A Secure World-Wide-Web Daemon

by Dr. Frederick B. Cohen ‡

Abstract: In this paper, we begin by discussing some of the protection-related history of world-wide-web servers and clients, some of their better-known vulnerabilities, and the need for a more secure server environment. We then discuss the protection goals we believe to be of import to a world-wide-web server, outline some of the principles we believe to be important to attaining such a server, and analyze the design of a server that we believe to be secure relative to our stated goals. Finally, we discuss some of the experience we have had with this server, the development of a secure gopher server using nearly the same code, and future work.

Search terms: world-wide-web, gopher, secure daemons, secure servers, client-server, operating systems security, integrity, availability, confidentiality.

‡ This research was sponsored by *Management Analytics*, PO Box 1480, Hudson, OH 44236, USA

1 Background and Introduction

The *World Wide Web* (W3) is probably the most widely used distributed information system ever to exist. [3] It currently has in excess of 25 million users and its user population has more than doubled over the last year. Architecturally, it consists of; a set of *servers* (typically implemented as *daemons* in Internet *hosts*) which receive one request at a time and respond to that request without preserving any state information; and a set of *clients* (also known as *browsers*) that make requests based on user input and present the results.

1.1 World Wide Web Services and Operation

Information in W3 is accessed by way of a *Universal Resource Locator* (URL) that refers to any particular item and consists, in its most general form, of a request type, a site name or IP address, a port number, and a pathname for the requested item. For example,

http://all.net:80/index.html

indicates a *hyper-text transfer protocol* access to the machine on which this work was done (all.net) on port 80 (the default IP port for W3 service) requesting the item named *index.html*. Since 80 is the default port number for *httpd* and *index.html* is the default file name, the URL above could also be expressed as *http://all.net*.

In the typical implementation, most files being accessed are *hyper-text markup language* (HTML) files which consist of formatting information, text, pointers to other information (in the form of URLs), and recently, *form* filling out commands.

Typically, browsers are graphical interfaces that present one or more pages worth of information and mark *pointers* to other information (generally consisting of a URL and some associated text or graphics) by coloring or underlining. The user can then *point and click* on any *pointer* and the browser will retrieve the item pointed to by the URL and present it on the screen. In this way, W3 resources point to each other and present a world-wide information retrieval system.

Anyone in the world with the means to do so can place a server on-line and try to get people who run other servers to provide pointers to their server or otherwise get people to use their service. Probably because of the graphical user interfaces, the extreme ease of use, and the emergence of a low-cost global information infrastructure, W3 has taken off, and there are now over 40,000 servers and millions of users. As is almost always the case, the designers of W3 did not adequately consider protection when implementing their service and, as a result, we now have millions of users and tens of thousands of servers potentially at risk.

1.2 Risks Associated With W3

The main risk to the user comes from the content of the information interpreted by their W3 client and the potential for servers to gather and exploit information resulting from client requests. For example, a server could present a *postscript* file containing a Trojan horse for viewing by browsers. Each browser that interprets the file runs the attacker's *postscript* program on their machine and suffers the consequences. By using the logging capability associated with most daemons, demographic information gathered by an on-line HIV help-line server might be used to blackmail people. Although these are very important areas of concern, they are not what this paper is about and they will be ignored from this point forward.

The main risk to providers of W3 services is that someone might be able to fool their server daemon (commonly known as an *http daemon* or *httpd*) software into doing something it is not supposed to do, thus allowing an attacker to break into their server and do some harm. The harm might include planting Trojan horses in files being provided to clients, corrupting other server information, disrupting services, gathering data about users and their usage, releasing information from the server, or using the server as a platform to launch other attacks.

In recent months several such vulnerabilities have been found in commonly used server software, and historically, this has been the source of many security problems. For example:

- The February 17, 1995 *Computer Emergency Response Team* (CERT) alert titled *NCSA HTTP Daemon for UNIX Vulnerability* reveals that "A vulnerability in the NCSA HTTP Daemon allows it to be tricked into executing shell commands." This was the result of being able to overrun an internal buffer. The suggested fix was to increase the buffer size.

- In another CERT Advisory on April 10, 1995 titled *Vulnerability in SATAN*, CERT warned "This vulnerability exploits a weakness in the HTML server started by SATAN on a random, high-numbered TCP port."
- Another vulnerability was found early in 1995 that allows a client to modify log files created by *httpd*. The impact on the server is not as severe as the previous examples, but such a failure indicates a potential for other exploitation.

Clearly, the identification of three actively exploited vulnerabilities in as many months indicates that the potential for abuse is high and that the current daemons are not designed with adequate protection.

The lack of secure design is reflected in many ways. One way is the sheer size of the current source code for *httpd*, which is more than 6500 lines of code written in C and uses about 1200 lines of auxiliary C source code and several hundred lines of configuration files, for a total of about 8000 lines of source code. Considering that the main function of this code is to copy a requested file, it seems excessive.

The most common *httpd* programs austensibly offer security features including directory-by-directory and file-by-file access control lists, however, in practice, these mechanisms ignore the underlying operating system protection features that could make protection effective in favor of ad-hoc creations of the authors. The protection mechanisms are also based on unauthenticated information provided by the client from over the network, and are thus easily bypassed.

These *httpd* programs create numerous processes and handle interprocess communications, provide aliasing capabilities so that names of directories and files can be modified for remote access, process *mime* and similarly encoded messages to allow automatic decompression of compressed files stored on-line, and allow processing of several different kinds of forms, all of which do essentially the same thing in different ways. And yet none of these functions are required in order to provide W3 services.

Many similar problems have been identified in other daemons, and in most cases, these vulnerabilities lead to the total collapse of protection on the server. For example, forged authentication daemon replies caused vulnerabilities in a large portion of existing electronic mail (*smtp*) servers early in 1995, the second such vulnerability revealed in those servers in as many months. *Gopher* daemons, which perform almost the same functions as W3 servers, have had similar problems over the last several years, but the lessons learned were never translated into W3 servers.

In a general sense, this paper is about server software designed to prevent this sort of exploitation. In a specific sense, it is about the design of one secure server that provides almost all of the services provided by more common *httpd* servers using less than 1% of the amount of source code and with reasonable assurance of effective protection.

1.3 Secure Server Design

In order to reduce the risks associated with servers of this sort while still providing a commercial presence for these services to the Internet community, organizations have taken many strategies, ranging from leasing space on provider systems to creating elaborate bastion hosts on network firewall systems to handle these services. The costs associated with these solutions are relatively high, primarily because of the added complexity of securely operating and maintaining systems set up in this way.

An alternative solution to the protection problem with servers is to design a server with protection properties that can be explicitly demonstrated. This is the approach we took with this daemon.

