Privoxy Developer Manual

$Id: developer−manual.sgml,v 1.46.2.11 2002/12/11 13:12:15 hal9 Exp $

The developer manual provides guidance on coding, testing, packaging, documentation and other issues of importance to those involved with Privoxy development. It is mandatory (and helpful!) reading for anyone who wants to join the team.

Please note that this document is constantly evolving. This copy represents the state at the release of version 3.0.3. You can find the latest version of the this manual at http://www.privoxy.org/developer−manual/. Please see the Contact section on how to contact the developers.

Table of Contents

Table of Contents

1. Introduction

Privoxy, as an heir to Junkbuster, is an Open Source project and licensed under the GPL. As such, Privoxy development is potentially open to anyone who has the time, knowledge, and desire to contribute in any capacity. Our goals are simply to continue the mission, to improve Privoxy, and to make it available to as wide an audience as possible.

One does not have to be a programmer to contribute. Packaging, testing, and porting, are all important jobs as well.

1.1. Quickstart to Privoxy Development

The first step is to join the developer's mailing list. You can submit your ideas, or even better patches. Patches are best submitted to the Sourceforge tracker set up for this purpose, but can be sent to the list for review too.

You will also need to have a cvs package installed, which will entail having ssh installed as well (which seems to be a requirement of SourceForge), in order to access the cvs repository. Having the GNU build tools is also going to be important (particularly, autoconf and gmake).

For the time being (read, this section is under construction), you can also refer to the extensive comments in the source code. In fact, reading the code is recommended in any case.

2. The CVS Repository

If you become part of the active development team, you will eventually need write access to our holy grail, the CVS repository. One of the team members will need to set this up for you. Please read this chapter completely before accessing via CVS.

2.1. Access to CVS

The project's CVS repository is hosted on SourceForge. Please refer to the chapters 6 and 7 in SF's site documentation for the technical access details for your operating system. For historical reasons, the CVS server is called `cvs.ijbswa.sourceforge.net`, the repository is called `ijbswa`, and the source tree module is called `current`.

2.2. Branches

Within the CVS repository, there are modules and branches. As mentioned, the sources are in the `current` "module". Other modules are present for platform specific issues. There is a webview of the CVS hierarchy at http://cvs.sourceforge.net/cgi−bin/viewcvs.cgi/ijbswa/, which might help with visualizing how these pieces fit together.

Branches are used to fork a sub−development path from the main trunk. Within the `current` module where the sources are, there is always at least one "branch" from the main trunk devoted to a stable release series. The main trunk is where active development takes place for the next stable series (e.g. 3.2.x). So just prior to each stable series (e.g. 3.0.x), a branch is created just for stable series releases (e.g. 3.0.0 −> 3.0.1 −> 3.0.2, etc). Once the initial stable release of any stable branch has taken place, this branch is *only used for bugfixes*, which have had prior testing before being committed to CVS. (See Version Numbers below for details on versioning.)

This will result in at least two active branches, which means there may be occasions that require the same (or similar) item to be checked into to two different places (assuming its a bugfix and needs fixing in both the stable and unstable trees). This also means that in order to have access to both trees, both will have to be checked out separately. Use the `cvs −r` flag to check out a branch, e.g: `cvs co −r v_3_0_branch current`.

2.3. CVS Commit Guidelines

The source tree is the heart of every software project. Every effort must be made to ensure that it is readable, compilable and consistent at all times. There are differing guidelines for the stable branch and the main development trunk, and we ask anyone with CVS access to strictly adhere to the following guidelines:

Basic Guidelines, for all branches:

- Never (read: *never, ever*) be tempted to commit that small change without testing it thoroughly first. When we're close to a public release, ask a fellow developer to review your changes.
- Your commit message should give a concise overview of *what you changed* (no big details) and *why you changed it* Just check previous messages for good examples.
- Don't use the same message on multiple files, unless it equally applies to all those files.
- If your changes span multiple files, and the code won't recompile unless all changes are committed (e.g. when changing the signature of a function), then commit all files one after another, without long delays in between. If necessary, prepare the commit messages in advance.
- Before changing things on CVS, make sure that your changes are in line with the team's general consensus on what should be done.
- Note that near a major public release, we get more cautious. There is always the possibility to submit a patch to the patch tracker instead.

Stable branches are handled with more care, especially after the initial *.*.0 release, and we are just in bugfix mode. In addition to the above, the below applies only to the stable branch (currently the `v_3_0_branch` branch):

- Do not commit *anything* unless your proposed changes have been well tested first, preferably by other members of the project, or have prior approval of the project leaders or consensus of the devel list.
- Where possible, bugfixes and changes should be tested in the main development trunk first. There may be occasions where this is not feasible, though.
- Alternately, proposed changes can be submitted as patches to the patch tracker on Sourceforge first: http://sourceforge.net/tracker/?group_id=11118&atid=311118. Then ask for peer review.
- Do not even think about anything except bugfixes. No new features!

3. Documentation Guidelines

All formal documents are maintained in Docbook SGML and located in the `doc/source/*` directory. You will need Docbook, the Docbook DTD's and the Docbook modular stylesheets (or comparable alternatives), and either jade or openjade (recommended) installed in order to build docs from source. Currently there is *user−manual*, *FAQ*, and, of course this, the *developer−manual* in this format. The *README*, *AUTHORS privoxy.1* (man page), and *config* files are also now maintained as Docbook SGML. These files, when built, in the top−level source directory are generated files! Also, the Privoxy `index.html` (and a variation on this file, `privoxy−index.html`, meant for inclusion with doc packages), are maintained as SGML as well. *DO NOT edit these directly*. Edit the SGML source, or contact someone involved in the documentation (at present Hal).

`config` requires some special handling. The reason it is maintained this way is so that the extensive comments in the file mirror those in *user−manual*. But the conversion process requires going from SGML to HTML to text to special formatting required for the embedded comments. Some of this does not survive so well. Especially some of the examples that are longer than 80 characters. The build process for this file outputs to `config.new`, which should be reviewed for errors and mis−formatting. Once satisfied that it is correct, then it should be hand copied to `config`.

Other, less formal documents (e.g. `LICENSE`, `INSTALL`) are maintained as plain text files in the top−level source directory. At least for the time being.

Packagers are encouraged to include this documentation. For those without the ability to build the docs locally, text versions of each are kept in CVS. HTML versions are also now being kept in CVS under `doc/webserver/*`.

Formal documents are built with the Makefile targets of `make dok`, or alternately `make redhat−dok`. If you have problems, try both. The build process uses the document SGML sources in `doc/source/*/*` to update all text files in `doc/text/` and to update all HTML documents in `doc/webserver/`.

Documentation writers should please make sure documents build successfully before committing to CVS, if possible.

How do you update the webserver (i.e. the pages on privoxy.org)?

1. First, build the docs by running `make dok` (or alternately `make redhat−dok`). For PDF docs, do `make dok−pdf`.
2. Run `make webserver` which copies all files from `doc/webserver` to the sourceforge webserver via scp.

