Application-Integrated Data Collection for Security Monitoring*

Magnus Almgren and Ulf Lindqvist

System Design Laboratory, SRI International
333 Ravenswood Ave, Menlo Park CA 94025, USA
{almgren,ulf}@sdl.sri.com

Abstract. This paper describes a new approach to collecting real-time transaction information from a server application and forwarding the data to an intrusion detection system. While the few existing application-based intrusion detection systems tend to read log files, the proposed application-integrated approach uses a module coupled with the application to extract the desired information. The paper describes the advantages of this approach in general, and how it complements traditional network-based and host-based data collection methods. The most compelling benefit is the ability to monitor transactions that are encrypted when transported to the application and therefore not visible to network traffic monitors. Further benefits include full insight into how the application interprets the transaction, and data collection that is independent of network line speed. To evaluate the proposed approach, we designed and implemented a data-collection module for the Apache Web server. Our experiments showed that the required implementation effort was moderate, that existing communication and analysis components could be used without incurring adaptation costs, and that the performance impact on the Web server is tolerable.

Keywords: Intrusion detection, application, application-integrated, module, Web server, Apache

1 Introduction

Intrusion detection systems (IDSs) can be categorized with respect to several different dimensions, of which the commonly used (but somewhat oversimplified) dichotomy between *misuse detection* and *anomaly detection* is one example. Another dimension for categorization is the type of event data analyzed by the IDS. An IDS that monitors traffic flowing in a network is usually called *network based*, while an IDS that analyzes data produced locally at a host is often referred to as *host based*.

* The work described here is currently funded by DARPA/ATO under contract number F30602-99-C-0149 and contract number F30602-98-C-0059. The views herein are those of the author(s) and do not necessarily reflect the views of the supporting agency.

Copyright © 2001 Springer-Verlag Berlin Heidelberg

We subdivide the host-based category further, depending on the abstraction level at which data is collected. Most existing host-based systems gather audit data at the operating system (OS) system-call level, but an IDS could get its data from higher as well as lower abstraction levels. Below the OS level, we could, for example, look at the executed processor instructions. Above the OS level, we could collect data from service applications such as database management systems, Web servers or e-mail daemons, or from end-user applications. As different types of security violations manifest themselves on different levels in a system, one could argue that it is important for the IDS to collect data at the most meaningful abstraction level(s) for the event in question. It should be kept in mind that independent of the type of data collected, it can be sent to any type of analysis engine (e.g., signature based, model based, probabilistic).

In this paper, we focus on collection of data produced by applications (above the OS level) and refer to an IDS analyzing such data as *application based*. Although the concept of application-based IDS is not new, there is a striking absence of commercial IDSs for applications other than firewalls [6]. The approach presented in this paper shows how the data collection for an application-based IDS can be integrated with the monitored application.

The remainder of this paper is organized as follows. Section 2 discusses limitations of network-based and host-based IDSs, respectively. In Section 3, we present our application-integrated approach, and discuss its advantages and how it complements the other methods. Section 4 describes an implementation for a Web server to validate our reasoning. In Section 5, we examine the performance characteristics of the implementation. Section 6 describes related work, while ideas for future work are outlined in Section 7. Our conclusions are summarized in Section 8.

2 Background

Many researchers have recognized that there is no single "silver bullet" approach to automatic detection of security violations. By combining and integrating complementary approaches, better attack space coverage and accuracy can be achieved. In this section, we look at some specific problems with the network-based and host-based approaches, respectively.

2.1 Network-Based Data Collection

The popularity of Ethernet technology paved the way for network-based IDSs. When traditional broadcast Ethernet is used, a whole cluster of computers can be monitored from a dedicated machine that listens to all the traffic. As no changes in the infrastructure are required, and there is no performance penalty, most free and commercial IDSs use this approach. The system can be completely hidden by having a network card that listens to the network but never transmits any traffic itself. However, by decoupling the system from the actual hosts it supervises, we lose valuable information.