The general properties of interest to us are (in order of highest to lowest priority):

- Information Integrity - We want to assure that the information residing in the server is not corrupted by the actions of legitimate or illegitimate clients as a result of their use or abuse of the service.
- Availability of Service - We want to assure that clients cannot make the server unusable for other clients as a result of their use or abuse of the service.
- Confidentiality - We want to assure that the service only provides information to clients that is explicitly authorized for outside access.

The reason these properties are of particular import are that we wish to provide assurance that the information placed on the server is not modified as a result of the provision of Web services, that Web services are provided with a high degree of reliability and without lowering the reliability of other services provided on the server, and that no information not designated for release through the Web server is released by the Web server. Other properties may be of interest to other designers of Web servers, however, these were our design goals.

We would like to assure these properties to a higher degree for information not explicitly designated for outside use in order to allow the server to operate on a firewall or other system without increasing the risk to that system through the presence of the server.

2 Design Features of Our Secure Daemons

In order to assure the desired security properties, we have made several important design decisions:

- **Small Size:** The size is kept to a minimum to enable analysis and minimize potentially complex interactions.
- **Confinement of the Daemon's Privileges:** Privileges of the daemon are minimized to limit the impact of any errors or corruptions. This is the well known principle of least privilege.
- **Confinement of Processes and Memory:** Minimal resource usage and allocation are done to reduce the interactions and thus the complexity between the daemon and its environment.
- **Confinement of the Daemon's File System:** File system access is restricted to limit the ability of the daemon to examine or modify unrelated information.
- **Limited Function:** Limited function is desired in order to make viruses and Trojan horses non-functional within the daemon's operating space.
- **Confinement of File Output:** File outputs are confined to limit potentials for corruption.
- **Confinement of File Inputs:** Accessible files are restricted to limit the ability of the daemon to leak information.

- **Confinement of Information Flow:** Information flow is limited to implement a POset and thus limit transitive corruption and leakage.
- **Finite Runtimes:** Finite runtimes are enforced to assure that denial of services attacks cannot succeed based on a lack of adequate input, a large number of requests, failures due to simulataneous access constraints, or other similar events.
- **Defense-in-Depth:** Defense-in-depth is used to protect against configuration errors and provide for detection of flaws before they become vulnerabilities.

2.1 Small Size

In the field of information security, it has long been recognized that writing secure software becomes far more complex as the size of the software increases. Proof of program correctness to verify even simple security properties, for example, grows almost exponentially with the number of program statements. Verifying a 100 line limited-language program for the simple security properties associated with the Bell-LaPadula model of security takes many hours of CPU time on the fastest supercomputer. The 8,000 lines involved in most *http* daemons makes it impossible to formally verify their security (or more likely demonstrate their insecurity).

The size of our secure daemon is very small when compared to the size of the other daemons now being used to provide similar services. The current secure *http* daemon contains only about 70 lines of source code. (see appendix A) This is less than 1/100th of the size of the more common server daemons. If we removed the audit capabilities from the server, it could fit into about 30 lines of source code, and simplistic versions without security features containing only 20 lines of source code have been demonstrated.

2.2 Confinement of the Daemon's File System

In the first few instructions, the daemon is confined to operate in a subpart of the normal server's file structure. This is accomplished through the *Unix Chroot* operating system call that makes the file structure appear to the daemon as being rooted in the subtree of the file structure containing the information provided for export.

Many attacks against servers concentrate on getting a copy of the password file and then using a password guessing program to find a valid entry point. With the confined file system provided by *Chroot*, there is no password file available within the virtual file system seen by the program. If properly configured, the only information that resides in the virtual file system seen by the daemon is the information designated for release.

2.3 Confinement of the Daemon's Privileges

At startup, daemons that process network information in most current operating environments are initiated with *superuser* privileges. In the case of *Unix*, the *inetd* daemon starts other daemons and grants them privileges defined in a configuration file. Privileges are required in order to limit the file system of the daemon, but as soon as that change is made, privileges are revoked and the daemon operates with no more privileges than a normal non-privileged user on the server. This is accomplished by use of the *Unix SetUID* operating system call that sets the effective user identity (UID) of the daemon's process to that of a user specified for the purpose. We will call this user *www*.

By running the daemon as user *www*, all of the basic operating system protection features (e.g., access control, process separation, limited input and output capabilities) that are used by millions of users every day are used to protect the server against the actions of the daemon.

By running the daemon as user *www*, we also provide an ability to trace the activities of the daemon and to readily differentiate those activities from all other server activities. This makes automatic detection of anomalous behavior very easy, and when combined with other confinement properties, makes detection of security violations by the daemon very simple.

2.4 Confinement of Processes and Memory

We prefer to use only one process per request and to use no multiprocessing or memory allocation. The use of multiple processes introduces complexities associated with interprocess communication and increased dependencies on other operating system features, and may result in exhaustion of resources. Avoiding unnecessary memory allocation eliminates undecidability issues like the allocation problem and eliminates complexities relating to the handling of failure conditions.

The use of static variables of fixed length facilitates easy understanding and analysis, but flies in the face of the common programming styles now being taught. It is our general belief that writing programs with the goal of effective protection is a very different sort of programming than writing programs with other goals. The present program was written in a very short period of time and was written for ease of understanding. Variations with no memory allocation or compiler function calls have been written for those who wish to be even more certain in their analysis.

2.5 Limited Function

The function of the secure daemon is quite simple. It gets a request for a file, verifies the propriety of the attempted access, sends the file or an appropriate error response to the client, and logs the activity in a predetermined log file. For this limited purpose, there is no requirement to interpret anything stored in the server, so the daemon only implements a limited function and executes no external program or function.

The limited function of this daemon makes it useful for providing well over 99 percent of all W3 services in common use today, but the inability to process the *post* function, which involves executing arbitrary programs on the server on behalf of the client, is a limitation that is severe in some environments. More information on this is provided in the section on further work.

Since the daemon runs with no special privileges and there are no programs or other interpretation capabilities in the available file system area that can extend privileges, the daemon is limited to the functions described earlier.

2.6 Confinement of File Output

The daemon only writes output to one log file which is defined at compile time. In order to corrupt information, it is necessary that there be a means by which the corrupt information can travel from the attacker to the area in the server being corrupted. By limiting output to a single pre-defined activity log file, the ability to corrupt any of the files provided by the server to clients is severely limited. Unless the attacker can cause the daemon program to be modified as it operates in the server's memory or attack the server by some method unrelated to the daemon, it is impossible for the daemon or a client acting through the daemon to modify any information on the server other than the activity log file.

This does not mean that it is impossible for an attacker to corrupt the daemon or the server. It only means that it is impossible to do so through the daemon itself unless the attacker can cause the daemon to become corrupt as it operates on the client's request. For example, the normal system logs kept by operating systems may be overrun by excessive use of the server without adequate log-file maintenance, thus causing denial of services. Since this is not within the realm of the daemon to control, it is beyond the scope of our analysis, but it is a possible side effect of having such a server.