Finished docs should be occasionally submitted to CVS (`doc/webserver/*/*.html`) so that those without the ability to build them locally, have access to them if needed. This is especially important just prior to a new release! Please do this *after* the `$VERSION` and other release specific data in `configure.in` has been updated (this is done just prior to a new release).

3.1. Quickstart to Docbook and SGML

If you are not familiar with SGML, it is a markup language similar to HTML. Actually, not a mark up language per se, but a language used to define markup languages. In fact, HTML is an SGML application. Both will use "tags" to format text and other content. SGML tags can be much more varied, and flexible, but do much of the same kinds of things. The tags, or "elements", are definable in SGML. There is no set "standards". Since we are using Docbook, our tags are those that are defined by Docbook. Much of how the finish document is rendered is determined by the "stylesheets". The stylesheets determine how each tag gets translated to HTML, or other formats.

Tags in Docbook SGML need to be always "closed". If not, you will likely generate errors. Example: `<title>My Title</title>`. They are also case−insensitive, but we strongly suggest using all lower case. This keeps compatibility with [Docbook] XML.

Our documents use "sections" for the most part. Sections will be processed into HTML headers (e.g. `h1` for `sect1`). The Docbook stylesheets will use these to also generate the Table of Contents for each doc. Our TOC's are set to a depth of three. Meaning `sect1`, `sect2`, and `sect3` will have TOC entries, but `sect4` will not. Each section requires a `<title>` element, and at least one `<para>`. There is a limit of five section levels in Docbook, but generally three should be sufficient for our purposes.

Some common elements that you likely will use:

<para></para>, paragraph delimiter. Most text needs to be within paragraph elements (there are some exceptions).

<emphasis></emphasis>, the stylesheets make this italics.

<filename></filename>, files and directories.

<command></command>, command examples.

<literallayout></literallayout>, like `<pre>`, more or less.

<itemizedlist></itemizedlist>, list with bullets.

<listitem></listitem>, member of the above.

<screen></screen>, screen output, implies `<literallayout>`.

<ulink url="example.com"></ulink>, like HTML `<a>` tag.

<quote></quote>, for, doh, quoting text.

Look at any of the existing docs for examples of all these and more.

You might also find "Writing Documentation Using DocBook − A Crash Course" useful.

3.2. Privoxy Documentation Style

It will be easier if everyone follows a similar writing style. This just makes it easier to read what someone else has written if it is all done in a similar fashion.

Here it is:

- All tags should be lower case.
- Tags delimiting a *block* of text (even small blocks) should be on their own line. Like:

```
<para>
 Some text goes here.
</para>
       Tags marking individual words, or few words, should be in−line:

Just to <emphasis>emphasize</emphasis>, some text goes here.
```

- Tags should be nested and step indented for block text like: (except in−line tags)

```
<para>
 <itemizedlist>
  <para>
   <listitem>
     Some text goes here in our list example.
    </listitem>
  </para>
 </itemizedlist>
</para>
       This makes it easier to find the text amongst the tags ;−)
```

- Use white space to separate logical divisions within a document, like between sections. Running everything together consistently makes it harder to read and work on.
- Do not hesitate to make comments. Comments can either use the <comment> element, or the <!−− −−> style comment familiar from HTML. (Note in Docbook v4.x <comment> is replaced by <remark>.)
- We have an international audience. Refrain from slang, or English idiosyncrasies (too many to list :). Humor also does not translate well sometimes.
- Try to keep overall line lengths in source files to 80 characters or less for obvious reasons. This is not always possible, with lengthy URLs for instance.
- Our documents are available in differing formats. Right now, they are just plain text, TML, and PDF, but others are always a future possibility. Be careful with URLs (<ulink>), and avoid this mistake:

  My favorite site is <ulink url="http://example.com">here</ulink>.

  This will render as "My favorite site is here", which is not real helpful in a text doc. Better like this:

  My favorite site is <ulink url="http://example.com">example.com</ulink>.
- All documents should be spell checked occasionally. aspell can check SGML with the `−H` option. (ispell I think too.)

3.3. Privoxy Custom Entities

Privoxy documentation is using a number of customized "entities" to facilitate documentation maintenance.

We are using a set of "boilerplate" files with generic text, that is used by multiple docs. This way we can write something once, and use it repeatedly without having to re−write the same content over and over again. If editing such a file, keep in mind that it should be *generic*. That is the purpose; so it can be used in varying contexts without additional modifications.

We are also using what Docbook calls "internal entities". These are like variables in programming. Well, sort of. For instance, we have the `p−version` entity that contains the current Privoxy version string. You are strongly encouraged to use these where possible. Some of these obviously require re−setting with each release (done by the Makefile). A sampling of custom entities are listed below. See any of the main docs for examples.

- Re− "boilerplate" text entities are defined like:

```
<!entity supported SYSTEM "supported.sgml">
```

In this example, the contents of the file, `supported.sgml` is available for inclusion anywhere in the doc. To make this happen, just reference the now defined entity: `&supported;` (starts with an ampersand and ends with a semi−colon), and the contents will be dumped into the finished doc at that point.

- Commonly used "internal entities":

p−version: the Privoxy version string, e.g. "3.0.3".

p−status: the project status, either "alpha", "beta", or "stable".

p−not−stable: use to conditionally include text in "not stable" releases (e.g. "beta").

p−stable: just the opposite.

p−text: this doc is only generated as text.

There are others in various places that are defined for a specific purpose. Read the source!

---

4. Coding Guidelines

4.1. Introduction

This set of standards is designed to make our lives easier. It is developed with the simple goal of helping us keep the "new and improved Privoxy" consistent and reliable. Thus making maintenance easier and increasing chances of success of the project.

And that of course comes back to us as individuals. If we can increase our development and product efficiencies then we can solve more of the request for changes/improvements and in general feel good about ourselves. ;−>

4.2. Using Comments

4.2.1. Comment, Comment, Comment

Explanation:

Comment as much as possible without commenting the obvious. For example do not comment "aVariable is equal to bVariable". Instead explain why aVariable should be equal to the bVariable. Just because a person can read code does not mean they will understand why or what is being done. A reader may spend a lot more time figuring out what is going on when a simple comment or explanation would have prevented the extra research. Please help your brother IJB'ers out!

The comments will also help justify the intent of the code. If the comment describes something different than what the code is doing then maybe a programming error is occurring.

Example:

```
/* if page size greater than 1k ... */
if ( PageLength() > 1024 )
{
    ... "block" the page up ...
}

/* if page size is small, send it in blocks */
if ( PageLength() > 1024 )
{
    ... "block" the page up ...
}

This demonstrates 2 cases of "what not to do".  The first is a
"syntax comment".  The second is a comment that does not fit what
is actually being done.
```

4.2.2. Use blocks for comments

Explanation:

Comments can help or they can clutter. They help when they are differentiated from the code they describe. One line comments do not offer effective separation between the comment and the code. Block identifiers do, by surrounding the code with a clear, definable pattern.

