Auditing a web server - Hobbit's ``webs'' [1]

Jeff Schaller

July 24, 2001

Contents

- Contents
- Introduction
    - Motivation
    - Overview
- Software Auditing
    - Source code auditing tools
        - GCC
        - ITS4
        - PScan
        - Scanner
        - Qaudit
        - RATS
        - Flawfinder
        - Stanford Checker
    - Black box testing
        - Bfbtester
        - Fuzz - Woodard
        - Fuzz - Maxwell
        - Fuzz - Miller
    - Known Exploits
        - Jill
        - Webscan
        - Nessus
        - Whisker
- Audit Results for ``webs''
    - Auditing tool results
        - GCC
        - ITS4
        - PScan
        - Scanner
        - Qaudit
        - RATS
        - Flawfinder
        - Bfbtester
        - Fuzz - Woodard
        - Fuzz - Maxwell
        - Fuzz - Miller
        - Jill
        - Webscan
        - Nessus
        - Whisker
    - Audit Evaluation
        - Is it enough?
    - Future work
- Patch for webs.c
- Embedded web servers
- Bibliography

Introduction

Motivation

Web servers are huge. Many people equate the Internet with websites, and for good reason - as I write this, the Google[1] search engine reports a database of over 1.3 billion pages and Netcraft[2] reports over 29 million web sites. Web servers are also being found in a growing number of embedded devices[2]. The most popular web servers today are Apache, Microsoft's IIS, and Netscape/iPlanet. All of them are powerful and extensible programs. At the other end of the spectrum are numerous special-purpose web servers, written to fulfill special requirements. These requirements may involve space, speed, or security.

I've become interested in special-purpose web servers while building a customized personal firewall. My requirements for the web server are:

Size
: The entire operating system should reside on one 1.44 megabyte floppy disk. A minimal kernel and filesystem leaves less than 500 kilobytes for extra programs.

Source code
: The code for the web server must be available for me - and others - to review for possible bugs.

Security
: A minimum of known or possible bugs in the code.

Simplicity
: The web server only needs to serve static files (HTML and graphics).

The speed at which the web server sends files is not a concern; about half of my target audience will be requesting the files over a dialup connection. The firewall and web server will reside on an Intel 386 desktop, which will most likely provide most of the performance bottlenecks. After gathering several small HTTP daemons, I discovered Hobbit's ``webs''[3]. It caught my eye for several reasons:

- Hobbit uses it to serve his content.
- It redirects initial requests to an unprivileged port for the remainder of the requests, which caught me off-guard.
- The comments at the top of the program claim security (safety versus HTTPS) as a goal.
- The code resides in two small files: webs.c (18k) and generic.h (11k)

This article focuses on ``webs'', but the tools and techniques introduced can and should be applied to other web servers.

Overview

There exist several general web server security guidelines:

- Lincoln Stein's WWW Security FAQ[4] covers both server and client security concerns.
- Cheswick and Bellovin's Firewalls book[5, Section 2.8.1] includes several aspects of web services that are dangerous.
- Practical Unix and Internet Security[6, Chapter 18], like Stein's FAQ, covers general aspects of server-side security concerns.
- A web server should reside on a secure operating system; many guidelines exist for building them. Students of SANS courses have done work in this area (`http://www.sans.org/giactc/cert.htm`), and Julia Allen's book[7, Chapters 2 and 3] is an excellent combination of CERT's experience.
- The Web Security Sourcebook[8] includes chapters on server-side web security, and has several good concepts.

Most of these guidelines assume that the web server itself is well-written. They instead focus on other aspects, such as: securing the operating system, writing security policies, being careful about server-side execution of programs, and configuring the web server appropriately. These are all important concerns, but with the increasing demand for features comes larger and larger code bases for web servers. The WWW Security FAQ explains this phenomenon as: buggy software results in security holes; large software inevitably has some bugs; web servers are large pieces of software. As a result, I believe that more effort needs to be spent in ensuring that web servers are thoroughly audited. One notable exception to the above guidelines has appeared recently. This article, written by Mixter, provides specific suggestions for auditing C programs: `http://mixter.void.ru/vulns.html`.

I found examples of two different types of auditing: source code checking and black box testing. Source code checking can be manual or automatic; experienced programmers have built up guidelines for robust ways to perform various functions. Peer review, code review meetings, and source code checking tools incorporate this experience. Black box testing can occur once the program is functional; various test cases are sent to the program and the results are observed. Test cases may be carefully crafted to exploit known weaknesses, be examples of common usage (``beta testing"), may test protocol conformance, or send random input (``fuzz testing").

Software Auditing

Software auditing can occur at two times: before and after release to the public. Two different groups of people can conduct the audit, depending on when it takes place. Before release, the programmer himself is the primary auditor. After release, the programmer may continue to audit the software, but now the outside world also has a chance. There is another dimension of auditing which differentiates the type of testing that can be done after release: whether the source code for the program is available (``open source"). Closed source programs allow only black box testing, whereas open source programs allow source code review and auditing tools as well as black box testing.

For example, a program might use the ``gets" function to read user input. Common knowledge has it that the ``gets" function is dangerous to use, since it does not limit the amount of input that will be assigned to the input variable. The gcc compiler will issues a warning if a program uses the function. With a closed source program, black box testing may never exceed the finite buffer assumed by the author of the program. However, with an open source program, even a simple tool such as ``grep" will uncover the unsafe function call.

The following sections cover source code auditing, black box testing, and the related tools.

Source code auditing tools

These tools inspect the source code of a given program for dangerous activity. Dangerous activity could include certain function calls, function call parameters, reliance on user input, lack of error checking, and improper use of variables, among other things. All of these activities have been shown to allow a compromise in security when exposed to hostile environments. A source code auditing tool will flag these dangerous sections of code and possibly provide suggestions for improvement.

By their very definition, source code auditing tools are both limited in scope and reactive. They are limited by their own author's knowledge of potentially dangerous activity in a certain language, and they are reactive because they have to be run after the bulk of the coding has taken place. There are many source code auditing tools available and presumably many more that are not publicly available. Tools which perform similar checks should be combined into fewer, more effective tools; this is already happening with two of the projects. Another deficiency is false positives; these tools often have very simple logic for detecting dangerous activity. The language that they are inspecting may be too complex to allow for more precise checks or external prerequisites must be met. Unfortunately, false positives dilute the effectiveness of the tool.

There are also several subjective areas that source code auditing tools must make assumptions about. The environment of the program can have a large impact on its expectations and use - a program that passes the audit may end up residing on an insecure operating system. A program that fails the input validation tests may be judged acceptable in an environment where the input is known to be safe. After all of the source code auditing tools have been run, there is an additional question: how good is the code?

The quality of a certain program is almost certainly a subjective question, but the elements of bad coding are easy to spot: inconsistent coding style such as indentation, variable names, etc; hard-to-read formatting; inappropriate commenting; and failure to check return codes from critical functions, to name a few. More guidelines can be found in the Appendix of Kernighan and Pike's book[9]. All of these elements detract from the maintainability of the code, which means that the next person to make changes to it - possibly the original author - is more likely to make mistakes. Source code auditing tools have a clear advantage over subjective tests: they give clear indication whether the code passes or not. All of the tools listed below will complain loudly if they find dangerous sections of code; they give little to no output when given safe code.

