Detecting Attacks That Exploit Application-Logic Errors Through Application-Level Auditing

Jingyu Zhou and Giovanni Vigna
Department of Computer Science
University of California, Santa Barbara
{jzhou, vigna}@cs.ucsb.edu

Abstract

Host security is achieved by securing both the operating system kernel and the privileged applications that run on top of it. Application-level bugs are more frequent than kernel-level bugs, and, therefore, applications are often the means to compromise the security of a system. Detecting these attacks can be difficult, especially in the case of attacks that exploit application-logic errors. These attacks seldom exhibit characterizing patterns as in the case of buffer overflows and format string attacks. In addition, the data used by intrusion detection systems is either too low-level, as in the case of system calls, or incomplete, as in the case of syslog entries. This paper presents a technique to enforce non-bypassable, application-level auditing that does not require the recompilation of legacy systems. The technique is implemented as a kernel-level component, a privileged daemon, and an off-line language tool. The technique uses binary rewriting to instrument applications so that meaningful and complete audit information can be extracted. This information is then matched against application-specific signatures to detect attacks that exploit application-logic errors. The technique has been successfully applied to detect attacks against widely-deployed applications, including the Apache web server and the OpenSSH server.

1. Introduction

The security of a host depends on both the operating system and the privileged applications that run on top of it. However, application-level vulnerabilities account for the majority of the vulnerabilities that are found and made public through mailing lists and advisories. A large number of the vulnerabilities in applications are caused by the lack of dynamic checks on input data, which makes it possible to perform buffer overflow and format string attacks.

Another type of attacks are those exploiting application-logic errors. Application-logic errors happen when an application performs actions that were not originally considered in the application design. For example, suppose that a privileged application is designed to read and print a specific file, such as "/etc/services". An application-logic error would allow an attacker to exploit an unexpected interaction with the shell environment to force the application to access (and print) a different file, such as "/etc/shadow", resulting in a security compromise.

The goal of Intrusion Detection Systems (IDSs) is to detect attacks against networks, operating systems, and applications. The mainstream approaches to intrusion detection use attack signatures to identify evidence of malicious activity in an event stream. The two most common types of intrusion detection systems are network-based intrusion detection systems (NIDSs), which use network packets as the data source for analysis [30], and host-based intrusion detection systems (HIDSs), which use operating system audit data as input [22, 14].

Unfortunately, both types of systems are unsuitable to detect attacks that exploit application-logic errors. These systems rarely have access to the semantically-rich audit data needed to identify this type of attacks. In fact, to detect these attacks, NIDSs need to reconstruct the data stream and parse application-level protocols, which are two expensive procedures. In addition, attackers can "desynchronize" the view of a NIDS with respect to the view of the attacked application, to evade detection or produce false alarms [27]. In some cases, NIDSs cannot even access the payload of the packets because encryption is used.

HIDSs may seem more suitable to detect application-level attacks because they have direct access to the system calls invoked by the application [18, 38]. Unfortunately, attacks exploiting application-logic errors rarely change the execution flow of the application. Therefore, approaches based on the analysis of system call sequences will not detect these attacks [10, 37]. Another problem is that the information needed to detect application-level attacks may be missing or too difficult to extract from the low-level information included in system call traces and in the audit re-

cords produced by the operating system.

To detect attacks exploiting application-logic errors, it is desirable to be able to perform selective, application-specific auditing in certain points of the application's control flow. The problem is that few applications provide hooks for instrumenting their control flows, and, even if these hooks are available, they may not be in the right places. In addition, the instrumentation technique would be application-specific and not easily portable to different applications.

This paper presents an instrumentation technique based on dynamic binary rewriting. The technique is application-independent and supports the collection of auditing information at any point within the application control flow. This technique is used in conjunction with application-specific signatures to detect attacks that exploit application-logic errors.

The approach has been validated through several case studies of attacks against real-world applications, namely Apache and OpenSSH. In both cases, auditing routines to collect data at critical points of the application's control flow have been developed. In addition, signatures to detect the attacks have been defined. Some of these attacks were not detectable by analyzing the network traffic or the data produced by existing OS-level auditing mechanisms.

The overhead introduced by the instrumentation technique has been evaluated quantitatively and compared to the overhead introduced by other types of auditing. The results are promising and show a low run-time overhead.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the approach used to instrument applications and detect attacks. Section 4 discusses several case studies. Section 5 evaluates the overhead introduced by the technique. Section 6 discusses some limitations of the approach. Finally, Section 7 draws conclusions and outlines future work.

2. Related Work

The work presented hereinafter is mostly related to auditing and program binary instrumentation.

2.1. Auditing

Auditing is a mechanism to collect information regarding the activity of users and applications. The *Trusted Computer System Evaluation Criteria* (TCSEC) requires that all computer systems evaluated at division C and higher produce audit data, and that audit mechanisms be both tamper-resistant and non-bypassable. Since the operating system is usually regarded as a trusted entity because it controls access to resources (e.g., memory and files), most existing audit mechanisms are implemented within the operating system.

Operating system audit data is not designed specifically for intrusion detection. Therefore, in many cases the audit records produced by OS-level auditing facilities contain irrelevant information, and sometimes lack useful data. As a result, IDSs often have to access the operating system directly for relevant information.

Lunt [23] suggested that specialized audit trails with only those data relevant to intrusion detection are needed. Daniels et al. identified the audit data that operating systems need to provide to support the detection of attacks against the TCP/IP stack [8]. Then, Kuperman et al. extended that work and developed a dynamic library interposition for application auditing [20]. However, their technique does not ensure non-bypassability because it cannot intercept function calls of statically-linked programs.