Example:

```
/*********************************************************************
 * This will stand out clearly in your code!
 *********************************************************************/
if ( thisVariable == thatVariable )
{
   DoSomethingVeryImportant();
}


/* unfortunately, this may not */
if ( thisVariable == thatVariable )
{
   DoSomethingVeryImportant();
}


if ( thisVariable == thatVariable ) /* this may not either */
{
   DoSomethingVeryImportant();
}
```

Exception:

If you are trying to add a small logic comment and do not wish to "disrupt" the flow of the code, feel free to use a 1 line comment which is NOT on the same line as the code.

4.2.3. Keep Comments on their own line

Explanation:

It goes back to the question of readability. If the comment is on the same line as the code it will be harder to read than the comment that is on its own line.

There are three exceptions to this rule, which should be violated freely and often: during the definition of variables, at the end of closing braces, when used to comment parameters.

Example:

```
/*********************************************************************
 * This will stand out clearly in your code,
 * But the second example won't.
 *********************************************************************/
if ( thisVariable == thatVariable )
{
   DoSomethingVeryImportant();
}

if ( thisVariable == thatVariable ) /*can you see me?*/
{
   DoSomethingVeryImportant(); /*not easily*/
}


/*********************************************************************
 * But, the encouraged exceptions:
 *********************************************************************/
int urls_read     = 0;     /* # of urls read + rejected */
int urls_rejected = 0;     /* # of urls rejected */

if ( 1 == X )
{
   DoSomethingVeryImportant();
}


short DoSomethingVeryImportant(
   short firstparam,   /* represents something */
   short nextparam     /* represents something else */ )
{
   ...code here...

}   /* -END- DoSomethingVeryImportant */
```

4.2.4. Comment each logical step

Explanation:

Logical steps should be commented to help others follow the intent of the written code and comments will make the code more readable.

If you have 25 lines of code without a comment, you should probably go back into it to see where you forgot to put one.

Most "for", "while", "do", etc... loops _probably_ need a comment. After all, these are usually major logic containers.

4.2.5. Comment All Functions Thoroughly

Explanation:

A reader of the code should be able to look at the comments just prior to the beginning of a function and discern the reason for its existence and the consequences of using it. The reader should not have to read through the code to determine if a given function is safe for a desired use. The proper information thoroughly presented at the introduction of a function not only saves time for subsequent maintenance or debugging, it more importantly aids in code reuse by allowing a user to determine the safety and applicability of any function for the problem at hand. As a result of such benefits, all functions should contain the information presented in the addendum section of this document.

4.2.6. Comment at the end of braces if the content is more than one screen length

Explanation:

Each closing brace should be followed on the same line by a comment that describes the origination of the brace if the original brace is off of the screen, or otherwise far away from the closing brace. This will simplify the debugging, maintenance, and readability of the code.

As a suggestion , use the following flags to make the comment and its brace more readable:

use following a closing brace: } /* −END− if() or while () or etc... */

Example:

```
if ( 1 == X )
{
   DoSomethingVeryImportant();
   ...some long list of commands...
} /* −END− if x is 1 */

or:

if ( 1 == X )
{
   DoSomethingVeryImportant();
   ...some long list of commands...
} /* −END− if ( 1 == X ) */
```

4.3. Naming Conventions

4.3.1. Variable Names

Explanation:

Use all lowercase, and separate words via an underscore ('_'). Do not start an identifier with an underscore. (ANSI C reserves these for use by the compiler and system headers.) Do not use identifiers which are reserved in ANSI C++. (E.g. template, class, true, false, ...). This is in case we ever decide to port Privoxy to C++.

Example:

```
int ms_iis5_hack = 0;
```

Instead of:

```
int msiis5hack = 0; int msIis5Hack = 0;
```

4.3.2. Function Names

Explanation:

Use all lowercase, and separate words via an underscore ('_'). Do not start an identifier with an underscore. (ANSI C reserves these for use by the compiler and system headers.) Do not use identifiers which are reserved in ANSI C++. (E.g. template, class, true, false, ...). This is in case we ever decide to port Privoxy to C++.

Example:

```
int load_some_file( struct client_state *csp )
```

Instead of:

```
int loadsomefile( struct client_state *csp )
int loadSomeFile( struct client_state *csp )
```

4.3.3. Header file prototypes

Explanation:

Use a descriptive parameter name in the function prototype in header files. Use the same parameter name in the header file that you use in the c file.

Example:

```
(.h) extern int load_aclfile( struct client_state *csp );
(.c) int load_aclfile( struct client_state *csp )
```

Instead of:

```
(.h) extern int load_aclfile( struct client_state * ); or
(.h) extern int load_aclfile();
(.c) int load_aclfile( struct client_state *csp )
```

4.3.4. Enumerations, and #defines

Explanation:

Use all capital letters, with underscores between words. Do not start an identifier with an underscore. (ANSI C reserves these for use by the compiler and system headers.)

Example:

```
(enumeration) : enum Boolean { FALSE, TRUE };
(#define) : #define DEFAULT_SIZE 100;
```

Note: We have a standard naming scheme for #defines that toggle a feature in the preprocessor: FEATURE_>, where > is a short (preferably 1 or 2 word) description.

Example:

```
#define FEATURE_FORCE 1

#ifdef FEATURE_FORCE
#define FORCE_PREFIX blah
#endif /* def FEATURE_FORCE */
```

4.3.5. Constants

Explanation:

Spell common words out entirely (do not remove vowels).

Use only widely−known domain acronyms and abbreviations. Capitalize all letters of an acronym.

Use underscore (_) to separate adjacent acronyms and abbreviations. Never terminate a name with an underscore.

Example:

```
#define USE_IMAGE_LIST 1
```

Instead of:

```
#define USE_IMG_LST 1 or
#define _USE_IMAGE_LIST 1 or
#define USE_IMAGE_LIST_ 1 or
#define use_image_list 1 or
#define UseImageList 1
```

4.4. Using Space

4.4.1. Put braces on a line by themselves.

Explanation:

The brace needs to be on a line all by itself, not at the end of the statement. Curly braces should line up with the construct that they're associated with. This practice makes it easier to identify the opening and closing braces for a block.

Example:

```
if ( this == that )
{
   ...
}
```

Instead of:

if (this == that) { ... }

or

if (this == that) { ... }

Note: In the special case that the if−statement is inside a loop, and it is trivial, i.e. it tests for a condition that is obvious from the purpose of the block, one−liners as above may optically preserve the loop structure and make it easier to read.

Status: developer−discretion.

Example exception:

```
while ( more lines are read )
{
   /* Please document what is/is not a comment line here */
   if ( it's a comment ) continue;

   do_something( line );
}
```

4.4.2. ALL control statements should have a block

Explanation:

Using braces to make a block will make your code more readable and less prone to error. All control statements should have a block defined.

Example:

```
if ( this == that )
{
   DoSomething();
   DoSomethingElse();
}
```

Instead of:

if (this == that) DoSomething(); DoSomethingElse();

or

if (this == that) DoSomething();

Note: The first example in "Instead of" will execute in a manner other than that which the developer desired (per indentation). Using code braces would have prevented this "feature". The "explanation" and "exception" from the point above also applies.