First, probably the most serious problem with the network-based approach is encrypted traffic, which in effect makes the network monitor blind to many attacks. Today, encryption is typically used to protect the most sensitive data, and there are indications that encryption will become more ubiquitous in the near future, making today's network-based monitors ineffective.[1]

Second, the IDS can be deceived in several ways. Most Internet standards in the form of RFCs carefully define valid protocol syntax, but do not describe in detail how the application should behave with deviant data. To be accurate, the IDS needs to model how the application interprets the operations, but this is almost an impossible task without receiving feedback from the application. Minor differences in operations play a major role in how they are interpreted. For example, consider a user requesting a certain Web page, by sending `http://www.someHost.com/dir1\file1` (note the backslash character). If the receiving Web server is Microsoft-IIS/5.0 under Microsoft Windows, the server looks for the file `file1` in the directory `dir1`. However, if the server is Apache version 1.3.6 under Unix, the server looks for a file named `dir1\file1` in the root directory. Even if an IDS could model the different popular servers, it cannot account for every implementation difference in every version of these. This problem is not limited to application-level protocols, but as Ptacek and Newsham [9] describe, the same goes for lower-level protocols.

Third, efforts to increase bandwidth in networks pose serious problems for network-based IDSs. Higher line speed is in itself a difficulty, and switching (unicast) technology ruins the possibility to monitor multiple hosts from a single listening point. Some IDS developers try to address this problem by placing a network data collection unit on every host. That solves this problem while introducing others, which are similar to the problems faced by host-based analysis.

2.2 Host-Based Data Collection

The host-based approach addresses some of the problems described above, with the primary advantage being access to other types of information. As it is installed on a host, it can monitor system resources as well as look at operating system audit trails or application logs. It is also independent of the network speed as it monitors only a single host. However, the system administrator now needs to install a number of monitors instead of just one, thus incurring more administrative overhead. Also, the user could experience a performance penalty as the monitor is on the same host as the application.

Furthermore, most monitors on the OS level cannot detect attacks directed at the lower network protocol levels because network information typically does

[1] There are some ways a network-based IDS could read encrypted traffic. For example, it could act as a proxy with the encrypted channel being only to the IDS (or a similar proxy), thus introducing unnecessary overhead in the form of extra programs that need to be supervised and also exposure of data before it reaches its final destination. The network monitor can also be given the private key of the server, increasing the exposure of the key and forcing the network monitor to be able to keep track of user sessions and their associated keys.

not become available in the audit event stream until it has reached the higher protocol levels. See [3, 4] for an approach to include network data in OS audit data.

An application-based monitor in the traditional sense (such as the one described in [1]) reads data from log files or other similar sources. By the time the information is written to the log, the application has completed the operation in question and thus this monitor cannot be preemptive. The information available is often also limited to a summary of the last transaction. For example, a Web request in the Common Logfile Format (CLF) is

```
10.0.1.2 - - [02/Jun/1999:13:41:37 -0700] "GET /a.html  HTTP/1.0" 404 194
```

Without going into the meaning of the different fields, the following recounts the scenario: The host with address 10.0.1.2 asked for the document `a.html`, which at that time did not exist. The server sent back a response containing `194` bytes.

The log entry does not contain all the information an IDS needs for its analysis. Were the headers too long or otherwise malformed? How long did it take to process the request? How did the server parse the request? What local file did the request get translated into?

In some applications, logging can be customized and contain much more information. Nevertheless, we have not yet seen a system where all internal information needed to understand the interpretation of an operation is available for logging. Furthermore, by turning on all log facilities, we increase the risk of running out of storage space for the logs and incurring performance degradation.

3 Application-Integrated Data Collection

As we have shown in the previous section, there are problems associated with both network-based and host-based approaches. Some of these can be solved by collecting data directly from the single critical application that we want to monitor. In this section, we present the general principles of this approach, while Section 4 describes a prototype implementation for monitoring Web servers.

3.1 Rationale

Today's network structure within organizations makes a few applications critical. These need to be available around the clock from the outside of the organization; they are sensitive to attacks but seldom sufficiently protected. Examples include Web servers, e-mail servers, FTP servers, and database systems.

To minimize security concerns, most such critical applications run on dedicated machines and are protected by firewalls. Typically, no other application is running on these machines. If remote login is allowed, it is very restricted (such as only ssh). Thus, the malicious user must go through the channels of the critical application to subvert the host. By having the IDS monitor the inner workings of the application, analyzing the data at the same time as the application interprets it, we have a chance of detecting malicious operations and perhaps even stopping them before their execution. However, for us to successfully integrate a monitor into the application, the application must provide an interface. Some applications provide an API and, as the advantage with an application-integrated monitor becomes clear, we hope that more vendors will provide such interfaces. Other venues for integration are found in the open-source movement.