2.7 Confinement of File Inputs

Except for the log file described earlier, the daemon only opens regular files in a read-only mode and only opens those files if they are owned by the user *www* and readable by *world* (a common protection method and the one used by Unix systems differentiates read, write, and execute privileges for owner, group, and world) . This means that the daemon will not read from any source that hasn't explicitly been designated for the purpose of providing information to outside users.

Even if, by mistake, some other user on the server places information in the server area, that information cannot be read by an attacker unless the ownership of the file is explicitly given to the user *www* and the protection setting is made to allow *read* access by *world*. Even then, the file cannot be over-written by the daemon.

To allow access to information, a user on the server's computer must set ownership to the user *www*, put it in a file in the special *chroot* area, and make it readable by *world*. By default, information does not meet these constraints, and it takes a specific effort to make information available via the server. System defaults do not allow access.

2.8 Confinement of Information Flow

The daemon only reads one request of fixed maximum length from one *Transmission Control Protocol* (TCP) channel setup by the operating system and stores the request in a fixed length array for analysis and use. It also reads two values corresponding to the requesting client's host name and user name from output generated by the *TCP wrappers* program now in widespread use. [2]

Since these are the only inputs to the daemon that come from a possible attacker, tracing the flow of these inputs through the program can assure us that the information provided by an attacker cannot end up anywhere it should not go and that it cannot affect the server in any but a limited number of well-defined ways.

The input buffers cannot be overrun because the read operations limit the number of input bytes to less than the buffer size. From the point where input is taken by the program through program termination, all information derived from external sources is confined to a specific set of routines and variables and it is demonstrated later in this paper that these inputs can only affect a limited set of things within the daemon. By exhaustion, it is shown that none of the things that inputs can affect can result in program misoperation.

The combined information flows form a trivial partially ordered set which has been shown to effectively limit the transitive corruption and/or leakage of information within any environment. [1]

2.9 Finite Runtimes

Finite runtime assures that a finite amount of resources are used on each request. This, in turn, makes denial of services attacks more difficult by limiting the total affect of any finite number of requests. Finite runtimes are assured in two ways:

- All routines are finite in terms of the number of operations they perform. Thus, unless any specific operation takes infinite time, the daemon as a whole must take finite time. All operations except input and output are guaranteed to take a finite number of instructions with each instruction being finite in its time of operation.

- Input and output are impossible to limit because, among other things, input is controlled by the client and not by the server. To assure that all input and output have finite runtimes, an *Alarm* is used to terminate operation of the daemon in a finite time unless the input or output is completed in time and the alarm is turned off. All inputs and outputs are surrounded by alarms so that they are individually guaranteed to have finite runtimes or terminate the operation of the daemon.

2.10 Defense-in-Depth

If the previous information is carefully reviewed, it can be seen that there is redundancy in the protection provided by different methods used by the secure daemon. This is intentional.

Many authors use a single protection mechanism under the assumption that it is perfect. They invariably find that some assumption is flawed, some software component is defective, or some unanticipated combination of events causes protection to break down. Our perspective is quite different.

We expect that if any one of these precautions were taken, the daemon would probably be moderately secure, but we don't want a moderately secure daemon. Experience tells us that as soon as one person finds a way through a moderately secure daemon they will leave a path that others will follow. We take a multitude of precautions so that if and when one fails, the others will prevent harm, give warning, and allow response to counter the detected weakness before it becomes a vulnerability. [4]

By way of example, consider the following scenarios:

- **Scenario 1:** The *Chroot* function fails to change the root directory and also fails to return the proper value indicating its failure. Alternatively, the *Chroot* function doesn't work properly under some previously unknown condition.

  Result: The server responds to requests by looking in areas of the file system not explicitly setup for its use. It finds files not owned by user *www* and reports the failures. The systems administrator looks at the audit trail and determines that the failures were caused by non-ownership or non-existence of specific files and determines what has occurred. The server remains safe from corruption and leakage and suffers only limited denial of service (i.e., the *httpd* service will be denied, but other server services will continue to operate unhindered) while repairs are made.

- **Scenario 2:** The *SetUID* function fails to change users and fails to report the error.

Result: The daemon operates with root privileges, but cannot provide any files to clients because it does not own the files being requested, and cannot be corrupted to perform other functions because of the confinements on information flow. It logs the errors which the administrator analyzes and responds to. The server remains safe from corruption and leakage and suffers denial of service for the *httpd* function while repairs are made.

- **Scenario 3:** The systems administrator of the server accidentally sets all files on the system to be able to be read by anyone and changes the ownership of all the system files to the user *www*.

  Result: The *Chroot* environment prevents outside access to any areas other than those explicitly designated for remote access. Normal system activities begin to fail from protection violations and thus the problem is detected. The *httpd* daemon remains operational and safe from corruption and denial of services while some information placed in the area provided for dissemination but not previously protected to allow its release may be released between the time the mistake is detected and the time the repairs are made.

2.11 Summary of Design Principles

The secure *http* daemon is designed in such a way that we can demonstrate (subject to the propriety of compilers, operating system functions, and other things in the environment) that once the daemon is started, only the desired affects result.

In addition, the redundant protective features cause the daemon to behave well even under conditions where the surrounding environment has succumbed to security failures or is fundamentally flawed. This makes the daemon secure even when some of the assumptions made about its environment break down.

3 Detailed Code Walkthrough

We now provide a code walkthrough that examines the W3 daemon in detail to show that it properly implements protection as discussed earlier. Complete copies of the secure *http* and the closely related secure *gopher* daemons are provided in the appendices. [1] In our discussion, we italicize and capitalize variable and function names for readability.

3.1 The Global Variables and Macros

We begin this code walkthrough with the definition of the global variables so that the rest of the code can be understood when it references them.

```
int CHECKUSER=1;int DOCHROOT=1;
#define BUFSIZE 4096
#define MAXSIZE 2048
char line[BUFSIZE],name[BUFSIZE],bs1[BUFSIZE],bs2[BUFSIZE],bs3[BUFSIZE], timestamp[64],
logline[BUFSIZE], remotehost[BUFSIZE], remoteuser[BUFSIZE];
struct stat buf; FILE *F;
int i,n,staterr;FILE *F;
time_t *tloc;time_t t;
```

Note that all arrays are of fixed length and predefined. This is done so that no space allocation is required by the program through its variables once the daemon is operating. Although these operations can be done reliably, it is desirable for simplicity that we limit the operations to those absolutely required for the program's function. Notice also that DOCHROOT=1 defines *Chroot* as desired. This can be altered at compile time to prevent the *Chroot* function if so desired.