4.4.3. Do not belabor/blow−up boolean expressions

Example:

```
structure->flag = ( condition );
```

Instead of:

if (condition) { structure−>flag = 1; } else { structure−>flag = 0; }

Note: The former is readable and concise. The later is wordy and inefficient. Please assume that any developer new to the project has at least a "good" knowledge of C/C++. (Hope I do not offend by that last comment ... 8−)

4.4.4. Use white space freely because it is free

Explanation:

Make it readable. The notable exception to using white space freely is listed in the next guideline.

Example:

```
int firstValue   = 0;
int someValue    = 0;
int anotherValue = 0;
int thisVariable = 0;

if ( thisVariable == thatVariable )

firstValue = oldValue + ( ( someValue − anotherValue ) − whatever )
```

4.4.5. Don't use white space around structure operators

Explanation:

− structure pointer operator ("−>") − member operator (".") − functions and parentheses

It is a general coding practice to put pointers, references, and function parentheses next to names. With spaces, the connection between the object and variable/function name is not as clear.

Example:

```
aStruct−>aMember;
aStruct.aMember;
FunctionName();
```

Instead of: aStruct −> aMember; aStruct . aMember; FunctionName ();

4.4.6. Make the last brace of a function stand out

Example:

```
int function1( ... )
{
   ...code...
   return( retCode );

}   /* −END− function1 */


int function2( ... )
{
}   /* −END− function2 */
```

Instead of:

int function1(...) { ...code... return(retCode); } int function2(...) { }

Note: Use 1 blank line before the closing brace and 2 lines afterward. This makes the end of function standout to the most casual viewer. Although function comments help separate functions, this is still a good coding practice. In fact, I follow these rules when using blocks in "for", "while", "do" loops, and long if {} statements too. After all whitespace is free!

Status: developer−discretion on the number of blank lines. Enforced is the end of function comments.

4.4.7. Use 3 character indentions

Explanation:

If some use 8 character TABs and some use 3 character TABs, the code can look *very* ragged. So use 3 character indentions only. If you like to use TABs, pass your code through a filter such as "expand −t3" before checking in your code.

Example:

```
static const char * const url_code_map[256] =
{
   NULL, ...
};


int function1( ... )
{
   if ( 1 )
   {
      return( ALWAYS_TRUE );
   }
```

```
   else
   {
      return( HOW_DID_YOU_GET_HERE );
   }

   return( NEVER_GETS_HERE );

}
```

4.5. Initializing

4.5.1. Initialize all variables

Explanation:

Do not assume that the variables declared will not be used until after they have been assigned a value somewhere else in the code. Remove the chance of accidentally using an unassigned variable.

Example:

```
short anShort = 0;
float aFloat  = 0;
struct *ptr = NULL;
```

Note: It is much easier to debug a SIGSEGV if the message says you are trying to access memory address 00000000 and not 129FA012; or arrayPtr[20] causes a SIGSEV vs. arrayPtr[0].

Status: developer−discretion if and only if the variable is assigned a value "shortly after" declaration.

4.6. Functions

4.6.1. Name functions that return a boolean as a question.

Explanation:

Value should be phrased as a question that would logically be answered as a true or false statement

Example:

```
ShouldWeBlockThis();
ContainsAnImage();
IsWebPageBlank();
```

4.6.2. Always specify a return type for a function.

Explanation:

The default return for a function is an int. To avoid ambiguity, create a return for a function when the return has a purpose, and create a void return type if the function does not need to return anything.

4.6.3. Minimize function calls when iterating by using variables

Explanation:

It is easy to write the following code, and a clear argument can be made that the code is easy to understand:

Example:

```
for ( size_t cnt = 0; cnt < blockListLength(); cnt ++ )
{
   ....
}
```

Note: Unfortunately, this makes a function call for each and every iteration. This increases the overhead in the program, because the compiler has to look up the function each time, call it, and return a value. Depending on what occurs in the blockListLength() call, it might even be creating and destroying structures with each iteration, even though in each case it is comparing "cnt" to the same value, over and over. Remember too − even a call to blockListLength() is a function call, with the same overhead.

Instead of using a function call during the iterations, assign the value to a variable, and evaluate using the variable.

Example:

```
size_t len = blockListLength();

for ( size_t cnt = 0; cnt < len; cnt ++ )
{
   ....
}
```

Exceptions: if the value of blockListLength() *may* change or could *potentially* change, then you must code the function call in the for/while loop.

4.6.4. Pass and Return by Const Reference

Explanation:

This allows a developer to define a const pointer and call your function. If your function does not have the const keyword, we may not be able to use your function. Consider strcmp, if it were defined as: extern int strcmp(char *s1, char *s2);

I could then not use it to compare argv's in main: int main(int argc, const char *argv[]) { strcmp(argv[0], "privoxy"); }

Both these pointers are *const*! If the c runtime library maintainers do it, we should too.

4.6.5. Pass and Return by Value

Explanation:

Most structures cannot fit onto a normal stack entry (i.e. they are not 4 bytes or less). Aka, a function declaration like: int load_aclfile(struct client_state csp)

would not work. So, to be consistent, we should declare all prototypes with "pass by value": int load_aclfile(struct client_state *csp)

4.6.6. Names of include files

Explanation:

Your include statements should contain the file name without a path. The path should be listed in the Makefile, using −I as processor directive to search the indicated paths. An exception to this would be for some proprietary software that utilizes a partial path to distinguish their header files from system or other header files.

Example:

```
#include <iostream.h>     /* This is not a local include */
#include "config.h"       /* This IS a local include */
```

Exception:

```
/* This is not a local include, but requires a path element. */
#include <sys/fileName.h>
```

Note: Please! do not add "−I." to the Makefile without a _very_ good reason. This duplicates the #include "file.h" behavior.

4.6.7. Provide multiple inclusion protection

Explanation:

Prevents compiler and linker errors resulting from redefinition of items.

Wrap each header file with the following syntax to prevent multiple inclusions of the file. Of course, replace PROJECT_H with your file name, with "." Changed to "_", and make it uppercase.

Example:

```
#ifndef PROJECT_H_INCLUDED
#define PROJECT_H_INCLUDED
 ...
#endif /* ndef PROJECT_H_INCLUDED */
```

4.6.8. Use `extern "C"` when appropriate

Explanation:

If our headers are included from C++, they must declare our functions as `extern "C"`. This has no cost in C, but increases the potential re−usability of our code.

Example:

```
#ifdef __cplusplus
extern "C"
{
#endif /* def __cplusplus */

... function definitions here ...

#ifdef __cplusplus
}
#endif /* def __cplusplus */
```

4.6.9. Where Possible, Use Forward Struct Declaration Instead of Includes

Explanation:

Useful in headers that include pointers to other struct's. Modifications to excess header files may cause needless compiles.

Example:

```
/*********************************************************************
 * We're avoiding an include statement here!
 *********************************************************************/
struct file_list;
extern file_list *xyz;
```

Note: If you declare "file_list xyz;" (without the pointer), then including the proper header file is necessary. If you only want to prototype a pointer, however, the header file is unnecessary.