3.2 Advantages

Access to unencrypted information. In almost all cases, data must be accessible inside the application in unencrypted form for meaningful processing, even if it is encrypted in lower layers. Consequently, the unencrypted data is also accessible to an application-integrated data collection module. This is a major advantage compared to a network-based IDS. Moreover, it should be noted that because encryption is used for the most sensitive data, the functions handling that data are probably among the most interesting from an attacker's point of view and therefore important to monitor.

Network speed is not an issue. The module is part of the application, and takes part in the normal processing cycle when analyzing operations. Thus, the limiting factor is the application speed rather than the network speed. For example, if the original application can accept a certain number of connections per second, the application equipped with the module must be able to perform equally well. Care must be taken so that the module does not become a bottleneck and does not consume too many of the host's resources. We discuss our solution to this problem in detail in the next section.

More information available. Being part of the application, the module can access local information that is never written to a log file, including intermediate results when interpreting operations. It can monitor how long it takes to execute a specific operation, to detect possible denial-of-service (DoS) attacks. Furthermore, we expect an application-integrated monitor to generate fewer false alarms, as it does not have to guess the interpretation and outcomes of malicious operations. For example, a module in a Web server can see the entire HTTP request, including headers. It knows which file within the local file system the request was mapped to, and even without parsing the configuration file of the Web server, it can determine if this file will be handled as a CGI program (not otherwise visible in either network traffic or log files).

True session reconstruction. Information of interest to an IDS often concerns transactions (request–response) and user sessions. To extract that information, a network-based monitor must perform expensive reconstruction of transactions and sessions from single network packets, and it is still not guaranteed to correctly mimic the end-point reconstruction. In contrast, the application-integrated module is handed complete transaction and session records directly from the application, and there is consequently no discrepancy between the interpretations.

The IDS could be preemptive. IDSs are at times criticized for being of limited use as they can only report intrusions, usually too late to stop any damage. By being part of the application, the module could supervise all steps of the processing cycle and could react at any time. For example, it could deny a single operation that appears malicious without otherwise compromising server performance.

3.3 Disadvantages

The disadvantages of the application-integrated monitor coincide with some of the disadvantages of the host-based monitors in general, as described in Section 2.

Any monitoring process running on the same host as the monitored service risks to impact the performance of the server. It is therefore important that a goal in the monitor design and implementation is to minimize this performance impact.

Given that one needs to have a distinct application-integrated monitor for every single type of application one wants to monitor, the development efforts could be significant. However, in today's situation where a handful of products dominate the field of network server applications, the efforts and costs required for a satisfactory coverage could be lower than they might first appear. This is particularly true for applications that are open source and/or provide an API for modules.

The application-integrated monitor can only be a complement to other types of IDSs. As it is part of the application, it sees only the data reaching the application. By targeting a protocol below the application layer, an attacker could evade detection by our module, but would be within the scope of a network-based IDS or possibly another host-based sensor specialized in lower-level protocols.

4 Design Principles and Implementation

As discussed in the previous section, current network infrastructure makes a few applications critical. Of these, the Web server stands out as being both ubiquitous and vulnerable. First, most organizations need a Web server, and it is the service most users associate with the Internet. Second, even though the server software might be as stable (or unstable) as other types of software, many sites have customized programs connected to the server allowing access to legacy database systems. Because these programs are easy to write, they are often developed by junior programmers who do not properly consider the security aspects of letting any user in the world supply any input to programs run locally. As a result, new vulnerabilities in CGI programs are discovered daily. Furthermore, the Web server is among the first to be probed during a reconnaissance. The vulnerabilities are easy to comprehend (e.g., add a semicolon after the request in your browser), and the gratification is instant (change the Web pages and all your friends can admire your deed).

For these reasons, we chose to focus our prototype on a Web server. The major Web server brands also have an API which provides the capabilities we desire for application-integrated event data collection. The remaining question is which server product to target.

