Experimenting with a Policy-Based HIDS Based on an Information Flow Control Model

Jacob Zimmermann, Ludovic Mé, Christophe Bidan
{firstname.lastname}@supelec.fr

Abstract

In [1], we proposed a model for policy-based intrusion detection, based on information flow control. In the present paper, we show its applicability and effectiveness on a standard OS. We present results of two set of experiments, one carried out in a completely controlled environment, the other on an operational server with real network traffic. Our results results show that the model fulfills its goals and serves as a successful runtime policy-based intrusion detector.

1. Introduction

Beginning with the first works in the intrusion detection field [2], an intrusion has been defined as a *security policy violation.* Paradoxically, the most traditional IDS design does not provide any means to actually implement a policy. Instead, the two most widely used techniques rely on signatures or learned profiles to detect event traces that are believed to be symptoms of intrusions :

Scenario-based (also known as signature-based) IDSes detect actions and behaviour that are known to be able to violate some security policy;

Anomaly-based IDSes detect behaviour that differ from some statistical "normal usage" profile.

Most IDS designers and users agree that current systems based on these techniques are suboptimal. Signature bases used for scenario-based detection are usually too simplistic to discriminate real exploits from benign actions, thus advanced correlation techniques must be used. Even so, it is often impossible to tell whether the action executed through a known exploit is actually malicious or benign. Similarly, an anomaly-based detector may report alerts when facing some unusual behaviour, even if no policy violation actually occurs.

An alternative approach with increasing interest is **policy-based** intrusion detection. Its goal is to enforce a logically-defined security policy; i.e. a policy defined in terms of mandatory usage profile, system behaviours, etc. Such a policy-based IDS would ultimately offer high reliability and detection accuracy, provided that the policy definition and enforcement mechanisms are sufficiently precise.

As we agreed that the logical starting point for detecting security policy violations is a policy definition, in [1], we proposed a policy-based intrusion detection model based on information flow control. In this paper, we demonstrate the applicability and effectiveness of this model on a standard OS.

The rest of this paper is organized as follows:

In section 2, we discuss related work. Section 3 reintroduces the reader to the proposed approach. Sections 3.4 and 4 cover the experiments we conducted, the former in a strictly controlled environment (using Vigna's BuggyHTTP server), the latter in an authentic operational environment (using our team's operational mail and web servers.)

2. Related work

When considering the work outlined in this paper, two bodies of knowledge are of special interest. The first is the previous work done in policy-based intrusion detection; the second is previous work done in the information flow control field.

The idea of enforcing a security policy by monitoring process behaviour is not new. To overcome the problems inherent to traditional misuse and statistical anomaly detection [3], **specification-based** detection has been proposed [4, 5]. Instead of building upon description of known attacks, means to enforce **expected program behaviour** are provided in the form of behaviour specification languages and sets of specifications for critical programs.

While this approach has proved to be flexible and effective, it has the disadvantage of requiring know-

ledge of the monitored program behaviour. This may not be a problem in the case of well-documented software with source code available, or even in the case of "closed-source" but simple programs, but dealing with large and complex systems is difficult. In contrast, our proposed approach requires no prior knowledge of expected behaviour, as the security policy is defined in terms of **information flows** instead of executed actions.

In [6], the authors show how race-condition attacks can be detected by checking the interleaving of system calls executed by privileged and unprivileged processes. This model requires only knowledge of the system calls semantics to define conditions of system call execution interleaving legality, so it has also the potential do detect unknown and novel attacks. Our proposal follows a similar path, but because the chosen theoretical foundation is a superset of the one used by the authors, our detector is able to handle a wider class of attacks.

In the information flow control field, runtime policy enforcement mechanisms [7, 8] have been proposed. These models operate in specific environments such as the Java platform, where the set of possible information flow channels is deduced from knowledge available about program components (such as object interfaces) and other properties (method visibility scope, etc.). Our approach is similar to these models in that no static analysis in needed, as the set of legal flow channels is inferred from known and observed information such as the system calls semantics and process access to objects. However, our model is intended to be used over a generic operating system kernel, running unaltered, existing applications.