Status: Use with discretion.

4.7. General Coding Practices

4.7.1. Turn on warnings

Explanation

Compiler warnings are meant to help you find bugs. You should turn on as many as possible. With GCC, the switch is "−Wall". Try and fix as many warnings as possible.

4.7.2. Provide a default case for all switch statements

Explanation:

What you think is guaranteed is never really guaranteed. The value that you don't think you need to check is the one that someday will be passed. So, to protect yourself from the unknown, always have a default step in a switch statement.

Example:

```
switch( hash_string( cmd ) )
{
   case hash_actions_file :
      ... code ...
      break;

   case hash_confdir :
      ... code ...
      break;

   default :
      log_error( ... );
      ... anomaly code goes here ...
      continue; / break; / exit( 1 ); / etc ...
```

```
} /* end switch( hash_string( cmd ) ) */
```

Note: If you already have a default condition, you are obviously exempt from this point. Of note, most of the WIN32 code calls `DefWindowProc' after the switch statement. This API call *should* be included in a default statement.

Another Note: This is not so much a readability issue as a robust programming issue. The "anomaly code goes here" may be no more than a print to the STDERR stream (as in load_config). Or it may really be an ABEND condition.

Status: Programmer discretion is advised.

4.7.3. Try to avoid falling through cases in a switch statement.

Explanation:

In general, you will want to have a 'break' statement within each 'case' of a switch statement. This allows for the code to be more readable and understandable, and furthermore can prevent unwanted surprises if someone else later gets creative and moves the code around.

The language allows you to plan the fall through from one case statement to another simply by omitting the break statement within the case statement. This feature does have benefits, but should only be used in rare cases. In general, use a break statement for each case statement.

If you choose to allow fall through, you should comment both the fact of the fall through and reason why you felt it was necessary.

4.7.4. Use 'long' or 'short' Instead of 'int'

Explanation:

On 32−bit platforms, int usually has the range of long. On 16−bit platforms, int has the range of short.

Status: open−to−debate. In the case of most FSF projects (including X/GNU−Emacs), there are typedefs to int4, int8, int16, (or equivalence ... I forget the exact typedefs now). Should we add these to IJB now that we have a "configure" script?

4.7.5. Don't mix size_t and other types

Explanation:

The type of size_t varies across platforms. Do not make assumptions about whether it is signed or unsigned, or about how long it is. Do not compare a size_t against another variable of a different type (or even against a constant) without casting one of the values. Try to avoid using size_t if you can.

4.7.6. Declare each variable and struct on its own line.

Explanation:

It can be tempting to declare a series of variables all on one line. Don't.

Example:

```
long a = 0;
long b = 0;
long c = 0;
```

Instead of:

long a, b, c;

Explanation: − there is more room for comments on the individual variables − easier to add new variables without messing up the original ones − when searching on a variable to find its type, there is less clutter to "visually" eliminate

Exceptions: when you want to declare a bunch of loop variables or other trivial variables; feel free to declare them on 1 line. You should, although, provide a good comment on their functions.

Status: developer−discretion.

4.7.7. Use malloc/zalloc sparingly

Explanation:

Create a local struct (on the stack) if the variable will live and die within the context of one function call.

Only "malloc" a struct (on the heap) if the variable's life will extend beyond the context of one function call.

Example:

```
If a function creates a struct and stores a pointer to it in a
list, then it should definitely be allocated via `malloc'.
```

4.7.8. The Programmer Who Uses 'malloc' is Responsible for Ensuring 'free'

Explanation:

If you have to "malloc" an instance, you are responsible for insuring that the instance is `free'd, even if the deallocation event falls within some other programmer's code. You are also responsible for ensuring that deletion is timely (i.e. not too soon, not too late). This is known as "low−coupling" and is a "good thing (tm)". You may need to offer a free/unload/destructor type function to accommodate this.

Example:

```
int load_re_filterfile( struct client_state *csp ) { ... }
static void unload_re_filterfile( void *f ) { ... }
```

Exceptions:

The developer cannot be expected to provide `free'ing functions for C run−time library functions ... such as `strdup'.

Status: developer−discretion. The "main" use of this standard is for allocating and freeing data structures (complex or nested).

4.7.9. Add loaders to the `file_list' structure and in order

Explanation:

I have ordered all of the "blocker" file code to be in alpha order. It is easier to add/read new blockers when you expect a certain order.

Note: It may appear that the alpha order is broken in places by POPUP tests coming before PCRS tests. But since POPUPs can also be referred to as KILLPOPUPs, it is clear that it should come first.

4.7.10. "Uncertain" new code and/or changes to existing code, use FIXME

Explanation:

If you have enough confidence in new code or confidence in your changes, but are not *quite* sure of the repercussions, add this:

/* FIXME: this code has a logic error on platform XYZ, * attempting to fix */ #ifdef PLATFORM ...changed code here... #endif

or:

/* FIXME: I think the original author really meant this... */ ...changed code here...

or:

/* FIXME: new code that *may* break something else... */ ...new code here...

Note: If you make it clear that this may or may not be a "good thing (tm)", it will be easier to identify and include in the project (or conversely exclude from the project).

4.8. Addendum: Template for files and function comment blocks:

Example for file comments:

```
const char FILENAME_rcs[] = "$Id: developer−manual.sgml,v 1.46.2.11 2002/12/11 13:12:15 hal9 Exp $";
/*********************************************************************
 *
 * File        :  $Source$
 *
```

```
 * Purpose     :  (Fill me in with a good description!)
 *
 * Copyright   :  Written by and Copyright (C) 2001 the SourceForge
 *                Privoxy team. http://www.privoxy.org/
 *
 *                Based on the Internet Junkbuster originally written
 *                by and Copyright (C) 1997 Anonymous Coders and
 *                Junkbusters Corporation.  http://www.junkbusters.com
 *
 *                This program is free software; you can redistribute it
 *                and/or modify it under the terms of the GNU General
 *                Public License as published by the Free Software
 *                Foundation; either version 2 of the License, or (at
 *                your option) any later version.
 *
 *                This program is distributed in the hope that it will
 *                be useful, but WITHOUT ANY WARRANTY; without even the
 *                implied warranty of MERCHANTABILITY or FITNESS FOR A
 *                PARTICULAR PURPOSE.  See the GNU General Public
 *                License for more details.
 *
 *                The GNU General Public License should be included with
 *                this file.  If not, you can view it at
 *                http://www.gnu.org/copyleft/gpl.html
 *                or write to the Free Software Foundation, Inc., 59
 *                Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 * Revisions   :
 *    $Log$
 *
 *********************************************************************/


#include "config.h"

   ...necessary include files for us to do our work...

const char FILENAME_h_rcs[] = FILENAME_H_VERSION;
```

Note: This declares the rcs variables that should be added to the "show-proxy-args" page. If this is a brand new creation by you, you are free to change the "Copyright" section to represent the rights you wish to maintain.