Netcraft continuously conducts a survey of Web servers on the Internet [7]. Some results from the February 2001 survey are shown in Table 1. It shows that there are three main players in the market covering more than 85%. We decided to build our first application-integrated data collection module for the Apache Web server, as it is the most popular one.

Table 1. Market shares for the top Web server products [7]

Product	Developer	Market Share
Apache	Apache	60.0%
Microsoft-IIS	Microsoft	19.6%
Netscape-Enterprise	iPlanet	6.2%

The market penetration for SSL is very different. Unfortunately, the data is considered commercially valuable and is available as a commodity only for a prohibitively high price. Furthermore, the Netcraft survey counts each site equally. This reflects neither the popularity of the site nor the risk and associated cost of attacks. We are not aware of any other survey of this kind, and even if the numbers above are subject to discussion, there is no doubt that Apache has a large share of the market.

4.1 Implementation

As it turns out, it is quite easy to extend the Apache server with our data collection module, primarily due to the following reasons.

– There is a well-defined API for modules and the data concerning each request is clearly distinguishable in distinctive data structures.
– Each request is processed in several stages, and there are so-called hooks or callbacks in each of these stages that a module can use (see Fig. 1).
– After each stage, the module can give feedback to the server as to whether it should continue executing the request.
– There is support for extensive logging capabilities through the reliable piped logs interface (explained below).

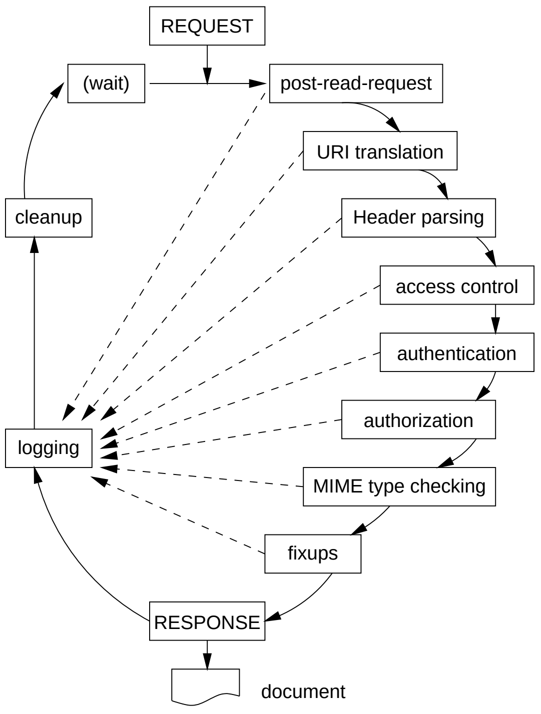

Fig. 1. The Apache request loop [12]. The solid lines show the main transaction path, while the dashed lines show the paths that are followed when an error is returned

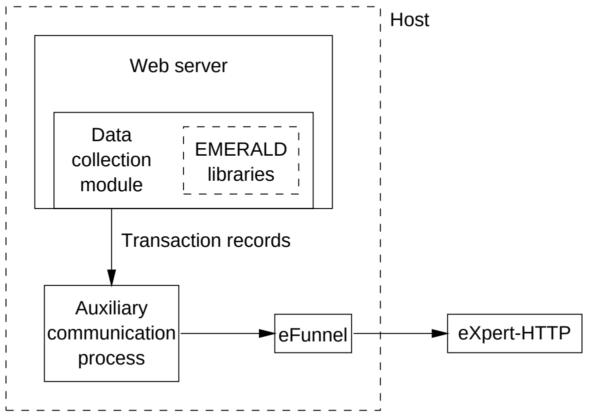

Fig. 2. The architecture and data flow in the implementation

We built the module within the framework of EMERALD [8], which among other components include the eXpert-HTTP analysis engine and the eFunnel communications process described below. The layout of our system is depicted in Fig. 2. For each request, the following happens:

1. The Web server passes control to our module in the logging stage of the request cycle.
2. The module takes relevant data of the request and packs it into a format the analysis engine eXpert-HTTP can understand.
3. Through the reliable piped logs interface of Apache, we pass the information to an auxiliary program, which just hands the information to a second program, eFunnel, through a socket.
4. eFunnel, in turn, communicates with an eXpert-HTTP on an external host.
5. The eXpert-HTTP performs the analysis.