Other source code auditing techniques are less precise, but valuable nonetheless. Peer review and public inspection have two benefits. The first benefit occurs before the code is released - the author typically wants to maintain a good reputation, and so is encouraged to do a good job before presenting the program. The second benefit comes from having other people, with their differing environments, backgrounds, and experiences review the code. Once the code is functional, having people try to use the code in their environment often exposes it to new conditions and expectations. These new cases can illuminate errors in the original program, resulting in revisions that make the program more robust. Other programmers may be aware of data structures or algorithms that are more efficient.

GCC

GCC[10] is produced by the EGCS Steering Committee. GCC is now an acronym for the ``GNU Compiler Collection", and it is able to compile several high level languages, including C, C++, and Fortran. There are many options to influence GCC's behavior; the following are of interest to the security community:

-Wall
: Combines several other warning options that look for dangerous usage.

-Wconversion
: Emit a warning if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype.

-Wstrict-prototypes
: Emit a warning if a function is declared or defined without specifying the argument types.

-Wmissing-prototypes
: Emit a warning if a global function is defined without a previous prototype declaration.

-Wmissing-declarations
: Emit a warning if a global function is defined without a previous declaration.

ITS4

Written by John Viega, ITS4[11] scans C and C++ source code for potential security vulnerabilities. For dangerous function calls, ITS4 outputs a short description of the problem and suggestions on how to fix the code.

PScan

Alan DeKok created PScan[12], a limited problem scanner for C source files. PScan scans C source files for printf-style functions; these functions are susceptible to format string bugs. PScan does not check for buffer overflow errors. It works by scanning for specific function calls with a formatting parameter that is not a static string. Any such problematic calls are flagged with their line number and function name.

Scanner

Antonomasia (ant@notatla.demon.co.uk) wrote Scanner[13], which analyzes C source code files for file race conditions. It has a list of file-related functions and identifiers which indicate whether the parameter(s) to the function are being checked or being used. Parameters that are used before they are checked could be subject to race conditions. It flags any unsafe file-related operations with the function name and line number.

Qaudit

Qaudit[14] was written by v9@fakehalo.org. It checks C and C++ code for possible buffer overflows, format bugs, environment variable usage, and execution of other programs. Possible problems are reported by listing the filename, type of check, line number, and the text of the offending line.

RATS

The RATS team at Secure Software Solutions wrote the Rough Auditing Tool for Security[15]. It works by scanning source code for potentially dangerous function calls; it is provided as a starting point for a security audit. It uses an XML file of vulnerable function calls to report dangerous usage by listing the line number and severity. Each error is followed by a short description of the possible problem and suggestions for improvement.

Flawfinder

David A. Wheeler wrote Flawfinder[16]; it examines C and C++ source code for security weaknesses. It has a database of functions and patterns that are common causes of security flaws. Flawfinder reports any flaws found in order of decreasing severity.

Stanford Checker

A group of students at Stanford are working on a Meta-level Compilation project[17]. The goal of the project is to allow others to build domain- and application-specific extensions to check, optimize, and transform code. They have used these tools to audit several systems, including Linux. They have a web page listing errors found in the Linux kernel at http://hands.stanford.edu/linux/. As of this writing, the tools are not publicly available.

Black box testing

Black box testing can be very valuable - it insulates the tester from detailed knowledge about the program's design. This environment forces the tester to be both general and specific: general in the sense that they do not know what assumptions and limits the programmer put into the code. They must also be specific in their tests to trigger certain behavior in the program. Black box testing can be separated into two types: random input (``fuzz testing") and typical usage (``beta testing"). Fuzz testing involves sending random inputs to the program and then watching for failure. Beta testing involves putting the program in front of typical users and asking them to try it out. Both methods have their benefits: fuzz testing is simple to construct, execute, and evaluate. Beta testing is useful once the program is mostly functional; users will try the out the functionality that they expect, using the expected protocols. This helps programmers understand which features people use; if there is an aspect of the program that is unused, it's not as important.

Fuzz testing has certain limitations: by its nature it is not protocol-aware. Random inputs to a protocol-driven program will typically not cause it to do anything useful; the program may have very specific requirements for the order and characteristics of the input. A web server will respond with an error to any requests that do not comply with HTTP protocol, for example. Fuzz testing's simple nature results in a vastly simplified evaluation criteria, however: did the program crash or not? If the program did not crash, did it behave as expected? That is, did it consume a reasonable amount of resources - CPU time, memory, and disk space? Did it hang or stop responding?

Fuzz testing, for all of its simplicity, has turned up surprising results. The original ``fuzz" study[18] found bugs in over 24% of the standard Unix utilities. A later study[19] done on a Windows system found that more than 21% of the programs crashed or hung. Theo DeRaadt of the OpenBSD project reports the failure of several programs in a default OpenBSD install in this article: http://lwn.net/2000/0803/a/openbsdfuzz.php3.

Bfbtester

Bfbtester[20] is able to test a program's usage of command-line arguments and environment variables. Also included is code that checks for a program's temporary file usage, which could be exploited by a local user. Bfbtester will report if the program crashed, and if so, with what arguments or environment variables.

Fuzz - Woodard

This fuzz[21] project, led by Ben Woodard, tests assumptions made by the program about the size or format of its input stream. The purpose of the project is to test common Linux utilities for bugs.

Fuzz - Maxwell

Scott Maxwell's fuzz[22] program is part of his ``Bulletproof Penguin" site, which aims to apply the original fuzz project's goals to Linux. Included on the site are patches to programs that were found to have bugs.

Fuzz - Miller

Written in 1989, the fuzz generator[23] is the original fuzz program. It was written by Lars Fredriksen, Bryan So, and Barton Miller to test the robustness of system utilities. The fuzz program generates random characters, which are then fed to target programs. If the target program crashes, the input and output streams are saved so that they can be reviewed.

Known Exploits

Jill

Jill is a C program that attempts to overflow an IIS 5.0 buffer, resulting in a remote command shell. The instructions say to set up a netcat[24] listener on a certain port; if the overflow is successful, the IIS server will create a command-shell connection to that port. Jill can be downloaded from http://packetstormsecurity.org/filedesc/jill.c.html.

Webscan

Webscan is a C program that scans a list of IP addresses for a large list of CGI vulnerabilities. Vulnerable instances are reported in an output file, ``vuln.log''. Webscan is available from http://mixter.void.ru/findex.html.

Nessus

Nessus is a free, powerful, open source security scanner with plugin capabilities. It uses a client/server architecture to scan and report on systems and services. Nessus provides detailed reports in various formats, grouped by system and service. Plugins are constantly being created to scan for new vulnerabilities; many plugins exist to check for web server vulnerabilities. Nessus can be downloaded from http://www.nessus.org/.

Whisker

Whisker was written by rfp (.rain.forest.puppy) to combine known CGI vulnerabilities with intelligent ordering, server information recognition, and IDS avoidance. Whisker uses a perl module, libwhisker, which can be used as the engine for future CGI scanners.

