RZ 3129 (# 93175) 08/02/99
Computer Science/Mathematics 62 pages

Research Report

Design and Implementation of a Lightweight Tool for Detecting Web Server Attacks

Magnus Almgren

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

IBM Research
Almaden · Austin · Beijing · Haifa · T.J. Watson · Tokyo · Zurich

Design and Implementation of a Lightweight Tool for Detecting Web Server Attacks

Magnus Almgren

IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland

Abstract

This thesis describes a monitoring tool for analyzing web server log files. This tool has several interesting characteristics. First, it can monitor in real time the requests sent to web servers and send alarms to a system management tool. Second, its layered architecture allows the detection of numerous abnormal events, not limited to requests for vulnerable cgi scripts. Finally, it includes mechanisms for learning of new attacks and for reducing the rate of false alarms. The design and implementation of the tool are described together with the experimental results obtained when the tool was tested at real sites.

Contents

Chapter 1

Background

Computers play a major role in our daily lives. Even mundane tasks often require a microprocessor hidden in the background. The growth of this dependency has been exponential, and more and more vital functions of our infrastructure need some sort of computer support.

The advantage of this trend is the enhancement of services offered, but the disadvantage is our increased dependence on computers. Let us take the example of banking. Twenty years ago, one had to go to a bank during its opening hours and then typically stand in a queue to withdraw money. Then came ATMs and phone banking, and today one can have a cash card reader connected to one's home phone. Withdrawing money from one's account could not be easier, and it is also possible to transfer money and buy stocks by connecting to the bank via the Internet. One's physical presence at the bank is no longer necessary.

The bank relies on computers to offer these services. However, customers not only want the service, they also require security. Unfortunately, each new channel to the bank means yet another way to break into it.

Customers can reach their bank through the Internet; so can criminals. There is no longer any need for them to use explosives if they have computers, which gave rise to the term *cybercrime*. The security issues concerning computer infrastructure are different from those of physical security. Unlike physical property, digital information can be duplicated and spread very easily.

As banks depend on their customers's trust, they would probably not report successful break-ins to the public. There have been reported cases where credit card numbers have been stolen from online web sites, though [1].

This thesis presents a lightweight tool to detect attacks against web servers, which constitute the electronic front door of many companies. We justify the usefulness of such a tool, and then describe the flexibility of its design.

The structure of this thesis is as follows. In Chapter 1, we introduce the concepts of intrusion detection and present the web server. In Chapter 2, we specify the requirements imposed on the work. Chapter 3 deals with design and implementation issues, whereas Chapter 4 describes findings when the monitor was tested at real sites. Chapter 5 describes the performance of the monitor. Some improvements are suggested in Chapter 6, and the conclusions are given in Chapter 7.

1.1 Intrusion Detection

1.1.1 Rationale

In theory a totally secure system is conceivable, but in practice, compromises have to be found between security and functionality:

- The users of computer systems are not interested in the constraints imposed by a secure system. They use computers to facilitate their daily tasks, and they want to be able to use JavaScript and email, and to download files in PDF and PostScript format. Their goal is to increase their productivity by using computers. They do not want to be denied functionalities in an effort to avoid the theoretical risk of a break-in.

- Even if users did agree to a more rigid control, it would be very costly to change existing systems to remove all vulnerabilities. As seen with the Y2K problem, many systems that have been in use for a relatively long time are still working, so there is no use in replacing them until necessary. A subtle point concerning vulnerabilities in the code is that a company may not mind them, as long as no one is exploiting them.

- In security the concept of composition is problematic. Even if object A and object B are secure, the composition of them both may not work in a secure fashion. As software development is otherwise built on the concept of modules being assembled to create a larger entity, the proof of the total security of the system can be difficult. When better formal methods are developed, this problem may be alleviated.

1.1.2 Techniques

When trying to detect intrusions, one can follow two main paths [2]:

A knowledge-based approach consists of detecting symptoms of known attacks. This scheme is often implemented with so-called signatures, which define known attacks. By defining what are considered attacks, this method should lead to few false alarms, but it requires an extensive database to define nefarious behavior. The time and knowledge required to update the database are substantial.

A behavior-based approach consists of defining a model of normality, instead of defining malicious behavior. If the tool observes behavior that does not correspond to the model, an alarm is issued. This method can be implemented with neural networks, statistical models or other constructs. There is no longer a need for a database with the newest attacks, but somehow the model must be trained. Usually this approach is associated with a high false alarm rate.

Almost all commercial products today use the knowledge-based approach [2]. The database of attacks must be updated regularly to include new attacks, or they will be overlooked. There are two ways to do this:

- Suppliers of commercial security products often include a method to distribute new databases at given intervals, exactly like a virus protection program, because all new, unknown attacks will be overlooked. This distribution channel may be made available by subscription.

- Users themselves can add new signatures if the language and architecture of the product are open.

The best situation is when both methods are allowed. An ordinary user may find it difficult to keep up with the flood of vulnerabilities found and hence need an automatic distribution. Nevertheless, the user might want to customize the database to reflect the policies of the company. Some usage may not be malicious in itself, but might be interpreted as suspicious or undesirable to a company. If the user can add his own entries to the signature database, these cases can also be covered.

1.2 The Web Server

The problems of intrusion detection grow when a vulnerable computer is connected to a network. If that network also happens to be the Internet, anyone in the world can reach the computer [3].

The World Wide Web was originally a networked information project at CERN, in the ECP/PT group of which Tim Berners-Lee was a member [4]. Formerly confined to the academic world, it has since grown tremendously and has now become an important channel for businesses and their customers. It is often one of the first electronic interfaces used by people looking at a new company. The web server can be regarded as a company's electronic front door, and its omnipresent nature makes it a natural target for attacks in terms of [5, 6]:

- Image. A break-in might be seen by thousands of people within a matter of hours and give fame to the culprits. This is being done more and more often to stage a political protest, or even to commit low-cost, high-reward terrorism.

- Many web servers are involved in commerce, and they may constitute a repository of financial information such as credit card numbers. Information found might also be used for blackmail and other criminal activities.

- Web servers are a natural bridge between external and internal networks containing proprietary information. This facilitates industrial espionage.

1.2.1 Faults of Web Servers

Put simply, there are three ways to break into a web server host [7]:

- Attacking services other than the web server (this method is not discussed in the current report).
  This includes a misconfigured machine, denial of service against the IP stack, etc.

- Using HTTP to attack the server software (http daemon).
  As with any complicated program, and web servers and clients are very complex, there are bugs that may be used maliciously to break into the system. There are mailing lists, such as *bugtraq* [8], that discuss a multitude of vulnerabilities anyone can learn about.

  One finds that malformed headers in a request may wreak havoc. Long cookie names may cause trouble, and long strings carry the risk of causing buffer overflow. There are also more advanced attacks and they can be easily deployed because the necessary programs are freely available for downloading from the Internet [9].

- Exploiting weaknesses in installed programs executed on a server (where we concentrate on scripts using the common gateway interface, cgi).
  Web servers are vulnerable to one of the easiest attack methods, that target installed cgi programs. Anyone who has read the publicly available "Internet Security FAQ" [3] can launch an attack by typing a few characters into their web browser. The success of the attack depends on the know-how of the system administrator at the receiving end. Unfortunately, many identified security problems remain unpatched [6].

  The ease with which these attacks can be performed increases the risk they pose. As an analogy, few people would actually go out and buy a ladder to enable them to break into an apartment, but if presented with a ladder and shown an open window, certain individuals may be more likely to do so.

1.3 Common Gateway Interface

This section describes the common gateway interface, and how it creates dynamic web pages. Then the dangers associated with this practice are discussed.

1.3.1 Introduction

The first web servers were designed to provide static pages for a user to download to his client software for rendering. Even though this idea was revolutionary at the time, its use was limited. As the pages were static, there was no user interaction and the most a company could do was to present predefined information. What was missing was the ability to build dynamic web pages on the fly. This would enable a user to modify what he could see depending on his preferences. Today JavaScript, Java applets/servlets, ASP, and PHP have closed this gap,[1] but the first solution to do so was an interface between the web page and any program located on the server. This interface is called the Common Gateway Interface (cgi) and is still widely used.

By letting the web server run programs on the same computer and pass arguments from www forms to these programs, almost everything can be implemented. Virtually any user who has surfed the web has encountered such programs. For example, consider search engines such as AltaVista. The user composes a query, the web server executes a program to locate

[1] These new techniques also have had security vulnerabilities. The interested reader is referred to [5].

the matching pages, and presents the results to the user. This could never be implemented with only static information.

1.3.2 Vulnerabilities

There are several vulnerability issues concerning cgi programs. First, to be able to present useful information, the web server may need to be connected to the intranet of the company if mission-critical databases are located there. This means that if the web server is compromised, the intranet may also be in danger.

Second, all enterprises require their own hand-made cgi scripts to access their database and present the information correctly. This has led to a multitude of programs written by programmers who are experts in providing information services, but are not necessarily aware of the security implications of network programming [7].[2] Let us compare this situation with that of actual web servers or their underlying operating system. No company would develop its own web server, and hence there are only a handful of widely used servers. These are programmed by experts who have had to learn about security the hard way [3]. The situation is mirrored when it comes to OS, with only a few serious players in the game.

Third, to create dynamic content, cgi programs get their input interactively from any user having access to a www page. A few years ago the critical databases of a company were run by competent staff who were very experienced in handling the pertinent software, whereas today any type of input should be expected.

Fourth, cgi programs are run locally at the server and the consequences of the exposed vulnerabilities reflect on the web server host, not on the client.

1.3.3 Example: The test-cgi program

Let us take one example of a vulnerable cgi script [10]. The NCSA and Apache web servers supply a program called `test-cgi`. It is used to check whether the web server is set up correctly for running cgi programs. Once the server is up, a prudent administrator should remove the program, which implies that he knows about the security concerns related to the script. However, the script is often left on the computer [6], and it has happened that it was renamed `test-cgiBad`, because this seemed safer than irrevocably deleting it.

The `test-cgi` program lists a few variables in the environment passed to it. As it was originally designed to be used only by administrators, no checks of the input were made. To run the program, one simply types in the location field of the browser:[3]

`http://vulnerable.host.site/cgi-bin/test-cgi/ExtendedPath?QueryStr` (example 1.1)

The program then lists the value of the query string, as well as the value of the extended path, among other things (output 1.1). A problem arises if a user lets the value of the query string be a wildcard (`*`), in which case the script prints the contents of the `cgi-bin` directory.[4] If a malicious user can see the contents of this directory, he can look for other

[2]Unlike the Java sandbox, the Common Gateway Interface has no inherent security model, and this leaves the entire security issue to the programmer.

[3]The Extended Path and the Query String are two arguments that can be provided to the invoked cgi script.

[4]More accurately put, the contents of the directory where the file is located are shown.

vulnerable programs and then launch targeted attacks against them.

In 1996, there was a lively discussion about this program. In more recent web servers, a wildcard in the query string or the extended path no longer expands a listing of a directory. But, in early 1999, a new vulnerability was discovered. The `test-cgi` program also prints the method used to access the program. When using a browser, this always has the value `GET`, but a user may supply any value by using the `telnet` program, even a wildcard. With this program, one can type (the result is shown in output 1.2):

```
telnet vulnerable.host.site 80
```
(example 1.2)

```
* /cgi-bin/test-cgi
```
(example 1.3)

```
CGI/1.0 test script report:

argc is 1. argv is QueryStr.

SERVER_SOFTWARE = Apache/1.3.3 Ben-SSL/1.28 (Unix)
SERVER_NAME = tournai.gsal.be
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/0.9
SERVER_PORT = 8080
REQUEST_METHOD = GET
HTTP_ACCEPT =
PATH_INFO = /ExtendedPath
PATH_TRANSLATED = /usr/local/apache/1.3.3/share/htdocs/ExtendedPath
SCRIPT_NAME = /cgi-bin/test-cgi
QUERY_STRING = QueryStr
REMOTE_HOST =
REMOTE_ADDR = 10.10.20.41
REMOTE_USER =
AUTH_TYPE =
CONTENT_TYPE =
CONTENT_LENGTH =
```

Output 1.1: Output from the normal use of the `test-cgi` program.

Thus, we are back at square one, because no one expected a user to be able to modify the value of the method.[5] This can be compared to fixing a leaky bucket by plugging one hole after another, instead of replacing the bucket with a new one. If security was not a concern from the beginning, it is very difficult to add it later on. All regular users would be able to access the `test-cgi` program through a browser, and then there would be no way to change the value of the method. No one considered years ago that a cracker may use another, more

[5]The command `set -f`, which disables filename globbing in a Bourne derivative-type of shell, was present in the Apache web server script tried above. The line was commented out to show the desired effect.

```
CGI/1.0 test script report:

argc is 1. argv is QueryStr.

SERVER_SOFTWARE = Apache/1.3.3 Ben-SSL/1.28 (Unix)
SERVER_NAME = tournai.gsal.be
GATEWAY_INTERFACE = CGI/1.1
SERVER_PROTOCOL = HTTP/0.9
SERVER_PORT = 8080
REQUEST_METHOD = form.cgi printenv test-cgi
HTTP_ACCEPT =
PATH_INFO = /ExtendedPath
PATH_TRANSLATED = /usr/local/apache/1.3.3/share/htdocs/ExtendedPath
SCRIPT_NAME = /cgi-bin/test-cgi
QUERY_STRING = QueryStr
REMOTE_HOST =
REMOTE_ADDR = 10.10.20.41
REMOTE_USER =
AUTH_TYPE =
CONTENT_TYPE =
CONTENT_LENGTH =
```

Output 1.2: Output from the `test-cgi` program when sent wildcards in the method name. Note the list of the `cgi-bin` directory returned.

flexible way to access the program. For more information on this vulnerability, the interested reader is referred to [10, 8].

1.3.4 Search Engines Issue

To make these vulnerabilities even more serious, search engines collect information about all the documents and programs installed at a site. A cracker need only go to AltaVista, for example, and request a list of all sites running a vulnerable program. The search engine indexes many Internet sites, and will promptly deliver a list of vulnerable hosts. Today's commercial search engines do filter queries, however, so a search for `test-cgi` would most likely yield only documents discussing the vulnerability, but not actual sites where the software is installed. Nevertheless, new search engines may not have implemented this feature, and nothing stops a cracker from building his own search engine either. Newly discovered vulnerabilities are not filtered out in the search engines either, so by following the discussions on the bugtraq mailing list, and immediately searching for the vulnerable program, it would not be too difficult to break into a new site.

At least the well-behaved search engines can be stopped by using the file `robots.txt`, which restricts indexing in the `cgi-bin` directory [11, 12].[6]

[6] The most commonly used directory for executables is `cgi-bin`, but it is site-dependent. There are also

As there is no way to enforce the contents of the `robots.txt` file, a system administrator may want to keep track of the documents being accessed by robots. One practical study shows that a human usually exhibits different behavior than a robot [13]. For one thing, humans take an average of 107 seconds to read a document. By keeping track of documents accessed by robots, it is possible to see whether they ignore the wishes put down in `robots.txt`, such as not indexing programs in the `cgi-bin` directory. The system administrator might want to consider blocking this site (robot) if that is the case.

1.3.5 Safe CGI Programs

Simply put, there is one main area of concern: *checking user input* [14, 15, 3, 16]. The frequently overlooked fact is that a cgi script takes its input from the web, which any user in the world can supply. The programmer designs the program with a desired functionality in mind, but he must also consider what the program should not do. How does the program behave when confronted with abnormal input? If enough care is taken to sanitize the input, many of the pitfalls described above can be avoided.

The following anecdote may serve as a reminder. Once, a routine was supplied with a web server to automate the checking for malicious input. Many programmers used it over time, and had a false feeling of security. Later it was discovered that this routine had actually missed a character it should have filtered out. The lesson to be drawn is to check for what is expected, and not to attempt to check for the unexpected.

If you expect two numbers, you should check to see that you actually have two numbers on the line, and not to try to see whether you have any other characters there. It is easy to check that the input conforms to something expected, but almost impossible to take into account all input that could be wrong. Unfortunately, this may not always work because correct input is sometimes indistinguishable from the malicious input.

This fault has two facets, depending on the chosen implementation language of the cgi program:

- If using a scripting language, such as Perl, care should be taken to verify each character. In a scripting language, it is easy to pass on the input directly to the operating system, which may then misinterpret the information.