Below we discuss each of these steps in detail. The design reflects mainly three concerns. Most importantly, we do not want to introduce vulnerabilities into the server software. For this reason, we decided to keep as little code as possible within the server. The second issue is performance. If the module makes the server slow, it will not be used. By limiting the analysis on the server host, we gain speed but we lose interactivity between the module and the server. Third, we wanted to reuse as much code as possible, both in the server and from the EMERALD system.

As our major concern was the risk of introducing vulnerabilities into the server, we decided against letting the analysis engine be part of the server. This would have added a lot of extra code, thus increasing the complexity and making the server more vulnerable to bugs [2, Theorem 1]. Although the analysis engine could have been placed on the same host, we wanted to demonstrate that larger sites can use this approach and satisfy critical performance requirements. However, removing the analysis engine from the server in turn means we limit the preemptive capabilities we described in Section 3, as the distance introduces latency between the receiving of a request and the final analysis. In Section 7, we propose a two-tier architecture that offers an acceptable trade-off between the ability to react in real time while still minimizing performance penalties. Note that the setup with the server existing on a separate host is more complicated than having the analysis engine on the same host. On sites where performance is not of critical importance, we recommend the simpler approach.

The introduced latency described in the previous paragraph restricts the reactive capabilities of the module. For this reason, we decided to let our module be called only in the last step of the request cycle—the logging step. By this time, the server has interpreted the request and sent the data to the client, and all information about the request is available for logging, which makes it easy for our module to extract it. Even though this seems to be marginally better than a system reading log files, we would like to point out two advantages with our proposed application-integrated module.

First, we have access to more information. For example, consider the request *http://www.w3c.org/phf*. The application-integrated monitor can determine if the server will handle the request as a CGI script (possibly bad), or if it accesses an HTML file (that, for example, describes the phf attack). Second, the information is immediately available with the application-integrated approach. A monitor watching a log file must wait for the application to write the information to the file, the caching within the operating system, and possibly the next monitor polling time.

The last steps of our design are explained by considering code reuse. Within the EMERALD framework, we have a component called eFunnel. This program accepts incoming connections where EMERALD messages can be transmitted, and passes the information to outgoing connections. It can duplicate incoming information (e.g., having two different analysis engines for the same application) or multiplex several incoming flows into one outgoing connection (e.g., comparing the results of a network-based monitor with an application-integrated monitor for discrepancies). This program takes into account problems that might appear in interprocess communication, such as lost connections or necessary buffering. Thus, eFunnel exactly matches our needs to send information from the module to other components.

On the server side, Apache provides a reliable pipe log interface. This interface sends the log information directly to a program. The term *reliable* signifies that Apache does some checking on the opened channel, such as making sure the connection is still accepting data. If that is not the case, it restarts the program [12, p. 563]. We also hope to capitalize on future advances within the implementation of this interface.

As noted, we would like to use both eFunnel and Apache's "reliable log format" but we run into a practical problem. In our tests, Apache started the receiving program twice [12, p. 59], but eFunnel binds to certain predefined sockets locally, and can thus be started only once. Our solution involves the auxiliary program described in Step 3 above, which provides a clear interface between the Web server and the IDS. Apache can restart this program as often as necessary, without the IDS being affected.

4.2 Analysis Engine

Within EMERALD, we are developing a package of network-based IDS components, one of them being *eXpert-HTTP*, an analysis component for HTTP traffic. It was developed to receive event messages from the network data collection component *etcpgen*, but can equally well receive the messages from the application-integrated module. Actually, we can use exactly the same knowledge base independent of the source of the data and no additional development cost is necessary for the analysis engine. After we had completed the data collection module, we could directly test it by having it send the data to eXpert-HTTP. It is not surprising that as more information is available through the module than through network sniffing, we have the possibility of constructing new rules to detect more attacks as well as refining existing rules to produce more accurate results.

4.3 Summary

Basically, our module extracts all transaction information from the Web server and packs it into a format that the analysis engine can understand. The module then ships off the information (through multiple steps due to the aforementioned implementation issues) to the analysis engine located at a separate host. No changes to the analysis engine were necessary even for detection of attacks using the encrypted SSL channel, so the module and the network-based event data collector could be used interchangeably with the same knowledge base. If we want to capitalize on the extra information we gain by using the module, we obviously need to develop new detection rules.