3. Outline of the approach

We begin by presenting three well-known attacks, each exploiting a different class of vulnerability (race condition, input validation error, Trojan horse). We then present our approach, demonstrating its key features by applying it to these three attacks.

3.1. Attack examples

3.1.1. *lpr* vulnerability No current system is affected by this vulnerability anymore; nevertheless, we present it here as it provides a straightforward instance of a proverbial race-condition exploit scenario.

Please refer to table 1 for a simple exploit. First, the attacker requests the printing of some benign file (for example, his own file *doc.txt*). His rights are checked by the *lpr* command and, if access is granted, the request is put into the printer daemon queue. Then, before the printing actually starts, the attacker removes the printed file and replaces it by a link to some file he is not allowed to access (for example, */etc/shadow*). As a result, the latter file will eventually be printed on his behalf.

3.1.2. *Apache* vulnerability This vulnerability [CAN-2003-0084] allows attackers to execute arbitrary operations on a target system through a format string error in Apache's *mod_auth_any* module.

The *mod_auth_any* module provides a client authentication interface for the Apache server. Information sent by a connecting client is passed to some external program that implements the actual authentication mechanism. Some versions of this module lack rigorous checking of data provided by clients. As strings sent through the module are used as command line arguments, a carefully formed string can therefore result in executing arbitrary shell commands.

The Apache server can thus be abused as a confused deputy to achieve various effects, such as denial-of-service (by shutting down the Apache server) or integrity violations (by overwriting files on the target systems that are writable under the Apache process user identity). It can even yield general security protection violations, if the attack is combined with some other "user to root" technique such as the *ptrace_kmod* exploit [CAN-2003-0127].

3.1.3. *OpenSSH* vulnerability This is a sophisticated "user to root" attack through a security hole in *OpenSSH* [9]. Originally, the telnet daemon was also affected by the same problem. Although both *telnetd* and *OpenSSH* have been patched, more software is potentially compromised.

Table 2 presents the major steps of the attack. The attacker creates a shared library (here called *libroot.so*) that provides a *setuid()* function whose purpose is to call the actual *setuid()* primitive provided in the system's API, but with *0* as parameter regardless of what the calling program may have requested. If this library is loaded by some process, it overloads the default *setuid()* function, so that any *setuid(u)* call made by that process is translated into *setuid(0)*.

The *OpenSSH* daemon may allow users to declare their own environment variables in their ssh profile. If the attacker declares *LD_PRELOAD=libroot.so*, any programs launched by the ssh session will be linked against that library, including the *login* program used to authenticate users. Because of the overloading of

	lpd daemon	attacker
1		lpr doc.txt
2	read /var/spool/lpd/job-****	
3		rm doc.txt
4		ln -s /etc/shadow doc.txt
5	read doc.txt	
6	send document to printer	

Table 1. The *lpr* attack

	sshd	attacker
1		create *libroot.so*
2		set *LD_PRELOAD* variable
3		start ssh session
4	execute */bin/login*	
5	load *libroot.so*	
6	authenticate the attacker	
7	execute *setuid(0)*	
8	execute */bin/sh*	
9		*root* shell available

Table 2. OpenSSH attack

setuid(), the attacker's authentication procedure will eventually execute the *setuid(0)* system call.

So this is a way for the attacker to infect the *OpenSSH* system with a Trojan horse, which leads ultimately to getting a root shell.

3.2. Detecting Intrusion Symptoms

In all three attacks presented above, the respective security policies (more precisely, data confidentiality and/or integrity requirements) are violated the transfer of information into forbidden locations:

- in the *lpr* attack, the contents of a file the attacker is forbidden to access is copied into a destination where he has full access rights (the printer), thus achieving *de-facto* access privilege granting [10];
- in the second example, uncontrolled information flows occur between local files stored on the target system, and the remote client connected on a TCP socket without proper authentication;
- in the third example, protected information (the executable memory image of a *root* process) is overwritten by uncontrolled information (the *libroot.so* library) produced by an unprivileged attacker.

More generally, many actual attacks consist precisely in achieving some **illegal information flow** between system objects[1]. We can further observe that the very existence of such an information flow is a condition *sine qua non* for the attack to be effective: for example, from our point of view, using the *lpr* exploit to print a public-readable document does not violate any confidentiality requirement.