Next come the defined macros and constants, again so we can understand the context.

```
#define ERRORLINE "The requested document has moved to
                       <A HREF=http://all.net/>here</A>.<P>\n"
#define REDIRECT "Location: http://all.net/\n"
#define WWWUID 101
#define WWWDIR "/u/www/htdocs"
#define WWWDefFile "/testserver.html"
#define WWWlog "/log"
```

[1] These program listings have been slightly altered to fit the page. The current sources to these programs are available on the World Wide Web through URL http://all.net which demonstrates and uses these servers. Licensing information is also available from that site.

```
#define Ktime 10
#define IOtime 15
#define LOG2(x,y) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y);} fclose(F);}
#define LOG3(x,y,z) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y,z);} fclose(F);}
#define LOG4(x,y,z,w) {F=fopen(WWWlog,"a+");
        if (F != NULL) {logfile(F);fprintf(F,x,y,z,w);} fclose(F);}
```

3.2 The Main Program

Now we will demonstrate operation by starting at the main program, where the program starts when it is run, and showing what it does, why it does it that way, and hopefully, why the operation is safe. We have added line numbers for reference purposes.

```
01 main(argc,argv,envp)
02 int argc; char *argv[],*envp[];
03 {alarm(Ktime);if (0 != chdir(WWWDIR))error("Cannot change to WWW directory");
04 if (DOCHROOT == 1) if (chroot(".") != 0)  error("Cannot change root directory to .");
05 if (0 != setuid(WWWUID)) error("setUID failed"); /* become user www or die */
06 if (argc>1) strncpy(remotehost,argv[1],MAXSIZE); else strcpy(remotehost,"nowhere");
07 if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE); else strcpy(remoteuser,"nobody");
08 remotehost[MAXSIZE]='\0';remoteuser[MAXSIZE]='\0';alarm(IOtime);
09 read(0,line,MAXSIZE); sscanf(line, "%s %s %s", bs1, name, bs2); /* get request */
10 if (name[strlen(name)-1] == '\r') name[strlen(name)-1]='\0'; /* some browsers! */
11 if ((name[0]=='/') && ((name[1]=='\0') || (name[1]==' '))) strcpy(name,WWWDefFile);
12 if (DOCHROOT!=1) {strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);}
13 alarm(0);if (strncasecmp(bs1,"get",5) == 0) fetch(); /* get */
14 error("Unknown request"); /* all other requests fail */ }
```

Lines 01 and 02 are standard *C* program startup information defining the main program and the three normal arguments. Notice that there are no local variables in the main program.

Line 03 sets an alarm which will limit the time allowed between the start of the program and reaching the *alarm(0)* statement in line 08. If *Ktime* seconds pass before reaching it, the operating system will terminate the daemon forcibly. It then attempts to change directories to the area being used for W3 services. If this cannot be done, the program immediately fails, never having examined any outside information. The *Error()* routine will be detailed later. Line 03 is required in order to implement the *Chroot* function that limits access to a subset of the file system.

Line 04 changes the *root* directory to the current directory. Note that since no outside input has been brought into the program yet, it cannot affect the program's operation. If the *Chroot* function fails, the program immediately fails, never having examined any outside information. This line is required in order to operate in a *Chroot* environment.

Line 05 changes the effective UID of this program to the special *www* UID provided for the server function. Note that in order for the *Chroot* function above to be performed, it is necessary to begin operation as the *superuser*, and that after the necessary and sufficient operations are performed with privileges, this function removes those privileges in favor of the lower privileges of the *www* user. If the *Setuid* function fails, the program immediately fails, never having examined any outside information. This line is required in order to operate in a less privileged environment after doing a *Chroot*.

Line 06 verifies that there is a first command line argument, and if one is present, copies its value into the *Remotehost* array. If it is not present, it copies the fixed string *nowhere* into the *remotehost* array. The command line arguments, if any, are provided by the *TCP wrapper* program noted earlier or by the *inetd* daemon used to invoke the daemon. The array is 4096 bytes long, which is larger than the system-defined limit on the size of a command-line argument and larger than the largest possible argument generated by the *TCP wrappers* program. In addition, *strncpy* limits the number of bytes copied to *MAXSIZE* and line 08 enforces this boundary with a terminating '0' byte. Thus, the input, which is provided indirectly from an outside *Domain Name Server*, is confined to the array *Remotehost*. Line 07 does the same operation for the second command-line argument and stores the result in *Remoteuser*, with line 08 enforcing the associated confinement property. Once confinement is enforced, the alarm is reset to allow the program to continue functioning for another *IOtime* seconds.

Line 09 reads the only input provided by the user through the normal *TCP* channel into the array *Line*, limiting the number of bytes read to *MAXSIZE*. It then uses the *Scanf* function to split the line into *Bs1*, *Name*, and *Bs2*, each of which has the size *BUFSIZE*. Again, this enforces confinement of the input to *Line Bs1*, *Name*, and *Bs2*.

Line 10 removes any trailing *return* characters from the input line. This is not required for the confinement or security of the daemon, but addresses incompatibilities between browsers and leaves the data confined to *Name*. Line 11 checks to see if *Name* is empty, in which case, the W3 protocol requires the use of *WWWDefFile* as the name of the file to be retrieved. In this case, the lack of input results in *WWWDefFile* being contained to *Name*.

Line 11 is for installations not desiring to use the *Chroot* environment. In this insecure form, the program prepends the pathname of *WWWDIR* to the filename so that operation is transparent. NOTE: If *DOCHROOT* is not 1 the program is not operating in a secure mode! Also note that if *DOCHROOT* is 1 at program initialization, it remains 1 at this point because it has not been explicitly changed and all input has been confined so as to not affect this variable.

Line 12 determines if the request was one that this daemon handles (i.e., *GET*) by looking at the first 5 bytes of *Bs1* (non-case sensitive). Note that the only possible effect of this examination is that the routine *Fetch* is called. At most the first 5 bytes of *Bs1* are examined, and the only side effect is the calling of *Fetch*.

Line 13 first disables the alarm clock. If the program reaches this line in time, it has succeeded in getting all external input within the allotted time, and will be permitted to continue operating. Line 13 then causes an error result if the request was not valid or a call to *Fetch* if the request begins properly.

It has been commented that: 'Since *GET* is only 3 bytes long no more than 4 (3 + null) characters of *Bs1* will ever be examined.' You may call it paranoia, but in case the implementation copies both strings before checking lengths and has too-small fixed-length or erroneously generated storage, the limitation of 5 forces additional constraints. It was correctly commented that it might be a better idea to write a little routine to do this check, or to simply use an *If* with a proper conditional. In an earlier version, another reader commented on the lack of elegance in using the *If* statement. It's impossible to please everyone.

At this point in the program's execution, only two possibilities exist for program flow. Either *Error* is called with the fixed string argument *Unknown request*, or *Fetch* is called with no arguments. In either case, all results of input are stored in the first *MAXSIZE* bytes of arrays *Line*, *Remotehost*, *Remoteuser*, *Bs1*, *Name*, and *Bs2*.