Almgren and Lindqvist proposed an extension to the Apache web server to collect application-specific audit data [1]. However, their work is specific to the Apache web server, which has a well-defined set of hooks to extend and instrument the server process. The problem with this technique is that it is not directly applicable to other applications that may provide a different set of hooks, or even no possibility of instrumentation at all.

The technique described in this paper overcomes the limitations of the approaches described above. The approach is based on binary rewriting, but, different from Kuperman's approach, the technique described here can be used to collect information about both the application and the library functions used by the application, even if an application has been linked statically. Therefore, it is independent of any particular application-specific interface to application extensions (e.g., the Apache module interface). The approach includes a language that allows one to specify the location in the application control flow where auditing must be performed. The language is independent of a particular application and allows a programmer to specify which format should be used for the audit records (e.g., [2]).

2.2. Binary Rewriting

Binary rewriting is a post-compilation technique that directly changes the binary code of executables. Different from dynamic library interposition, binary rewriting works for both statically-linked and dynamically-linked programs. Binary rewriting also provides more access to the application internals, because, in addition to library functions, static functions of the application can also be instrumented.

There are two types of binary rewriting techniques: *static rewriting* and *dynamic rewriting*. Static rewriting techniques modify the file-system image of program binaries, while dynamic rewriting techniques change the memory image of a process. ATOM [34], EEL [21], Purify [12], and Etch [31] are examples of tools based on static rewriting techniques. Dyninst [3] and Detours [13] are exam-

ples of dynamic rewriting tools. The Dyninst tool is used in this paper for dynamic rewriting.

Compared to the static techniques, dynamic rewriting techniques have the advantage of keeping the executables intact. In addition, by using dynamic rewriting it is possible to modify an application binary image in different ways, depending on the context of the invocation of the application. Dynamic rewriting also allows instrumentation code to be kept in memory after a `fork()` call, so that the child process is also instrumented.

Broadly speaking, dynamic binary rewriting is an interposition approach. Interposition is generally performed at the system call interface, the library interface, and the application interface for various purposes: Curry uses dynamic library interposition for profiling and tracing library calls [6]; Detours is a debugging and profiling tool that uses dynamic binary rewriting for intercepting Win32 functions [13]; Bypass extends the functionality of existing software systems for distributed computing using shared library interposition techniques [35]; Interposition Agents is a toolkit for interposing user code between a program and the operating system kernel [15].

System call interposition has been used for performance profiling and debugging of software systems. Interposition is also widely used for security purposes. SLIC [7], Generic Software Wrappers [11], and Systrace [26] enhance operating system security by interposing code at the kernel interface. The TIS firewall toolkit [29] and the TCP Wrappers [36] interpose code at the application level, but they only provide very limited application audit data. The technique described here uses interposition at the application and library interfaces to gather a complete set of security-relevant information.

3. Design and Implementation

An application-level auditing facility should be designed to meet a number of requirements.

1. **Non-bypassability.** This is a requirement of all auditing systems. For application-level auditing, non-bypassability essentially means that the auditing code has to be effective since the beginning of the application's execution.

2. **Compatibility.** The auditing mechanism should be compatible with legacy applications and operating systems. The changes to the applications and operating systems required by the mechanism should be minimal, in order to reduce the cost of deployment and administration. In addition, if the auditing mechanism does not modify the application binary, integrity checkers, such as Tripwire [17], will not be affected.

3. **Programming support.** The auditing facility should provide a simple mechanism for programmers to develop application-level audit routines.

4. **Performance.** The additional auditing capability should not exact much performance penalty on applications.

There are several possible alternatives for the implementation of application-level auditing. In the following, we review the advantages and disadvantages of three possible approaches.

Source modification. This approach directly modifies the source code of applications. Both the internal sensors approach for intrusion detection described in [16] and the DoS-resistant software approach described in [28] are examples of this approach. While achieving most of the design objectives, this approach requires recompilation of the applications. In addition, the configuration of the auditing system (i.e., what functions are instrumented, and what information is logged) is determined at compile time, once for all. This is a limitation to the flexibility of the system.

Virtual machine modification. Interpreted languages such as Java or Perl use virtual machines or interpreters to execute programs. Therefore, it is possible to extend a virtual machine or the interpreted code to add auditing functionalities [33]. Approaches like [4, 25] apply different access control policies by using Java bytecode rewriting techniques. Approaches that operate at the virtual machine level may impose notable overhead to the interpretation of the application's code. In this paper, we focused on high-performance, privileged applications written in C/C++.

Dynamic binary rewriting. This approach separates the audit routines from the original application. Thus, a new audit module can be added without re-compilation of the application binary. Audit module selection is postponed until the application is actually invoked, allowing for flexible selection of the auditing configuration. In addition, the audit routines run in the same address space as the application, with performance comparable to the source modification approach.

We implemented an application-level auditing tool by using a dynamic binary rewriting approach. The tool leverages an existing dynamic rewriting mechanism for inserting audit routines into applications at runtime. We implemented a loadable kernel module and a user-space daemon that allow for non-bypassable monitoring of applications. We also developed a language to support the development of both the instrumentation code and the detection procedures. The language hides the details of instrumentation from the programmers.