Generally speaking, confidentiality and integrity goals of any policy can be expressed as an information flow control problem. Thus, an illegal information flow is the symptom of a policy violation, i.e. of an attack. No hypothesis has to be made about the actual scenario that leads to the attack, so controlling information flows generated by running programs provides a reliable confidentiality and integrity violation detection mechanism, allowing to detect known and unknown attacks[2].

Table 3 shows the percentage of each of the 7 considered CVE vulnerability classes[3] in exploits that

1 An *object* is any information container that has a precise identifier and whose state can be accessed through a precise set of primitive operations. Examples of objects are files, sockets, messages, memory pages etc.

2 Please notice that we do not handle system availability requirements.

3 We do not take "Configuration error" vulnerabilities into ac-

achieve each of the 4 policy violation goals defined by CVE. In our case, in addition to Confidentiality and Integrity violations themselves, we have also to deal with Protection violations, since by getting for example *root* privileges on some host, an attacker can steal confidential information and/or modify information that he is not allowed to access. We can thus observe that each vulnerability class has a substantial usage importance in at least one of these 3 categories. By detecting Confidentiality and Integrity attacks, as well as confidentiality and/or integrity violations that eventually occur through Protection attacks, we can thus deal with a wide range of actual attack scenarii.

3.3. Implementing security policies

Our proposed approach to intrusion detection and the policy implementation model have been discussed in detail in [11]. In this section, we briefly describe its application to Discretionary Access Control-based operating systems, such as Linux, Unix and Windows.

Our purpose is to enforce an existing confidentiality and integrity policy. In DAC-based systems, confidentiality and integrity policies are implemented using access control rights. Such an access control policy implies a set of legal information flows between objects. The basic idea is that if the access control policy allows some user to gather information from some object A (for example, to read a file) and to modify the state of some object B (for example, to write a file), then by definition it allows him to create an information flow from A to B.

For example, the DAC policy shown on table 4 allows *Alice* to create information flows from o_1 to o_2 and o_3, and Bob to create flows from o_2 and o_4 to o_1 and o_4 (please refer to figure 1).

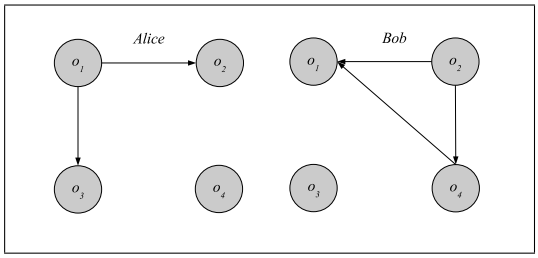

Figure 1. Legal information flows

count, since these can actually be considered as policy specification errors.

Information flows between system objects are created by calling operations on these objects, i.e. by executing system calls. Given a security policy specification in terms of legal information flows and a system call execution trace, the intrusion detector checks whether the executed system calls do not create illegal flows, either directly (for instance, a *sendfile* call copying illegally the contents of a file into a network socket), or by transitive combination of legal flows (in the example above, such a combination would be for instance a flow from o_1 to o_2 created by *Alice*, followed by a flow from o_2 to o_4 created by *Bob*. That would result in a flow from o_1 to o_4, which is not legal given this policy).

To evaluate the effectiveness of our detection approach, we tested it using attacks against Vigna's BuggyHTTP server [12] which has several vulnerabilities of different classes. These can be easily exploited; however, real-world products such as Apache or MS-IIS have identical security issues.

This choice was made after the model design and implementation were frozen, so the BuggyHTTP daemon provided a neutral, independently developed application with attacks that were unknown to us.

3.4. Testbed environment

The tests were performed on a standard Linux system, with no special assumptions about the behaviours of the observed daemons. The enforced security policy was derived from existing access privileges as described in paragraph 3.2. Since no authentication mechanism is used on HTTP client connection, only information from world-readable objects could flow to the HTTP client's socket, and information sent by the client could only flow into world-writable objects.