- If written using a compiled language, such as C, an application is less likely to mistakenly execute unintended commands. Nevertheless, problems can arise with memory management. If the input is too long, there may be a buffer overflow. By carefully crafting the input arguments, a cracker can then get the program to execute an arbitrary command.

There are other problems related to network programming. Care has to be taken to avoid race conditions, for example. If one assumes that two things happen subsequently but this is not always the case, the program may be vulnerable.

examples of `bin`, and at certain universities running cgiwrap, any user name followed by `cgi-bin` is a valid directory. For simplicity, `cgi-bin` is used in this thesis to signify all executable directory names.

By carefully exercising the user privileges in a proper operating system, the damage can be restricted. The cgi program must still go through the checks of the operating system, so if a cracker manages to compromise the program, he may not gain much if the server is correctly set up. Unfortunately, many servers are running as `root`, which means that a cracked server provides open access to the entire machine.

To simplify the administrator's task, many web servers are shipped with as much enabled functionality as possible. It requires some work to make the servers secure by turning off many of the "features." This also needs to change. Administrators must be made aware that the more functionality is turned on, the more pathways there are into the intranet [3].

1.4 Similar Tools

There are many programs on the market to protect a system from intrusions. Few concentrate on the www server, and of those, the functionality they offer is different from the requirements given in Chapter 2. A representative, yet not exhaustive, list of products follows:

WWWstat [17] is mainly a program to collect statistics of the web server usage. Even though it can be an invaluable tool, it does not provide much functionality for intrusion detection.

Autobuse [18] is a framework for analyzing log files. It offers some support for analyzing firewall logs and web server logs. The web server attack signatures accept Perl regular expressions, but they are limited. They only look at the `url` and for method names of type GET. The implementation is also slow.

LogCheck [19] is a program that accepts Perl regular expressions, and takes action if it finds them in a log file. The actions supported are *mail, page, exec, file, and ignore.*

Logscanner [20] is a framework in which rules can be incorporated. It automatically contacts a responsible person if necessary, and it feeds the logs into functions made by the user. There is not much included functionality to test for malicious behavior.

Commercial Intrusion Detection Software Owing to the limited scope of my survey, I will omit the names of commercial IDSes. For more information, see [21]. Few products exist that specialize in interpreting the logs of web servers. Some systems offer limited support in detecting www server attacks, but others do not even support the encoding scheme defined in HTTP (as specified in [22]), which means a cracker can easily circumvent the IDS.

Overall, there is little protection against www server attacks. Of the programs listed above, none fulfills the requirements given in Chapter 2. The way of writing attack signatures is too limited, they provide no support for the encoding scheme defined in the HTTP protocol, and they have no way to keep track of known suspicious hosts.

They do, however, supply a nice platform for reporting the errors to an administrator through diverse means. Unfortunately, this feature is of little use because we desire a product that runs in real time and sends messages to a system management tool. Many of the products found can only run on the Wintel platform.

Some www servers also have the option of saving the log files into an SQL database [23]. By writing powerful queries and using data mining, a lot of information can be recovered from the logs. The disadvantage is that this database is very space-consuming, and the queries cannot be run in real time. It is not a simultaneous process: The logs have to be saved, and then the queries can run.

For programs not related to security, but pertaining more to statistics, web site administration, and keeping track of users, see the overview at:

`http://ipw.internet.com/analysis/index.html` (example 1.4)

Chapter 2

Requirements

The requirements imposed on the design of the monitor are that it be

- lightweight in terms of resources required;
- portable, i. e. able to run on various platforms with different servers;
- able to send events to a system management tool, in our case the Tivoli Enterprise Console;
- able to process log files in real time.

None of the existing tools surveyed in Section 1.4 fulfills the requirements above. Moreover, the desired functionality calls for

- flexible attack signatures for the knowledge-based approach;
- the ability to track hosts that have exhibited malicious behavior, the assumption being that their intentions are malicious and that they might repeat such efforts. By studying the behavior of a suspicious host, we may deduce new signatures to add to the database;
- some sort of functionality or intelligence to decide what is a real alarm, and to remove false alarms;
- the ability to learn by itself, to avoid having to update a database;
- modular design to facilitate future extensions.

When examining attacks against a www server, the first thing one looks for are known vulnerable cgi scripts. The program must be able to read a database and find these known scripts. In principle, the various attack scenarios can be divided into three areas depending on the goal of the hacker:

1. **Penetration of the system**
    - known vulnerable cgi programs
      This includes simple pattern matching of names, but also extremely complex patterns.

- password guessing
  For example, consider the case of a resource protected by a password, where a user keeps failing to access it. Is he trying to guess the password?
- guessing installed cgi programs
  A user tries to access `/cgi-bin/prog1`, `/cgi-bin/prog2`, `/cgi-bin/prog3` etc several thousand times. This is clearly an attack to find out whether the site has any vulnerable cgi programs installed.

2. **Denial of service**

   - repeated accesses of non-existing resources,
   - repeated accesses of resources that cause a server error.

   What does it mean if a host fails to access a document *several thousand times*? Even if it is not a hostile attack, an administrator would like to know about it because each request served uses server resources.

   If it is a broken link, it should be corrected, and if it is a poorly configured robot, the site in question should be informed. This case is less critical than the similar ones described above, where more than server resources is at stake.

3. **Legal but undesirable activity**

   - strange use of the HTTP protocol
     Certain behavior is allowed in the HTTP protocol, but its use is very undesirable and should be questioned. As described in Section 1.3.3, any method name can be used when accessing dynamic resources, and this can be used to exploit vulnerabilities.
   - sensitive documents
     Certain documents should not be accessed through the web server because their contents are confidential. Examples are listings of the `cgi-bin`-directory, configuration files of the web server, and password files.

By examining the information supplied, one realizes that a vulnerable cgi program *and* a successful status code (200) pose a greater threat than only a failed attempt. The conclusion to draw is that the signatures used should be able to span several fields of the request, and they should obviously be very flexible.

2.1 Monitor Input Channel

To protect the www server, we need to be able to follow what kind of input is sent to it. As the www server in itself is quite stable (as described in Section 1.3.2) and the main vulnerabilities are in connection with the programs executed, the log files can be used to see what happens. The advantage of this approach is its portability. The program merely has to be able to read a file, and does not have to act as a wrapper, or a type of transparent filter on the port level.

The disadvantage is that if the server itself fails, no log entry is written and we have no information on what has happened. The monitor also misses low-level attacks, like SYN flooding. This is still not a severe restriction because this tool is intended to be used in conjunction with other intrusion-detection software.

Also, by relying on log files, the program may later be easily extended to new functionalities.

2.2 WWW Servers and Log Files

The most widely used web servers are generally quite similar with regard to the kind of information written, but we decided to concentrate on the Common Log Format (CLF) [24] because most brands of servers seem to be able to support that option.

As the Extended Log Format (ECLF) is very similar to CLF, it should not be difficult to extend the program to include more fields. We decided not to build the first prototype using this format because most of the log files at our disposal were written in CLF. The extra fields in ECLF (browser agent, referrer field,[1] etc) were available in other files, but it was not possible to correlate this information with the access logs in a fast and flexible way.

Moreover, the error logs were omitted. The format is not fixed, and the information found is duplicated many times in the access logs. Each file access takes time, and reducing the necessary files to be read by 50% saves a lot of time.

The servers differed in what type of information they saved in the logs. Some also had direct support of SQL databases.[2] Normally a CLF format log grows 1 MB for 10,000 requests.

2.3 The CLF Format

A typical log file entry written in CLF [24] looks like:

`host - authenticated_user [date] "request_string" status bytes` (example 2.1)

The first field is the host name of the computer wanting access to a resource. Depending on the server configuration, this may be written as a host name or as an IP address. The latter is recommended because IP spoofing is more difficult than DNS spoofing [5]. The next field, marked above with a dash, is the remote login name if the `ident` daemon is running. This is seldom the case, and even if a user name is present it should not be used for authorization purposes because any person in control of the remote host may set this field to any value.

The next field is the authenticated user according to the HTTP protocol. This result may also be erroneous if the status code is `401` (unauthorized).

The next field is the date, followed by the request within double quotes. Then the status field specifies the status code of the request, and the field bytes is the number of bytes

[1] This field usually occurs misspelled as *referer field*. For this reason the referral logs used in the program occurs as *refererLog*.

[2] Please refer to [23, 25] for information on how to connect the Apache www server to an external database.

returned to the client, excluding headers.

If a field is empty, or zero, a dash "-" is used to mark its place.

Finally, it should be pointed out that, although the CLF format is a de facto standard adopted by many web server vendors, it is not a formal standard.

Chapter 3

Design Principles and Implementation

3.1 Building Blocks of the Monitoring Tool

From the CLF log, we have several fields to work with. The most important part is the *request string*, but the field *host* is also very useful. The program mainly uses a knowledge-based approach to detect attacks. By searching for known attacks, the monitor can raise so-called *alerts*. The *alerts* signify that something unusual is going on, and if enough malicious actions are shown, the alerts turn into *warnings*, which are passed on to a console.

The structure of the program is represented in Figure 3.1. There are several building blocks, which perform various checks on the logs. The layered architecture makes it easy to change the functionality of a certain block and adapt the program to other uses. Before discussing in detail the functionality of each module, a superficial description is proposed.

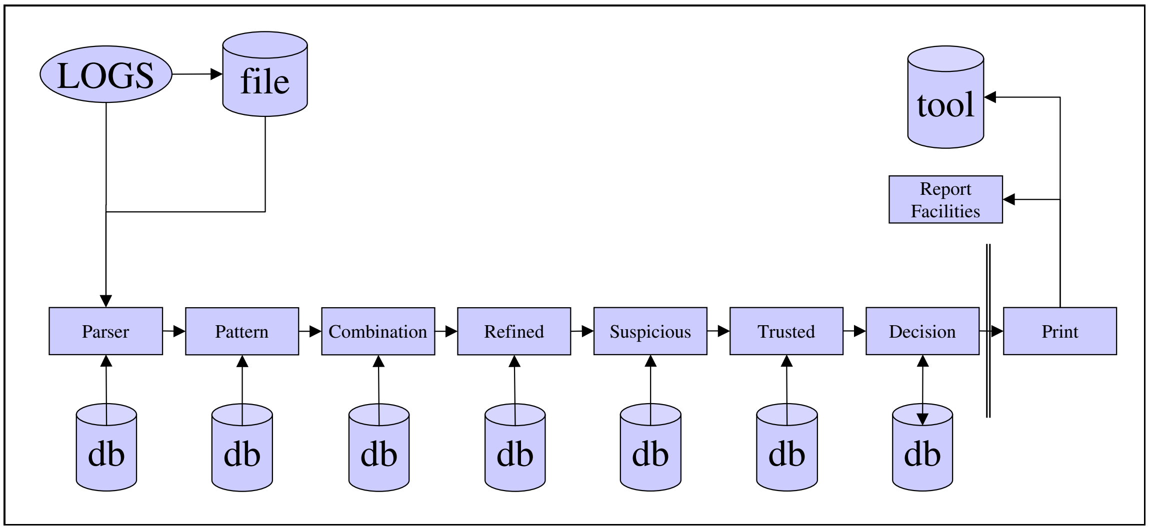

Figure 3.1: The layout of the monitor with all the building blocks.

When the program receives a new request, it creates a data structure encapsulating the log entry. Each module performs certain tests, and adds more fields to this data structure, which is *pipelined* through all the blocks. Each subsequent block will be able to use information stored in the object by a previous module. That is, each block merely agglomerates more information to the object going through.[1]

- The *Parser Module* breaks the request into pieces to ensure that a valid URL has been written into the log files. It creates attributes of the data fields found in the CLF format. Other modules later use these attributes to search for malicious signatures.

- The *Pattern Module* uses these stored attributes to look for its attack signatures, and it stores any findings (as *alerts*) in the object. Each field of the CLF can be searched explicitly for a special signature in the database. The signatures are Perl regular expressions, ranging from simple strings to complex structures. The module ensures that the overhead introduced by regular expressions is kept to a minimum, in particular by compiling all expressions once at the beginning of execution.

- The *Combination Module* allows attributes to be grouped under a common name. As some conjunctional conditions are more dangerous than their separate pieces would suggest, this module lets a user create a new alert depending on old information found. There is no new search in the CLF log; instead the module uses information already found by previous modules. It allows new alerts to be created if a Boolean logic expression is true: `if alert1 is present but not alert2, please create alert3`.

- By using the *Refined Module*, a user may refine a search completed by the *Pattern Module*. The refined search may be performed on the same field from the CLF or any other attribute in the data structure.

- The *Suspicious Module* checks whether the request originates from a host marked as malicious. This feature enables the monitor to keep track of hosts that raised an alarm while analyzing a previous request. By studying these new requests, the database can be kept up-to-date. This is a manual process in the current implementation, but could be automated in the future, at least partially pending review and creation of the appropriate regular expression.

- The *Trusted Module* removes[2] alerts written by the previous modules if it can match a *trusted signature* against some of the fields stored by the *Parser Module*. Without this module, there would be many false alarms and no way to regulate the flow. This module makes it possible to approve manually certain actions performed by certain hosts, or to say that a special URL is safe.

- The *Decision Module* analyzes the resulting object and decides whether it should be sent on to the management console (file), based on the amount of malicious activity the suspicious host has previously generated. The goal is to define a threshold for reporting in order to prioritize alarms. A single instance of a vulnerable CGI script

[1] The *Combination Module* and the *Refined Module* are implemented as only one module because the functionalities are similar.

[2] It was decided in the implementation phase that it was more efficient to delete alerts than to mark them as invalid. Thus we deviate slightly from the design given above, where we specify agglomeration as a key word.

request should be reported immediately, whereas accesses to non-existing documents are reported only when the number of requests is associated to a likely denial-of-service attempt.

- The *Print Module* prints the received object, if any.

To simplify the presentation, we let the monitor read one request from a log file, and show the resulting data structure after each module/section below. The request used is:

```
hacker.paradise.bad - - [30/Apr/1999:22:25:50 +0200]
"GET /\%*7cgi-bin/ph\%66?cat\%20passwd.txt" 400 -
```

This is a failed attempt to go through the vulnerable cgi program `phf` to access the password file. The data structure shown in the following consists only of the new parts added, to make it easy to see the changes made. For some fields, we have also changed the output to a more comprehensible form.

3.2 Implementation Language

The choice of language for the prototype was quite easy. Perl has many advantages for prototyping lightweight tools, especially when one has to treat text. Its dynamic memory allocation and powerful and fast regular expressions made it a suitable choice. Thanks to the included data structures, the *time to market* is quite short.

Perl has also proven itself to be stable and free of severe bugs in the later releases. Furthermore, the www server often has Perl installed to run cgi programs, and thus there is no space penalty to use it also to check the logs. Of course, the monitoring tool can be placed on another computer to ease the load on the www server machine. The program does not support this option in real time in this version, but by adding yet another module, certain web servers will have no trouble sending their logs to another machine where the analyzing takes place. Last, but not least, Perl exists on many platforms, which makes it a very portable option.

A final remark: The code was optimized for speed by sacrificing memory.

3.3 Parser Building Block

The four tasks performed by the *Parser Module* are to:

- read the request,
- break the log entry into its constituent parts,
- refine the URL into its parts,
- decode any encoded characters.

Each step has its own set of checks made, as described in detail below.

3.3.1 Program Input

The program needs to access the data from the log server. In the current version, there is the option of either piping the logs directly into the program, or letting the program read a complete file. By concentrating on Apache, we chose to have this www server pipe the logs directly to the program.

The option of reading a file allows off-site analysis and the ability to go back and check saved logs. This is not as powerful as a real-time analysis, mostly because the only information available are the contents of the logs. The whole server layout of pages may have changed since the time of the attack.

3.3.2 Interpreting the Logs

The log format is then broken up into parts and saved into the data structure as attributes. Sanitizing checks are made to ensure that the entry conforms to what is expected. If the monitor fails this step, a warning is issued. This condition can prevail:

- if a severe error of the web server has occurred (serious), or
- when web servers write the format of the logs as the first line of their output (harmless).

The monitor does not differentiate between these two situations, but leaves this judgment to the administrator.

The second step is crucial for the rest of the functionality of the program. If the fields cannot be identified, no subsequent tests can be done. Moreover, if there is no host found in the request line, there is no way to trace the request back to the originator.[3] As soon as the line has been successfully parsed, all other checks can be performed.

After this step, the data structure containing the request has several new attributes, all of which correspond to the fields in the CLF log.