The tool has been implemented for the Linux 2.4 kernel. The dynamic rewriting mechanism is implemented by using an extended version of the Dyninst API [3]. The design and implementation of the tool's components are detailed in the following sections.

3.1. Runtime Environment

The tool uses a kernel module and a user-space daemon to perform application instrumentation. The purpose of these components is to monitor the operating system for events that represent the invocation of a monitored application and to inject the appropriate audit routines into the memory of the application at startup time.

The loadable kernel module allows for a custom application invocation procedure. Without support from the kernel, the use of dynamic binary rewriting would either cause incompatibility with legacy systems or fail to achieve the non-bypassability goal. On systems supporting ptrace, application instrumentation could be implemented using the `PTRACE_TRACEME` semantics. This mechanism allows a parent process to instrument its child after the child calls the `execve()` system call. Unfortunately, this solution would violate the compatibility requirement because the application's first invocation would have to be performed by some special "parent" program. In addition, ptrace's *attach* operation does not meet the non-bypassability goal, because there is a time window between the creation of a process and the beginning of the instrumentation procedure. Therefore, we implemented a kernel module that ensures non-bypassable instrumentation by intercepting the `execve()` system call. By doing this, the application is stopped before its first instruction is executed and the audit routines can be inserted into the application before its execution starts.

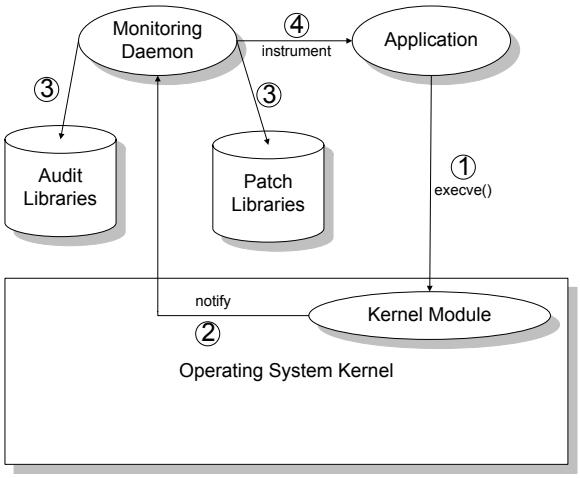

Figure 1: Runtime environment of the tool.

The actual instrumentation of an application is performed by the monitoring daemon, which is a privileged user-space process. The daemon manages two repositories: a patch repository and an audit repository. The patch repository contains the code for instrumenting the monitored applications. The audit repository contains the auditing code to be inserted into an application. The code in both the audit and the patch repositories is in the form of dynamic libraries. By using dynamic libraries, it is possible to update the code in the libraries while the daemon is still running. In addition, multiple versions of the libraries can exist at the same time.

Figure 1 describes the runtime environment. The monitoring daemon is invoked at system startup time. When a monitored application is invoked by calling the `execve()` system call (step 1, in the figure), the kernel module intercepts the call, stops the application, and then simulates the `PTRACE_TRACEME` semantics by setting the process's parent to the monitoring daemon and stopping the process by sending a SIGTRAP signal to it. By doing this, it is possible to achieve non-bypassability without having to modify the application binaries. We extended the Dyninst API to support the modified ptrace attach procedure.

Then, the kernel module notifies the user-space daemon that a monitored application has started (step 2). The monitoring daemon performs an "attach" operation to the process. Then, it consults its configuration file to locate the patch and audit libraries that are appropriate for the application (step 3). The patch libraries are loaded into the daemon process's address space and the instrumentation code in the patch libraries is executed (step 4). The instrumentation code loads the audit libraries into the application's address space and inserts audit function calls at certain points in the application's code. Once the application has been instrumented, the daemon "detaches" and resumes the execution of the instrumented application.

3.2. Auditing Specification Language

The language component of the tool supports the development of application-specific audit routines. The development of patch and audit libraries is complex and error-prone. Therefore, a simple language has been developed to support the instrumentor and to hide the details of the instrumentation from the developer of the auditing code. Thus, the programmer can focus on important issues, such as what data is needed, where the data should be collected, and what to do with the data.

Figure 2 gives an overview of the audit routine development process. An audit programmer writes an audit routine for an application in a superset of the C language. The audit routine contains auditing code and specifies the points in the application where the auditing code is called. A translator takes the audit routine as input and generates two C

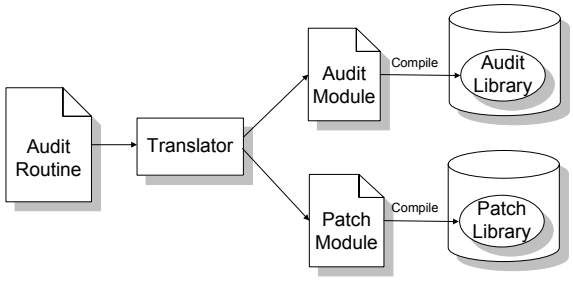

Figure 2: Off-line procedure.

modules: an auditing module and a patching module. These two modules are compiled separately into an audit library and a patch library, respectively.

The challenging part of the off-line component was to design a simple language that could be easily translated into C, and, at the same time, that would give enough flexibility for instrumentation, hiding the details of binary rewriting from audit programmers. The Dyninst API [3] provides a rich set of instrumentation primitives, such as variable allocation, arithmetic operations, if-then-else statement construction, and function replacement. In fact, it is possible to insert almost arbitrary code at an instrumentation point. However, this would complicate the translation process, as it would be necessary to translate the inserted code into different Dyninst API calls. In addition, these primitives are not uniformly implemented on different platforms. Therefore, it is desirable to use a minimal set of the primitives to achieve compatibility and flexibility.