5 Monitor Performance

From a performance standpoint, the application-integrated module does not do anything computationally intensive. It simply accesses a few variables and formats the information before it is sent on. This should not decrease the server performance. However, we would like to have a more substantial claim that our approach is viable. One way would be to let the module include a call to a timing function (in C) to show how much execution time is spent inside the module. We decided against this measure, as we are more interested in measuring the user experience when an entire monitor is running. This means that we configured the module to send the data to the eXpert-HTTP analysis engine on another host, as depicted in Fig. 2.

WebLoad from RadView Software is an advanced Web application testing and analysis package [10]. Among its many features and options, it allows us to specify a single URL that will be continuously fetched during a specified time interval. WebLoad measures the round-trip time for each transaction, giving a fair measure of the user experience with the network and the server. We used the free evaluation version of WebLoad, which has some restrictions compared to the full-blown commercial version. However, those restrictions (no support for SSL and a maximum of 12 virtual clients) did not significantly limit our ability to evaluate the performance of our module.

We set up four runs, each lasting 60 minutes and using 10 virtual clients on a single physical client host. We used two different types of URLs, the first returning a Web page consisting of about 50 KB of text and a 12 KB JPEG image, while the second caused the execution of a CGI program. A summary of the results is in Table 2. The absolute values may seem somewhat high, but are due to the relatively low-end server hardware configuration. The relative difference between running the monitor or not is so small that it is probably only caused by the CPU load imposed by the auxiliary communications process and eFunnel running in parallel with the Web server. In fact, we believe that the results confirm that a future application-integrated module can have *zero* performance impact (with respect to response time) on the Apache application, as explained below.

Table 2. Performance measurements

Round-trip time (seconds)	Page without monitor	Page with monitor	Impact	CGI without monitor	CGI with monitor	Impact
median	1.486	1.517	2.1%	1.192	1.229	3.1%
average	1.499	1.521	1.5%	1.195	1.238	3.6%
std. deviation	0.057	0.059		0.034	0.048	

In Fig. 1, we show the request cycle for the server. The module performs logging *after* the request has been sent back to the client. For this reason, we expect that it is possible to postpone the penalty of the module until the next request. In Fig. 3, we show details of a possible scenario of what could happen in the server. The server spawns several child processes to handle new requests. As long as there are enough children, the penalty of the module does not need to be noticeable, as a child can finish logging before it receives the next request to handle. Under heavy load, where a request would have to wait for logging of the previous request, this would of course be different.

Apache/1.3.6 behaves in a different way from what is depicted in Fig. 3. Depending on the content sent back to the client, we observed a different behavior. We suspect that this depends on different implementations of the so-called Content-Handlers [12].

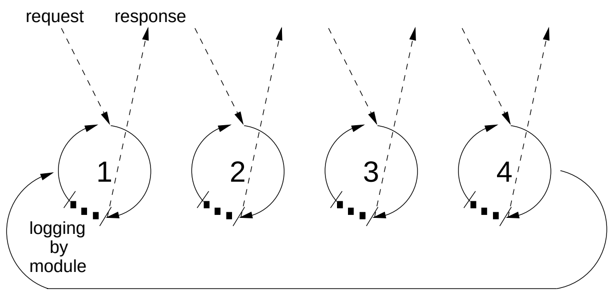

Fig. 3. Details of a possible scenario for Web server request handling. The numbers denote parallel processes ready to serve a new request

We did not stress test the Web server in the measurements described above, for two reasons. First, we have measured the whole system and we cannot attribute the difference in time to any specific reason. A stress test also affects the server and the operating system, neither of which is under investigation in this article. Second, we believe that many commercial server parks are built for peak needs and are therefore normally underutilized.

6 Related Work

The concept of application-based IDSs is not new, and there are indications that this would be the next big field to explore for commercial IDS vendors. There are some IDSs that are capable of monitoring firewalls [6]. In contrast, we are currently aware of only one single example of a commercial Web server IDS, AppShield from Sanctum, Inc. [11]. It is a proxy that overlooks the HTML pages sent out. For example, it scans HTML forms sent from the server, and makes sure that the information returned is valid (e.g., allows the use of hidden fields). Even though it actively monitors the Web server, it is not integrated into a greater IDS framework.