3.3.3 Refine the URL

The third step is to parse the request string. It should have a format like:

`method url?query` (example 3.1)

Two different alerts are signaled depending on the format of this string:

empty string. If an old web server receives an empty `url` string, it identifies itself. A cracker can then use a directed attack and more easily compromise the web server. These requests do not necessarily imply an attack, though, because bad network conditions and time-outs may also cause them.

strange format. The format of the string does not conform to the rules specified in the HTTP protocol [22]. This usually means that the request was made using a non-standard client. The intent may have been malicious and the strange format applied to confuse the server, or it may simply be a typo.

[3]Note that the host name found in the logs does not necessarily correspond to the host that sent the request because DNS spoofing or a similar scheme may have been used to hide the tracks. Nevertheless, it gives an indication of the origin.

This step adds the two additional attributes of *url* and *query* to the data structure wrapping the request. Note that the variable *extended path* is not added separately. Its location is included either in the url or in the query attribute.

3.3.4 Decode All Characters

The fourth step of the parser module is to decode any possible encodings found in either the `url` or the query string. The HTTP protocol allows characters to be sent in an encoded form, and a control character must be so sent. Other characters can be encoded optionally. The encoding scheme is very simple: the ASCII code of a character is written in hexadecimal notation and is preceded by a percentage sign (%). The percentage sign is in itself a reserved character, and if it is present the next two characters must be the hexadecimal ASCII code for an encoded character. The only way to submit the % in a `url` is to encode it as `%25`.

The monitor starts by scanning the relevant attributes for the % sign and tries to decode the postfixed hexadecimal code. If that is not possible, an alert is issued. This may happen for one of three reasons:

experimental protocol in use. As of HTTP version 1.1, only a hexadecimal code may follow the % sign; any other character is illegal. The HTTP protocol allows for experimental protocol versions and it may be that one of them is used. For example, they may have assigned a special value to the sequence `%*$`.

a non-standard agent is used. The commercial agents are thoroughly tested and seldom send strange requests. It may be that a request was generated by a non-standard agent, such as `telnet`, and the encoded character contains a typo.

an attack. The reason for the unusual encoding is to circumvent detection and confuse a cgi program in order to break into the computer hosting the web server.

If there are encoded characters, we also perform a check to see which characters were encoded and add them to the data structure containing the request. As we pointed out above, the HTTP protocol allows any characters to be encoded, but there should be no reason to encode alphanumeric characters. Sometimes such encodings are made for legal reasons, but they might also be to avoid some security controls. If a company has deployed an intrusion-detection system that raises an alarm for packets to the web server containing the string `test-cgi`, the scheme fails if the cracker instead sends `%74est-cgi`. The firewall lets the packet through, the web server decodes the `url` and then launches the vulnerable cgi program.

3.3.5 Resulting Data Structure

After these steps we have a complete request in memory. The various fields have been parsed into different variables, and several checks have been made to ensure the correct format of the entry. In Table 3.1, the resulting data structure can be seen, with examples of the values stored in the various attributes. The reason that some fields are empty is that

Attribute Name	**Value**
reqId	`924182493_1`
accessLog	`hacker.paradise.bad - - [30/Apr/1999:22:25:50 +0200]` `"GET /%*7cgi-bin/ph%66?cat%20passwd.txt" 400 -`
host	`hacker.paradise.bad`
rfc931	`-`
user	`-`
date	`30/Apr/1999:22:25:50 +0200`
reqStr	`GET /%*7cgi-bin/ph%66?cat%20passwd.txt`
status	`400`
bytes	`-`
method	`GET`
url	`/%*7cgi-bin/phf`
query	`cat passwd.txt`
version	`0.9`
decodedUrl	`f`
decodedQuery	`[sp]`
alerts	*see Table 3.2*
warnings	`<empty>`
hostParts	`<empty>`
errorLog	`<empty>`
refererLog	`<empty>`
agentLog	`<empty>`

Table 3.1: Resulting data structure after the *Parser Module* has finished. The actual parsed log entry can be found in the attribute `accessLog`. All of the attributes are searchable in later modules.

Alert Parameter	**Value**
engine	`parser`
class	`suspiciousHexCodesUrl`
sig	`f`
text	`f`
engine	`parser`
class	`invalidHexCodesUrl`
sig	`%*7`
text	`%*7`

Table 3.2: Alerts included in the data structure shown in Table 3.1, shown separately for simplicity. The parameters shown explain where and why the alert was created, see discussion in Section 3.4.4.

- a functionality has been turned off, such as in the attribute *refererLog*,
- a later module adds the value, such as in the attribute *warnings* (created in the *Decision Module*).

There are two things to note in Table 3.1. First, even though an unrecognized hexcode was found, there is no special field in the data structure showing its value. It is only reported in the alert. The coded structure is also left in the url because there is no correct way to decode it. It cannot be deleted since it is part of what the web server sees.

Second, if there were two alphanumeric characters encoded, only one would be saved in the alert. The reason for this is to avoid unnecessary searches. The agglomeration of this error does not make it more severe, and for this reason the monitor can stop looking as soon as it has determined that there is at least one encoded alphanumeric character.

A lot of information is duplicated in the structure for faster access. As stated above, the implementation sacrifices memory for speed.

3.4 Pattern Module

The most important part of the program is where we look for patterns, *signatures*, found in real attacks. When writing the attack signatures, three different steps are involved:

- creating a signature,
- deciding where the signature should match,
- grouping similar signatures together.

3.4.1 Creating a Signature

As described in Section 1.1.2, there are two ways to store the database. We have chosen an open design, where anyone can add a new attack signature. By simply typing the offending program name, the monitor issues an alert if it finds that program name. As soon as a new vulnerable cgi script is found, its name can simply be inserted into a file to make the monitor react to it. This means *any* person can make additions to the database, even if they have never programmed before.

The signatures can be much more complex, though. As a simple signature might trigger many false alarms, it is best to customize it until it detects the attack but never triggers false alarms at the site where the monitor is running.

For example, imagine we have heard the program `wrap` is vulnerable.[4] By simply writing `wrap` into the correct place in the signature file, the monitor reacts to any urls containing that program name.

The problem we notice is that this signature is slightly too general. It creates false alarms too. The site we are monitoring, also runs the wrapper `cgiwrap` [26], and the accesses to

[4] According to a Bugtraq message, this program lists the contents of any directory with permissions set to 755 when receiving the URL `http://sgi.victim/cgi-bin/wrap?/../../../../../etc`.

this program should not trigger an alert. We need to find a way to differentiate the use of the vulnerable program from accesses of this wrapper.

`http://sgi.victim/cgi-bin/wrap?/../../../../../etc` (example 3.2)

`http://SERVER/cgi-bin/cgiwrap/USER/SCRIPT` (example 3.3)

The easiest way is to specify that the preceding character should be a slash (`/`), and not an alphanumeric character, as in `/wrap`. No more false alarms will be generated.

The signatures can be any Perl-compliant regular expression. The powerful regular expression engine lets the signatures be refined to match exactly a vulnerable program with few false alarms. As a user gains proficiency with regular expressions, he is able to craft more powerful signatures but does not need this knowledge to create the first signatures.

3.4.2 Matching a Specific Part

The second design question is to which field should the signature be matched. Any of the fields shown in Table 3.1 can be specified. This is usually a very straightforward decision because the signature is tailored to a particular context. If we look for a vulnerable cgi script, we want to match the `url`. If we are looking for a status code indicating an error, we want to match the `status` field.

We have two advantages from this design:

- speed,

- fewer false alarms with intuitive matching, and simpler signatures.

The program does not need to search for the pattern in all fields available to it. It is sufficient to look at a single attribute. When we are looking for a vulnerable cgi program, there is no reason to spend computing resources by trying to match the signature against the date when the only valid match can be in the field of the `url`.

We know where we want the match to be when we design the signature. By being able to specify this information explicitly, many false alarms can be avoided. Compare this with using `grep` on the log files. With `grep`, the entire access log (*only*) will be examined. If we are looking for status code `404` (not found), we can get results matching the bytes sent back, the host name (`dial-up404.some.site`), or `url`s such as `/images/pict404.gif`. If we did not have this context-sensitive search, the signatures in the database would have to be much more complex to filter out false alarms.

As another example, we can take the file `/etc/passwd`. Many exploits concern reading this file. Thus we create a general signature `/etc/passwd` and match it against the `query` attribute of the data structure shown in Table 3.1. We may be able to catch a few unknown attacks with this signature because we are targeting one objective of the attack and not its amorphous structure.

3.4.3 Matching to Enforce a Policy

In the previous section we showed how to write a signature that would trigger an alert if the pattern was found. We have also enabled signatures that *must* be present within a request. These may uphold a policy.

For example, consider an internal web server that is supposed to be accessed only by internal hosts. These hosts comply with a certain name convention (such as `ibm.com`). If a request lacks this signature, something has gone wrong and an alert should be sent.

These signatures are created exactly like ordinary signatures, with the difference that when the field it should be match against is specified, we say that it is a policy enforcement signature that *must* be present in the field.

3.4.4 Grouping Similar Signatures into *Classes*

In many cases, the signatures created are very similar in nature. They may all match on the `url` field. They may all concern vulnerable cgi scripts, or the danger posed is about the same. There is a feature to *group* similar signatures into *classes*, which has several advantages:

- **speed**
  The monitor searches for signatures in each *class* separately. As soon as one signature in the class is found, the monitor jumps to the next class. The signatures contained in the same class should be more or less equal, and there is no use to continue searching after an alert of a certain severity has been found.

- **easier creation**
  As signatures can be grouped together, an informal structure is kept in the database file and less information must be input for each signature instance. This reduces the risk of errors.

- **easier interpretation**
  The output includes the name of the class containing the malicious signature found. If the name is descriptive, it is much easier to see the problem. As seen in Table 3.3, the four fields of an alert are `engine`, `class`, `sig`, and `text`. The `engine` indicates the part of the monitor where the alert was created, and the `class` further refines this description to a specific class. The `sig` field contains the actual signature, whereas the `text` field might give a more detailed explanation in some cases.

As an example where it is useful to group similar signatures into classes, we can take the following cases:

- **vulnerable cgi scripts**
  This is a list of vulnerable cgi scripts. They can be executed, and the arguments to these programs should be analyzed carefully. It is possible to differentiate this group into smaller classes, based on the vulnerability of each script (gives away server information, allows user to become root).

- **sensitive files**
  These files should not be accessible for downloading. They contain confidential information, but because the files cannot be executed, this group should be treated differently than the class described above.

3.4.5 Resulting Data Structure

The module fulfills most of the requirements specified in Chapter 2. We can now find any text expressed as a regular expression in any field of the request. In terms of a concrete example: any request that encodes the character *t* as `%74` can be found without false alarms.

Extensions made to the *Parser Module* automatically enhance this module too. When the former also reads referral fields and browser agent information (found in ECLF), signatures using the additional attributes can be written immediately. Even if these fields are not present in the current logs, the signatures can be in the database and they are ignored.

The data structure containing the request for the `phf` program has a few extra fields, shown in Table 3.3. As can be seen, these alerts were created in the `engine pattern`. This corresponds to the module *Pattern Module* (other modules may also give this engine name, though). The class names should describe the objective of the signature. There are three alerts:

- **cgi**
  This `class` contains vulnerable cgi programs. The monitor discovered an attempt to execute phf.

- **clientError**
  The signature `^400` matches the `status` field of the data structure. The code signifies that the server received a request it did not understand. The abnormal encoding of `%*7` is probably the reason. If this is removed from the request, our server replies with status code `404` (not found).

- **badStatus**
  This signature also matches the `status` field, but has a much higher granularity than the signature above. It issues an alert for all status codes that are not successful (2xx) or for the status codes indicating redirection (3xx).

3.5 Combination Module

The combination of certain alerts might be more dangerous than the single parts give reason to believe. In the request shown in Table 3.1 there are one vulnerable cgi program *and* encodings of normal characters with the HTTP encoding scheme. Even though the severity of the first is quite high, the combination of these two events is even more suspicious.

The monitor allows the user to create new signatures with Boolean logic on already found alerts. It is possible to specify a signature in the following way: *Raise an alert if alert1 and alert2 exists but alert3 is not found.*

Alert Parameter	**Value**
engine	`pattern`
class	`cgi`
sig	`phf`
text	`phf`
engine	`pattern`
class	`clientError`
sig	`^400`
text	`^400`
engine	`pattern`
class	`badStatus`
sig	`^[23]`
text	`^[23]`

Table 3.3: Alerts added by the *Pattern Module*.

In Table 3.4 we show the result after this module. This alert is signaled if the class `cgi` has issued an alert, and also if suspicious hex codes have been found in either the url or the query.

This module can also be used to create signatures spanning several fields. The *Pattern Module* can be used to create alerts for single fields, and this module can be used to combine them into a single name. If truly multifield signatures are needed, we suggest using the field `accessLog` directly instead.

3.6 Refined Module

Certain patterns should not be looked for if some previous pattern has not been found. An example is the successful status code (`200`). This code is not in itself malicious. On the contrary, it is desirable that every request handled by the server return this code. This means the server understood the request and was able to fulfill it.

What happens if the client asks for a vulnerable cgi program, and the server successfully completes the request? We want to have the ability to launch additional searches if certain

Alert Parameter	**Value**
engine	`pattern`
class	`UrlSuspiciousHexCodes`
sig	`.`
text	`.`

Table 3.4: Alert added by the *Combination Module*. This module is closely linked with the *Pattern Module* and the *Refined Module*. For this reason, the engine is still `pattern`, and the value of the `sig` field is a period, which matches anything in a regular expression.

signatures have been found.

The functionalities offered by the *Pattern Module* and the *Combination Module* together are the same as what this module does. The difference in using this module instead is:

- **speed**
  No unnecessary searches are launched for the status code `200` when no other suspicious activity has been found.

- **memory**
  No memory is used for the additional alert of `200` for each normal request. Also, a later module (*Suspicious Hosts*) keeps track of all hosts that have triggered an alert. If the signature `200` was put in the *Pattern Module*, *all* requests from all hosts would be processed into the internal structure of *Suspicious Hosts*. This is clearly not acceptable.

Table 3.5 shows the alert added by this module. The monitor has a set of words, or files, that should be checked if a suspicious cgi program is found. In this list, we have `passwd`.

For implementation reasons, this module is combined with the *Combination Module*. If a combination of alerts is found, a refined search is made. If the refined search is a period (`.`), we have the *Combination Module*. If the combination of alerts is just a single one, with a real regular expression, we have the *Refined Module*. Any combination in between is also allowed.

The refinement is done in only two steps. The *Pattern Module* creates alerts, which may be refined in the *Refined Module*. These alerts may not be further refined, and no signature in the latter module should be dependent on information added there.

3.7 Suspicious Model

This module keeps track of hosts that are deemed malicious by the:

- *Decision Module*, which will add the host name to the list when it causes an *alert*,

- system administrator who can manually add entries to the list.

Every request submitted by the host is passed on to the *Decision Module* for further analysis, even if no malicious attacks have been found in the request.

The functionality can be regarded as specialized signatures tailored to host names. They are added dynamically, depending on the requests sent to the server, or manually by a system administrator.

Alert Parameter	**Value**
engine	`pattern`
class	`suspiciousCgi`
sig	`passwd`
text	`passwd`

Table 3.5: This alert was added by the *Refined Module*.

As all requests have to be matched against the list of suspicious hosts, it is very important that the module be efficient. It would have been possible to reuse the *Pattern Module* for this task, but the regular expressions this module utilizes are too slow and do not scale well. For this reason, the module is kept separate, and it performs an *exact* match on host names through hash tables. If this is considered too severe a restriction, the entries added manually can also be written as general-purpose signatures in the *Pattern Module*.

The module allows different classes of suspicious hosts and specifies how future requests should be handled.[5] New requests originating from a suspicious host can be printed to the console:

- always,
- when an alert has been raised for the request,
- when a warning exists for the request (reported anyway, but it emphasizes the importance).

A system administrator can decide what level of suspiciousness is appropriate for a particular host. As the program adds new hosts at runtime, the administrator can also specify how these new hosts should be treated by setting a default value. The actual addition of this field is done in the *Decision Module*, and we refer to Table 3.6 for the additional entry in the data structure for our sample request.

As written in Section 1.1.2, there is a penalty associated with keeping the database of malicious attacks up-to-date. We hope to alleviate this problem by this module. New signatures can be learned by looking at new behavior of suspicious hosts. This has to be done manually in the current version of the monitor.