The current tool implementation allows instrumentation at a function's entry and exit point only. At an instrumentation point, only a function call can be used. The function call must declare a subset of the parameters of the instrumented function. Experience with the tool so far has shown that this approach provides enough flexibility in the instrumentation of applications.

Figure 3 shows an example audit routine for the Apache web server. The routine contains two sections: an audit section and a patch section. The audit section is composed of the audit functions and associated data structures (lines 1-20). The patch section specifies which audit functions are called and where they are called (lines 22-26).

The audit section is translated into an auditing module, which is then compiled into a shared library. In the example, the audit section defines two functions. Function `audit_function_1` logs the request URI. Function `audit_function_2` logs the mapped path, file name, and handler name for the request. Note that both functions use the parameter of the `ap_run_log_transaction` function. Function `_init` is the initialization function for the shared library, which is called when the audit library is loaded into an application. In the example, the function is used to open a log file.

The patch section is introduced by the keywords "At Function" followed by a function signature, an optional entry declaration, and an optional exit declaration. The function signature is a function of the application to be instrumented. An entry declaration is specified by the keyword "entry" followed by a code block. The code block contains the function to be called at the entry point of the instrumented function. A similar syntax is used for the exit declaration. In the example, `audit_function_1` is inserted at the entry point of `ap_run_log_transaction` and `audit_function_2` is inserted at the function's exit point.

The patch section is translated into a patching module that uses the Dyninst API to create code snippets and patching code. A pseudo-code example of the patching module is given in Figure 4. The patching module is extracted by the translator and compiled into a patch library.

```
1  /* audit section */
2  FILE *fp = NULL;
3
4  void _init()
5  {
6    fp = fopen("/var/applogd/apache.log", "a");
7  }
8
9  void audit_function_1(request_rec *r)
10 {
11   fprintf(fp, "Original URI: %s\n", r->uri);
12   fflush(fp);
13 }
14
15 void audit_function_2(request_rec *r)
16 {
17   fprintf(fp, "Mapped path %s, file %s, handler %s\n",
18           r->path_info, r->filename, r->handler);
19   fflush(fp);
20 }
21
22 /* patch section */
23 At Function int
24    ap_run_log_transaction(request_rec *r)
25 entry { audit_function_1(r); }
26 exit{ audit_function_2(r); }
```

Figure 3: Example of the language.

4. Case Studies

The binary rewriting technique has been tested on two complex real-world applications: Apache and OpenSSH. Each application is susceptible to one or more attacks that exploit application-logic errors. For each attack, the audit data requirements have been identified, audit routines to collect the data have been written, and rules for attack detection have been developed. These case studies exemplify the inadequacies of existing application-level logging and show how binary rewriting overcomes these problems.

```
/* load the audit library into Apache */
load_audit_lib();

/* generate audit code snippets */
p1=find_func_entry("ap_run_log_transaction");
p2=find_func_exit("ap_run_log_transaction");

/* 0: 1st parameter of instrumented func. */
param = new func_param(0);
func_1 = find_function("audit_function_1");
call_1 = new func_call(func_1, param);
func_2 = find_function("audit_function_2");
call_2 = new func_call(func_2, param);

/* insert snippets at right places */
insert_snippet(p1, call_1);
insert_snippet(p2, call_2);
```

Figure 4: Example of pseudo-code of the patching library.

4.1. Apache

According to the NetCraft survey (`http://www.netcraft.com/survey/`), Apache is used in 67.70% of the Internet web servers as of August, 2004. Because of its wide deployment, Apache is the target of various types of attacks. Some of these attacks can be detected by examining the network traffic directed to the server. Other attacks can be detected by analyzing the application-level auditing produced by the application.

Apache's logging is provided by several modules, i.e., `mod_log_config`, `mod_log_referer`, and `mod_log_agent`. In the following sections, three attacks are presented together with the audit requirement for each of them. Our analysis found that, in every case considered, the logging provided by Apache is insufficient for effective intrusion detection.

4.1.1. CGI Script Source Code Disclosure Attack. The CGI script source code disclosure attack allows an attacker to access the source code of a CGI script [1]. The problem is that if a directory has both the WebDAV and the CGI behavior enabled, Apache will incorrectly consider the script as WebDAV content for an HTTP POST request and send back the script's source code in the reply.

In order to detect this attack, it is necessary to know whether the Apache server is considering a requested URL to be a CGI script or not. Unfortunately, none of the standard Apache logging modules provides this information. Therefore, an auditing routine that logs what handler Apache uses for a request has been developed (see Figure 3, lines 17-18). Note that, in this case, the instrumented function is Apache's logging routine. Thus, each time Apache performs logging, the auditing functions are executed.

A rule for detecting this attack is: *An Apache CGI script has to be handled by the "cgi-script" handler*. Apache was tested with the developed audit routine and the results showed that when the attack happened, the handler Apache used was "dav-handler," instead of the usual "cgi-script". This information allows one to easily detect the attack using a simple Perl script that analyzes the information produced by the auditing routine introduced through binary instrumentation. Note that, during the attack, Apache's standard access log contains an HTTP POST request only. That is, there is no information to infer that the attack occurred.