Moreover, the environmental conditions were as follows:

- The BuggyHTTP daemon ran on a default Linux workstation along with other standard applications, including a full graphical desktop, a Web navigator, an email client, a ssh server and a game server;
- No incoming traffic to that host from outside of the local network was allowed, so that we could precisely evaluate the number of attack attempts and the rate of successful detections vs. false positives
- The tested attacks achieved confidentiality and integrity violations in different ways. All were performed locally as well as across the network.

14 system calls, which could generate 18 different types of information flows, were controlled by the detector.

	Confidentiality	Integrity	Protection	Availability
Boundary conditions	8.9%	2.6%	4.0%	8.8%
Buffer overflow	3.4%	7.6%	59.2%	28.8%
Access validation error	58.1%	53.8%	21.4%	3.4%
Exception error	15.8%	6.4%	5.7%	24.7%
Environment error	8.0%	4.6%	3.6%	1.3%
Race condition	2.9%	24.0%	4.3%	1.2%
Configuration error	N/A	N/A	N/A	N/A
Design error	11.3%	11.4%	8.0%	2.9%

Tab. 3. Vulnerabilities vs. policy violations

	Alice	Bob
o_1	*read*	*write*
o_2	*write*	*read*
o_3	*write*	$\emptyset$
o_4	$\emptyset$	*read,write*

Tab. 4. Sample DAC policy

3.5. Attack detection

3.5.1. Malicious URLs Illegal file access through a malformed path in the HTTP request is the easiest attack against a web server. Such a request can access:

1. a file that is part of the website, but referenced through a needlessly complicated pathname, e.g. `GET ../../www/html/index.html`
2. a file that is declared world-readable by the access control policy, but for which it is unclear whether it is supposed to be available to web clients. For example: `GET ../../etc/hostname`
3. a file that must not be accessed by anyone. In this case the web server can be used as a confused deputy, thereby granting access to the file: `GET ../../etc/shadow`

Any access results in a flow from a file to the client's socket. In the first case, this flow is by definition legal, so no alert must be generated. In the third case, the confidentiality policy is violated by a flow from */etc/shadow* to a random location, which is by definition illegal, so an alert is raised. In the second case, the security policy is actually ambiguous since by default, the access rights are too coarse-grained to make difference between "local access" and "remote access" (however, this can be easily corrected using extended ACLs, available in all modern OSes). From a strict access control point of view, no security policy violation occurs and our detector raises no alert. Nevertheless, many administrators would probably consider that being able to download a random file outside of the *www* tree is a security threat, even if that file is supposed to be readable by anyone.

In comparison, a signature-based IDS such as *Snort* usually contains a rule matching all URLs containing the *"../.."* pattern. Note that this signature would trigger false positives for the first and (possibly, depending upon interpretation) the second attack scenarios outlined above. It would also fail to recognize several valid attacks: for instance if the attacker first uses some other attack technique to replace the *index.html* file by a link to */etc/shadow*.

3.5.2. Attacks through *cgi-bin* Similarly, we can execute arbitrary programs on the server as *cgi-bin* scripts. If the server runs as *root*, this opens a wide spectrum of possible effects, including:

1. accessing any file in the system, for instance `GET /cgi-bin/../../bin/cat /etc/shadow`
2. overwriting any file, for instance `GET /cgi-bin/../../bin/echo "hacked" > /etc/shadow`
3. executing arbitrary commands, even a *root* shell: `GET /cgi-bin/../../bin/sh`

Let us consider a *root* shell example, as it is the most general. Please refer to table 5 for a possible use. Although launching the shell at step 1 is a security protec-

tion violation by itself, no illegal information flow occurs yet, so no alert is expected. Step 2 produces an information flow from *index.html* to the socket, which is by definition legal, so no alert is raised[4]. On the other hand, reading */etc/shadow* at step 3 is of course illegal and an alert is raised.

The same comparison to signature-based detection (e.g. *snort*) as in paragraph 3.5.1 applies. Moreover, given that the sequence of commands possibly executed this way is unpredictable, providing exhaustive and reliable signatures is almost impossible. To deal with such an attack using a traditional IDS, one would have to detect essentially the mere fact of launching a shell. Not only does this raise false positives (as in step 2, or if the attacker only started the shell and immediately exited without actually executing *anything*), its very detection is actually largely nontrivial. In this case, some of the advantages of our approach are:

- we need to take no further special hypotheses to handle this case, the detector configuration is unchanged;
- a forbidden information flow is reliably detected, no matter how it is achieved.