3.8 Trusted Model

Some computers are allowed to do more things than others. A system administrator may run probing software once a month to check the security of the system. A user may mirror a Linux distribution, which includes the file `passwd`.

This module tries to filter out false alarms, and it contains two features:

- *trusted signatures*, and
- the *level of trust* given to them.

As this module does not cancel any alerts of the data structure of the sample request (given in Section 3.1), there is no table showing the fields here.

[5] This functionality is provided by both this module and the *Decision Module*.

3.8.1 Trusted Signatures

To avoid false alarms being generated by perfectly safe activities, one can specify signatures which, if found, cancels certain alerts triggered in the previous modules. This module works in the opposite way of the *Pattern Module*. Whereas the latter triggers an alert if the signature is found, this one *removes*[6] an alert if the signature is found. All features of the *Pattern Module* are at users' disposal in this module as well.

For example, consider the user providing a mirror of the Linux kernel. He has the login name `mjn`, and thus the trusted signature

`/~mjn/linux-distr/` (example 3.4)

is created, to be searched for in the field `url` of the object wrapping the request.

3.8.2 Level of Trust

The monitor can discover many different attacks, and even if one computer is allowed to perform one action that raises a (false) alert, it might not be allowed to launch more serious attacks.

It is necessary to decide how much trust should be given to the *trusted signatures* described above. Each signature must have a specification of what types of alerts it is allowed to cancel. This may be a single alert, a list of them, or the special keyword *all*. The latter option gives total trust to the signature, and no action will be reported if it is found in the request. It is suitable for the system administrator's computer. By specifying his host name as the signature, and then allowing all alerts to be canceled, there will be no alarms even though he checks security by running probes at regular intervals.

3.8.3 Creating Signatures

The best way to remove most false alarms is by modifying the actual attack signatures found in the *Pattern Module*. If they are written with care, there should be very few false alarms from similar `url`s. The attack signatures can and should also be ported between different sites because they do not depend on the local data.

Some situations do require the functionality of this module. Below are three examples of when it is appropriate to create a *trusted signature*:

- **system administrator's computer**
  As shown above, the system administrator might probe his own system for weaknesses. The name of his computer should be written as a safe signature because the actual attacks are real and the signatures in the *Pattern Module* should not be modified. In this case, we use the additional information available to recognize that it is not a real attack.

- **previously vulnerable script installed**
  If a site uses a script that previously had a vulnerability, there will be an alarm for

[6] The alert is physically removed, as if it never existed. This departs from the specification, but it improves the implementation.

each request accessing the script. If there is no way to differentiate the actual attack from normal use when looking at the arguments supplied, a *trusted signature* should be created to cancel these alarms. This will not influence the security at the site because the program is no longer vulnerable. It will, however, hide any probes for this vulnerable program. This is better practice than simply removing the attack signature from the *Pattern Module*. The signatures found for this module should be rewritten for each new site, whereas the attack signatures are generally platform-independent. This way, it is easy to update the vulnerability database.

- **wrong context**
  The best way to discover new attacks is to write general signatures. For some sites, these signatures will create false alarms. Instead of reducing the generality of the attack signature, it is better to write a *very specific trusted signature* to cancel the safe behavior. New attacks can still be detected, whereas some behavior is explicitly allowed.

When the monitor is deployed at a new site, the latest attack signature database should be used and no trusted signatures at all. By analyzing all alerts from the monitor and carefully evaluating the requests that caused the alerts, trusted signatures can be created. It is important to understand that they are of an empirical nature, and can only be created *after* the monitor has been deployed at a site.

After running the monitor for a few thousand requests, the system administrator sees that most false alarms come from the text file `passwd` being downloaded. He creates a signature that will match the whole path of the Linux distribution, and specifies the level of trust to be one alert: the downloading of the file `passwd`.

3.8.4 Dangers Involved

It is important to be aware of the dangers involved when using the trusted signatures. If the pattern of the signature matches, some alerts are canceled. If complete trust is given to the signature, all alerts are canceled. In itself that is not dangerous, but what happens if the signature is poorly written and matches requests that were never supposed to be matched? What happens if a safe signature is forgotten, and a user later gets the computer of the system administrator? That user can then wreak havoc without triggering a single alarm.

There are some points that should be observed:

specific The signatures should be very specific, and even though regular expressions are allowed, it is better to write the signature in a way that cannot be misunderstood.

limited There should only be a few trusted signatures. The system administrator will be unable to keep track of them if they are numerous. Also, the monitor requires some time to check and cancel an alert. Each *trusted signature* adds a time penalty.

confidential If a hacker finds out which signatures are trusted, he can use this information to write requests that escape the notice of the monitor.

As an example of the latter case, consider the Linux distribution in Section 3.8.1. If a cracker knows about the safe signature, he can simply write a query such as:

`/cgi-bin/phf/~mjn/linux-distr/?mail_passwd_file` (example 3.5)

The latter part is ignored by the web server and the cgi program, but will cancel any alerts in the monitoring program. The signature above should have been written:

`^/~mjn/linux-distr/` (example 3.6)

where the first character specifies that the line must begin with this text. This is much more specific than the former, and thus safer.

3.9 Decision Module

This module analyzes the resulting object and decides whether it should be sent to the management console (file), based on the amount of suspicious activity the suspicious host has previously generated. The goal is to define a threshold for reporting in order to prioritize alarms. A single instance of a vulnerable cgi script request should be reported immediately, whereas accesses to non-existing documents are reported only when the number of requests is associated with a likely denial-of-service attempt.

The module stores information of all alerts a host has created.[7] When the number of alerts of a certain type and for a certain host is greater than a predefined threshold, the module changes the alert to a warning and sends it to the console. Internally, the module builds a tree from the host name parts, going from the general (com) to more specific on each branch (ibm - zurich - www). The saved statistics are both for the host, but also for the domain. The monitor can thus track a specific domain (such as a service provider, or a university where users do not have the same host names at every session) as well as specific hosts. If either one performs too many suspicious activities, the monitor issues a warning.

There is a mechanism for lowering the value of old alerts, so that the module does not base its judgment on too old of requests. The monitor takes into account the *age*[8] of a stored alert, and decreases its importance if it is old. This reduces the number of false alarms (see Section 3.9.1).

This is implemented with a *half-life function*: $Q_n = 1 + 2^{-kr} Q_{n-1}$. Each alert type has its own half-life function, where Q_n is the current *level* of the alert. If this value is greater than the preset user-defined threshold, the alert will turn into a *warning*. The name of the function comes from the fact that old values of Q_n will slowly age, and after a certain number of valid requests, r, Q_{n-1} will count as only half of its real value. The index k is a constant set to decide how many requests are required until only half the value is counted. If $k = 1/10,000$, it means that it will take $10,000$ requests ($r = 10,000$) before the old Q-value will be halved.

[7] This module requires the attribute `host` and cannot function without it.

[8] Age can either be dependent on time or on how many successful requests have been sent since the last malicious one. The latter is used in the monitor.

3.9.1 Example of Password Protected Resources

The use of the module becomes clear when looking at the example of password protected documents. The HTTP protocol has a scheme for accessing authorized documents via passwords. Unfortunately, there are no automatic restrictions on how to handle failures such as those built into a normal login procedure. There is no limit to the number of attempts a user can make, nor is there a delay between failed attempts. If status code `403` (forbidden) is repeatedly shown for a host, it may imply that the host is trying to break the password protecting the resource. A single instance does not have to mean a malicious attack is in progress. The problem is that the HTTP protocol uses this code to signal to the client that a password is required to access the resource.

A typical exchange is therefore (also shown in Figure 3.2):

- Client sends a request for a page.
- Server replies with status code `403` to signal that an optional header for authentication is required.
- Client (somehow) acquires a password from the user, and sends a new request with the authentication included in some headers.
- Server provides the document to the client.

As can be seen, a single instance of the status code `403` merely implies that a client wanted to access a document, and that he was prompted for a password. Probably up to four errors should be allowed, because a user may mistype his password a few times. It is only beyond this point that the system administrator wants to be notified that something unusual is occurring.

The importance of aging alerts also comes into play in the example above. One can imagine the user Anne having to access a protected resource every day at work. As she logs out from her computer every evening, there is at least one alert for each working day, and after a week she has collected five alerts. If the monitor simply counted the number of failed requests, Anne would very soon raise an alert even though she is merely performing a routine task. By letting the alerts collect age, either by time or by successful requests, there will be fewer alarms.

This module also allows detection of denial-of-service attacks, with several thousand instances of status code `404` (not found). By defining a high threshold, the monitor only reports hosts that persistently ask for broken links.

3.9.2 Resulting Data Structure

Table 3.6 shows the new fields added by this module. The fields concerning the `followup` class is for the feature described in Section 3.7. For implementation reasons, the fields are added at this step instead of previously.

Each type of signature has an internal threshold defined and a value for k (see Section 3.9). The alerts raised by previous modules turn into warnings if the Q-value (level) for the class and host is greater than the threshold. The text field of the warning contains this information.

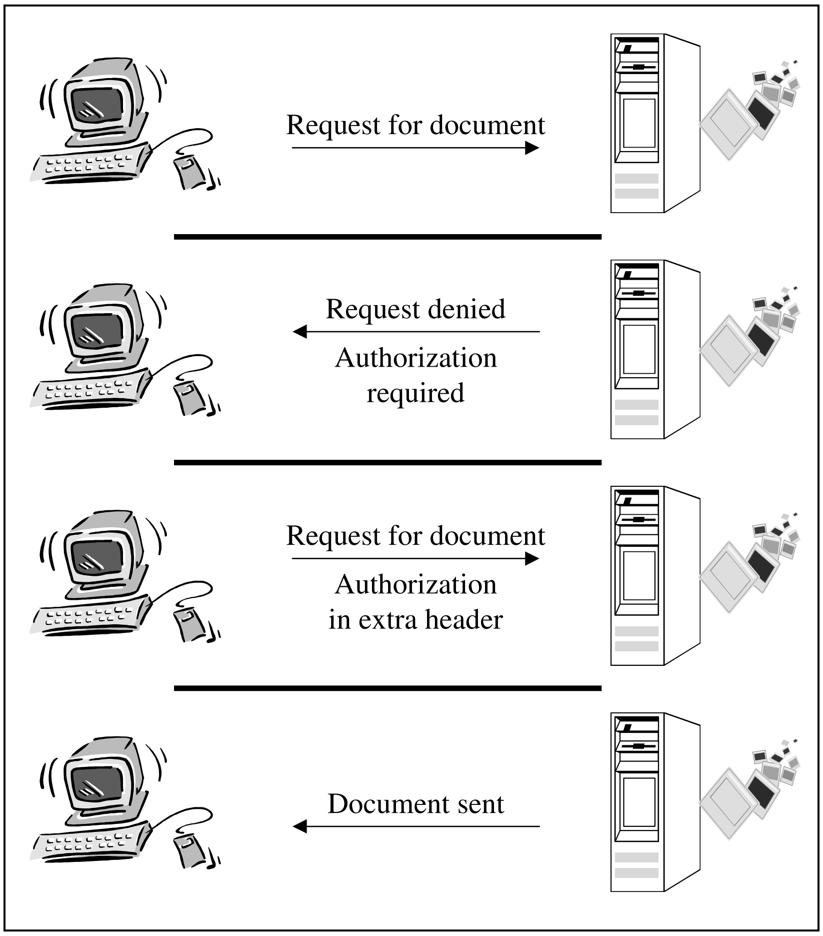

Figure 3.2: Steps necessary to access a protected resource through HTTP.

3.10 Print Module

If the *Decision Module* decides a host has created enough alerts, it forwards the data structure wrapping the request to the *Print Module*. This module prints selected information to `syslog` if running in real time. If the monitor gets its input via files, it prints the warnings directly to the screen or into a file.

The output includes the original request and why it is judged to be malicious.

3.11 HTML Reporting Facility

This is actually not a module belonging to the monitor, but an external program. It was designed to facilitate the interpretation of the output from the monitor, when it is run in batch mode on old log files.

If the monitor analyzes old log files, it prints the output to `STDOUT`. On `UNIX`, this can be redirected to a file. Each warning has one entry in the file, and it contains values of internal fields of the object.

This format works well if the program happens to run in real time, or if there are few

Alert Parameter	Value
engine	`decision`
class	`followup`
sig	`all`
text	`warning`

Warning Parameter	Value
engine	`pattern`
class	`suspiciousCgi`
text	`level: 1 > threshold: 1`
engine	`pattern`
class	`cgi`
text	`level: 1 > threshold: 1`
engine	`pattern`
class	`UrlSuspiciousHexCodes`
text	`level: 1 > threshold: 1`
engine	`decision`
class	`followup`
text	`All requests reported after a warning`

Table 3.6: Fields added by the *Decision Module*. The class `followup` concerns the functionality described in Section 3.7. The `level` parameter is the value of Q_n (see Section 3.9) and the threshold belongs to the attack signature.

warnings to evaluate. However, if a troublesome log file is being analyzed, it is difficult to acquire an overview of the contents.

The reporting program reads the output from the monitor and presents it as summaries in HTML.[9] The choice of HTML was easy:

- It is platform-independent.
- It has good capabilities for displaying information. Different pieces may be marked with different tags, and it suits both graphic screens and other presentational devices.
- It can link different pieces of documents together.
- As the monitor works with a www server, chances are high that a system close by also has a client to display documents.

The system uses style sheets to facilitate the interpretation of data. It also has two options, JavaScript enabled/disabled. It presents four different *views* of the warnings collected:

simple view This is a list of all the hosts with the `urls` they have accessed.[10]

[9] A future version will also include a text-only summary file. In the distribution, there is a beta version included, which prints only text.

[10] Only the requests that caused a warning are included.

host view This view lists the hosts, and then a list of the warnings they created. In the warning list, the offending `url`s are found. Please note that because each request may cause several warnings but contains only one `url`, some `url`s may be duplicated and printed under each warning list heading. See Figure 3.3 for a screen shot.

url view The summary is sorted by `url`, and under each point one can find the host that accessed this `url`.

warning view The warnings are highlighted, and the hosts are shown in the minor list.

There is logic built through links, so an administrator can scroll through all requests made by a host, all requests accessing a certain resource, or all requests that caused a certain warning.

To simplify the treatment of `url`s, there is a feature for grouping similar requests into a single entry. For example, the `url` to access the search engine may look like:

`GET /cgi-bin/searchquery?q=security` (example 3.7)

`GET /cgi-bin/searchquery?q=vulnerabilities` (example 3.8)

The program can group these entries into one `url` if the user specifies how many characters should be significant. Above, a choice of 25 characters would be suitable, because then all entries to this program would be grouped together.

The reporting facility does not provide a real-time GUI. The GUI is provided by the Tivoli environment to which the tool reports.