4.1.2. Signals to Non-Apache Process Attack. This attack exploits a vulnerability in Apache's shared memory scoreboard [2]. The attack allows code running with the Apache UID to send a signal (SIGUSR1) to any process as root. This privilege can be obtained by exploiting a vulnerability in Apache, particularly in a web-based application that uses scripting (e.g., PHP and Perl scripts). Any local user with a legitimate Apache scripting resource can perform this attack. When the target process receives the signal, it will react according to the corresponding signal handler. By default, the signal will terminate the process.

To detect this attack, the audit data should contain the process IDs of both the signal sender and the signal receiver, as well as the signal number. Unfortunately this information is not provided by the Apache logging facility. Because Apache sends signals by calling libc's `kill()` function, an audit routine has been developed to log the data at `kill()`'s entry point, as shown in Figure 5.

```
void kill_logging(int pid, int sig)
{
  int self = getpid();
  fprintf(log_fd,
    "PROCESS␣%d␣KILL␣(%d)␣WITH␣SIG␣=␣%d\n",
    self, pid, sig);
}

At Function int kill(pid_t pid, int sig)
entry { kill_logging(pid, sig); }
```

Figure 5: Audit routine for the signal attack.

A rule that detects this attack is: *Apache should not send signals to processes other than its children*. We tested this audit routine by performing the attack and found that the signal being sent was the alarm signal (SIGALRM), not the SIGUSR1 as reported by the advisory. By looking into Apache's source code, we found that the actual signal being sent depends on a compile-time flag. If Apache is compiled with OPTIMIZE_TIMEOUTS flag, then the signal sent is SIGALRM, otherwise it is SIGUSR1.

4.1.3. Slow Client Connection DoS Attack. This Denial-of-Service (DoS) attack is outlined in [28]. The idea is to

1 CVE entry CAN-2002-1156, `http://www.cve.mitre.org/`

2 CVE entry CAN-2002-0839, `http://www.cve.mitre.org/`

create a TCP connection that is as slow as possible. Because the server can only handle a limited amount of TCP connections, an attacker can exhaust all the server resources without flooding the server. Apache defines a time limit for receiving requests and sending out responses. The default value is 300 seconds, i.e., 5 minutes. Thus, a single request can tie up the server for about 10 minutes by using both slow send and slow receive. In addition, Apache supports HTTP 1.1 persistent connections and by default it allows 100 client requests per connection, with a 15 seconds idle timeout between two requests. Therefore, a single client connection can tie up the server for 10 * 100 + 15 * 99 / 60 $\approx$ 1025 minutes.

The Apache log entries contain the total time for a request. However, the logging system does not provide fine-grained information about a request's reading time, processing time, and result-sending time. The breakdown of the server cycle time for a request can help to identify the attack. In particular, the request reading time is required to detect the attack. Two audit functions have been inserted at the entry and exit points of Apache's request reading function `ap_read_request`. Thus, the time Apache spends on reading a client request can be calculated as the time difference between these two function invocations. The implementation of the audit routines is given in Figure 6.

```
static time_t start, end;

void read_entry(conn_rec *r)
{
  start = time(NULL);
}

void read_exit(conn_rec *r)
{
  int t;
  char time_str[32];

  end = time(NULL);
  t = end - start;
  strftime(time_str, 32, "%m/%d/%Y␣%H:%M:%S",
             localtime(&(start)));
  fprintf(log_fd,
    "%s␣request␣at:␣%s,␣duration:␣%d␣sec\n",
    r->remote_ip, time_str, t);
}

At Function request_rec*
   ap_read_request(conn_rec *conn)
entry { read_entry(conn); }
exit  { read_exit(conn); }
```

Figure 6: Audit routines for the Apache DoS attack.

A rule to detect this attack is: *The number of simultaneous slow client connections from an IP address can not exceed a threshold.* The rule has been implemented as a Perl script.

4.2. OpenSSH

OpenSSH is an open-source implementation of the SSH protocol. Different from Apache, OpenSSH uses the syslog facility for logging. Certain versions of OpenSSH [3] are vulnerable to a user-to-root attack that exploits client-defined environment variables. More specifically, if the "UseLogin" option is enabled, an attacker can pass environment variables, such as `LD_PRELOAD`, to the server using the key authentication method. Since the environment variable is set before the "login" program is invoked, the malicious library referenced by `LD_PRELOAD` can execute arbitrary code as root. It is worth pointing out that this attack is not new. Telnet daemons that supported environment passing have suffered from similar problems before [4].

To detect this kind of attack, it is necessary to include in the audit data the environment variable values set at the server, which are not available in the syslog entries produced by OpenSSH. Therefore, an audit routine that logs every environment variable's value at the server has been implemented. The environment variables are logged at the entry point of the `child_set_env` function, as shown in Figure 7.

```
static int log_fd;

void _init()
{
  log_fd = open("/var/applogd/openssh.log",
                O_WRONLY | O_CREAT | O_APPEND);
}

void env_entry_call(char ***envp, u_int *envsizep,
         const char *name, const char *value)
{
  char msg[1024];
  int i, size = 1024;

  i = snprintf(msg,size,"NEW␣ENV:␣%s=",name);
  size -=i;

  if ( size > 0 && value != NULL)
    i += snprintf(msg+i, size, "%s", value);

  write(log_fd, msg, i);
}

At Function void child_set_env(char ***envp,
   u_int *envsizep, const char *name, const char *value)
entry { env_entry_call(envp,envsizep,name,value); }
```

Figure 7: Audit routines for the OpenSSH User-To-Root attack.