3.5.3. Buffer overflow exploits The BuggyHTTP server contains a buffer overflow vulnerability. It can be exploited either:

1. to achieve a denial-of-service by making the server crash; or
2. to let it execute an operation, for instance by using a shell-code.

As we do not enforce availability requirements, a denial-of-service attack is undetected by our approach. However, an attempt to execute some operation can be detected: since in paragraph 3.5.2 no hypothesis was made on *how* the operation execution is achieved, it can be done using a cgi-bin exploit as well as a buffer overflow exploit.

3.5.4. Race conditions exploits We patched the BuggyHTTP server to introduce an additional race condition vulnerability. By default, the server operates in two steps: first, the client's request is written into a log file, then it is processed. By letting the server first execute the request, then log it, it is possible to violate the system integrity using a scenario similar to the one on table 6.

Of course, there exists many ways to achieve the effect of the first 3 steps, i.e. to create the symbolic link. In any case, step 5 results in a flow from the client's socket into */etc/shadow* and as such, it is illegal.

The important part is that since, by design of the server, flows from the socket to the log files directory are legal (as input from the client is copied into the log), creating the symlink is not illegal *per se*. Actually, the same effect could be achieved by sending a string similar to the contents of */etc/shadow*. However, the flow from the client's socket to */etc/shadow* which results from it violates the integrity policy.

3.6. Proof of concept

The results presented in this section show that our approach is able to reliably detect illegal information flows, even if the means to achieve them are unusual, unknown and/or highly complex. It fulfills its role as a policy-based intrusion detector.

4. Real-world usage

4.1. Environment

To evaluate the intrusion detection abilities of this approach in a real environment, we installed our detector on two hosts: a mail and web server used daily by our team (13 researchers and PhD students) and a test server running an older Linux distribution known to be affected by many vulnerabilities (RedHat Linux 6.2 on x86). The experiment lasted ten days; another test for a several month-long period is currently underway.

The exploits used against these systems were chosen at random, from collections of exploits targeting the current versions of running software. As we selected these exploits at random, we consider them to represent a fair replica of attacks that might be conducted by an external attacker with no particular knowledge of our operational network. Moreover, this allows us to test the ability of our IDS to detect unknown attacks. Indeed, some successfully detected exploits were unknown to us when the tests started.

On this network, we run the open source IDS *snort*, which raises between 100 and 1000 alerts per day, depending upon configuration. Upon further investigation by our security officer, virtually all are false positives. This detection pattern has remained stable over time in our environment.

4.2. Security policy

In addition to a default local security policy handled as discussed in §3.3, we deal with the network services as follows.

4 Of course, this is a rather unusual way to access a web site, but strictly speaking, it does not violate the security policy in any way !

step	command	alert
1	GET /cgi-bin/../../bin/sh HTTP/1.1	
2	cat /www/html/index.html	
3	cat /etc/shadow	•

Table 5. Sample *netcat* session

step	operation	alert
1	send request *GET /cgi-bin/../../bin/sh HTTP/1.1*	
2	*ln -s /etc/shadow /www/log/127.0.0.1.num*	
3	*exit*	
4	server logs the request into */www/log/127.0.0.1.num*	
5	*/etc/shadow* is overwritten	•

Table 6. Sample race condition attack

4.2.1. Web In our Linux installation, the Apache server runs as *www-data*. It can be configured to manage its log files either through the standard *syslog* system, or by directly accessing them. In the first case, the default security policy needs no amendment. In the second case, we note that in addition to information flows resulting from default access control rights, flows from the client socket to objects writable by *www-data* are authorized. The "client socket" is any socket created by connecting to TCP port 80.

More precisely, The Apache server security policy thus allows some flows between TCP port 80 and objects readable or writable by *www-data*. A HTTP client socket initialization is by definition a flow from port 80 to the newly created port. In effect, this is a straightforward translation of the standard policy that restricts Apache access rights to those of *www-data*.