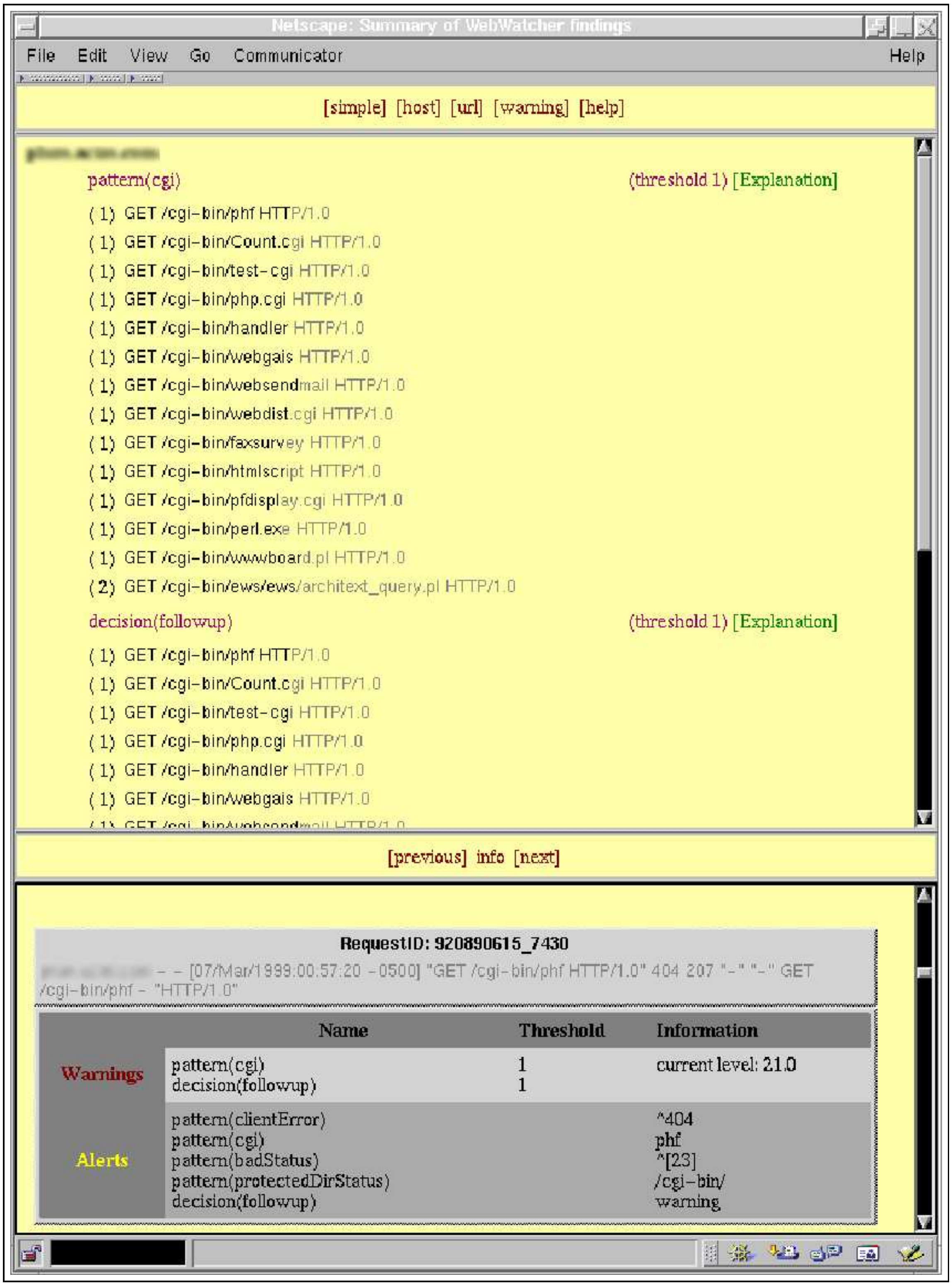

Figure 3.3: Screen shot of the HTML facility.

Chapter 4

Testing on Real Sites

The program has been run against several sets of log files from busy sites. To collect a few examples, we have made weekly runs against the following web sites:

- two medium-sized commercial sites, with around $8,000$ hits a week for the smaller, and around $58,000$ hits for the larger. The diagrams shown later are results from the smaller site;
- log files from a university (size varied among web servers, but they were generally $20,000 - 30,000$ hits per week;
- logs saved from the Olympic Games in Nagano, 1998. See below for size information.

The latter was the busiest site to date, having gone into The Guinness Book of World Records with almost $650 * 10^6$ requests during its 16 days of operation. The top request speed was $110,414$ requests per minute, which is what the medium-sized server above experienced during two weeks of operation. We limited the tests to one day of the Olympic Games, with about 37 million requests.

We also let the program run in real time, but by using the substantial log files we were able to simulate years of traffic with the monitor running. This showed the behavior in a longer-term perspective including resources used (processor and memory usage). We found it better to use real data than simulated requests with a mixture of harmless and malicious requests.

4.1 Analysis at All Sites

In this chapter, the traffic at the smaller commercial site has been thoroughly analyzed. We chose to use these results in this thesis for several reasons:

- It had the most attacks, and was thus more interesting to analyze.
- The attack patterns found there are consistent with the results of the other sites.
- Our findings can be shown concisely because the dataset is smaller.

All sites were different, and required their own trusted signatures. Different problems were also encountered at different sites:

- The Nagano logs showed a problem with the local installation of Perl. The monitor crashed after 30 million analyzed requests. It worked fine with the newest Perl release.
- The university logs had a vulnerable script installed. Thus many warnings were created. These log files also included the longest supervision (15 months).
- The larger commercial site was the one that changed the most, with many broken links (thousands of them) and users providing mirrors of software distributions. It was kept under supervision for almost one year.

For each new site analyzed, the monitor matured and it became clear how to handle various problems. The sites above are very diverse, and this is reflected by the data the monitor had to analyze. As the supervision lasted for over one year, we also covered vacation periods as opposed to normal accesses during work sessions.

4.2 Tests Performed

Before we deployed the program at various sites, we created a signature file of malicious behavior. Most of the information was found in the extensive vulnerability database maintained by the Global Security Analysis Lab at IBM Research.[1]

The following checks were made:

- sanity check on log format,
- empty `url` string,
- non-standard format of `url` string,
- non-conforming encoding of characters in the `url`, and in the query (two separate tests),
- suspicious encoding of normal characters in the `url` and the query,
- unknown method name in the `url`,
- directory tricks, such as ../ in the `url`,
- listings of executable or sensitive directories,
- vulnerable cgi programs (more than 50 signatures, including some permutations),
- interpreter in the `cgi-bin` directory,
- accesses to backup files, especially old cgi programs,

[1]This database is for IBM internal use only, and not available to the public.

- accesses to files containing sensitive information,
- checks for excessively numerous bad status codes,
- checks for unauthorized accesses of protected resources,
- warnings if the query string contains strange characters in conjunction with a vulnerable cgi program,
- special warning if a malicious behavior is found, and also if status code `200` (success) is returned,
- the combination of malicious behavior and strange hex codes is worse than any single component,
- guesses of programs existing in executable directories,
- special warning for a strange method name combined with a vulnerable cgi program.

For each site, we added specially crafted *safe signatures* suitable for that particular site. Also, the list of suspicious hosts was empty at first, but after several runs it grew. We differentiated between hosts that were considered suspicious because of relatively innocent behavior (e.g. accessing too many broken links) and those that launched more malicious attacks, such as probing for vulnerable cgi programs.

4.3 Traffic Analyzed

In Figure 4.1, all the traffic at the smaller commercial site has been plotted. The analysis was made for 69 days, and during this time the site received 80,030 valid requests.[2] It is easy to distinguish weekends from working days because traffic at the site is lighter on weekends.

When the server tries to serve the client's request, a status code is generated that states whether the server managed to complete the request successfully. As shown in Figure 4.2, the majority of all requests have a successful completion. This shows that the server analyzed is well setup. The other status codes generated may not be very severe. A few documents not found may only be a typo in the client when trying to access a resource.

The site was visited by 7,049 distinct host names, and we were interested in finding out how many requests each separate host made. As can be seen in Figure 4.3, most hosts ask for only a few documents. As many as 30% of the hosts requested only one document, and then never came back while the monitor was active at the site. The two computers that requested the most documents (2320 and 5255, respectively) were internal hosts.

Figure 4.4 shows another view of the logs. We have grouped the hosts according to the kind of status codes their requests have generated. For example, if *all*[3] requests were

[2] The access log actually had one more entry: `.txt - "HTTP/1.0"` The format of this is obviously not valid, and this line belongs with the line above. The line break has most likely appeared in some of the transfers of the logs, and we ignore this line in the analysis.

[3] That is, all requests made during the supervised interval.

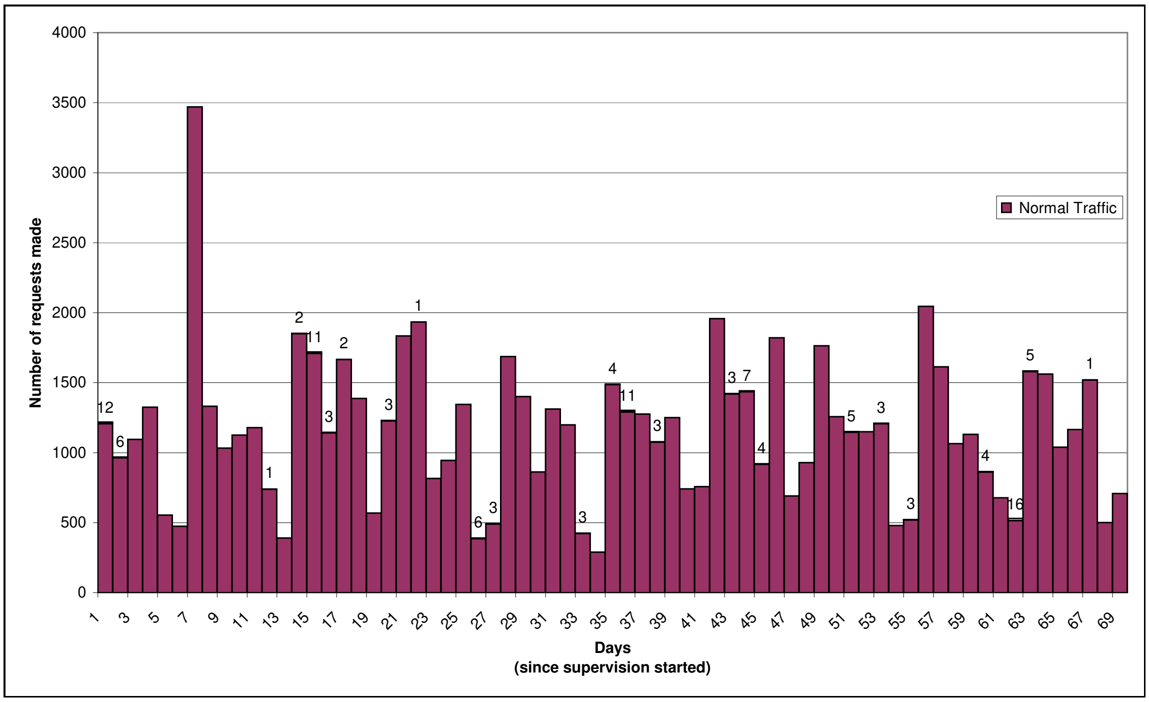

Figure 4.1: All traffic at a commercial site is shown. As the malicious traffic is far lighter than the normal traffic, all days have been marked with a number, signifying the alarms issued by the monitor that particular day.

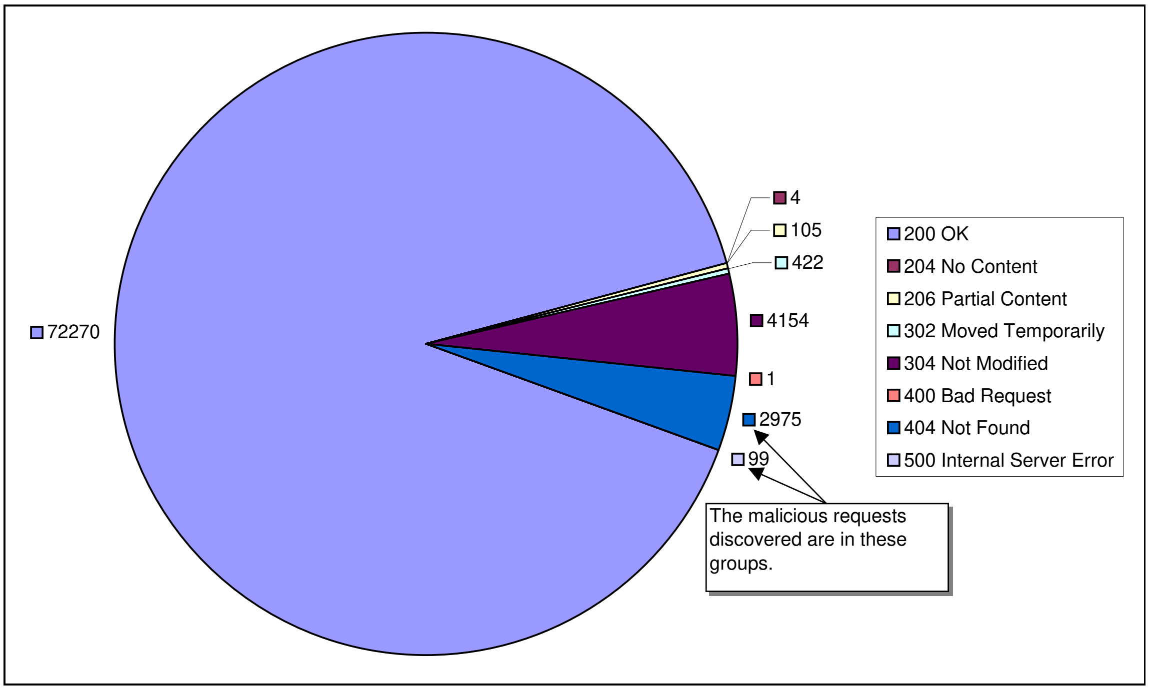

Figure 4.2: The status codes returned by the server for all requests handled (80, 030). Most requests handled by the server are successful, with 96% benign ones and only 4% in the category of client or server error.

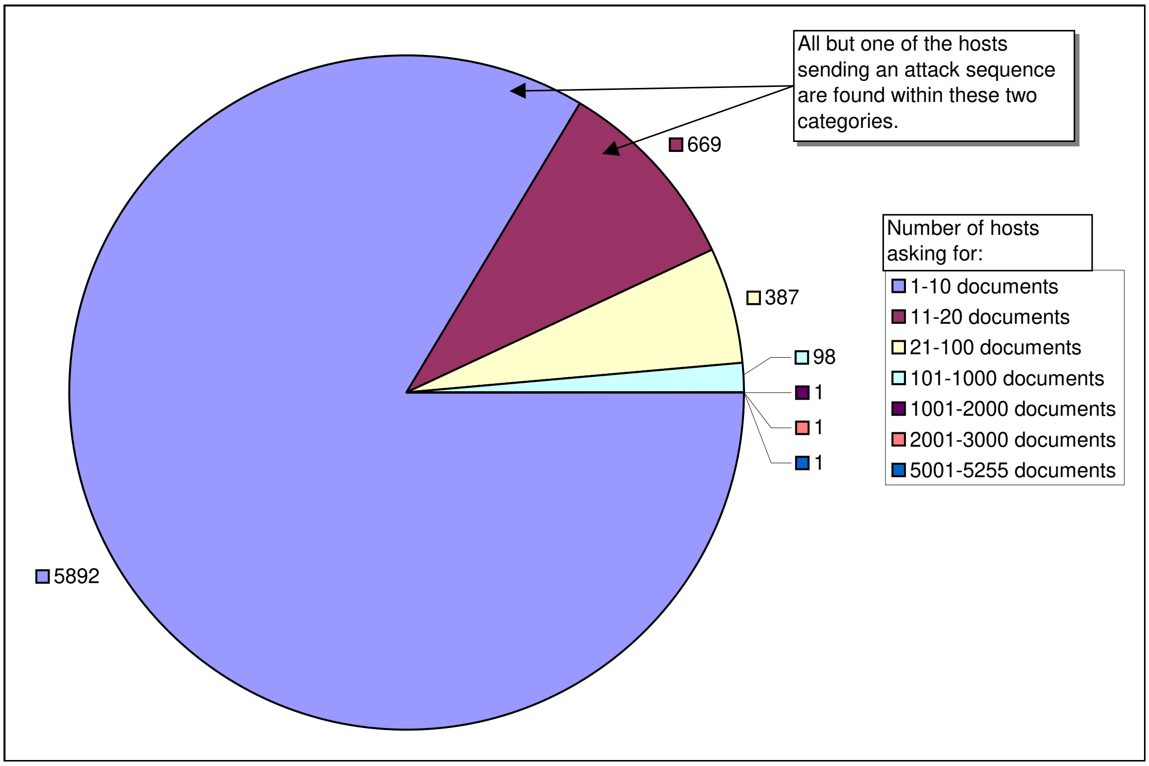

Figure 4.3: Number of requests made by the 7,049 distinct hosts visiting the site. Most hosts accessing the site request only 1–10 documents. 2,124 hosts requested only one document during the period of site supervision. This number obviously depends on the primary page for the site (index.html) and how many links (images) it contains.

completed with a status code of `2xx`, the host is placed in category `2xx`. The same goes for status codes `3xx`, `4xx`, and `5xx`. The hosts that sent requests which were completed with different status codes are grouped together in the category *mixed status codes*.

As can be seen, 88% of the hosts that accessed the site only supplied requests that were handled successfully by the server. That is, all requests they sent were completed with status code `2xx`. This can be explained by people's behavior of accessing the main page and then following links. If the site is setup correctly, all links should work.

Some people mistype the url they want, documents may be deleted, and obsolete links not updated immediately. This causes some hosts to have mainly status code `2xx` on their documents, but intermixed with a few error codes (`4xx`, `5xx`). Thus, these hosts show up in the category *mixed status codes*.