The closest to our approach is the work done by Burak Dayioglu [5]. His module simply matches the current request with a list of known vulnerable CGI programs. As the analysis is performed in line with the data collection, there is a greater risk of reduced server performance, especially if the list grows large.

7 Improvements

The module prototype described in this paper is the first step in the exploration of the possibilities of application-integrated data collection for IDS. By extending the knowledge base of the IDS to fully utilize the information available through the module, we hope to improve detection rates and reduce false-alarm rates. A natural step is also to develop similar data collection modules for other server applications, such as FTP, e-mail, databases, and Web servers other than Apache. We already have a working prototype for the iPlanet Web server.

It could also be interesting to have an analysis engine compare the transaction data from the application-integrated module with data from network sniffing. The subset of data items that is available to both collectors should normally be identical, except when someone actively tries to fool one of the collectors. This could potentially enable detection of advanced attacks that try to circumvent the IDS.

In Section 4, we described the trade-off between preemptive capabilities (application-integrated analysis) and low performance impact (application-integrated data collection but external analysis). We could have a two-tier architecture where the application-integrated module performs a quick and simple analysis before the request is granted by the application and then passes the data on to an external advanced analysis engine. If the first analysis detects an attack attempt, it could cause the request to be denied. The second analysis could also pass feedback to the first, such as the name of a host that has launched an attack and should not be serviced again. Instead of outright denying the request, the application-integrated module could suspend a request it finds suspicious until the external analysis engine has reached a conclusion. By tuning the first filter, we could restrict the performance penalty to a small subset of all requests, but still be able to thwart attacks.

8 Conclusions

We have presented an application-integrated approach to data collection for intrusion detection. By being very specialized, the module is closely tailored to be part of the application and have access to more information than external monitors, thus being able to discover more attacks but also to reduce the number of false alarms. Because each module is specific to one application product, coverage of many products could lead to an increased development cost, but we showed several reasons why that is not a severe limitation of this approach.

If this approach becomes popular, we expect vendors to provide an API for similar products in the future to stay competitive. This means very little extra effort is needed to include this type of monitor in a component-based IDS, such as EMERALD. We also showed the advantages with our prototype for the Apache Web server. It gave us access to information internal to the server, which helped the IDS understand how the server actually parsed the request. The module had access to the decrypted request, even if it was transported through SSL over the wire. As we clearly separated the data collection from the analysis engine, the performance penalty was negligible. The knowledge base could be used with no change, thus leveraging previous investments.

Acknowledgments

Phillip Porras and Steven Dawson of SRI International made an early prototype design and implementation of a transaction-data collection module for Apache. Their work served as an example and inspiration upon which we based the implementation presented in this paper.

References

[1] M. Almgren, H. Debar, and M. Dacier. A lightweight tool for detecting web server attacks. In *Proceedings of the 2000 ISOC Symposium on Network and Distributed Systems Security*, pages 157–170, San Diego, California, Feb. 2–4, 2000.

[2] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, 1994.

[3] T. E. Daniels and E. H. Spafford. Identification of host audit data to detect attacks on low-level IP vulnerabilities. *Journal of Computer Security*, 7(1):3–35, 1999.

[4] T. E. Daniels and E. H. Spafford. A network audit system for host-based intrusion detection (NASHID) in Linux. In *Proceedings of the 16th Annual Computer Security Applications Conference*, New Orleans, Louisiana, Dec. 11–15, 2000.

[5] B. Dayioglu, Mar. 2001. *http://yunus.hacettepe.edu.tr/~burak/mod_id/*.

[6] K. A. Jackson. Intrusion detection system (IDS) product survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, Los Alamos, New Mexico, June 25, 1999. Version 2.1.

[7] The Netcraft Web server survey, Feb. 2001. *http://www.netcraft.com/survey/*.

[8] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, Oct. 7–10, 1997. National Institute of Standards and Technology/National Computer Security Center.

[9] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Calgary, Alberta, Canada, Jan. 1998. *http://www.clark.net/˜roesch/idspaper.html*.

[10] RadView Software, Inc., Mar. 2001. *http://www.radview.com/*.

[11] Sanctum, Inc., Mar. 2001. *http://www.sanctuminc.com/*.

[12] L. Stein and D. MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, 1999.