4.2.2. E-mail We implemented the following email policy: SMTP clients may write mail into any mailbox. That is, information incoming from clients connecting to the SMTP service can flow into any of the mailboxes in */var/spool/mail*. Furthermore, the contents of the mailboxes may flow to clients connecting to the POPS and IMAPS services.

A naive approach to implementing this policy would be to rely on pure access rights. However, a plain Unix-like access control policy in not precise enough to suit a *mailbox*. For each user u, either it forbids flows from TCP port 25 to */var/spool/mail/u*, or it allows flows from port 25 to any object writable by u. Thus, we authorize explicitly the following flows:

- from TCP port 25 to */var/spool/mail/u*;
- from */var/spool/mail/u* to port 993 (IMAPS) and 995 (POPS).

We did not define any special policy regarding outgoing mail in these tests.

4.3. Executed tests

We exploited several vulnerabilities (CVE-2001-0690, CAN-2002-0392, CAN-2003-0127, CVE-2000-0666) to achieve 50 actual security policy violations on these two systems, i.e., an average of 5 succeeded attacks per day:

- Denial of service;
- Stealing confidential files (for instance */etc/shadow*, */etc/ssh/ssh_host_dsa_key*, etc...);
- Overwriting arbitrary files;
- Executing local operations (for instance appending */etc/shadow* to */var/spool/mail/jzimm*).

In addition, we used some of these exploits to achieve goals that do not violate the system confidentiality and integrity or that are irrelevant, service availability issues aside. For instance:

- Capturing a root shell and not executing anything;
- While logged on as *mail*, gaining *root* privileges but only executing operations that *mail* is authorized to execute;
- Killing the Apache daemon, binding a trivial "root shell server" to port 80, then using it to access */home/httpd/index.html*;
- etc.

4.4. Results

Results for the test period are summarized in table 7. Given the large number of processed events (> 26 million system calls), the attacks themselves are buried in the noise floor generated by benign operations.

4.4.1. True positives Of the 50 succeeded attacks, 38 were accurately detected on the basis of the implemented security policy. All were violations of confidentiality or integrity of resident information. Some goals (i.e. stealing a non-public file) were achieved using several different attack techniques, most notably:

- capturing a root shell and reading the file;
- moving (remotely or locally) the file into a mailbox and then reading mail;
- moving (remotely or locally) the file into a location where it could be accessed as a web page;
- etc.

Given the security policy implemented as described in section 4.2, with no change to the software nor the user's capabilities, we assert that **all of our attempts to violate this policy were successfully detected**.

Moreover, when we launched an identical attack procedure without violating this policy, no alert was raised. Thus we assert that our detector satisfied its primary goal. To recap, given a security policy, our system can reliably detect violations of that policy, regardless of the attack technique used to accomplish that violation; furthermore, when the same attack techniques are used to benign effect, false positives do not occur.

4.4.2. False negatives The 12 undetected attacks achieved effects that include the following:

1. accessing world-readable files that are not explicitly part of the website;
2. modifying an user's mailbox contents using only data incoming from the network;
3. modifying world-writable files;
4. a denial of service.

As we discussed in §3.5.1, case 1 can be considered as a legitimate action given the default security policy; it can be prevented by refining the policy to forbid some information to flow to remote locations. The same remark applies to case 3.

Case 2 is a pure integrity of transiting information problem. To detect such attacks, one has to check the contents of incoming information, which is not the purpose of our model. It is also possible to define manually extremely fine-grained policy regarding utilities such as *echo* or *cat* that these attacks can use. So while detecting such attacks seems theoretically possible to some extent, it clearly falls outside of our approach.

4.4.3. False positives and additional benign alerts The systems also generated 6 benign alerts and only 4 false positives. Alerts we call *benign* refer to actions that do violate the implemented security policy strictly speaking, but the achieved effect cannot be qualified as really malicious. These alerts were raised for example:

- when we logged on the mail server and tried to use a console mail client. By construction of the policy regarding e-mail, this was detected as an illegal flow;
- similarly, when we created files in a *home* directory and tried to access them through the IMAP server;
- when we launched as root a recursive grep search, that generated flows from many different objects (including mailboxes and other private files) to one single destination.