The interesting portion of this graph is populated by hosts that have only made requests that were completed with status code `4xx` or `5xx`. In this case, the hosts never even succeeded in obtaining a document from the site because these two status code categories signify errors. This may point to external broken links, or simply malicious attacks.

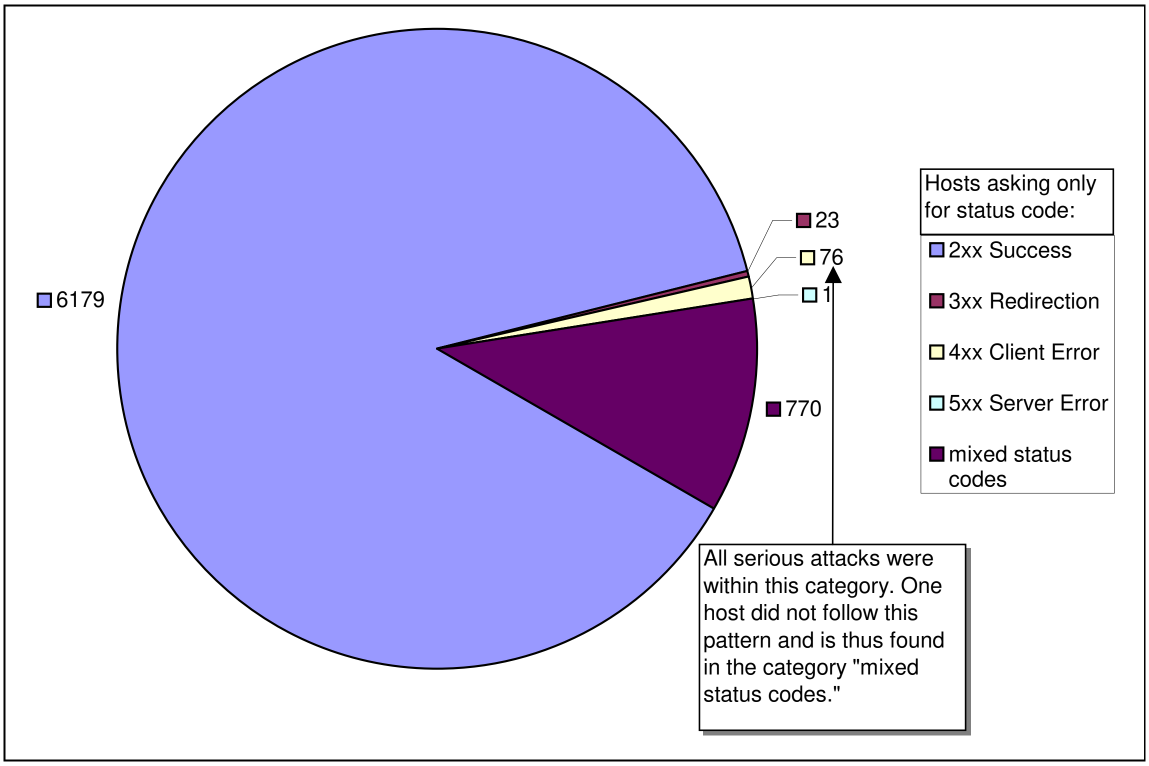

Figure 4.4: Status codes generated by hosts. The server was accessed by a total of 7,049 distinct host names during the analyzed time. The majority of the hosts only place requests that are handled successfully by the server. The likely cause is that these hosts access the main page and then follow one or two links. If the site is working properly, this should not cause errors.

4.4 Attack Findings

In Figure 4.5 all malicious attacks detected by the monitor are shown. As it happened, the site supervised was only attacked once by each host, so each bar represents a different host.

Quite often, only the most simple of the cgi script vulnerabilities are tested. About once a week do these appear in the logs. The most commonly found attack sequence was: `phf`, `test-cgi`, and `handler`. It seems strange that an attacker would try only these three requests when probing a site, because these vulnerabilities are quite old and well known. These three attacks are usually sent in the order above, and within seconds of each other (see Table 4.1). This may indicate that a tool is sending the requests. We have been unable to identify it, though, and only three (old) attacks against the www server seem less than optimal for an attack program. One explanation may be a malicious Trojan horse hidden in a downloadable application. It probes computers, and sends the results to the cracker. It may also be a tool that is more versatile than www server attacks. There may have been other types of probes on the network simultaneously with this one.

Less often, a series of vulnerable scripts is tried. This may be a scan by a serious cracker who is checking a company's defense. If this is successful, the danger is relatively acute. There may be other attacks staged from this computer, and web jacking is also likely. Of

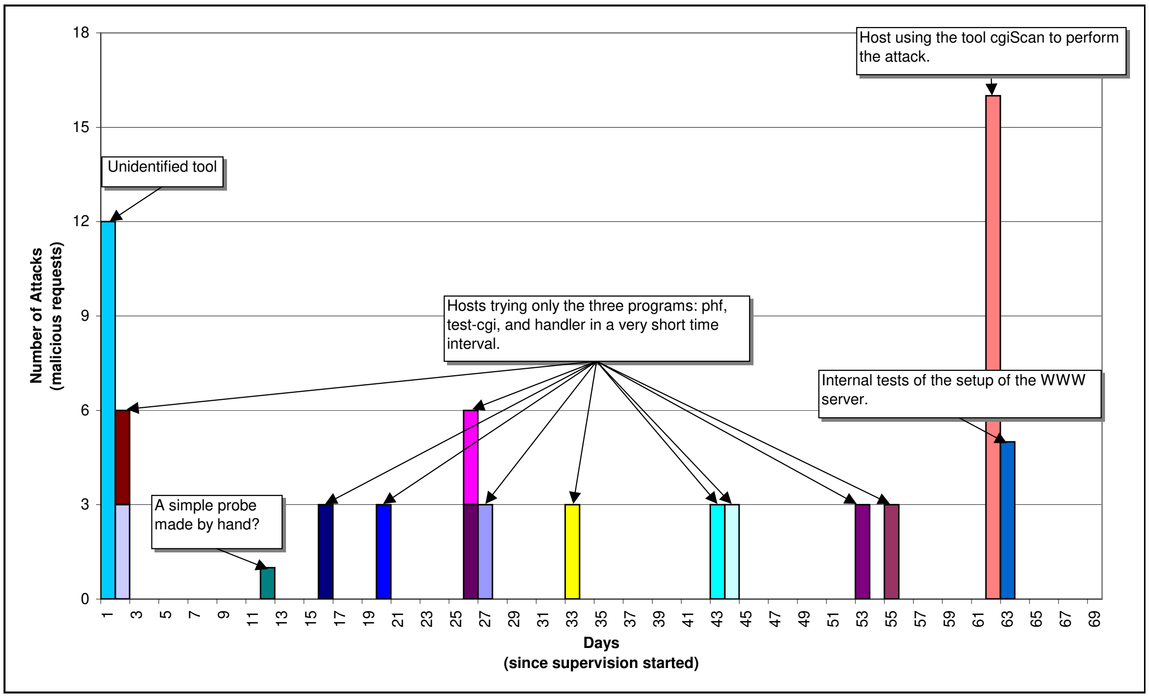

Figure 4.5: Attack pattern as detected by the monitor, excluding the accesses made by the host shown in Figure 4.7.

phf	test-cgi	handler
19:55:13	t+2 s	t+3 s
18:02:49	t+6 s	t+7 s
00:54:23	t+1 s	t+2 s
17:36:29	t s	t+1 s
21:31:43	t+1 s	t+2 s
10:54:06	t+1 s	t+1 s
11:26:36	t+6 s	t+12 s
03:53:27	t+2 s	t+3 s

Table 4.1: Time between the access attempts of three vulnerable programs: `phf`, `test-cgi`, and `handler`. Not all attempts at the site have been included.

the attacks we analyzed, the serious cracker very often included one of the simpler types of known vulnerabilities in his probe, so it seems that even if the monitor does sometimes miss new ways to launch attacks, it very seldom misses the actual host trying the attacks.

In one case (see Figure 4.5, day 62), we also managed to identify the program used to perform the scans (see [27] for the actual program). There are several attack scripts for downloading on the Internet, and it seems that these programs are also employed by several people. In the example we found, the user did extend the attacks slightly.

The commercial sites where the monitor was tried can be regarded as having been expertly set up. Every week, the sites are scanned internally for vulnerabilities, and, if found, the servers are disabled immediately. This had the consequence that we never found successful attacks – only many attempts. If we had been at another site, the results would probably have been very different. One of the university logs had the program `test-cgi` (see Section 1.3.3) active, and there were a few requests that succeeded in accessing it. The requests were harmless, but this proved that some of the more advanced features of the monitor worked.

4.5 Suspicious Host Results

We noticed that attacking hosts usually only launched one attack sequence, and then never returned to the site again (shown in Figure 4.6). Admittedly, we only checked log files for 69 days at this site, but the results found at the other sites support this fact. There were a few exceptions to this rule, one of which can be seen as *Host 9* in Figure 4.6.

When we later concluded a more detailed investigation, we noted that several of the hosts seemed to have come from dynamically allocated addresses. This conclusion is drawn from the fact that we were unable to find a server that recognized the names and also from the type of names used (`dial-up76.some.site`).

Other attacks originated from quite respectable companies, from which we assume the culprit is a single employee, or the company is used as a staging area for a cracker.

The attacks studied originated from many different countries. From one site, where our supervision only lasted 10 weeks, we detected 16 serious attackers, 8 from the US and 8 from other countries (Norway, Sweden, Brazil, Romania, Switzerland, Italy, Australia, and Chile).

We concluded that the scheme of tracking suspicious hosts generates few false alarms. As soon as a host has performed an action causing it to be put in the suspicious database, all requests should be printed and compared with the known attacks. This way, the program can learn by itself. After a certain time, this database should be purged, but since the control is quite effective and fast for suspicious hosts, the penalty of keeping it is moderate.

Host 9 in Figure 4.6 was the exception, and we followed its behavior more closely. In Figure 4.7 all accesses of this host to the site are shown. The host sent only one request that was interpreted by the monitor as malicious. It was a request to access a protected resource belonging to a vulnerability scanner. It may well be that this request was made by a user who has access to this resource by other means, and merely wanted to ensure that it would not be accessible over the web.

Finally, the attacks seemed to be localized in time. That is, the probing was done within a very limited period. As any unknown attack launched before a host is marked as suspicious is lost, the program could probably be enhanced with a small, time-limited buffer which is

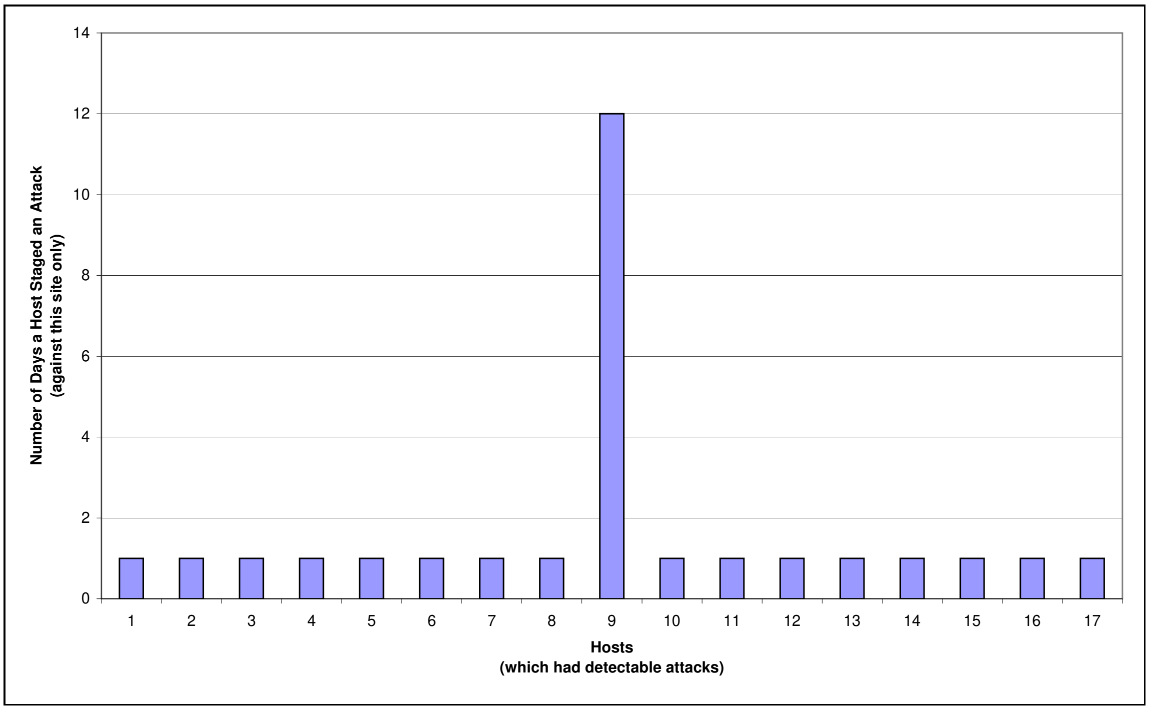

Figure 4.6: Attack behavior of hosts. Most hosts perform a scan with several malicious requests at a site, and then they never come back again. There was only one exception, shown in this diagram as host 9. This host actually belonged to the company, and only performed one attack sequence. The accesses on other days were not of a malicious nature, but related to the business of the company. They were still followed up by the monitor tool.

sorted by host names (see Chapter 6). When the monitor adds a new suspicious host to the list kept in the *Suspicious Module*, it also scans this small buffer for previous requests originating from that host. If any are found, they are analyzed for malicious behavior. Even if the monitor sees only the third attack from a given host, the previous requests are stored in the buffer and can easily be found. The buffer is purged regularly because most attacking hosts send the malicious requests in sequence.

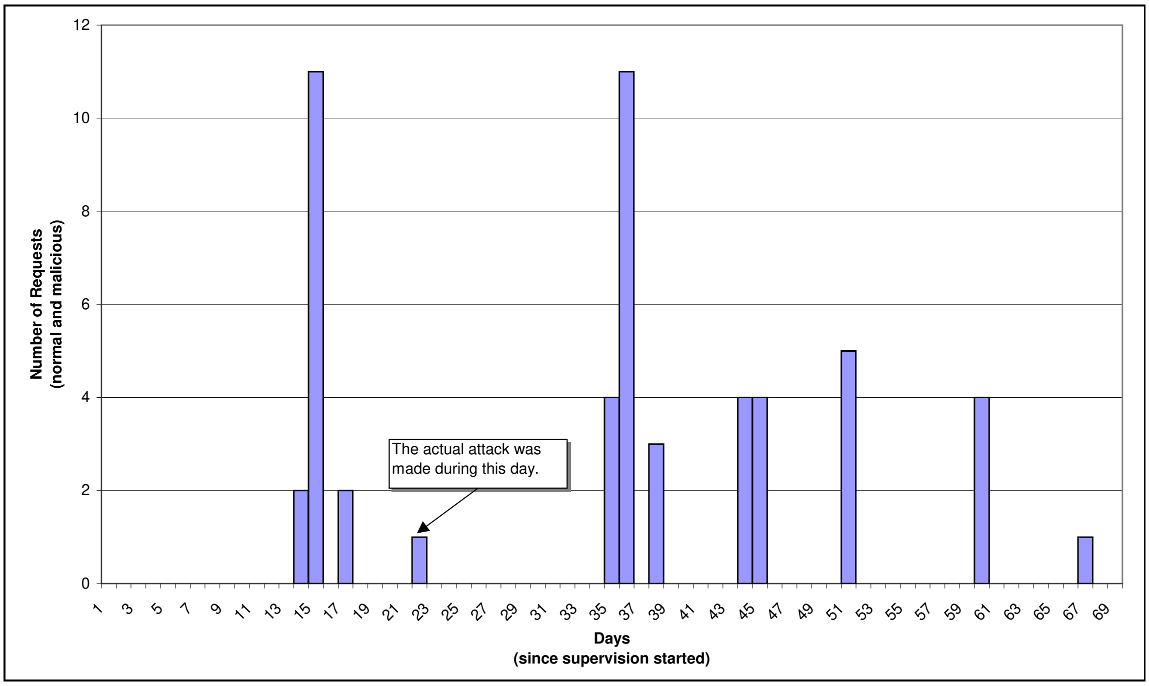

Figure 4.7: Access pattern of the host (host 9 in Figure 4.6) that launched an initial attack, but subsequently made only normal requests. As the single malicious request sent was to read a configuration file for a vulnerability scanner, it is probable that the user meant no harm but was merely looking for information.