Audit Results for ``webs''

Auditing tool results

GCC

```
$ gcc -DLINUX -Wall -Wconversion -Wstrict-prototypes -Wmissing-prototypes \
-Wmissing-declarations -o webs webs.c
webs.c:118: warning: function declaration isn't a prototype
webs.c: In function `bail':
webs.c:126: warning: implicit declaration of function `sleep'
webs.c: At top level:
webs.c:138: warning: function declaration isn't a prototype
webs.c: In function `tmtravel':
webs.c:140: warning: implicit declaration of function `alarm'
webs.c: At top level:
webs.c:149: warning: function declaration isn't a prototype
webs.c:189: warning: function declaration isn't a prototype
webs.c: In function `fgetss':
webs.c:198: warning: passing arg 2 of `fgets' as signed due to prototype
webs.c: At top level:
webs.c:223: warning: function declaration isn't a prototype
webs.c: In function `parseurl':
webs.c:228: warning: unused variable `s2'
webs.c:228: warning: unused variable `s1'
webs.c: At top level:
webs.c:283: warning: function declaration isn't a prototype
webs.c:307: warning: function declaration isn't a prototype
webs.c: In function `handfile':
webs.c:337: warning: implicit declaration of function `read'
webs.c:343: warning: implicit declaration of function `write'
webs.c:362: warning: implicit declaration of function `close'
webs.c: At top level:
webs.c:375: warning: return-type defaults to `int'
webs.c:375: warning: function declaration isn't a prototype
webs.c: In function `main':
webs.c:438: warning: implicit declaration of function `chroot'
webs.c:441: warning: implicit declaration of function `chdir'
webs.c:445: warning: implicit declaration of function `setgid'
webs.c:446: warning: implicit declaration of function `setuid'
webs.c:448: warning: implicit declaration of function `getuid'
webs.c:450: warning: implicit declaration of function `geteuid'
webs.c:474: warning: passing arg 1 of `ntohs' with different width due \
to prototype
```

ITS4

```
$ ./its4 ../../webs099/webs.c
../../webs099/webs.c:105:(Urgent) fprintf
../../webs099/webs.c:106:(Urgent) fprintf
../../webs099/webs.c:107:(Urgent) fprintf
../../webs099/webs.c:124:(Urgent) fprintf
../../webs099/webs.c:288:(Urgent) fprintf
../../webs099/webs.c:289:(Urgent) fprintf
../../webs099/webs.c:292:(Urgent) fprintf
../../webs099/webs.c:571:(Urgent) fprintf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
../../webs099/webs.c:518:(Urgent) printf
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
../../webs099/webs.c:129:(Urgent) syslog
../../webs099/webs.c:132:(Urgent) syslog
../../webs099/webs.c:298:(Urgent) syslog
../../webs099/webs.c:367:(Urgent) syslog
../../webs099/webs.c:369:(Urgent) syslog
../../webs099/webs.c:485:(Urgent) syslog
../../webs099/webs.c:578:(Urgent) syslog
../../webs099/webs.c:604:(Urgent) syslog
Non-constant format strings can often be attacked.
Use a constant format string.
----------------
../../webs099/webs.c:418:(Very Risky) sprintf
../../webs099/webs.c:430:(Very Risky) sprintf
This function is high risk for buffer overflows
Use snprintf if available, or precision specifiers, if available.
----------------
../../webs099/webs.c:275:(Very Risky) strcat
../../webs099/webs.c:426:(Very Risky) strcat
This function is high risk for buffer overflows
Use strncat instead.
----------------
../../webs099/webs.c:242:(Very Risky) strcpy
This function is high risk for buffer overflows
Use strncpy instead.
----------------
../../webs099/webs.c:441:(Risky) chdir
Can lead to process/file interaction race conditions (TOCTOU problems)
Manipulate file descriptors, not symbolic names, when possible.
----------------
../../webs099/webs.c:438:(Risky) chroot
Don't forget to chdir() first!  Also, can lead to process/file interaction\
race conditions (TOCTOU category A)
Manipulate file descriptors, not symbolic names, when possible.
----------------
../../webs099/webs.c:545:(Risky) fdopen
Can be involved in a race condition if you open things after a poor check. For
example, don't check to see if something is not a symbolic link before opening
it.  Open it, then check bt querying the resulting object.  Don't run tests on
symbolic file names...
Perform all checks AFTER the open, and based on the returned object, not a
symbolic name.
----------------
../../webs099/webs.c:315:(Risky) open
Can be involved in a race condition if you open things after a poor check. For
example, don't check to see if something is not a symbolic link before opening
it.  Open it, then check bt querying the resulting object.  Don't run tests on
symbolic file names...
Perform all checks AFTER the open, and based on the returned object, not a
symbolic name.
----------------
../../webs099/webs.c:401:(Risky) openlog
Can lead to process/file interaction race conditions (TOCTOU category B)
Manipulate file descriptors, not symbolic names, when possible.
----------------
```

```
../../webs099/webs.c:431:(Risky) stat
../../webs099/webs.c:589:(Risky) stat
Can lead to process/file interaction race conditions (TOCTOU category A)
Manipulate file descriptors, not symbolic names, when possible.
----------------
../../webs099/webs.c:337:(Some risk) read
Be careful not to introduce a buffer overflow when using in a loop.
Make sure to check your buffer boundries.
----------------
```

PScan

```
$ ./pscan -w ~/proj/giac/gsna/webs/webs.c
/home/schaller/proj/giac/gsna/webs/webs.c:106 Warning: fprintf uses \
non-constant string for format argument 1.
/home/schaller/proj/giac/gsna/webs/webs.c:129 Warning: syslog uses \
non-constant string for format argument 1.
```