It has been commented that: 'If the request and headers don't fit in *MAXSIZE* bytes, clients may hang waiting for the server to read the rest of the request.' and also that: 'On slow networks, read may return less than a full line of input so that valid requests will fail.' Both are probably true, however, almost all current requests are limited to less than the specified length, and in the insecure versions of *Httpd*, fixed string lengths are also used, and without proper bounds checking.

In the case of too-long service requests, *Httpd* will return an answer before the request is completed. All W3 clients we have tested respond properly to this condition. In the case of too-short requests, trying to wait for a request to complete could allow denial of service attacks in which an attacker could launch a series of incomplete *Http* requests designed to overrun the number of available processes in the process table of the server, and thus cause denial of services. By treating all requests as *one-shot*, we avoid the *allocation problem* which is unsolvable. In practice, this condition has never been detected, and if it were to happen, it would only deny services to select clients on rare occasions during periods of extremely poor network response.

3.3 Error Handling and Logging

The *Error* routine takes the fixed length compile-time argument and produces two results. It returns an error to the requesting client, and it appends error information to the daemon's log file.

```
01 error(s) /* simulate a 302 - document moved */
02 char *s;
03 {alarm(IOtime)printf("HTTP/1.0 302 Found\n");printf("Server: ManAl/0.1\n");
04 printf("MIME-version: 1.0\n");printf(REDIRECT);printf("Content-type: text/html\n");
05 printf("<HEAD><TITLE>Document moved</TITLE></HEAD>\n");
06 printf("<BODY><H1>Document moved</H1>\n");printf("%s (%s) </BODY>\n",ERRORLINE,s);
07 alarm(IOtime);LOG4("Error:%s - %s %s\n",s,bs1,name);exit();}
```

Lines 01 and 02 define the routine and it's character string input. Line 03 limits the operating time of the initial outputs from this routine to *IOtime* seconds. Lines 03, 04, 05, and 06 then print fixed length strings. None of these can be affected by the inputs at this point in the program because the inputs are still confined as described above. Line 07 executes the macro *LOG4* and then exits the program. *LOG4* (shown earlier) opens the fixed-name file specified as *Logfile* at compile time, executes the *Logfile* function (described next), and using the fixed format specified in line 07, prints internal variables and the external inputs stored in *Bs1* and *Name* into that log file. Error logging is permitted *IOtime* seconds to be completed before a failure results. Line 07 then closes the log file. Hence *Bs1* and *Name* are confined to the output file and cannot otherwise affect the daemon. Only the *Logfile* function remains to be examined to assure that the *Error* execution path retains the confinement properties that prevent inputs from adversely affecting the daemon.

```
01 void logfile(F)
02 FILE *F;
03 {alarm(IOtime);t=time(NULL);strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));
04 fprintf(F,"%s %s %s ",remotehost,remoteuser,timestamp);alarm(0);}
```

Lines 01 and 02 of *Logfile* define the function as having a single argument, that being the file pointer to the log file. Line 03 sets a timeout in *IOtime* seconds and then prepares a time stamp by calling system *Time* functions and storing the results in the first 20 bytes of the *Timestamp* array (which is far longer than 20 bytes). Line 04 prints the external information in confined variables *Remotehost* and *Remoteuser* to the log file and resets the alarm to allow operation to continue. Hence, these values are confined to the log file, and the whole *Error* routine maintains the confinement principles that protect the daemon from external attack.

As a side note, the time printed by the daemon is affected by the *Chroot* environment. In the worst case, this causes all times to be reported in *Grenwich Mean Time* (GMT). If properly installed per the installation instructions, the daemon reports local times properly.

3.4 The Get Command

The only remaining execution path is through the *Fetch* routine which we will now examine.

```
01 fetch() /* if www owns it, it can be put - else, forget it */
02 {alarm(IOtime);staterr=stat(name,&(buf));alarm(0);
03 if (staterr != 0) error("Can't stat file");
04 if (0 == S_ISREG(buf.st_mode)) error("Can't fetch directories");
05 if (CHECKUSER==1) if (buf.st_uid != geteuid()) error("Not owner of file");
06 if (0 != (S_IROTH & buf.st_mode)) {cat(name); LOG2("cat %s\n",name);exit();}
07 error("Access Denied");}
```

Line 01 defines the routine. Line 02 limits operating time and then uses the confined input *Name* as the name of the file to get the status of. This can only effect the results stored in *Staterr* and the repository for results *Buf*. If the routine finishes in a reasonable time, the alarm clock is disabled. If there is no file of the specified name or it cannot be detected by the *Stat* routine for any other reason (line 04), *Staterr* will return a non-zero value, which will result in calling the safe *Error* routine described earlier. If the name specified is the name of a legitimate file, then the result placed in *Buf* contains specific information relating to that file.

Line 04 prevents fetching directories. Although this does not present a security risk, fetching directories does not fit within the *html* protocol suite, so it is eliminated to prevent browser errors. Line 04 retains the confinement properties or calls *Error* which retains them.

Line 05 verifies that the named file is owned by the current *www* user. If it is not owned by that user, *Error* is called. Line 05 retains the confinement properties or calls *Error* which retains them.

Line 06 verifies that the file is readable by *world* and if so, sends the requested file to the requesting user via the *Cat* function (described later), and uses the *Logfile* procedure described earlier to report results to the log file. Thus, line 06 retains the confinement properties described earlier.

Line 07 exits the program with an error if no other exit has been exersized. This indicates an access denial because the requested file is protected against *world* access. This retains the confinement properties described earlier and leaves us only with the function *Cat* to verify.

```
void cat(s)
char s[];
{alarm(IOtime);i=open(s,0); while ((n=read(i,bs2,MAXSIZE)) > 0)
{alarm(IOtime);write(1,bs2,n);}close(i);alarm(0);}
```

The first two lines of the *Cat* function are purely definitional. In the last lines, an alarm timer is set for the initialization operations and the file named by the client is *Open*'ed for read-only access. If the file doesn't exist (even though it was previously confirmed as existing) or cannot be *Open*'ed, the *While* loop returns a non-positive result, no *Write* is done, the file pointer is removed via the *Close* operation, and the alarm is disabled. If the file exists and can be read, the contents of the file are sent to the standard output of the daemon, or in other words, to that client who made the request. The repeated use of *alarm(IOtime)* in this loop assures that each *Write* and subsequent *Read* operation occurs within *IOtime* seconds or the program terminates. The name of the file stored in S is confined so as to have no effect on the daemon or on any other part of the server other than the desired shipment of the result to the requesting client.

4 Commentary, Possible Holes, and Extensions

The security of this program depends on the proper operation of the system and compiler functions it calls. We doubt if these programs have been subjected to a similar analysis to assure that they retain confinement properties, and this introduces potential problems.

4.1 Confinement Properties

Notwithstanding these limitations, the secure daemon described here has been shown to enforce the confinement properties required for secure operation against intentional abuse as well as many of the other properties that are appropriate to secure daemon operation. That is not to say that a poorly implemented compiler and operating environment cannot make the daemon insecure.