As, given the implemented policy, users are not supposed to execute anything directly on the mail server, such actions actually violate the policy, even if they do not really violate confidentiality nor integrity.

The 4 remaining alerts were genuine false positives. Three were generated by the *ssh* server (accessing the host key database and using public key-based authentication, which is not handled by the policy definition at this time). Finally, the last alert remains unexplained.

5. Conclusion

In this paper, we show that the model we proposed in [1] and [11] provides an applicable and efficient means to detect security policy violations in runtime. By controlling information flows on a host, we are able to detect a wide range of confidentiality and integrity violations. Despite the fact that we pose no hypotheses on the vulnerabilities exploited by an attacker nor the attack scenario itself, and furthermore use no learning method to characterize normal system activity, we are able to detect unknown and/or novel attacks. We demonstrate that even in the case when a known exploit is used to accomplish a benign effect, unless the security policy is violated, no alert is triggered.

Our model focuses on local information flow, which allows one to detect local intrusions. However, most remote intrusions lead to a local information flow policy

Processed events	26276342
Our successful attacks	50
True positives	38
False negatives	12
False positives	4
Additional benign alerts	6

Table 7. Experience summary

# of operations	12051434
default	9m 40s
with active checker	11m 23s

Table 8. Linux kernel compilation benchmark *(make bzImage)*

violation and are thus detected as well. Our first experiment carried out on the BuggyHTTP-based platform demonstrates this point. This also shows that policy should specify how local objects may be remotely accessed, which is rarely done by default.

Our real-usage experiment shows that, despite the large number of analysed events (more than 26 million), the current implementation detects in runtime all 38 of the executed attacks that explicitly violate the confidentiality and integrity requirements of the policy, in addition to 6 cases in which the implemented policy is also violated. The 12 executed attacks that the system was not able to detect were either exploits violating availability policy or exploits against an insufficient security policy specification.

At the same time that our system reliably detected attacks, the rate of false positives was low. No false positives were generated during the controlled BuggyHTTP tests, and only 4 out of the 48 alarms generated during our real-usage tests were false positives. This low false positive rate held despite an extremely large number of system events.

At this stage of our research, we did not devote extensive time and energy to measuring the impact of our detector on system performance. We did perform a simple benchmark (please refer to table 8) demonstrating that on our test system (PIII 450 MHz, 128 Mb, SCSI), the overhead when compiling a Linux kernel is slightly over 15%.

A subsequent experiment is underway, in which our approach is being tested in a real-world operational environment over several months. To this date, observed results remain consistent with those reported in this paper. The next major step in our work is to propose an approach to support automated explanation of generated alerts, correlating them to observed events.

References

[1] J. Zimmermann, L. Mé, and C. Bidan. Introducing reference flow control for detecting intrusion at the os level. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection*, pages 292–306, October 2002.

[2] J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[3] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[4] Calvin Ko, George Fink, and Karl Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[5] R. Sekar, T. Bowen, and M. Segal. On preventing intrusions by process behavior monitoring. In *Proceedings of the 1999 USENIX Intrusion Detection Workshop*, pages 29–40, 1999.

[6] Calvin Ko and Timothy Redmond. Noninterference and intrusion detection. In *Proccedings of the IEEE Symposium on Security and Privacy*, 2002.

[7] Thomas P. Jensen, Daniel Le Metayer, and Tommy Thorn. Verification of control flow based security properties. In *IEEE Symposium on Security and Privacy*, pages 89–103, 1999.

[8] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140, 1997.

[9] CMU CERT/CC. Vu#40327: Openssh uselogin option allows remote execution of commands as root. http://www.kb.cert.org/vuls/id/40327, November 2001.

[10] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *Proceedings of the Seventh Symposium in Operating Systems Principles*, pages 45–54, December 1979.

[11] J. Zimmermann, L. Mé, and C. Bidan. An improved reference flow control model for policy-based intrusion detection. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2003.

[12] Giovanni Vigna. The buggy web server (tm). http://www.cs.ucsb.edu/ vigna/courses/CS279/HW5/buggy.c.