Scanner

```
$ ./scancode.plx -v -E ~/proj/giac/gsna/webs/webs.c
```

Qaudit

```
$ ./qaudit.pl ../../webs099/webs.c
qaudit.pl::q(uick)audit: version[02].  by: vade79[v9@fakehalo.org].
-----------------------------------------------------------------
-----------------------------------------------------------------
webs.c:*NOTICE:START: entering format bug check mode.
webs.c:*NOTICE:STOP: exiting format bug check mode.
webs.c:*NOTICE:START: entering bounds check mode.
webs.c:*NOTICE:INFO: character array(s) to scan: unknown webpath webidx bbuf \
fbuf aclient redir aflgs.
webs.c:BOF_CHK:75:unknown: static char unknown[] = "(UNKNOWN)";
webs.c:BOF_CHK:426:unknown: strcat (bbuf, unknown);
webs.c:BOF_CHK:76:webpath: static char webpath[] = WEB_PATH;
webs.c:BOF_CHK:242:webpath: strcpy (q, webpath);
webs.c:BOF_CHK:430:webpath: sprintf (fbuf, "%s/%s", webpath, webidx);
webs.c:BOF_CHK:438:webpath: x = chroot (webpath);
webs.c:BOF_CHK:491:webpath: x = sizeof (fbuf) - sizeof (webpath) - 2;   \
/* limit input length... */
webs.c:BOF_CHK:509:webpath: pp = fbuf + strlen (webpath);               \
/* "/u/web/blah" -> "/blah" */
webs.c:BOF_CHK:552:webpath: x = sizeof (fbuf) - sizeof (webpath) - 2;
webs.c:BOF_CHK:77:webidx: static char webidx[] = WEB_IDX;
webs.c:BOF_CHK:275:webidx: strcat (out, webidx);
webs.c:BOF_CHK:430:webidx: sprintf (fbuf, "%s/%s", webpath, webidx);
webs.c:BOF_CHK:78:bbuf: static char bbuf [BSIZE];       \
/* general-purpose big buffer */
webs.c:BOF_CHK:337:bbuf: x = read (f, bbuf, BSIZE);             \
/* just a simple lockstep xfer */
webs.c:BOF_CHK:341:bbuf: p = bbuf;
webs.c:BOF_CHK:414:bbuf: memset (bbuf, 0, BSIZE);
webs.c:BOF_CHK:417:bbuf: pp = bbuf;
webs.c:BOF_CHK:418:bbuf: sprintf (bbuf, "%s ", aclient);
webs.c:BOF_CHK:419:bbuf: pp += strlen (bbuf);
webs.c:BOF_CHK:424:bbuf: strncat (bbuf, hp->h_name, 128);
webs.c:BOF_CHK:426:bbuf: strcat (bbuf, unknown);
webs.c:BOF_CHK:485:bbuf: syslog (LOG_INFO, "%s given port %d", bbuf, lp);
webs.c:BOF_CHK:494:bbuf: pp = fgetss (bbuf, x, stdin);
webs.c:BOF_CHK:502:bbuf: pp = parseurl (bbuf, fbuf);
webs.c:BOF_CHK:504:bbuf: bogusurl ("first hit", bbuf);
webs.c:BOF_CHK:555:bbuf: pp = fgetss (bbuf, x, chan);
webs.c:BOF_CHK:559:bbuf: bogusurl ("input", bbuf);
webs.c:BOF_CHK:578:bbuf: lp, inet_ntoa (remend.sin_addr), bbuf);
webs.c:BOF_CHK:584:bbuf: pp = parseurl (bbuf, fbuf);
webs.c:BOF_CHK:586:bbuf: bogusurl ("parse", bbuf);
webs.c:BOF_CHK:79:fbuf: static char fbuf [256];         /* filename buffer */
webs.c:BOF_CHK:413:fbuf: memset (fbuf, 0, sizeof (fbuf));
webs.c:BOF_CHK:430:fbuf: sprintf (fbuf, "%s/%s", webpath, webidx);
webs.c:BOF_CHK:431:fbuf: x = stat (fbuf, &dafile);
webs.c:BOF_CHK:491:fbuf: x = sizeof (fbuf) - sizeof (webpath) - 2;      \
/* limit input length... */
webs.c:BOF_CHK:502:fbuf: pp = parseurl (bbuf, fbuf);
webs.c:BOF_CHK:508:fbuf: /* fbuf comes back with full web-tree-path.  Skip our \
private bits ... */
webs.c:BOF_CHK:509:fbuf: pp = fbuf + strlen (webpath);          \
/* "/u/web/blah" -> "/blah" */
webs.c:BOF_CHK:512:fbuf: fbuf[0] = '/';
webs.c:BOF_CHK:513:fbuf: fbuf[1] = '\0';
webs.c:BOF_CHK:514:fbuf: pp = fbuf;
webs.c:BOF_CHK:552:fbuf: x = sizeof (fbuf) - sizeof (webpath) - 2;
webs.c:BOF_CHK:584:fbuf: pp = parseurl (bbuf, fbuf);
webs.c:BOF_CHK:85:aclient: char aclient [64];           \
/* original client ascii-addr */
webs.c:BOF_CHK:298:aclient: aclient, bocount, reason, spec);
webs.c:BOF_CHK:367:aclient: syslog (LOG_INFO, "%s %.250s", aclient, path);
webs.c:BOF_CHK:369:aclient: syslog (LOG_ERR, "%s %.250s I/O err %d", aclient, \
path, y);  /* nonfatal */
webs.c:BOF_CHK:410:aclient: memset (aclient, 0, sizeof (aclient));
webs.c:BOF_CHK:411:aclient: sprintf (aclient, "[%.32s]", \
inet_ntoa (remend.sin_addr));
webs.c:BOF_CHK:418:aclient: sprintf (bbuf, "%s ", aclient);
webs.c:BOF_CHK:604:aclient: syslog (LOG_INFO, "%s %d timeout", aclient, lp);
webs.c:BOF_CHK:7:redir: Reasonably comprehensive logging.  Wanna run scripts?\
Bury redirects
webs.c:BOF_CHK:12:redir: redirect to same.  All subsequent interaction with a \
given caller uses
webs.c:BOF_CHK:58:redir: #define FULL_REDIRECT 1                \
/* include orig req in redirect */
webs.c:BOF_CHK:92:redir: /* local-listener redirect.  We fill in \
"<addr>:<port>" etc on the fly */
webs.c:BOF_CHK:93:redir: static char redir[] =
webs.c:BOF_CHK:94:redir: "HTTP/1.0 302 Port redirect [valid 15 \
seconds]\r\nLocation: http://";
webs.c:BOF_CHK:453:redir: /* get a socket and make it available *before* \
we redirect */
webs.c:BOF_CHK:501:redir: /* Find the request-path and include it in the \
redirect */
webs.c:BOF_CHK:517:redir: /* hand out the redirect, and clean up */
webs.c:BOF_CHK:518:redir: printf ("%s%s:%d%s\r\n", redir, \
inet_ntoa (lclend.sin_addr), lp, pp);
webs.c:BOF_CHK:165:aflgs: static unsigned char aflgs[] = {
webs.c:BOF_CHK:182:aflgs: }; /* aflgs */
webs.c:BOF_CHK:206:aflgs: q = aflgs[q];                 /* convert thru array */
webs.c:*NOTICE:STOP: exiting bounds check mode.
webs.c:*NOTICE:START: entering exec check mode.
webs.c:*NOTICE:STOP: exiting exec check mode.
webs.c:*NOTICE:START: entering environmental variable check mode.
webs.c:*NOTICE:STOP: exiting environmental variable check mode.
webs.c:*NOTICE:START: entering miscellaneous check mode.
webs.c:MSC_CHK:188:WARNING: char * fgetss (buf, len, from)
webs.c:MSC_CHK:198:WARNING: p = fgets (buf, len, from);         \
/* returns ptr to buf */
webs.c:MSC_CHK:242:WARNING: strcpy (q, webpath);
webs.c:MSC_CHK:275:WARNING: strcat (out, webidx);
webs.c:MSC_CHK:411:WARNING: sprintf (aclient, "[%.32s]", \
inet_ntoa (remend.sin_addr));
webs.c:MSC_CHK:418:WARNING: sprintf (bbuf, "%s ", aclient);
webs.c:MSC_CHK:426:WARNING: strcat (bbuf, unknown);
webs.c:MSC_CHK:430:WARNING: sprintf (fbuf, "%s/%s", webpath, webidx);
webs.c:MSC_CHK:463:WARNING: x = getsockname (0, (SA *) &lclend, &y);    \
/* get local-end addr */
webs.c:MSC_CHK:465:WARNING: bail ("local getsock 0");
webs.c:MSC_CHK:471:WARNING: x = getsockname (netfd1, (SA *) &lclend, &y);               /* find its number */
webs.c:MSC_CHK:473:WARNING: bail ("local getsock nfd");
webs.c:MSC_CHK:494:WARNING: pp = fgetss (bbuf, x, stdin);
webs.c:MSC_CHK:555:WARNING: pp = fgetss (bbuf, x, chan);
webs.c:*NOTICE:STOP: exiting miscellaneous check mode.
--------------------------------------------------------------------
exiting qaudit.
```