A detection rule has been implemented to enforce the property: *Environment variables such as "`LD_PRELOAD`" should not be set by the client.*

3 CVE entry CAN-2001-0872, `http://www.cve.mitre.org/`
4 CERT Advisory CA-1995-14, `http://www.cert.org/advisories/CA-1995-14.html`

4.3. Discussion

Effectiveness of audit data. Intrusion detection is only as good as the input data used for analysis. Meaningful and complete auditing information is a prerequisite for effective intrusion detection. This is made clear by comparing the effectiveness of different types of audit data with respect to the attacks presented in the case studies of the previous sections. More precisely, in Table 1 we compare network traffic, OS audit data, application logs, and syslog data to the application-level audit data produced by binary instrumentation.

For the Apache signal attack it is possible to use OS audit data to catch the attack, even though Linux does not have an OS auditing facility by default. The DoS attack can be detected by analyzing network packets, but the data is not as straightforward to interpret as the application audit data is. Notice that both the CGI attack and the SSH attack are too elusive to be detected by analyzing existing audit data streams, because they lack semantically-rich information. In all the cases considered, the application-level audit data produced by binary instrumentation contained all the information required for detection.

Attack	Network	OS	Appl. logs	Appl. Audit
CGI				√
DoS	√			√
SIG		√		√
SSH				√

Table 1: Effectiveness of different audit data for intrusion detection.

Selection of instrumentation points. A key step of the approach described in this paper is to identify the instrumentation points in an application. Specifically, function names and their parameter types are required. When the source code of the application is available, the information about the instrumentation points can be obtained by analyzing the source code. The audit routines given in the case studies were the results of such manual process. We are currently looking into using compiler extensions to automate this process [9].

If source code is not available, it is hard, but still possible, to instrument an application. For example, Miller et al. demonstrated that by using reverse-engineering techniques, it is feasible to identify the license checking functions in a commercial product (Adobe's Framemaker) and to disable them by patching the binary at runtime [24]. The same data flow and control flow analysis techniques can be applied here to determine the instrumentation points for the application.

It is worth pointing out that the names and parameters of the instrumented functions in the case studies did not change much across different versions. Only one of the instrumented functions was changed in the Apache code, when moving from version 1.3 to version 2.0 [5]. The consistency of function signatures between major versions allows audit routines to remain effective, even though the executable binaries may change.

Detection of unknown attacks. In all the case studies, we analyzed previously known attacks to derive attack signatures and audit routines. However, the most important goal of intrusion detection is to detect unknown, future attacks. While being able to detect unknown attacks is a goal that has yet to be achieved, we argue that it is possible to develop generic signatures to detect some previously unseen variations of known attacks. For example, the CGI script handling signature may detect other CGI source code disclosure attacks that exploit different vulnerabilities. Also, the OpenSSH attack suggests that it is possible to detect new attacks from lessons learned by analyzing known attacks. The audit routines developed in the previous sections only log appropriate data into an audit file, which suffices for our auditing needs. In practice, the code can perform different actions. For example, an auditor may choose to terminate the OpenSSH server by calling `exit()` when `LD_PRELOAD` is set by a client.

5. Performance Evaluation

The binary rewriting approach has been evaluated quantitatively to determine the amount of overhead introduced by the tool. The evaluation included three parts: the kernel module overhead, the instrumentation overhead, and the runtime overhead, which are discussed in the following sections.

5.1. Kernel Module Overhead

The kernel module used by the tool introduces some overhead at the kernel level because it wraps the `execve()` system call. The module spends some execution time comparing the file to be executed with a list of monitored files.

We developed a benchmark to evaluate the amount of overhead introduced. The benchmark performs 100,000 `execve()` operations. For every invocation, the kernel compares the file executed by the `execve()` call with a set of 20 files, which we believe is a reasonable size for a set of trusted applications. The average execution time and the standard deviation of 10 runs are

5 The function is instrumented in the CGI source code disclosure attack. In Apache 1.3, the function name is `ap_log_transaction`, which is changed to `ap_run_log_transaction` in version 2.0.

shown in Table 2. The host used for the test is a Pentium IV, with a 1.8 GHz CPU and 512MB of memory. The overhead is fairly low, less than one percent. Considering that the `execve()` system call is invoked much less frequently than other system calls (e.g., `open()`), the impact on the whole system performance is minimal.

	w/out LKM	w/ LKM	Overhead
Time (s)	69.0454	69.5083	0.4629 (0.67%)
Std. Dev.	0.17	0.32	

Table 2: Micro-benchmark of `execve()` performance.

5.2. Instrumentation Overhead

	CP	GI	PT	DT	Other	Total
Time (s)	5.41	0.28	5.32	0.31	0.23	11.55
Per. (%)	46.3	2.4	46.1	2.7	2.0	
Std. Dev.	0.04	0.04	0.29	0.17	0.43	0.06

Table 3: Breakdown of instrumentation overhead (CP, GI, PT, and DT represent CreateProcess, GetImage, Patch and Detach, respectively).

The instrumentation overhead is the delay introduced by performing the binary rewriting at application-startup time. The Apache web server has been tested with all three audit routines enabled for 10 runs. For each run, the time needed for each step of the instrumentation process was measured. The host used for the test was a PC with a 1.5 GHz Pentium IV CPU and 256 MB of memory. The results are given in Table 3.