4.2 Integrity

From a standpoint of integrity, we believe that this daemon is very safe and that it effectively protects the server it operates on from corruption caused by the daemon's operation. We have essentially shown that the only information flow from this program, regardless of input, is to the log file and any system audit trails resulting from its operation. It can only append to those files in the manner in which log files are intended to be used, and inputs are confined from causing the program to misoperate or provide invalid logging information. If there is a stronger argument to be made for integrity assurance, we are not aware of it.

4.3 Availability

The availability issue has been inadequately investigated in the scientific literature, and to the extent hat it has been investigated, little in the way of general understanding has been produced. This is one of the first examples we have seen where availability has even been considered, and we certainly have not advanced the art substantially through this limited analysis, however, we believe that, from a standpoint of denial of services, this implementation is far safer than the insecure versions of *httpd* currently in widespread use.

An attacker can use large numbers of requests to cause some requests to fail by overrunning the available number of file handles and processes on the server, however the sequential nature of file access in this daemon and the fact that a very small number of files are opened for only the minimum necessary time reduces this vulnerability compared to less secure implementations. It is far more likely that this sort of attack would work against the insecure versions of *httpd* than against this secure version because this version uses fewer processes, fewer file accesses, less memory, fewer instructions, and fewer resources in every other way we are aware of. It seems inherently obvious that a program that uses less of each type of resource available on a system is less likely to result in denial of services due to resource exhaustion.

There are few loops in this program and all of them can be easily shown to terminate in finite time, subject to finite file sizes. In addition, all input and output is surrounded by *Alarm* functions that force program termination if input or output is not completed within a prespecified amount of time. Thus it can be easily proven that this program halts, assuming the operating system calls work properly and the file structures and other operating system structures don't have infinite loops or sizes.

The fact that the daemon halts in finite and bounded time means that it uses finite overall resources and that we can, for any particular system, determine the limits of its resource usage and provide sufficient resources to assure that it cannot disrupt services under specified load conditions. With a little bit more effort, we can restrict loading to assure performance levels to clients.

4.4 Confidentiality

From a standpoint of confidentiality, we believe that these programs are at least as strong as the environment they operate in and that they do not lessen protection against information leakage. Specifically, we have shown that there is no explicit information flow from the server to the requesting party other than from the files made available for external access. An example of the sort of underlying weaknesses that this daemon does not protect against is a covert timing or resource utilization channel induced by other processes within the server environment. [5] For example, this sort of weakness could be exploited in a server providing multiple services as an indication of the success or failure of an otherwise open loop attack on that server.

4.5 Interrupts and Exceptions

The program has no interrupts at all and no exception conditions that are not properly handled in its current operation. Even in cases where the underlying environment changes during operation, the code operates without causing any protection problems that we are aware of.

This program has no interaction with outside programs or other elements of the environment other than those required for its proper operation, and it minimizes these required operations. To the extent it is possible, the program is independent of its environment and fails in a safe mode when its environment behaves unexpectedly.

Further analysis of environmental interaction was not done and a thorough analysis of this sort of interaction may be revealing, however, such an analysis was beyond the scope of this work.

4.6 Testing

This program has been tested with tens of thousands of requests including a substantial number of malicious requests specifically designed to exploit weaknesses found in other servers. None of these tests have resulted in a failure, but that does not mean that the program is secure.

Ideally, a complete test of all possible input and state conditions would provide proof that the program does as expected in all circumstances, however, such a test is infeasible because of the large number of possibilities.

Some testing has been under various load conditions. In the normal course of events, the server has operated at an average rate of more than one request per minute for several months. A simple loop test that requests the same URL repeatedly has been used over an Ethernet to simulate loads of up to several requests per second without impact. The Sun *inetd* program prevented testing at higher rates by detecting the loop condition and shutting down service when requests are made at sustained rates in excess of three per second, so extremes in performance were not tested in this study. By way of comparison, the most active W3 servers report request rates of under 1,000,000 requests per day, or an average of less than ten requests per second.

Other tests performed included:

- *Standard attack test suite:* This test suite attempts to exploit the most widely published vulnerabilities in Web servers.

  Result: None of these vulnerabilities were found to be present.

- *Input buffer overflow:* This test attempts to determine whether or not input buffers are limited so as to prevent corruptions of internal data as a result of malicious inputs.

  Result: Buffers for all inputs appeared to be properly limited.

- *Unusual characters and sequences:* This test attempts to determine whether the server reacts inappropriately when faced with character sequences not normally used in http transactions. Each byte code and pseudo-randomly selected sequences of bytecodes are used both as part of an otherwise valid request, and as an entire request.

  Result: No unexpected behavior was found and all expected behavior occurred.

- *Undocumented commands:* This test exhaustively searches the command space, thus verifying that only the *get* command works and that it works regardless of upper and lower case variations.

Result: No unexpected behavior was found and all expected behavior occurred.

- *Denial of service by resource consumption:* This test suite attempts to exhaust available resources such as input buffers, input ports, and number of recipients to determine whether failures in service result.

  Result: Test confirmed that all timeouts worked as expected and that services could only be denied by exhausting operating system limits on available processes.

- *Access control code tests:* This test exhaustively tests each access control setting for files and directories to determine whether any unanticipated access results or allowable access is prevented.

  Result: No unexpected behavior was found and all expected behavior occurred.

- *Audit trail tests:* These tests generate examples of each auditable type of event to determine whether proper audit trails are produced.

  Result: No unexpected behavior was found and all expected behavior occurred.

4.7 Server Management

It is vital that the operating environment be properly configured if protection is to be effective. Although this particular daemon appears to be fairly well behaved even in misconfigured environments, it would be useful to augment it with checking software to verify configurations and to allow configuration testing as a regular part of operational management. Simple testing programs to verify the W3 configuration have been demonstrated, but the mere existence of these auditing tools doesn't assure their proper use or that server errors not detected by these tools won't cause problems.

4.8 Extensions

Once the secure W3 daemon was in place, it was clear that the functions of W3 and another distributed network database service called *gopher* are quite similar. We decided it would be worthwhile to create a secure gopher daemon as well by reusing the code from the secure W3 server and altering the functions to meet *gopher* daemon standards. The result is the secure gopher daemon included in the second appendix. Although the gopher daemon has not been subjected to the same scrutiny as the W3 daemon, it soon will be. Other similar daemons are likely to follow.

5 Summary, Conclusions, and Further Work

In this paper, we have discussed design principles for secure daemons and demonstrated a secure *http* daemon which uses these principles. It appears from the limited experience with this daemon that it is quite effective in terms of providing integrity, availability, and privacy even in environments not otherwise properly configured to assure those properties. But there is still a great deal of work to be done.