Chapter 5

Monitor Performance

If the monitor is regarded as one part of an entire system of security tools, it works satisfactorily. It succeeded in analyzing several log files from different site configurations and server brands. The real-time trial was limited to the Apache www server, though, and limited to the time within which this thesis was completed.

5.1 False Alarms

Deploying the program at a new site most likely generates false alarms. Each site has its own peculiarities, and a signature file cannot simply be ported to a new system and be expected to work. As written in Section 3.8, there are several cases where it is necessary to create a trusted signature but many times it is sufficient to create a better attack signature for the *Pattern Module*.

As an example, we have an attack signature for the interpreter `sh`. At one site, there was also a GIF image, called `sh.gif`. Every time this resource was accessed, we received an alert. By changing our attack signature to exclude images, we could avoid this false alarm. Overall we found that improving the attack signatures decreased the number of false alarms.

As different sites may have different names for the directory where they allow executables, we did not want to restrict the signature to a specific directory but let the program look for the malicious scripts anywhere in the `url`. Obviously, this produced many false alarms. By specifying that cgi programs should only be searched for in the `cgi-bin` directory, we limit the number of alarms but may also miss some malicious scans.

5.2 Existing Vulnerabilities

One site also had a vulnerable cgi program running (`Count.cgi`). Each access generated an alarm, and it was very difficult to judge from the alert whether it signalled a nefarious attempt. As the program was run on old log files, there was no way to go back and check the version being tried. The way to use the vulnerability of this script was very similar to the normal operation, and there was no way to distinguish a scan from a valid request. If the program had been run in real time, we would have had the option of looking at each user running the program to find out whether the correct version was running. Then we would have been able to add a trusted signature for that user only, as we knew his program was safe. This way, the monitor would not report any accesses to this program, meaning that we would unfortunately miss initial probes of hackers checking the presence of the vulnerability. It did reduce the number of false alarms to a manageable number, though.

5.3 Timing and Memory Requirements

By timing[1] the program, we came to the conclusion that in general it takes $3-6$ ms to handle a request.[2] The longer the program runs, the more statistics it has to keep track of, including suspicious hosts. The slower result was for the Nagano logs, which were run on a slightly slower computer.

This should be well within the time limits to run the tool in real time on a moderately busy server. To save time, the program does not scan for all vulnerabilities if it finds some attacks. In practice, this means that valid requests take slightly longer to scan than nefarious ones because a valid request requires all tests to be performed.

The memory required was difficult to measure. We examined the amount of memory used by the Perl interpreter, and for the logs above with $80,030$ requests, it used about 7 MB for the core image of the Perl process.

Finally, the program also keeps some statistics of the status of the suspicious hosts. When the monitor quits, it saves these statistics in a text file. Its size was $150,968$ bytes, with many superfluous characters included for ease of importing it to other programs (again, this was for the logs with $80,030$ requests).

5.4 Monitor and Log Files

Of all the hosts accessing the smaller commercial site, the monitor decided to keep 333 under further supervision. Of these, 17 were judged to have a malicious intent and the monitor reported them to the console, Figure 5.1. The *Decision Module* based its conclusions on the thresholds defined for each *class* of signatures in the *Pattern Module* described in Chapter 3.

The pie chart in Figure 5.2 shows the requests instead of the hosts. 96% of the requests where judged to be normal, and did not lead to further analysis in the monitor. 4% did cause some concern, and the monitor saved some information from these requests to make refined checks as time went on. Only 0.15% of the requests where reported to the console. This was after the *safe signatures* had been written for the site, but the number of warnings sent on is very limited, and well within what a human administrator can analyze.

[1]The timing and memory operation was done in a primitive and granular way, so it is not very precise. With the monitor working, we plan to include a benchmark test of a web server with and without the monitor running on the same computer. Unfortunately, there was no time to do this before this thesis was printed.

[2]The machine used was of the RS/6000 43P family, with the processor PowerPC604e running at 200 MHz. The benchmark specs are SPECint95 7.9 and SPECfp95 4.9. These numbers may be slightly off due to the old information used. The computer was running AIX 4.3.

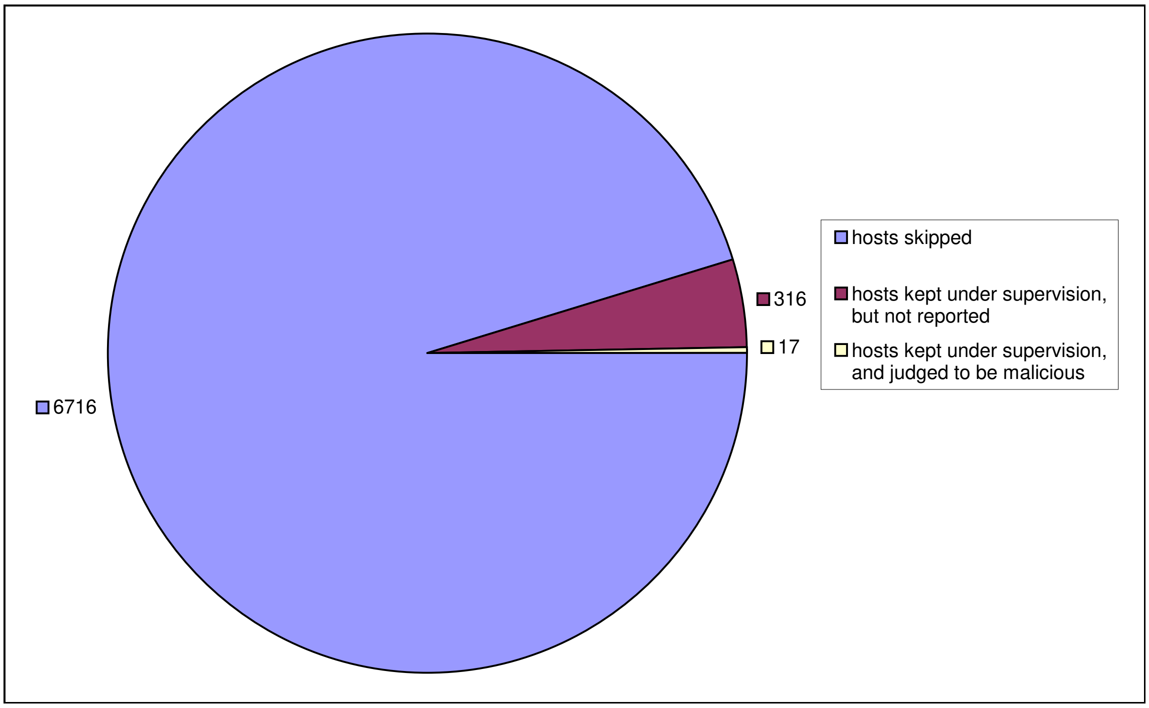

Figure 5.1: Monitor performance: analyzed hosts. There were 7,049 distinct host names in the log files for this site. The monitor will keep track of a minority of the hosts accessing a site, and will decide when to send a warning to the administrator.

5.5 Learning Capabilities

The current version of the signature file did not include all the vulnerabilities contained in the vulnerability database. By looking at attacking hosts and comparing the other accesses with the information found in the signature file, it was possible to deduce several new attack signatures. The comparison and the addition had to be done somewhat manually in this prototype.

The experiments show that it would be useful to be able to compare a host's actions with the attack signatures, and to suggest additions (to be cleared by an administrator), which can then automatically be added to the signature database. This approach alleviates the penalty of keeping an updated database of the latest vulnerabilities. They present themselves to the monitor, which suggests them to the administrator. He can then search the Internet to make sure how they work. The behavior of the attacking hosts, as shown in Figure 4.6, facilitates the learning process because most malicious hosts only sent requests to probe the site for vulnerabilities.

5.6 Proxy Issues

Proxies are machines that works as intermediaries for serving www documents (see Figure 5.3). The client computer contacts the proxy and asks for a document from a server. The proxy forwards the request if it cannot satisfy it itself. The server machine then sends

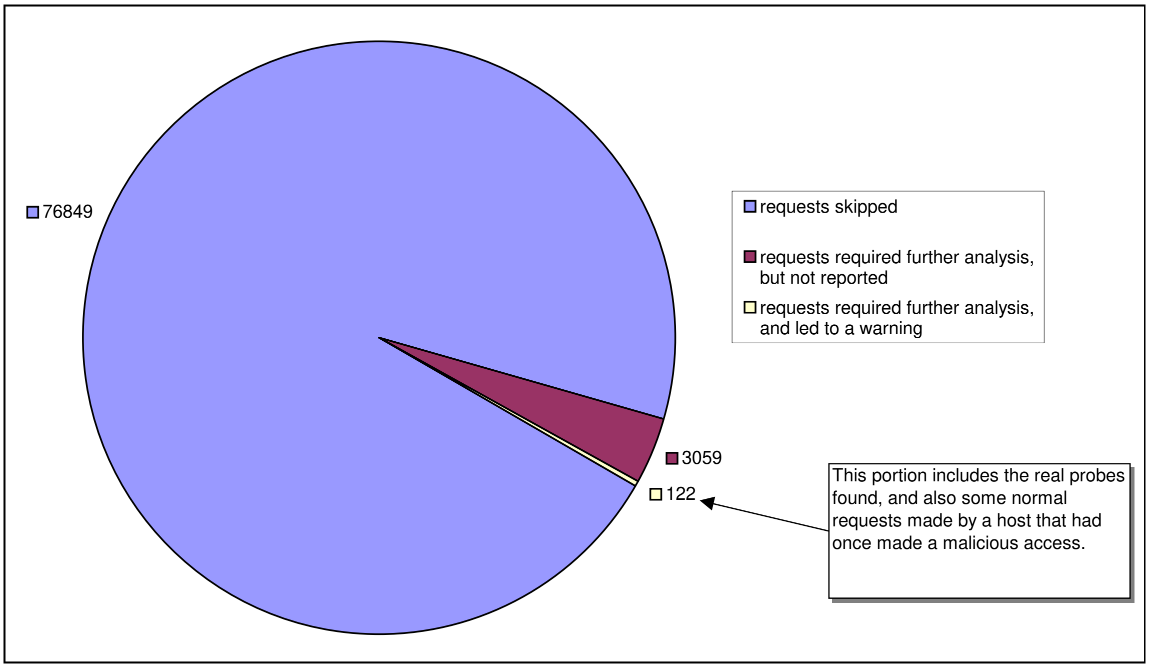

Figure 5.2: Monitor performance: analyzed requests. The access log contained 80,030 valid entries. After a perfunctory check of the logs, the monitor will decide whether the request analyzed shows suspicious behavior or it is innocuous. As can be seen, most requests are normal, but the versatility of the monitor allows many conditions to be analyzed further before a warning is issued.

back the requested resource to the proxy, which forwards it to the client. The rationale for proxies varies. They can be used

- as a caching facility, so that frequently used resources are stored locally;
- to anonymize requests from clients. The servers does not know anything about the original client, but only about the proxy. If the client trusts the proxy, a certain anonymity has been gained;
- as a policy enforcement point because a proxy can be set to block certain sites with dubious contents.

The problem with proxies and the monitor is that a malicious person may hide behind it in two senses. First, as the proxy serves several people, it may seem like the host name of the proxy generates a large stream of traffic. If there are many legitimate users and only one attacker with few requests, the malicious requests may drown in the normal traffic and the monitor will miss the alerts (see Section 3.9 on aging). By being able to adapt the parameters guiding the *Decision Module*, this can be avoided.

The second problem is the anonymizing effect of the proxy. There is no way we can distinguish whether one host is launching attacks or several hosts are launching a few attacks. Nor can we determine the host name directly, but may have to work in conjunction with an

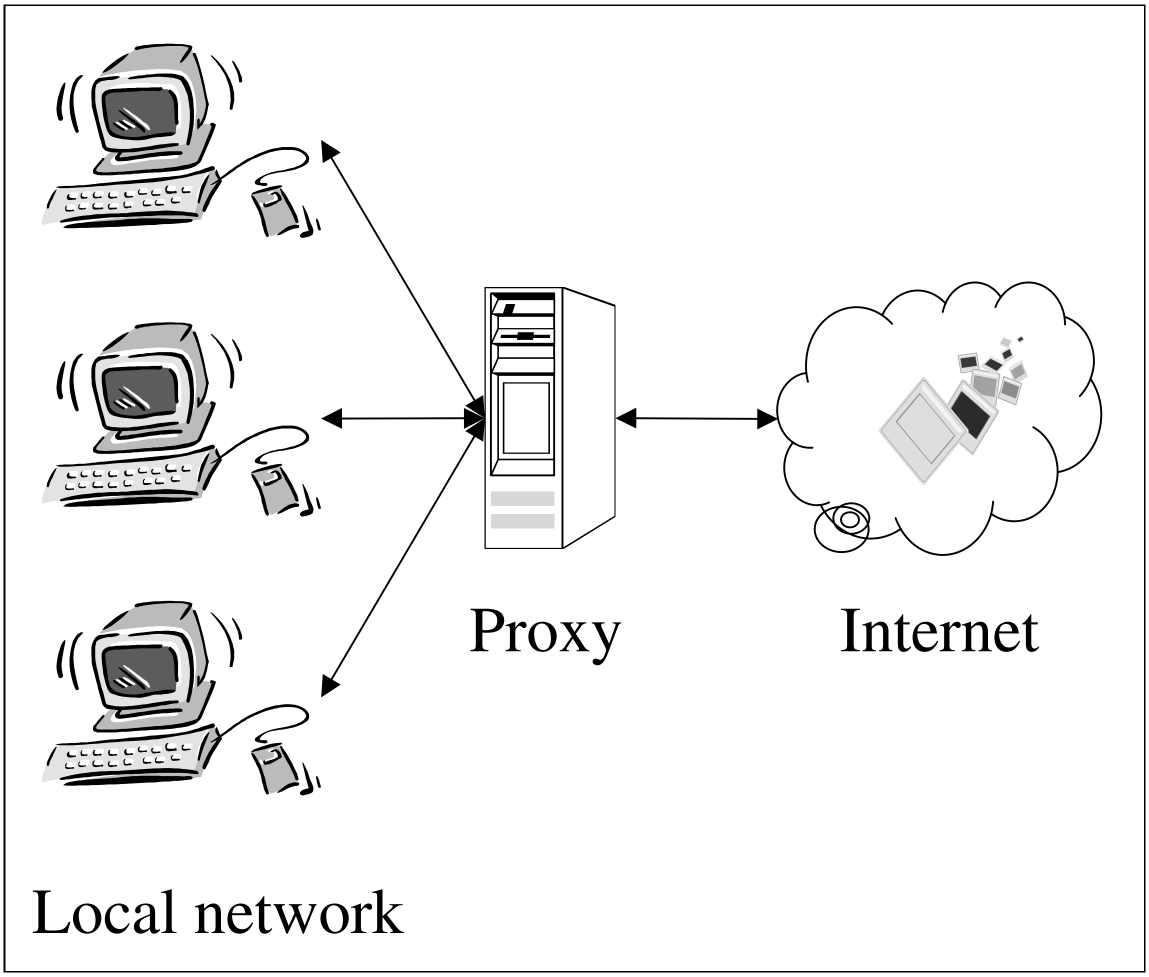

Figure 5.3: A proxy is an intermediary between an internal network and the Internet.

ISP. This is not optimal because the personnel employed may care little about trouble at another site.

5.7 Busy Web Servers

To test the monitor, we ran it on the logs saved from the Nagano Olympic Games. Extremely busy sites like this one (extreme case) may make the real-time running of the monitor difficult. The setup for the Olympic Games included several computers and routers [28, 29, 30]. As there were so many requests every minute, there was no way a single host could handle the load. The setup also had some redundancies, so when one part failed, there were others that could take over.

The important point is that two consecutive requests originating from the same client may be routed to different servers.[3] There would not be only one monitor running at one site, but many at different hosts. One monitor would not know what another one was doing, so it may not put enough weight on an attack because some of the malicious requests may have gone to server A, and others to server B.

Fortunately, the problem is not as severe as it seems. First, it should be emphasized that the Nagano site was an extreme case of a busy www server.

Second, the monitor tool spent most of the time reading the logs, breaking them apart, and deciding whether they are harmless. Each request is independent of all others because HTTP is a stateless protocol. This means there is inherent parallelism in the most time-consuming parts of the monitor, and several requests can be handled in parallel. The important part is the *Decision Module*. It must be able to get input from all sensors (the other parts of the monitor), and base its decision on the global state and not only on the locally collected information.