RATS

```
$ ./rats -i -r -w 3 ~/proj/giac/gsna/webs/webs.c
/home/schaller/proj/giac/gsna/webs/webs.c:106: High: fprintf
Check to be sure that the non-constant format string passed as argument 2 to
this function call does not come from an untrusted source that could have added
formatting characters that the code is not prepared to handle.

/home/schaller/proj/giac/gsna/webs/webs.c:129: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:132: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:297: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:367: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:369: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:485: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:577: High: syslog
/home/schaller/proj/giac/gsna/webs/webs.c:604: High: syslog
Truncate all input strings to a reasonable length before
passing them to this function

/home/schaller/proj/giac/gsna/webs/webs.c:242: High: strcpy
Check to be sure that argument 2 passed to this function call will not more
data than can be handled, resulting in a buffer overflow.

/home/schaller/proj/giac/gsna/webs/webs.c:275: High: strcat
/home/schaller/proj/giac/gsna/webs/webs.c:426: High: strcat
Check to be sure that argument 2 passed to this function call will not more
data than can be handled, resulting in a buffer overflow.

/home/schaller/proj/giac/gsna/webs/webs.c:418: High: sprintf
/home/schaller/proj/giac/gsna/webs/webs.c:430: High: sprintf
Check to be sure that the format string passed as argument 2 to this function
call doest not come from an untrusted source that could have added formatting
characters that the code is not prepared to handle.  Additionally, the format
string could contain `%s' without precision that could result in a buffer
overflow.

/home/schaller/proj/giac/gsna/webs/webs.c:80: Medium: non-function call \
reference: stat
A function call is not being made here, but a reference is being made to a name
that is normally a vulnerable function.  It could be being assigned as a
pointer to function.

/home/schaller/proj/giac/gsna/webs/webs.c:337: Medium: read: stat
Check buffer boundaries if calling this function in a loop and make sure \
you are not in danger of writing past the allocated space.

/home/schaller/proj/giac/gsna/webs/webs.c:78: Low: fixed size global \
buffer: stat
/home/schaller/proj/giac/gsna/webs/webs.c:79: Low: fixed size global \
buffer: stat
/home/schaller/proj/giac/gsna/webs/webs.c:85: Low: fixed size global \
buffer: stat
Extra care should be taken to ensure that character arrays that are allocated
with a static size are used safely.  This appears to be a global allocation
and is less dangerous than a similar one on the stack.  Extra caution is still
advised, however.

/home/schaller/proj/giac/gsna/webs/webs.c:198: Low: fgets: stat
Double check that your buffer is as big as you specify

/home/schaller/proj/giac/gsna/webs/webs.c:409: Low: memcpy: stat
Double check that your buffer is as big as you specify

/home/schaller/proj/giac/gsna/webs/webs.c:438: Low: chroot: stat
Reminder: Do not forget to chdir() to an appropriate directory before calling \
chroot()!

/home/schaller/proj/giac/gsna/webs/webs.c:315: Low: open: stat
A potential race condition vulnerability exists here.  Normally a call to this
function is vulnerable only when a match check precedes it.  No check was
detected, however one could still exist that could not be detected.
```

```
/home/schaller/proj/giac/gsna/webs/webs.c:431: Low: stat: stat
A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists.  This is
the first line where a check has occured.  No matching uses were detected.

/home/schaller/proj/giac/gsna/webs/webs.c:589: Low: stat: stat
A potential TOCTOU (Time Of Check, Time Of Use) vulnerability exists.  This is
the first line where a check has occured.  No matching uses were detected.

/home/schaller/proj/giac/gsna/webs/webs.c: 198: fgets
Double check to be sure that all input accepted from an external data source
does not exceed the limits of the variable being used to hold it.  Also make
sure that the input cannot be used in such a manner as to alter your program's
behaviour in an undesirable way.
```

Flawfinder

```
$ ./flawfinder --context ~/proj/giac/gsna/webs/webs.c
Flawfinder version 0.15, (C) 2001 David A. Wheeler.
Number of dangerous functions in C ruleset: 40
Processing /home/schaller/proj/giac/gsna/webs/webs.c
/home/schaller/proj/giac/gsna/webs/webs.c:106 [4] (format) fprintf: if format \
strings can be influenced by an attacker, they can be exploited. Use a \
constant for the format specification.
  fprintf (stderr, str, p1, p2, p3, p4, p5, p6);
/home/schaller/proj/giac/gsna/webs/webs.c:242 [4] (buffer) strcpy: does not \
check for buffer overflows. Consider using strncpy or strlcpy.
  strcpy (q, webpath);
/home/schaller/proj/giac/gsna/webs/webs.c:275 [4] (buffer) strcat: does not \
check for buffer overflows. Consider using strncat or strlcat.
      strcat (out, webidx);
/home/schaller/proj/giac/gsna/webs/webs.c:418 [4] (buffer) sprintf: does not \
check for buffer overflows. Use snprintf or vsnprintf.
  sprintf (bbuf, "%s ", aclient);
/home/schaller/proj/giac/gsna/webs/webs.c:426 [4] (buffer) strcat: does not \
check for buffer overflows. Consider using strncat or strlcat.
    strcat (bbuf, unknown);
/home/schaller/proj/giac/gsna/webs/webs.c:430 [4] (buffer) sprintf: does not \
check for buffer overflows. Use snprintf or vsnprintf.
  sprintf (fbuf, "%s/%s", webpath, webidx);
/home/schaller/proj/giac/gsna/webs/webs.c:411 [2] (buffer) sprintf: does not \
check for buffer overflows. Use snprintf or vsnprintf. Risk is low because the \
source has a constant maximum length.
  sprintf (aclient, "[%.32s]", inet_ntoa (remend.sin_addr));
There are probably other security vulnerabilities as well; review your code!
```

Bfbtester

```
# -a = all tests
# -t = watch temp file usage
# -x 100 = start 100 copies at a time

$ ./bfbtester -a -t -x 100 ~/proj/giac/gsna/webs/webs
=> /home/schaller/proj/giac/gsna/webs/webs
   * Single argument testing
   * Multiple arguments testing
   * Environment variable testing
Cleaning up...might take a few seconds
```