Table 3 shows that the average slowdown is of 11.55 seconds. Most of the instrumentation time is spent on the CreateProcess and Patch phases. The CreateProcess [3] function creates a `BPatch_thread` object by parsing a process image, including all linked shared libraries. The Patch phase finds function instrumentation points and inserts code snippets. Both phases spend a significant amount of time going through all the functions in the process image. GetImage and Detach are internal Dyninst API calls. The former gets an associated image object from `BPatch_thread` and the latter detaches from the instrumented process.

The overhead in this case is noticeable, and definitely perceivable by the user. On the other hand, the overhead is introduced at startup time only. This may be an acceptable trade-off for the additional security provided by the tool. One possible way to reduce the overhead is to combine the CreateProcess phase and the Patch phase, so that the search through all the functions must be performed only once.

5.3. Runtime Overhead

To evaluate the runtime overhead of the auditing routines on the Apache web server, the server was tested using the WebStone benchmark, version 2.5 (from `http://www.mindcraft.com/webstone/`).
The experimental setup consisted of one client machine that generates HTTP requests (Pentium IV, 1.8 GHz, 512 MB RAM, Linux 2.4) and one server machine (Pentium IV, 1.5 GHz, 256 MB RAM, Linux 2.4). Both machines are connected with a 100 Mb Ethernet. WebStone was configured with 10 to 100 clients in steps of 15 and each run took 5 minutes. The file set is the default static file set included in the benchmark

For each run, three cases were tested: Apache without any auditing, Apache instrumented with the auditing routines described in this paper, and Apache running with the Snare [32] OS-kernel auditing enabled. Snare includes a kernel module intercepting 34 different system calls and a user space daemon for writing audit data to a file. Client connection rate, response time, and throughput were measured. The results are shown in Figure 8, 9, and 10. As one can see from these figures, Snare performs poorly with 85 and 100 clients. The reason is that after about 25 minutes Snare used up all system resources, affecting the system performance. Sometimes, the system can not return to the normal state after heavy-load experiments. These two points were removed when performance impact was calculated. As one can see from Figure 8, the connection rate dropped slightly for the instrumented version of Apache. The average decrease is only 2.7 connections per second (about 0.4%). In the Snare case, the average decrease is 19.78 connections per second (about 3.44%). Figure 9 shows that there is no significant increase in the average response time for instrumented Apache (the average impact is 0.56%). Snare increases the response time by 4.3%. Figure 10 shows that the average throughput decreases by 0.4 Mbps (0.6%) for the instrumented version of Apache and by 3.3 Mbps (3.58%) when Snare is used.

In summary, the audit routines injected into Apache generate less than 1% overhead on average, which is significantly better than the overhead introduced by Snare. The reason for such a low overhead is that audit routines are running in the same address space as the Apache process. Thus, no context-switch cost or memory-copy cost is imposed. Snare uses a daemon process to periodically read audit data from the kernel and write the data to a log file. There are two additional memory copies of the data between the kernel space and the user space and two context-switch

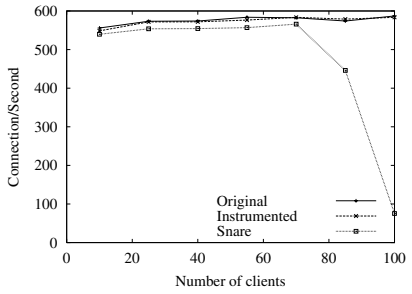

Figure 8: Client connection rate.

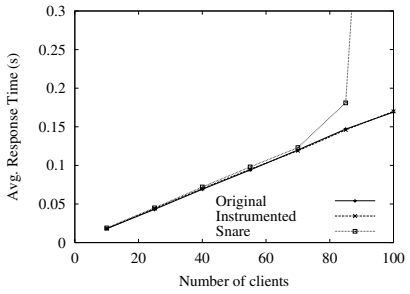

Figure 9: Average client response time.

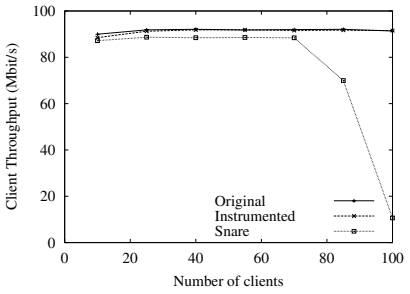

Figure 10: Client throughput.

costs. On the other hand, the audit routines are only effective for Apache, while Snare logs provide information for all processes. Considering that the experiments were conducted with all unrelated system services disabled, the majority of the slowdown was caused by Snare.

6. Limitations

The approach described in this paper has several limitations. First, some attacks, notably buffer overflow attacks, cannot be detected by this approach. Buffer overflow attacks modify the program's control flow and execute code on the stack, which is outside the original program code. Thus, a successful exploitation of buffer overflow will bypass inserted auditing routines. However, since some tools (e.g., StackGuard [5]) can prevent these attacks and many existing IDSs can successfully detect them, this limitation does not impact the overall applicability of this approach.

Second, there are some limitations of the approach that are related to the Dyninst API. One is that the Dyninst API does not allow further instrumentation once the monitoring daemon detaches from an application. Because of this limitation, a new audit library can only take effect when the corresponding application is restarted. Currently, this is not a problem, and future versions of Dyninst are expected to allow multiple instrumentations for an application.

7. Conclusions and Future Work

In this paper, a new approach for detecting attacks exploiting application-logic errors has been presented. The approach exploits a binary rewriting technique to collect application-specific data. A tool that uses the data to detect attacks has also been developed. The performance evaluation of the tool showed that effective intrusion detection can be achieved at a low cost.