5.1 Secure Execution Environments

The secure daemons discussed in this paper provide protection by, among other things, limiting the function of the daemon and its environment. This greatly simplifies protection in that the protection properties demonstrated for the daemon don't have to be demonstrated for all of the information in the environment in which it resides.

Unfortunately, in today's networking environment, servers are increasingly being called upon to perform services above and beyond the simple retrieval of information. Today, these services include database search and retrieval, filling out of information sheets, remote testing, and other similar processing services. In the coming months and years, these services will come to include electronic payment functions, remote execution of high-value computing functions, electronic voting and straw polls, and many other functions we cannot anticipate today. Clearly, in the coming environment, facilities for general purpose secure computing services will have to be seriously considered.

In the future, we hope to work further on developing improved test suites to verify that daemons such as this operate properly to as large an extent as possible. This should include performance testing and testing to assure that security properties remain intact when operating at or near performance limits, testing for boundary conditions on sizes, the use of obscure or obtuse input characters and sequences, verification of proper operations under all protection settings of files and directories, and other tests we have yet to devise.

5.2 Other Observations

Based on this work, we believe that the design of secure daemons is different than the design of other programs in several important ways:

- Many programmers view our design as *defensive* programming in that they believe that we have wasted time, effort, instructions, and space unnecessarily, and could write a much more efficient program. That is certainly true, but it is our belief that defensive programming is appropriate when the program is intended to defend against malicious attacks.
- Many programmers feel that the use of static variables is a step backwards and that it makes it harder to make certain that the program properly uses variables and doesn't improperly reuse the same variables. While it is true that the analysis would be far more complex for a large program, in the case of a small program, it is not hard to verify the use of all variables. On the other hand, avoiding allocation reduces the dependency on external operations and eliminates the need for resource sharing which can lead to deadlock.
- Many programs retain privileges in one process and create a second unprivileged process so that privileged operations can take place after much of the program has completed execution. We take the tactic that the design should be made to fit protection requirements of least privilege and that privileges should be abandoned at the earliest possible moment and never restored.
- Many programs take input as it becomes available, while the secure daemon delays input until it is necessary. This is done to reduce the interaction between inputs and the program with the hope of making confinement of untrusted data easy and

verifiable. Similarly, we believe that outputs should be provided as soon as they are available and not be unduely delayed. This prevents the possibility of corruption of outputs by interactions.

- We have clearly abandoned the *creeping featurism* of many daemons by providing only the features absolutely required in order for the server to operate. As a rule, we believe that secure servers must limit function to as large a degree as possible. While most programmers add functions to servers to support simpler design of clients, we believe that protecting servers means limiting what they do. As long as functions can be performed by the client, the server should not provide them.
- While many daemon designers implement their own protection features, we believe that well tested operating system protection features should be exploited to as large a degree as possible and that special-purpose daemon protection features should be avoided.
- While many daemon providers try to combine many functions into each daemon, we believe it is more secure to separate functions into separate daemons and to minimize their interactions.

5.3 Formal Analysis

The analysis provided in this paper is informal. In the near future, we hope to create formal methods to verify that the security properties discussed earlier may be proven to be properties of this daemon. We hope to publish results of this work in the near future and to apply these results to other daemons developed as a part of this work.

5.4 Finally

In Closing: We believe that this very compact and specially designed server is far more secure against corruption of the server, denial of services to clients, and unauthorized dissemination of information from the server than the standard daemons available today. It is also easy to use and easy to securely manage.

It is our sincere hope that future daemon designers perform similar analysis on all of their programs to assure that they meet the stringent requirements of protection in the modern computing environment.

A The Secure W3 Daemon

```
/* (c) 1995, Management Analytics (all.net) - ALL RIGHTS RESERVED */
#define ERRORLINE "The requested document has moved to
                     <A HREF=http://all.net/>here</A>.<P>\n"
#define REDIRECT "Location: http://all.net/\n"
#define WWWUID 101
#define WWWDIR "/u/www/htdocs"
#define WWWDefFile "/index.html"
#define WWWlog "/log"
#define Ktime 10
#define IOtime 15
int CHECKUSER=1;int DOCHROOT=1;

#define BUFSIZE 4096
#define MAXSIZE 2048
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <syslog.h>
#define LOG2(x,y) {F=fopen(WWWlog,"a+");
            if (F != NULL) {logfile(F);fprintf(F,x,y);} fclose(F);}
#define LOG3(x,y,z) {F=fopen(WWWlog,"a+");
            if (F != NULL) {logfile(F);fprintf(F,x,y,z);} fclose(F);}
#define LOG4(x,y,z,w) {F=fopen(WWWlog,"a+");
            if (F != NULL) {logfile(F);fprintf(F,x,y,z,w);} fclose(F);}

char line[BUFSIZE],name[BUFSIZE],bs1[BUFSIZE],bs2[BUFSIZE],bs3[BUFSIZE], timestamp[64],
logline[BUFSIZE], remotehost[BUFSIZE], remoteuser[BUFSIZE];
struct stat buf;
time_t *tloc;time_t t;
int staterr,i,n;FILE *F;

void logfile(F)
FILE *F;
{alarm(IOtime);t=time(NULL);strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));
fprintf(F,"%s %s %s ",remotehost,remoteuser,timestamp);alarm(0);}

error(s) /* simulate a 302 - document moved */
```

```
char *s;
{printf("HTTP/1.0 302 Found\n");printf("Server: ManAl/0.1\n");
printf("MIME-version: 1.0\n");printf(REDIRECT);printf("Content-type: text/html\n");
printf("<HEAD><TITLE>Document moved</TITLE></HEAD>\n");
printf("<BODY><H1>Document moved</H1>\n");printf(ERRORLINE);printf("(%s) </BODY>\n",s);
LOG4("Error:%s - %s %s\n",s,bs1,name);exit();}

void cat(s)
char s[];
{alarm(IOtime);i=open(s,0); while ((n=read(i,bs2,MAXSIZE)) > 0)
{alarm(IOtime);write(1,bs2,n);}close(i);alarm(0);}

fetch() /* if www owns it, it can be put - else, forget it */
{alarm(IOtime);staterr=stat(name,&(buf));alarm(0);
if (staterr != 0) error("Can't stat file"); /* can't stat the file - die */
if (0 == S_ISREG(buf.st_mode)) error("Can't fetch directories");
if (CHECKUSER==1) if (buf.st_uid != geteuid()) error("Not owner of file");
if (0 != (S_IROTH & buf.st_mode)) {cat(name); LOG2("cat %s\n",name);exit();} /* Send it*/
error("Access Denied");}

main(argc,argv,envp)
int argc; char *argv[],*envp[];
{alarm(Ktime);if (0 != chdir(WWWDIR))error("Cannot change to WWW directory");
if (DOCHROOT == 1) if (chroot(".") != 0)  error("Cannot change root directory to .");
if (0 != setuid(WWWUID)) error("setUID failed"); /* become user www or die */
if (argc>1) strncpy(remotehost,argv[1],MAXSIZE); else strcpy(remotehost,"nowhere");
if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE); else strcpy(remoteuser,"nobody");
remotehost[MAXSIZE]='\0';remoteuser[MAXSIZE]='\0';alarm(IOtime);
read(0,line,MAXSIZE); line[MAXSIZE]='\0'; sscanf(line, "%s %s %s", bs1, name, bs2);
if ((name[0] != '\0') && (name[strlen(name)-1] == '\r')) name[strlen(name)-1]='\0';
if ((name[0]=='/') && ((name[1]=='\0') || (name[1]==' '))) strcpy(name,WWWDefFile);
if (DOCHROOT!=1) {strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);}
alarm(0);if (strncasecmp(bs1,"get",5) == 0) fetch(); /* get */
error("Unknown request"); /* all other requests fail */}
```