Fuzz - Woodard

```
$ ./fuzz -r 5 nc localhost 80
Testing /usr/bin/nc done -- No faults found.

$ tail /var/log/messages
webs: [127.0.0.1] localhost given port 2322
webs: [127.0.0.1] strike 1: bogus first hit: .x#B##TH
webs: [127.0.0.1] localhost given port 2324
```

```
webs: [127.0.0.1] strike 1: bogus first hit: l##XW7n#y#5##lSjy7#dY\
y##n##F#5#m#n#rf##No7D_4
webs: [127.0.0.1] localhost given port 2326
webs: [127.0.0.1] strike 1: bogus first hit: #=#3yeL#Vn2WU#/P##uRC
webs: [127.0.0.1] localhost given port 2328
```

Fuzz - Maxwell

```
$ ./fuzz | nc localhost 80
<title>Get real!</title><h2>Not found</h2>
<p>That does not exist here.

$ tail /var/log/messages
webs: [127.0.0.1] localhost given port 2360
webs: [127.0.0.1] strike 1: bogus first hit: ##U#ul##g##I#-###qmEuB#x\
#1xUb8##Rp#F#k#Bv&##S#######=Hz#xs0Y#KU##dB#ObXE#dJZ
```

Fuzz - Miller

```
$ (echo -n "GET "; ./fuzz -0 -e '\n\n') | nc localhost 80
<title>Get real!</title><h2>Not found</h2>
<p>That does not exist here.

$ tail /var/log/messages
webs: [127.0.0.1] localhost given port 2366
webs: [127.0.0.1] strike 1: bogus first hit: GET -####qgy79###.j0=z#r#\
e4#UXN7Zlq#lt##e#u#GR0O
```

Jill

```
$ nc -l -p 8000 -v
$ ./jill localhost 80 localhost 8000
iis5 remote .printer overflow.
dark spyrit <dspyrit@beavuh.org> / beavuh labs.

connecting...
sent...
you may need to send a carriage on your listener if the shell doesn't appear.
have fun!
```

Webscan

```
$ cat ipfile
127.0.0.1

$ ./webscan ipfile
web scan 1.1 by Mixter
scanning from ipfile (pid: 17344)

$ sleep 200; cat status.log
Started new session. File: ipfile, PID: 17344
Finished session. File: ipfile

$ cat ver.log

$ cat vuln.log
```

Nessus

```
Nessus Scan Report
------------------
SUMMARY
```

```
 - Number of hosts which were alive during the test : 1
 - Number of security holes found : 0
 - Number of security warnings found : 1
 - Number of security notes found : 3

TESTED HOSTS

 localhost (Security warnings found)

DETAILS

+ localhost :
 . List of open ports :
   o ssh (22/tcp) (Security notes found)
   o www (80/tcp)
   o general/tcp (Security notes found)
   o general/udp (Security notes found)
   o unknown (3001/tcp) (Security warnings found)

 . Information found on port ssh (22/tcp)

    Remote SSH version :
     ssh-1.5-openssh_2.5.2p2


 . Information found on port general/tcp

    Nmap found that this host is running Linux 2.1.122 - 2.2.16

 . Information found on port general/udp

    For your information, here is the traceroute to 146.82.16.91 :
    146.82.16.91


 . Warning found on port unknown (3001/tcp)

    Nessus Daemon open on port TCP:3001, NessusD version:
     NTP/1.2

------------------------------------------------------
This file was generated by the Nessus Security Scanner
```

Whisker

```
$ ./whisker.pl -s scan.db -h localhost -i -v
-- whisker / v1.4.0 / rain forest puppy / www.wiretrip.net --
- Loaded script database of 1968 lines

= - = - = - = - = - =
= Host: localhost
- Did not return a Server: string; going to automatically rescan
- with dumb.db


- Loaded script database of 608 lines

= - = - = - = - = - =
= Host: localhost
- Did not return a Server: string; going to automatically rescan
- with dumb.db
```

Audit Evaluation

The gcc compiler emitted several warnings that indicated missing prototypes and unused variables; errors of this type are easy to find and fix, and do not typically represent security concerns. Gcc also warned when certain arguments to functions would be converted to a different data type; these warnings can be a sign that unexpected data could cause problems due to assumptions made by the program's author.

ITS4 and Pscan found several instances of variable format string usage. This class of bug has gained popularity, following buffer overflows. These warnings bring the programmer's attention to possible corruption due to user input being used as formatting commands. ITS4 reported several other file operations as being risky, particularly those using symbolic names instead of file descriptors. The danger here lies in the program being subjected to a race condition where the attacker aims to replace the symbolic object with one under their control in between the time that the program checks for it and when that the program performs the actual operation. Most of the warnings, upon inspection, would be difficult for an attacker to manipulate, since it would require priviledged local access; access at this level could be used to simply replace the web server.

Qaudit gives very brief output upon discovering possible problems; it would be more useful to an experienced programmer who simply wanted a place to start looking for problems. RATS' output was very similar to ITS4's, but provides more information on the possible problem(s) as well as more knowledge surrounding the function call.

The four black box testing tools (bfbtester and the three fuzz programs) gave no useful feedback, as suspected. ``Webs'' is not the type of program that these tools expect to test:

- It does not read from standard input.
- It expects requests to follow the HTTP protocol.
- It expects ``inetd'' to hand it a socket to read requests from.

Since ``webs'' does not expect any command-line parameters or switches, no amount of standard fuzz testing will have any ill effect. The best test I could come up with was to emulate the ``GET'' request of the HTTP protocol; these tests resulted in simple and immediate error reports from ``webs''.

Jill, Webscan, Nessus, and Whisker met similar fates - because ``webs'' does not support CGI scripts, they all found nothing to report. Nessus has exploit scripts available for older versions of popular web servers, but did not have the logic to follow the redirect sent by ``webs''. As a result, none of the tests affected the important section of code where files are read from disk and sent across the network.

Is it enough?

The members of the Secure Programming list[25] debated in October and November of 2000 over the definition of security. Many claimed it wouldn't be possible to ever declare a program ``secure''. Others suggest that improving tools and raising awareness helps raise the bar, and that anything we can do helps. Obviously, without a clear definition of what ``security'' means, it will be difficult to write programs that are ``secure''.

The tools that are available address currently-known security concerns, such as buffer overflows, format string vulnerabilities, and unsafe input usage. As noted above, though, all of these tests are reactive - they must be run after a program has been written. Proactive guidelines are needed to educate future programmers, particularly ones who will be writing code that should be ``secure''.

I've seen two definitions of security that look interesting:

- A program is reliable if it does everything it is specified to do. A program is secure if it does everything it is specified to do and nothing else. -Ivan Arce
- A program is secure if it behaves as expected. -Garfinkel and Spafford in [6].

These definitions sound good, but are difficult (if not impossible) to translate to real-world examples. Reliability is a large part of security (think Denial of Service attacks), but what does it mean to ``behave as expected''? In a narrow context such as web servers, we can define a certain level of functionality that is expected; it is this basic functionality (pass files from a certain subdirectory over a socket) that drew me to ``webs''.