Note: The formfeed character that is present right after the comment flower box is handy for (X|GNU)Emacs users to skip the verbiage and get to the heart of the code (via `forward-page' and `backward-page'). Please include it if you can.

Example for file header comments:

```
#ifndef _FILENAME_H
#define _FILENAME_H
#define FILENAME_H_VERSION "$Id: developer-manual.sgml,v 1.46.2.11 2002/12/11 13:12:15 hal9 Exp $"
/*********************************************************************
 *
 * File        :  $Source$
 *
 * Purpose     :  (Fill me in with a good description!)
 *
 * Copyright   :  Written by and Copyright (C) 2001 the SourceForge
 *                Privoxy team. http://www.privoxy.org/
 *
 *                Based on the Internet Junkbuster originally written
 *                by and Copyright (C) 1997 Anonymous Coders and
 *                Junkbusters Corporation.  http://www.junkbusters.com
 *
 *                This program is free software; you can redistribute it
 *                and/or modify it under the terms of the GNU General
 *                Public License as published by the Free Software
 *                Foundation; either version 2 of the License, or (at
 *                your option) any later version.
 *
 *                This program is distributed in the hope that it will
 *                be useful, but WITHOUT ANY WARRANTY; without even the
 *                implied warranty of MERCHANTABILITY or FITNESS FOR A
 *                PARTICULAR PURPOSE.  See the GNU General Public
 *                License for more details.
 *
 *                The GNU General Public License should be included with
 *                this file.  If not, you can view it at
 *                http://www.gnu.org/copyleft/gpl.html
 *                or write to the Free Software Foundation, Inc., 59
 *                Temple Place - Suite 330, Boston, MA  02111-1307, USA.
 *
 * Revisions   :
```

```
 *    $Log$
 *
 *********************************************************************/


#include "project.h"

#ifdef __cplusplus
extern "C" {
#endif

   ... function headers here ...


/* Revision control strings from this header and associated .c file */
extern const char FILENAME_rcs[];
extern const char FILENAME_h_rcs[];


#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* ndef _FILENAME_H */

/*
  Local Variables:
  tab-width: 3
  end:
*/
```

Example for function comments:

```
/*********************************************************************
 *
 * Function    :  FUNCTION_NAME
 *
 * Description :  (Fill me in with a good description!)
 *
 * parameters  :
 *          1  :  param1 = pointer to an important thing
 *          2  :  x      = pointer to something else
 *
 * Returns     :  0 => Ok, everything else is an error.
 *
 *********************************************************************/
int FUNCTION_NAME( void *param1, const char *x )
{
   ...
   return( 0 );

}
```

Note: If we all follow this practice, we should be able to parse our code to create a "self-documenting" web page.

5. Testing Guidelines

To be filled.

5.1. Testplan for releases

Explain release numbers. major, minor. developer releases. etc.

1. Remove any existing rpm with rpm −e
2. Remove any file that was left over. This includes (but is not limited to)
    - /var/log/privoxy
    - /etc/privoxy
    - /usr/sbin/privoxy
    - /etc/init.d/privoxy
    - /usr/doc/privoxy*
3. Install the rpm. Any error messages?
4. start,stop,status Privoxy with the specific script (e.g. /etc/rc.d/init/privoxy stop). Reboot your machine. Does autostart work?
5. Start browsing. Does Privoxy work? Logfile written?
6. Remove the rpm. Any error messages? All files removed?

5.2. Test reports

Please submit test reports only with the test form at sourceforge. Three simple steps:

- Select category: the distribution you test on.
- Select group: the version of Privoxy that we are about to release.
- Fill the Summary and Detailed Description with something intelligent (keep it short and precise).

Do not mail to the mailing list (we cannot keep track on issues there).

6. Releasing a New Version

When we release versions of Privoxy, our work leaves our cozy secret lab and has to work in the cold RealWorld[tm]. Once it is released, there is no way to call it back, so it is very important that great care is taken to ensure that everything runs fine, and not to introduce problems in the very last minute.

So when releasing a new version, please adhere exactly to the procedure outlined in this chapter.

The following programs are required to follow this process: `ncftpput` (ncftp), `scp, ssh` (ssh), `gmake` (GNU's version of make), autoconf, cvs.

6.1. Version numbers

First you need to determine which version number the release will have. Privoxy version numbers consist of three numbers, separated by dots, like in X.Y.Z (e.g. 3.0.0), where:

- X, the version major, is rarely ever changed. It is increased by one if turning a development branch into stable substantially changes the functionality, user interface or configuration syntax. Majors 1 and 2 were Junkbuster, and 3 will be the first stable Privoxy release.
- Y, the version minor, represents the branch within the major version. At any point in time, there are two branches being maintained: The stable branch, with an even minor, say, 2N, in which no functionality is being added and only bug−fixes are made, and 2N+1, the development branch, in which the further development of Privoxy takes place. This enables us to turn the code upside down and inside out, while at the same time providing and maintaining a stable version. The minor is reset to zero (and one) when the major is incremented. When a development branch has matured to the point where it can be turned into stable, the old stable branch 2N is given up (i.e. no longer maintained), the former development branch 2N+1 becomes the new stable branch 2N+2, and a new development branch 2N+3 is opened.
- Z, the point or sub version, represents a release of the software within a branch. It is therefore incremented immediately before each code freeze. In development branches, only the even point versions correspond to actual releases, while the odd ones denote the evolving state of the sources on CVS in between. It follows that Z is odd on CVS in development branches most of the time. There, it gets increased to an even number immediately before a code freeze, and is increased to an odd number again immediately thereafter. This ensures that builds from CVS snapshots are easily distinguished from released versions. The point version is reset to zero when the minor changes.

  Stable branches work a little differently, since there should be little to no development happening in such branches. Remember, only bugfixes, which presumably should have had some testing before being committed. Stable branches will then have their version reported as `0.0.0`, during that period between releases when changes are being added. This is to denote that this code is *not for release*. Then as the release nears, the version is bumped according: e.g. `3.0.1 −> 0.0.0 −> 3.0.2`.

In summary, the main CVS trunk is the development branch where new features are being worked on for the next stable series. This should almost always be where the most activity takes place. There is always at least one stable branch from the trunk, e.g now it is `3.0`, which is only used to release stable versions. Once the initial *.0 release of the stable branch has been done, then as a rule, only bugfixes that have had prior testing should be committed to the stable branch. Once there are enough bugfixes to justify a new release, the version of this branch is again incremented Example: 3.0.0 −> 3.0.1 −> 3.0.2, etc are all stable releases from within the stable branch. 3.1.x is currently the main trunk, and where work on 3.2.x is taking place. If any questions, please post to the devel list *before* committing to a stable branch!

Developers should remember too that if they commit a bugfix to the stable branch, this will more than likely require a separate submission to the main trunk, since these are separate development trees within CVS. If you are working on both, then this would require at least two separate check outs (i.e main trunk, *and* the stable release branch, which is `v_3_0_branch` at the moment).

6.2. Before the Release: Freeze

The following *must be done by one of the developers* prior to each new release.

