. In *UNIX Security III*, September 1992.
17. M. Wood. Integrating intrusion detection into the network/security infrastructure. In *Workshop on Recent Advances in Intrusion Detection*, September 1998.