The tool complements existing operating system auditing facilities and network auditing procedures. The approach can be used in those cases where semantically-rich data streams are needed for effective intrusion detection.

Since our approach inserts code into an application, it affects specification-based IDS [19] and systems that monitor the execution flow of an application [10, 37]. Future work will study how to address these problems.

A further evolution of this approach is to automatically generate the instrumentation routines directly from a high-level description of the attack signatures. By doing this, it will be possible to relieve the auditor from the task of identifying the instrumentation points and writing the auditing code. In addition, this approach would allow one to perform application auditing only if there is a signature that would actually use the data.

Acknowledgments

This research was supported by the National Science Foundation under grants CCR-0209065 and CCR-0238492 and by the Army Research Office, under agreement DAAD19-01-1-0484. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, the Army Research Office, or the U.S. Government.

References

[1] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, LNCS, pages 22–36, Davis, CA, October 2001. Springer.

[2] M. Bishop. A Standard Audit Trail Format. In *Proc. 18th NIST-NCSC National Information Systems Security Conference*, pages 136–145, Baltimore, MD, 1995.

[3] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[4] A. Chander, J. C. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *DARPA Information Survivability Conference and Exposition (DISCEX II)*, Anaheim, CA, June 2001.

[5] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *7th USENIX Security*, pages 63–78, San Antonio, TX, January 1998.

[6] T. W. Curry. Profiling and Tracing Dynamic Library Usage Via Interposition. In *USENIX Summer 1994 Technical Conference*, pages 267–278, Boston, MA, 1994.

[7] D. P. Ghormley and D. Petrou and S. H. Rodrigues and T. E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *Proceedings of the USENIX 1998 Annual Technical Conference*, New Orleans, LO, 1998.

[8] 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.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[10] S. Forrest. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996.

[11] T. Fraser, L. Badger, and M. Feldman. Hardening COTS Software with Generic Software Wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, Oakland, CA, 1999.

[12] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proc. of the Winter USENIX Technical Conference*, San Francisco, CA, January 1992.

[13] G. Hunt and D. Brubacher. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–144, Seattle, WA, 1999.

[14] K. Ilgun. USTAT: A Real-time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, Oakland, CA, May 1993.

[15] M. B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Symposium on Operating Systems Principles*, pages 80–93, Asheville, NC, 1993.

[16] F. Kerschbaum, E. H. Spafford, and D. Zamboni. Using Internal Sensors and Embedded Detectors for Intrusion Detection. *Journal of Computer Security*, 10(1-2):23–70, 2002.

[17] G. H. Kim and E. H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 18–29, Fairfax, VA, 1994.

[18] C. Ko and T. Redmond. Noninterference and Intrusion Detection. In *Proccedings of the IEEE Symposium on Security and Privacy*, Berkeley, CA, 2002.

[19] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, Oakland, CA, May 1997.

[20] B. A. Kuperman and E. Spafford. Generation of Application Level Audit Data via Library Interposition. Technical Report CERIAS TR-99-11, COAST Laboratory, Purdue University, October 1999.

[21] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[22] U. Lindqvist and P. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.

[23] T. F. Lunt. Detecting Intruders in Computer Systems. In *Proceedings of the Sixth Annual Symposium and Technical Displays on Physical and Electronic Security*, 1993.

[24] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing Inside the Black Box: Using Dynamic Instrumentation to Create Security Holes. In *the Second Los Alamos Computer Science Institute Symposium*, Sante Fe, NM, October 2001.

[25] R. Pandey and B. Hashii. Providing Fine-Grained Access Control for Java Programs via Binary Editing. In *Proc. of the 13th Conference on Object-Oriented Programming (ECOOP'99)*, pages 449–473, Lisbon, Portugal, 1999.

[26] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, 2003.

[27] T. Ptacek and T. Newsham. Insertion, Evasion and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, January 1998.

[28] X. Qie, R. Pang, and L. Peterson. Defensive Programming: Using an Annotation Toolkit to Build Dos-Resistant Software. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Boston, MA, December 2002.

[29] M. J. Ranum and F. M. Avolio. A Toolkit and Methods for Internet Firewalls. In *USENIX Conference*, pages 37–44, Boston, MA, 1994.

[30] M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, Seattle, WA, November 1999.

[31] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *USENIX Windows NT Workshop*, Seattle, WA, 1997.

[32] SNARE - System iNtrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`.

[33] S. Soman, C. Krintz, and G. Vigna. Detecting Malicious Java Code Using Virtual Machine Auditing. In *Proc. of the 12th USENIX Security Symposium*, pages 153–168, Washington, DC, 2003.

[34] A. Srivastava and A. Eustace. ATOM - A System for Building Customized Program Analysis Tools. In *Proceesings of the Symposium on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, 1994.

[35] D. Thain and M. Livny. Multiple Bypass: Interposition Agents for Distributed Computing. *Journal of Cluster Computing*, 4(1):39–47, March 2001.

[36] W. Venema. TCP Wrapper: network monitoring, access control, and booby traps. In *USENIX Proceedings of the Third UNIX Security Symposium*, Sept. 1992.

[37] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001. IEEE Press.

[38] J. Zimmermann, L. Mé, and C. Bidan. Experimenting with a Policy-Based HIDS Based on an Information Flow Control Model. In *Proceedings of the 19 Annual Computer Security Applications Conference (ACSAC)*, Las Vegas, NV, 2003.