- Make sure that everybody who has worked on the code in the last couple of days has had a chance to yell "no!" in case they have pending changes/fixes in their pipelines. Announce the freeze so that nobody will interfere with last minute changes.
- Increment the version number (point from odd to even in development branches!) in `configure.in`. (RPM spec files will need to be incremented as well.)
- If `default.action` has changed since last release (i.e. software release or standalone actions file release), bump up its version info to A.B in this line:

  ```
  {+add−header{X−Actions−File−Version: A.B} −filter −no−popups}
  ```

  Then change the version info in doc/webserver/actions/index.php, line: '$required_actions_file_version = "A.B";'
- All documentation should be rebuild after the version bump. Finished docs should be then be committed to CVS (for those without the ability to build these). Some docs may require rather obscure processing tools. `config`, the man page (and the html version of the man page), and the PDF docs fall in this category. REAMDE, the man page, AUTHORS, and config should all also be committed to CVS for other packagers. The formal docs should be uploaded to the webserver. See the Section

"Updating the webserver" in this manual for details.

- The *User Manual* is also used for context sensitive help for the CGI editor. This is version sensitive, so that the user will get appropriate help for his/her release. So with each release a fresh version should be uploaded to the webserver (this is in addition to the main *User Manual* link from the main page since we need to keep manuals for various versions available). The CGI pages will link to something like `http://privoxy.org/$(VERSION)/user-manual/`. This will need to be updated for each new release. There is no Makefile target for this at this time!!! It needs to be done manually.
- All developers should look at the `ChangeLog` and make sure noteworthy changes are referenced.
- *Commit all files that were changed in the above steps!*
- Tag all files in CVS with the version number with "**cvs tag v_X_Y_Z**". Don't use vX_Y_Z, ver_X_Y_Z, v_X.Y.Z (won't work) etc.
- If the release was in a development branch, increase the point version from even to odd (X.Y.(Z+1)) again in `configure.in` and commit your change.
- On the webserver, copy the user manual to a new top-level directory called `X.Y.Z`. This ensures that help links from the CGI pages, which have the version as a prefix, will go into the right version of the manual. If this is a development branch release, also symlink `X.Y.(Z-1)` to `X.Y.Z` and `X.Y.(Z+1)` to `.` (i.e. dot).

6.3. Building and Releasing the Packages

Now the individual packages can be built and released. Note that for GPL reasons the first package to be released is always the source tarball.

For *all* types of packages, including the source tarball, *you must make sure that you build from clean sources by exporting the right version from CVS into an empty directory* (just press return when asked for a password):

```
mkdir dist # delete or choose different name if it already exists
cd dist
cvs -d:pserver:anonymous@cvs.ijbswa.sourceforge.net:/cvsroot/ijbswa login
cvs -z3 -d:pserver:anonymous@cvs.ijbswa.sourceforge.net:/cvsroot/ijbswa export -r v_X_Y_Z current
```

Do NOT change a single bit, including, but not limited to version information after export from CVS. This is to make sure that all release packages, and with them, all future bug reports, are based on exactly the same code.

Please find additional instructions for the source tarball and the individual platform dependent binary packages below. And details on the Sourceforge release process below that.

6.3.1. Note on Privoxy Packaging

Please keep these general guidelines in mind when putting together your package. These apply to *all* platforms!

- Privoxy *requires* write access to: all `*.action` files, all logfiles, and the `trust` file. You will need to determine the best way to do this for your platform.
- Please include up to date documentation. At a bare minimum:

  `LICENSE` (top-level directory)

  `README` (top-level directory)

  `AUTHORS` (top-level directory)

  `man page` (top-level directory, Unix-like platforms only)

  `The User Manual` (doc/webserver/user-manual/)

  `FAQ` (doc/webserver/faq/)

  Also suggested: `Developer Manual` (doc/webserver/developer-manual) and `ChangeLog` (top-level directory). `FAQ` and the manuals are HTML docs. There are also text versions in `doc/text/` which could conceivably also be included.

  The documentation has been designed such that the manuals are linked to each other from parallel directories, and should be packaged that way. `privoxy-index.html` can also be included and can serve as a focal point for docs and other links of interest (and possibly renamed to `index.html`). This should be one level up from the manuals. There is a link also on this page to an HTMLized version of the man page. To avoid 404 for this, it is in CVS as `doc/webserver/man-page/privoxy-man-page.html`, and should be included along with the manuals. There is also a css stylesheets that can be included for better presentation: `p_doc.css`. This should be in the same directory with `privoxy-index.html`, (i.e. one level up from the manual directories).
- `user.action` is designed for local preferences. Make sure this does not get overwritten!
- Other configuration files should be installed as the new defaults, but all previously installed configuration files should be preserved as backups. This is just good manners :-)

- Please check platform specific notes in this doc, if you haven't done "Privoxy" packaging before for other platform specific issues. Conversely, please add any notes that you know are important for your platform (or contact one of the doc maintainers to do this if you can't).
- Packagers should do a "clean" install of their package after building it. So any previous installs should be removed first to ensure the integrity of the newly built package. Then run the package for a while to make sure there are no obvious problems, before uploading.

6.3.2. Source Tarball

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then do:

```
make tarball-dist
```

To upload the package to Sourceforge, simply issue

```
make tarball-upload
```

Go to the displayed URL and release the file publicly on Sourceforge. For the change log field, use the relevant section of the `ChangeLog` file.

6.3.3. SuSE, Conectiva or Red Hat RPM

In following text, replace *`dist`* with either "rh" for Red Hat or "suse" for SuSE.

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above).

As the only exception to not changing anything after export from CVS, now examine the file `privoxy-`*`dist`*`.spec` and make sure that the version information and the RPM release number are correct. The RPM release numbers for each version start at one. Hence it must be reset to one if this is the first RPM for *`dist`* which is built from version X.Y.Z. Check the file list if unsure. Else, it must be set to the highest already available RPM release number for that version plus one.

Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then do

```
make dist-dist
```

To upload the package to Sourceforge, simply issue

```
make dist-upload rpm_packagerev
```

where *`rpm_packagerev`* is the RPM release number as determined above. Go to the displayed URL and release the file publicly on Sourceforge. Use the release notes and change log from the source tarball package.

6.3.4. OS/2

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then get the OS/2 Setup module:

```
cvs -z3 -d:pserver:anonymous@cvs.ijbswa.sourceforge.net:/cvsroot/ijbswa co os2setup
```

You will need a mix of development tools. The main compilation takes place with IBM Visual Age C++. Some ancillary work takes place with GNU tools, available from various sources like hobbes.nmsu.edu. Specificially, you will need `autoheader`, `autoconf` and `sh` tools. The packaging takes place with WarpIN, available from various sources, including its home page: xworkplace.