Once CGI scripts and server-side processing (SSI, Enterprise Java Beans, Server-side javascript, etc) are introduced, it becomes more difficult to define and measure what is expected. I believe that a new approach, like webs, is needed: to start with a minimum of functionality and add new features as they are audited. While it sounds very basic, one person has noted that some companies aren't even aware that code needs to be audited: `http://www.securityfocus.com/archive/98/153537`.

I compared the auditing results for ``webs'' with two other small web servers: boa[26] and thttpd[27]. Both of these web servers consist of several source files written in C. Both projects aim to be small, fast, and secure web servers. RATS and ITS4 both reported more possible problems with boa and thttpd than with webs, primarily due to the larger code base.

Future work

Maintainers of software programs that are used in sensitive areas, such as network daemons and setuid programs, should use the available auditing tools to review their code. We need a future environment where security-related bugs are completely unacceptable. This can be accomplished by introducing security concepts into programming language courses, by combining and improving existing auditing tools, and by creating encouraging the public release of source code. Work is currently being done on the OpenBSD operating system to audit each line of code.

Current security threats are the result of research on new ways to interact with programs. Programs that accept input from the outside, whether it be a human or another program, should be careful about the assumptions it makes. A valuable concept here is one attributed to Jon Postel: ``be liberal in what you accept and conservative in what you send". Careful attention to this concept while creating programs can help avoid future security breaches. Note that programs need not accept all inputs, but rather be liberal in their assumptions about the size and composition of that input.

Work should continue on several ongoing security resources: David Wheeler's Secure Programming FAQ[28], the Shmoo site[29], and the SecurityFocus FAQ[30]. As these files grow in size and detail, future programmers will have more resources to draw upon.

Source code auditing tools do a decent job of alerting a conscientous programmer to possible problems. Black box testing tools do a good job in their area of expertise. What we need are protocol-aware auditing tools. These tools should understand a certain protocol; as a result, they would test more paths of the target program. These types of tests are also known as ``protocol conformance testing", and are often done in other areas, such as hardware manufacturing and telecommunications, where there are specific expectations regarding interactions.

Techniques and methods have been developed to test TCP/IP (`http://www-mice.cs.ucl.ac.uk/multimedia/misc/tcp_ip/8603.mm.www/0314.html`), OSI protocols (`http://www.computer.org/cspress/catalog/BP05352.htm`), and formal description methods (`http://ken.slctech.org/miscdatacomm/osi_protocol.htm`). These ideas, combined with black box testing, could be developed into a protocol-aware testing tool for HTTP. I have taken an interest in this area, and have begun a project called ``protofuzz" at Sourceforge[31].

The aim of the protofuzz project is to extend the original fuzz programs into network protocols. There will be three stages to the protofuzz testing architecture:

1. Generate a grammar for a certain protocol (SSH, NTP, DNS, HTTP, etc).
2. Generate a ``fuzz session" from the grammar.
3. Test the target with the fuzz session.

Protofuzz will build on as much earlier work as possible; I hope to be able to generate grammars from the RFC descriptions of the protocols. I will use the ideas from existing fuzz programs to generate the sessions and use tools such as netcat[24] to run the tests.

Improvements can only be made after problems are found, and problems can be found much more quickly and easily once the source code of a program is available. It is widely accepted in the cryptography community that secret algorithms are often weaker than publicly-available ones, due to the fact that the remaining publicly-available ones have been tested by many people. One concern that security practitioners have with making source code available is that the ``black hat" community then also has a chance to find bugs in the program that lead to security violations. It is only through the cycle of finding and patching bugs that a program can survive.

The Apache web server is one example of an open-source program that has survived this cycle. Many earlier versions of the program had bugs; some of these bugs could lead to unexpected behavior. After the bugs were found, however, a patch was soon available. With closed-source programs, insecure versions of programs may exist for months before a fix is available, if at all. The open source community, once it begins relying on programs, tends to find motivation to keep those programs running bug-free, and often has the means to provide suggestions for fixes, if not the fixes themselves. Such is not the case with closed-source software.

The open source movement has gained momentum recently with the success of several open source programs, including the Linux kernel. The movement has used several principles that have been beneficial. One of these is the idea of ``release early and often"[32, Lesson #1]. This guideline encourages the author of a program to look beyond himself as the expert. If the program is mostly working, but several bugs remain, other people may recognize the symptoms and suggest ways to fix the bug. Releasing the source code early can also result in suggestions for improvement that may not have been realized by the author alone.

As more people see and use the program, it is exposed to a variety of environments and expectations. This variety gives rise to another open source maxim: ``Many eyes make all bugs shallow"[32, Lesson #8]. Once a program becomes widely used, it is subjected to a variety of environments, some of which may not have been envisioned by the original author. If these new environments expose limitations or bugs in the program, then the program can be improved.

In the end, security requires vigilance from both the programmer(s) and the administrator(s). Auditing tools can assist both parties in determining areas of vulnerability, while community standards and review raise the bar of software quality.

Patch for webs.c

I have submitted the following patch to Hobbit for review. The main change is the addition of the suck_headers function, which reads the extra HTTP request headers that are sent by most browsers. The other changes exist to pacify gcc's -Wall option.

```
--- webs099/webs.c      Fri Apr 12 17:58:42 1996
+++ my_webs/webs.c      Mon Jul 16 09:11:50 2001
@@ -30,6 +30,8 @@

 #include "generic.h"           /* comes with netcat, too... */

+#include <unistd.h>            /* for sleep() */
+#include <stdlib.h>            /* for exit() */
 #include <stdio.h>
 #include <string.h>            /* strcpy, strchr, yadda yadda */
 #include <setjmp.h>            /* jmp_buf et al */
@@ -225,7 +227,6 @@
 {
   register char *p, *q, *qs;
   register int x;
-  register short s1, s2;       /* first/second found-space flags */

   if (strncmp (str, "GET ", 4))        /* only valid method here! */
     return (0);
@@ -369,17 +370,66 @@
     syslog (LOG_ERR, "%s %.250s I/O err %d", aclient, path, y);  /* nonfatal */
 } /* handfile */

+void suck_headers(FILE * channel) {
+       int stage = 0;
+       int pristine = 1;
+
+       /* read until you see 13,10,13,10 (\r,\n,\r,\n) sequence */
+       /*
+       Start at stage 0. With a \r, go to stage 1.
+       in stage 1, with a \n, go to stage 2.
+       in stage 2, with a \r, go to stage 3.
+       in stage 3, with a \n, go to stage 4, done.
+       Anything unexpected resets us to stage 0
+       */
+
+       /* if the very first thing we read is a \r\n, dump out. I've only seen
+          this come up with manual telnet testing. netscape and lynx (and
+          presumably most browsers) send some header junk. nc gets us EOF,
+          but telnet lets us enter more stuff. */
+       do {
+               int trash = fgetc(channel);
+               if (trash == EOF) {
+                       return;
+               }
+               if (trash == '\r') {
+                       if (stage == 0) {
+                               stage = 1;
+                       } else if (stage == 2) {
+                               stage = 3;
+                       } else {
+                               stage = 0;
+                       }
+               } else if (trash == '\n') {
+                       if (stage == 1) {
+                               if (pristine) {
+                                       return;
+                               } else {
+                                       stage = 2;
+                               }
+                       } else if (stage == 3) {
+                               return;
+                       } else {
```

```
+                               stage = 0;
+                       }
+               } else {
+                       pristine = 0;
+                       stage = 0;
+               }
+       } while (1);
+}
+
 /* main :
    and away we go... */
-main(argc, argv)
+int main(argc, argv)
   int argc;
   char ** argv;
 {
   register char * pp;
   register int x;
   int y;
-  int netfd1;
-  int netfd2;
+  int netfd1=0;
+  int netfd2=0;
   USHORT lp;
   struct hostent * hp = (struct hostent *) 0;

@@ -592,6 +642,11 @@
       continue;
     }
     x = 1;
+
+    /* read in (and toss) any remaining request header lines. */
+       /* this gets around the broken pipe thing that comes up with netscape */
+       suck_headers(chan);
+
     if ((dafile.st_mode & S_IFMT) == S_IFREG)
       handfile (pp, netfd2);
     else
@@ -601,7 +656,6 @@
 /* here when timeout fired; log it and die */
   (void) close (netfd2);
   (void) close (netfd1);
-  syslog (LOG_INFO, "%s %d timeout", aclient, lp);
   exit (0);
 } /* main */
```