B The Secure Gopher Daemon

```
/* (c) 1995, Management Analytics (all.net) - ALL RIGHTS RESERVED */
#define ERRORLINE "0’%s - %s’ does not exist                error.host      1\n.\n"
#define WWWUID 101
#define WWWDIR "/u/www/gopher"
#define WWWlog "/log"
#define SERVERNAME "all.net"
#define PORT "70"
#define Ktime 10
#define IOtime 15
int CHECKUSER=1;int DOCHROOT=1;int i;

#define BUFSIZE 4096
#define MAXSIZE 2048
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <syslog.h>
#include <dirent.h>
#define LOG2(x,y) {F=fopen(WWWlog,"a+");if (F != NULL)
{logfile(F);fprintf(F,x,y);} fclose(F);}
#define LOG3(x,y,z) {F=fopen(WWWlog,"a+");if (F != NULL)
{logfile(F);fprintf(F,x,y,z);} fclose(F);}
#define LOG4(x,y,z,w) {F=fopen(WWWlog,"a+");if (F != NULL)
{logfile(F);fprintf(F,x,y,z,w);} fclose(F);}

char line[BUFSIZE],name[BUFSIZE],bs1[BUFSIZE],bs2[BUFSIZE],bs3[BUFSIZE],
bs4[BUFSIZE],timestamp[64], logline[BUFSIZE], tmplog[BUFSIZE],
remotehost[BUFSIZE], remoteuser[BUFSIZE];
struct stat buf;
int i,n,staterr;FILE *F;
time_t *tloc;time_t t;
DIR *dirp;struct dirent *dp;

void logfile(F)
FILE *F;
{alarm(IOtime);t=time(NULL);
strftime(timestamp, 20, "%Y/%m/%d %T", localtime(&t));
```

```
sprintf(tmplog,"%s %s %s ",remotehost,remoteuser,timestamp);
fprintf(F,"%s",tmplog);alarm(0);
}

error(s) /* error return */
char *s;
{alarm(IOtime);printf(ERRORLINE,name,s);alarm(IOtime);
LOG3("GError:%s - %s\n",s,name);exit();}

void printdir(s)
char *s;
{alarm(IOtime);dirp = opendir(s);if (dirp==NULL) error("No such directory");
for (dp = readdir(dirp); dp != NULL; dp = readdir(dirp))
{alarm(IOtime);strcpy(bs3,dp->d_name);
strcpy(bs4,s);if (s[strlen(s)-1] != '/') strcat(bs4,"/");strcat(bs4,bs3);
if (bs3[0]!='.')
{staterr=stat(bs4,&(buf));
if ((staterr == 0) && (buf.st_mode & S_IFDIR))
printf("1%s\t%s\t%s\t%s\n",bs3,bs4,SERVERNAME,PORT);
else printf("0%s\t%s\t%s\t%s\n",bs3,bs4,SERVERNAME,PORT);
} }
alarm(IOtime);printf(".\n");closedir(dirp);alarm(IOtime);LOG2("ls %s\n",name);exit();
}

void cat(s)
char s[];
{alarm(IOtime);i=open(s,0); while ((n=read(i,bs2,MAXSIZE)) > 0)
{alarm(IOtime);write(1,bs2,n);}close(i);alarm(0);}

fetch() /* if www owns it, it can be put - else, forget it */
{alarm(IOtime);staterr=stat(name,&(buf)); alarm(0);
if (staterr != 0) error("Can't stat file"); /* can't stat the file - die */
if (buf.st_mode & S_IFDIR) printdir(name); /* list it */
if (0 == S_ISREG(buf.st_mode)) error("Can't fetch this kind of entity");
if (CHECKUSER==1) if (buf.st_uid != geteuid()) error("Not owner of file");
if (0 != (S_IROTH & buf.st_mode)) {cat(name); LOG2("cat %s\n",name);exit();} /* Send it*/
error("Access Denied");}

main(argc,argv,envp)
int argc; char *argv[],*envp[];
{alarm(Ktime);if (0 != chdir(WWWDIR))error("Cannot change to WWW directory");
if (DOCHROOT == 1) if (chroot(".") != 0)  error("Cannot change root directory to .");
if (0 != setuid(WWWUID)) error("setUID failed"); /* become user www or die */
if (argc>1) strncpy(remotehost,argv[1],MAXSIZE); else strcpy(remotehost,"nowhere");
if (argc>2) strncpy(remoteuser,argv[2],MAXSIZE); else strcpy(remoteuser,"nobody");
remotehost[MAXSIZE]='\0';remoteuser[MAXSIZE]='\0';alarm(IOtime);
read(0,name,MAXSIZE);name[MAXSIZE]='\0';
for (i=0;((i<MAXSIZE) && (name[i] != '\t') && (name[i] != '\0') &&
(name[i] != '\n') && (name[i] != '\r'));i++); name[i]='\0';
if (((name[0] == '1') || (name[0] == '0') ||  (name[0] == 'm')) && (name[1] == '/'))
{strcpy(bs2,&(name[1]));strcpy(name,bs2);} /* eliminate leading 1 or 0*/
if (DOCHROOT!=1) {strcpy(bs3,WWWDIR);strcat(bs3,name);strcpy(name,bs3);}
alarm(0);fetch(); exit();}
```

References

[1] F. Cohen, “Protection and Administration of Information Networks Under Partial Orderings”, IFIP-TC11 Computers and Security, V6(1987) pp118-128.

[2] Wietse Venema (wietse@wzv.win.tue.nl), On-line software p[rogram available from the author, Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[3] Network Working Group, T. Berners-Lee, “Universal Resource Identifiers in WWW - A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web”, Internet Request for Comments: 1630, June 1994

[4] F. Cohen, “Defense-In-Depth Against Computer Viruses”, IFIP-TC11 “Computers and Security”, V11#2?, 1992.

[5] B. W. Lampson. “A Note on the Confinement Problem”, Communications of the ACM V16(#10) October 1973:613–615.