The malicious information can either be distributed to n *Decision Module*s on n different hosts, or it can be sent directly to a single *Decision Module* on one host. As the malicious attacks are few, a single host has no trouble taking care of several busy www servers. The program should scale well, thanks to the parallelism allowed by the design.

5.8 Reporting Facility

The reporting facility is limited. The conversion to HTML documents happens in one step, and then the information is saved statically. When using the interface, it is easy to see that a dynamic interface would be more flexible. Still, it works well enough for the prototype, and the Tivoli environment provides a real-time GUI.

There is one serious problem associated with the HTML documents created in this way, however. Certain advanced features of style sheets are used, and some older browsers may not be fully compliant, causing them to crash when trying to read the summary. By using a modern browser, this situation can be avoided. As the development was done on AIX,

[3]There was a feature of the routing that the same client communicating with a server using SSL was likely to be routed the same way. Thus the computed keys could be kept.

the browser used is Netscape Navigator and the pages do not work as well with Internet Explorer. When trying different platforms for Netscape Navigator, we found that:

- it worked well on AIX 4.3.2.0 and NN version 4.51,
- it worked well on Win95 and NN version 4.5,
- it crashed on Win95 and NN 4.04,
- it crashed on Solaris 2.5 and NN version 4.04.

By either disabling style sheets, deleting the style sheet file created, or commenting out the entries on `border-width`, all browser clients above can display the documents.

There was another problem related with the reporting facility. It binds together a summary with the actual warnings raised by the monitor. If the monitor has created an excessive number of warnings (tens of MB), there are also problems. In the JavaScript-version of the report, all warnings are put into a single HTML page in the form of separate tables. When analyzing the Nagano logs, the browsers above could not display so many tables and crashed.

In the non-JavaScript version, each warning is stored in a separate file. It may fill up the file system's internal structure. One run at the Nagano logs resulted in $30,000$ files, but at that point the file system complained that it could not create more files, even though there was still physical space.

Even though this is a disadvantage of the reporting facility, it is a minor one. The summaries can still be displayed, and it would not be feasible for a human administrator to examine $30,000$ warnings to see whether they are valid. It should be pointed out that this only happened for the Nagano logs, and there was no problem at the other sites when the proper trusted signatures had been created. In Section 6, we discuss how the *Print Module* can be improved to avoid these problems. The problem is also alleviated by the fact that the tool should be run in real time on a busy site like the Olympic Games.

Finally, the JavaScript version with tables is slightly slow because the browsers take time to render pages with many tables. We hope this will be better when Netscape starts using the new rendering engine, Gecko [31].

5.9 Areas of Usage

The monitor can be used for several reasons:

- **to watch for attacks**
  - **real-time analysis**
    The main use of the monitor is to be run in real time and to send events to a system management tool. By combining several probes, the events can be correlated and the state of the network can more easily be supervised.

– **log files**
Log files are often analyzed in batch mode at another site. The monitor can use log files to look for break-ins to find out how far back one has to go to restore a secure system from backup tapes. Consultants also use log files to prove to management that their site has been probed and possibly seriously attacked.

- **for policy enforcement**
The program can be used on proxies to control the surfing habits of employees and make sure they do not submit confidential information.

For an example of the latter case, let us consider on-line businesses that ask for credit cards. If they use the `GET` method to give this information to the cgi program, the credit card numbers are written into the local proxy, the merchant's log files, *and* any other site the user visits afterwards through the referrer field. Almost the same scheme can be misused on a secure web server. Netscape up to version 4.05 offers the feature to log in directly by typing the following `url`:

`https://username:password@secure.site.com` (example 5.1)

This `url` is saved locally in proxies, on the secure web server itself in the access log, *and* at any other site the user visits afterwards. The tool can be used to verify that no such sensitive information leaks out to an external web server. This is possible if the tool is installed in the firewall through which requests to the outside world have to pass, or at any proxy server.

5.10 Shortcomings of the Monitor

No direct attacks against the actual server can be detected because they probably cause the server to crash before any information is written to the log file. In one case, we also noted that some valid requests disappear before the server actually wrote them to the log file. The setup at the site where this happened is very complex, and it did not work that way for our installed test server.

With certain methods, such as `POST`, the arguments are not written to the log file. Thus, it is more difficult to analyze these requests.

Another disadvantage is that the monitor only gives an indication, but not a proof, of a successful attack. Even if a program is executed successfully with the commands to mail the password file, the monitor cannot say whether the file was actually sent. Nevertheless, the monitor issues a warning of possible attacks, and it is the task of the administrator to control the machine. By including more specific functionality, the monitor may run the risk of being less portable to various operating systems.

The monitor did crash and issue strange error messages when running on the Nagano logs. After some research, we came to the conclusion that the crash was caused by memory problems of Perl. It may be that the Perl interpreter itself or the library routine `malloc()` was installed incorrectly. After we installed a later version (Perl 5.005_02) it worked fine, but the memory requirements were severe. Sites should run the latest version of Perl when using the monitor and it may be necessary to restart the monitor if memory consumption is critical.

The main weakness associated with a web server is the improper setup of the server and vulnerable cgi programs. The server software in itself is by comparison quite secure because the Internet community closely monitors new releases. This monitor is supposed to be run in conjunction with other types of sensors, preferably ones that detect low-level attacks.

5.11 Lessons Learned

The monitoring tool proved to be portable among various web server brands and operating systems. It uses few resources, with the exception of the Nagano logs. In Section 6 we give suggestions to improve the tool. It can be used in real time.

The results also showed that the learning of new attacks envisioned in the design is feasible. By means of this mechanism, the monitor can detect new attacks. The reporting in HTML may seem restricted, but correctly set up *trusted signatures* and a later-version browser circumvent this problem.

As the monitor depends on the log file as its source of information, only certain types of attacks are detected but, as explained in Section 1.3.2, this is where we expect the most problems to arise.

Chapter 6

Improvements

After having run the monitor against real data, it is easy to make a list of functionality that should be included in a later release of the program.

Pattern Module The attack signatures (and the safe signatures) are stateless. We save information based on hosts, and types of attacks, but there is no way to combine several types of signatures at several requests to raise an alarm. The ability to write a multi-line signature would be very good (example given in Section 3.3.3). Work is currently being done in that direction in the Zurich GSAL by sending the warnings to the Tivoli console.

Suspicious Module To make the suspicious module work slightly better, it would be advantageous to have an internal buffer of requests, sorted either by time or by host. Because many of the attacking hosts follow the pattern of including at least one known attack signature, and because the attacks are localized in time (if they are using a standard tool to send many requests), the program can search backwards to find a few requests it may have missed. This avoids the need to rerun the logs to search for new attack patterns.

Trusted Model The trust could be limited in time or to a given number of requests. It could also *age* to avoid having old, invalid signatures in the database.

Print Module If the program is run on a file, it would be convenient to receive a summary of warnings, printed either if the rate at which they are issued changes, or when an internal count is exceeded. In the latter case, the first warning would be printed, then the 10th, then the 100th etc.

Reporting Facility By writing a platform-independent applet, the reports can be made dynamic.

The monitor can also be used in conjunction with a database to explain the vulnerabilities found. We also see certain gains if the monitor is implemented as an Apache module, in which case it would profit from more power and better efficiency when [32, 33]:

- checking the url before the www server executes the request,

- checking the status code before sending the reply, and
- using more information available to an internal module.

The monitor might serve as an active protective shield against malicious attacks by stopping the server from executing malicious requests.

Chapter 7

Conclusions

In this thesis we have presented a tool for monitoring web servers. We have discussed its design principles, and shown the results of its behavior when tested at real sites. It has proved to have several interesting features.

First, it allows very general signatures to be written to detect attacks. The attack signatures can be:

- **simple** by using only the offending program name;
- **versatile** by allowing regular expressions, which filters out certain false alarms. These are compiled in the beginning to gain speed;
- **general** by using the entire request, not only the `url`
- **context-saving** by applying the signatures on a particular piece of the request, which decreases false alarms and makes the signatures simpler;
- **of a class** that allows similar vulnerabilites to be treated in the same way, which also increases speed;
- **combinable** to create signatures based on several fields of the request;
- **refined** to search for additional information, which increases speed and decreases memory usage;
- **compulsory** by checking for signatures that must be present as opposed to those that should not be present (allows policy enforcement versus attack detection);
- **host-specific** to match host names quickly to keep track of malicious hosts;
- **adaptable to thresholds** for each class to allow the detection of a wider range of attacks, from vulnerable cgi programs to denial-of-service attacks.

Second, it has a module to filter out false alarms. The module uses so-called *trusted signatures*, which can be any regular expression. Each trusted signature is given a list of types of alerts it is allowed to cancel. This reduces the risk of having too powerful of signatures. These signatures must be adapted to different sites.

To detect a wider range of attacks, the monitor will keep track of malicious hosts and their actions. It will differentiate between the actual host and the domain to allow for a university-like setup, or ISPs, where users may change hosts between sessions. The information kept will *age* in terms of a half-life function, meaning that old attacks will have less value. The supervision of suspicious hosts also alleviates the need to keep the attack signature database up to date because the behavior of attacking hosts will help deduce new signatures to be added.

We ran the monitor at many different types of sites to see how it behaved. The results presented show that the monitor can analyze logs in real time, and it does not use a lot of server resources in general. An exception were the log files from the Olympic Games at Nagano, Japan, but we believe this may be improved by adding a filter to the *Print Module*, and by regularly cleaning the statistics saved by the *Decision Module*.

The monitor discovered several attacks at all sites. We identified one tool available over the Internet that had been used for one attack. Thanks to the *Suspicious Module* and the behavior of the attacking hosts, the monitor can learn of new attacks by itself, pending a review by a system administrator.

Finally, we pointed out some features of the monitor that could be improved.

Chapter 8

Acknowledgment

Even though only one name appears on the front page, the making of this thesis is due to many people. First I would like to thank Marc Dacier and Hervé Debar for their support. Without their helpful suggestions and ideas, this thesis would hardly have been possible.

Many thanks to Andreas Wespi who somehow managed to keep up with my questions concerning new systems, and to Dominique Alessandri, who showed me new vulnerabilites relating to the web server.

This work was also dependent on access to different types of log files. Without the cooperation of Jim Challenger, Dave A. Curry, Michael Niksch, and Marie-France Declerfayt, the tool would not work as well as it did. Each new set of logs taught me valuable lessons.

I also want to thank Lilli-Marie Pavka, who made sure that the English in this thesis is comprehensible.

Then I would like to thank the Department of Scientific Computing in Uppsala, and especially my supervisor Tom Smedsaas for his supervision and flexibility.

Bibliography

[1] Anup K. Gosh. *E-Commerce Security Weak Links, Best Defences*. John Wiley & Sons, Inc., New York, New York, 1998.

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

[3] Lincoln D. Stein. The world wide web security faq. Internet. http://www.w3.org/Security/Faq/.

[4] Biography of Tim Berners-Lee with links. Internet. http://www.w3.org/People/Berners-Lee/.

[5] Lincoln D. Stein. *Web Security A Step-By-Step Reference Guide*. Addison Wesley Longman, Inc., Reading, Massachusetts, 1998.

[6] Simson Garfinkel and Gene Spafford. *Web Security & Commerce*. O'Reilly & Associates, Inc., Sebastopol, California, first edition, 1997.

[7] Aviel D. Rubin, Daniel Geer, and Marcus J. Ranum. *Web Security Sourcebook*. John Wiley & Sons, Inc., New York, New York, 1997.

[8] Bugtraq mailing list hypermail archive. Internet. http://www.geek-girl.com/.

[9] Hacking resources and information resources. Internet. http://www.playerslounge.com/hacking.htm.

[10] L0pht advisory released april 1996 about test-cgi vulnerability. Internet. http://www.l0pht.com/advisories/test-cgi-vulnerability.

[11] Html 4.0 specification. Internet. http://www.w3.org/TR/REC-html40/.

[12] Martijn Koster. A standard for robot exclusion. Internet, 1994. http://info.webcrawler.com/mak/projects/robots/norobots.html.

[13] Lincoln D. Stein. Gaining the upper hand with rude robots. *Web Techniques Magazine*, 3(5), May 1998. http://www.webtechniques.com/features/1998/05/webm/webm.shtml.

[14] Paul Phillips. Safe cgi programming. Internet, September 1995. http://www.go2net.com/people/paulp/cgi-security/safe-cgi.txt.

[15] Selena Sol. Security issues when installing and customizing pre-built web scripts. Internet, February 1998. http://Stars.com/Authoring/Scripting/Security/.

[16] Shishir Gundavaram and Tom Christiansen. Perl cgi programming faq. Internet. http://www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html.

[17] Roy Fielding. wwwstat: Httpd logfile analysis software. Internet. http://www.ics.uci.edu/pub/websoft/wwwstat/.

[18] Grant Taylor. Autobuse, a log-monitoring program. Internet. http://www.picante.com/˜gtaylor/download/.

[19] Logcheck. Internet. http://logcheck.web.aol.com/.

[20] Cole Tuininga and Rob Holak. Logscanner. Internet. http://logscanner.tradeservices.com/.

[21] Kathleen A. Jackson. Intrusion detection system (IDS). Internal report, Los Alamos National Laboratory, Los Alamos, New Mexico, 1999. Contact the author for a copy of the survey.

[22] Rfc 2068: Hypertext transfer protocol – http/1.1. Internet. http://info.internet.isi.edu:80/in-notes/rfc/files/rfc2068.txt.

[23] Lincoln D. Stein. Webmaster's domain. *Web Techniques Magazine*, 3(10), October 1998. http://www.webtechniques.com/toc/9810toc.shtml.

[24] The common logfile format. Internet.
http://www.w3c.org/Daemon/User/Config/Logging.html#common-logfile-format.

[25] Lincoln D. Stein. Cute tricks with perl and apache. Handout/Internet, July 1998. http://www.modperl.com/perl_conference/handout.html.

[26] Nathan Neulinger. Cgiwrap. Internet. http://www.unixtools.org/cgiwrap/.

[27] Tool to probe web servers: cgiscan. Internet. http://www.k-elektronik.org/arsip/eksploit/attack-tools/cgiscan.c.

[28] Arun Iyengar, Jim Challenger, Daniel Dias, and Paul Dantzig. Techniques for designing high-performance web sites. *IEEE Journal of Internet Computing*, October 1998. submitted to the IEEE Journal of Internet Computing.

[29] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable system for consistently caching dynamic web data. In *18th Annual Joint Conference of the IEEE Computer and Communications Societies*, New York, New York, March 1999.

[30] Jim Challenger, Paul Dantzig, and Arun Iyengar. A scalable and highly available system for serving dynamic data at frequently accessed web sites. In *1998 High Performance Networking and Computing Conference*, Orlando, Florida, November 1998.

[31] Gecko layout engine. Internet. http://www.mozilla.org/newlayout/faq.html.

[32] Lincoln Stein and Doug MacEachern. *Writing Apache Modules with Perl and C*. O'Reilly & Associates, Inc., Sebastopol, California, first edition, 1999. http://www.modperl.com/.

[33] The apache/perl integration project. Internet. http://perl.apache.org/.

[34] Simson Garfinkel and Gene Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc., Sebastopol, California, second edition, 1996.

[35] Sriram Srinivasan. *Advanced Perl Programming*. O'Reilly & Associates, Inc., Sebastopol, California, first edition, 1997.

[36] Tom Christiansen and Nathan Torkington. *Perl Cookbook*. O'Reilly & Associates, Inc., Sebastopol, California, 1998.

[37] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, California, second edition, 1996.

[38] Randal L. Schwartz and Tom Christiansen. *Learning Perl*. O'Reilly & Associates, Inc., Sebastopol, California, second edition, 1997.

[39] Teresa F. Lunt, Ann Tamaru, Fred Gilham, R. Jagannathan, Caveh Jalali, Peter G. Neumann, Harold S. Javitz, Alfonso Valdes, and Thomas D. Garvey. A real-time intrusion-detection expert system (IDES). Technical report, SRI International, February 1992. SRI Project 6784.

[40] Netcraft. The netcraft web server survey, 1999. http://www.netcraft.com/survey/.

[41] Arun K. Iyengar, Mark S. Squillante, and Li Zhang. Analysis and characterization of large-scale web server access patterns and performance. Research Report 98A00592, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1998.