Embedded web servers

August, 1996

http://www.emweb.com

Virata's website, powered by ``EmWeb'', their embedded web server product.

August, 1997

http://www.webreview.com/1997/08_08/developers/08_08_97_1.shtml

An article entitled ``The Incredible Shrinking Web Server'' introduces embedded web servers as a counter to bloat in mainstream web servers.

March, 1998

http://www.emweb.com/web_management/trade_articles/csd0398.html

Article on the EmWeb site explaining the advantages of web-based management of network devices.

January, 1999

http://wearables.stanford.edu/hardware.html

Features a matchbox-sized web server and PC, with instructions for building both. The web site is powered by the web server it describes.

April, 1999

http://dpnm.postech.ac.kr/ews/
An introduction to, and overview of, Embedded Web Server Technology.

July, 1999

http://www-ccs.cs.umass.edu/~shri/iPic.html
Explains their IPic - a match head sized web server.

May, 2000

http://www.tiqit.com
Tiqit Computers spins off from the Stanford University Wearable Computing Laboratory.

September, 2000

http://www.allegrosoft.com/
Allegrosoft offers RomPager, "an HTTP 1.1 compliant, portable embedded Web server designed for use in any network device."

October, 2000

http://www.jkmicro.com/emweb.html
JK Microsystems offers embedded TCP devices; they provide a working example at http://209.233.102.10/homepage.htm.

March, 2001

http://www.prosyst.com/press/online_household.html
ProSyst and Siemens present an intelligent online household, including an Internet-connected washing machine.

April, 2001

http://www.riverdale.k12.or.us/linux/toaster/
A high school student embeds most of a PC into a toaster oven, then installs Linux and makes a terminal server out of it.

June, 2001

http://www.uclinux.com/hand-powered_web_server/index.html
Explains how a Lineo employee created a hand-powered web server using a flashlight and Linux.

Bibliography

1
Google search engine.
http://www.google.com/.

2
Netcraft web server survey.
http://www.netcraft.com/survey/.

3
Source code to hobbit's webs.
http://avian.org/src/hacks/webs099.tgz.

4
Lincoln D. Stein.
Www security faq.
http://www.w3.org/Security/Faq/wwwsf1.html.

5
William R. Cheswick and Steven M. Bellovin.
Firewalls and Internet Security.
Addison-Wesley, 1994.

6
Simson Garfinkel and Gene Spafford.
Practical Unix and Internet Security.
O'Reilly & Associates, 2nd edition, April 1996.

7

Julia H. Allen.
The CERT Guide to System and Network Security Practices.
Addison-Wesley, 2001.

8

Aviel D. Rubin, Daniel Geer, and Marcus J. Ranum.
Web Security Sourcebook.
John Wiley & Sons, Inc., 1997.

9

Brian W. Kernighan and Rob Pike.
The Practice of Programming.
Addison-Wesley Professional Computing Series, February 1999.

10

EGCS Steering Committee.
Gcc compiler.
`http://www.gnu.org/software/gcc/gcc.html`.

11

John Viega.
Its4.
`http://freshmeat.net/projects/its4/`.

12

Alan DeKok.
Problem scanner.
`http://www.striker.ottawa.on.ca/~aland/pscan/`.

13

Antonomasia.
Scanner.
`http://www.notatla.demon.co.uk/SOFTWARE/SCANNER/`.

14

v9@fakehalo.org.
Quick audit.
`http://freshmeat.net/projects/qaudit/`.

15

RATS team.
Rough auditing tool for security.
`http://freshmeat.net/projects/rats/`.

16

David A. Wheeler.
Flawfinder.
`http://www.dwheeler.com/flawfinder/`.

17

Dawson Engler, Benjamin Chelf, David Yu Chen, Andy Chou, Seth Hallem, Wallace Huang, and Junfeng Yang.
Meta-level compilation.
`http://hands.stanford.edu/`.

18

Barton P. Miller, Lars Fredriksen, and Bryan So.
Fuzz testing of application reliability.
`http://www.cs.wisc.edu/~bart/fuzz/fuzz.html`.

19

Justin E. Forrester and Barton P. Miller.
An empirical study of the robustness of windows nt applications using random testing.

http://www.cs.wisc.edu/~bart/fuzz/fuzz-nt.html

20
Mike Heffner.
Brute force binary tester.
http://bfbtester.sourceforge.net/.

21
Ben Woodard.
Extended fuzz.
http://fuzz.sourceforge.net/.

22
Scott Maxwell.
The bulletproof penguin.
http://home.pacbell.net/s-max/scott/bulletproof-penguin.html.

23
Lars Fredriksen, Bryan So, and Bart Miller.
Original fuzz program.
ftp://grilled.cs.wisc.edu/fuzz/.

24
Hobbit.
Netcat.
http://www.l0pht.com/~weld/netcat/.

25
SecurityFocus.com.
Secure programming mailing list.
http://www.securityfocus.com/forums/secprog/intro.html.

26
Larry Doolittle and Jon Nelson.
Boa webserver.
http://www.boa.org/.

27
Jef Poskanzer.
thttpd - tiny/turbo/throttling http server.
http://www.acme.com/software/thttpd/.

28
David A. Wheeler.
Secure programming howto.
http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/book1.html .

29
Shmoo.com.
How to write secure code.
http://www.shmoo.com/securecode/.

30
Oliver Friedrichs at SecurityFocus.com.
Working document: Secure programming v1.00.
http://www.securityfocus.com/forums/secprog/secure-programming.html.

31
Jeff Schaller.
Protofuzz.
http://sourceforge.net/projects/protofuzz/.

32

Eric S. Raymond.
The cathedral and the bazaar.
http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

Footnotes

... ``webs''[1]
Submitted under the requirements of SANS GSNA Practical v1.0

... devices[2]
See part V for a brief list.

Jeff Schaller
2001-07-24