Change directory to the `os2setup` directory. Edit the os2build.cmd file to set the final executable filename. For example,

```
installExeName='privoxyos2_setup_X.Y.Z.exe'
```

Next, edit the `IJB.wis` file so the release number matches in the `PACKAGEID` section:

```
PACKAGEID="Privoxy Team\Privoxy\Privoxy Package\X\Y\Z"
```

You're now ready to build. Run:

```
os2build
```

You will find the WarpIN−installable executable in the `./files` directory. Upload this anonymously to `uploads.sourceforge.net/incoming`, create a release for it, and you're done. Use the release notes and Change Log from the source tarball package.

6.3.5. Solaris

Login to Sourceforge's compilefarm via ssh:

```
ssh cf.sourceforge.net
```

Choose the right operating system (not the Debian one). When logged in, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then run

```
gmake solaris−dist
```

which creates a gzip'ed tar archive. Sadly, you cannot use **make solaris−upload** on the Sourceforge machine (no ncftpput). You now have to manually upload the archive to Sourceforge's ftp server and release the file publicly. Use the release notes and Change Log from the source tarball package.

6.3.6. Windows

You should ensure you have the latest version of Cygwin (from http://www.cygwin.com/). Run the following commands from within a Cygwin bash shell.

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then get the Windows setup module:

```
cvs −z3  −d:pserver:anonymous@cvs.ijbswa.sourceforge.net:/cvsroot/ijbswa co winsetup
```

Then you can build the package. This is fully automated, and is controlled by `winsetup/GNUmakefile`. All you need to do is:

```
cd winsetup
make
```

Now you can manually rename `privoxy_setup.exe` to `privoxy_setup_X_Y_Z.exe`, and upload it to SourceForge. When releasing the package on SourceForge, use the release notes and Change Log from the source tarball package.

6.3.7. Debian

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then add a log entry to `debian/changelog`, if it is not already there, for example by running:

```
debchange −v 3.0.3−stable−1 "New upstream version"
```

Then, run:

```
dpkg−buildpackage −rfakeroot −us −uc −b
```

This will create `../privoxy_3.0.3−stable−1_i386.deb` which can be uploaded. To upload the package to Sourceforge, simply issue

```
make debian−upload
```

6.3.8. Mac OSX

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then get the Mac OSX setup module:

```
cvs -z3 -d:pserver:anonymous@cvs.ijbswa.sourceforge.net:/cvsroot/ijbswa co osxsetup
```

Then run:

```
cd osxsetup
build
```

This will run `autoheader`, `autoconf` and `configure` as well as `make`. Finally, it will copy over the necessary files to the ./osxsetup/files directory for further processing by `PackageMaker`.

Bring up PackageMaker with the PrivoxyPackage.pmsp definition file, modify the package name to match the release, and hit the "Create package" button. If you specify ./Privoxy.pkg as the output package name, you can then create the distributable zip file with the command:

```
zip -r privoxyosx_setup_x.y.z.zip Privoxy.pkg
```

You can then upload `privoxyosx_setup_x.y.z.zip` anonymously to `uploads.sourceforge.net/incoming`, create a release for it, and you're done. Use the release notes and Change Log from the source tarball package.

6.3.9. FreeBSD

Login to Sourceforge's compile-farm via ssh:

```
ssh cf.sourceforge.net
```

Choose the right operating system. When logged in, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then run:

```
gmake freebsd-dist
```

which creates a gzip'ed tar archive. Sadly, you cannot use **make freebsd-upload** on the Sourceforge machine (no ncftpput). You now have to manually upload the archive to Sourceforge's ftp server and release the file publicly. Use the release notes and Change Log from the source tarball package.

6.3.10. HP-UX 11

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then do FIXME.

6.3.11. Amiga OS

First, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then do FIXME.

6.3.12. AIX

Login to Sourceforge's compilefarm via ssh:

```
ssh cf.sourceforge.net
```

Choose the right operating system. When logged in, *make sure that you have freshly exported the right version into an empty directory*. (See "Building and releasing packages" above). Then run:

```
cd current
autoheader && autoconf && ./configure
```

Then run:

```
make aix-dist
```

which creates a gzip'ed tar archive. Sadly, you cannot use **make aix-upload** on the Sourceforge machine (no ncftpput). You now have to manually upload the archive to Sourceforge's ftp server and release the file publicly. Use the release notes and Change Log from the source tarball package.

6.4. Uploading and Releasing Your Package

After the package is ready, it is time to upload it to SourceForge, and go through the release steps. The upload is done via FTP:

- Upload to: ftp://upload.sourceforge.net/incoming
- user: `anonymous`
- password: `ijbswa-developers@lists.sourceforge.net`

Or use the **make** targets as described above.

Once this done go to http://sourceforge.net/project/admin/editpackages.php?group_id=11118, making sure you are logged in. Find your target platform in the second column, and click `Add Release`. You will then need to create a new release for your package, using the format of `$VERSION ($CODE_STATUS)`, e.g. *3.0.3 (beta)*.

Now just follow the prompts. Be sure to add any appropriate Release notes. You should see your freshly uploaded packages in "Step 2. Add Files To This Release". Check the appropriate box(es). Remember at each step to hit the "Refresh/Submit" buttons! You should now see your file(s) listed in Step 3. Fill out the forms with the appropriate information for your platform, being sure to hit "Update" for each file. If anyone is monitoring your platform, check the "email" box at the very bottom to notify them of the new package. This should do it!

If you have made errors, or need to make changes, you can go through essentially the same steps, but select `Edit Release`, instead of `Add Release`.

6.5. After the Release

When all (or: most of the) packages have been uploaded and made available, send an email to the announce mailing list, Subject: "Version X.Y.Z available for download". Be sure to include the download location, the release notes and the Changelog. Also, post an updated News item on the project page Sourceforge, and update the Home page and docs linked from the Home page (see below).

7. Update the Webserver

The webserver should be updated at least with each stable release. When updating, please follow these steps to make sure that no broken links, inconsistent contents or permission problems will occur (as it has many times in the past!):

If you have changed anything in the stable−branch documentation source SGML files, do:

```
make dok dok−pdf # (or 'make redhat−dok dok−pdf' if 'make dok' doesn't work for you)
```

That will generate `doc/webserver/user−manual`, `doc/webserver/developer−manual`, `doc/webserver/faq`, `doc/pdf/*.pdf` and `doc/webserver/index.html` automatically.

If you changed the manual page sources, generate `doc/webserver/man−page/privoxy−man−page.html` by running "**make man**". (This is a separate target due to dependencies on some obscure perl scripts [now in CVS, but not well tested]. See comments in `GNUmakefile`.)

If you want to add new files to the webserver, create them locally in the `doc/webserver/*` directory (or create new directories under `doc/webserver`).

Next, commit any changes from the above steps to CVS. All set? If these are docs in the stable branch, then do:

```
make webserver
```

This will do the upload to the webserver (www.privoxy.org) and ensure all files and directories there are group writable.

Please do *NOT* use any other means of transferring files to the webserver to avoid permission problems. Also, please do not upload docs from development branches or versions. The publicly posted docs should be in sync with the last official release.

---

8. Contacting the developers, Bug Reporting and Feature Requests

We value your feedback. In fact, we rely on it to improve Privoxy and its configuration. However, please note the following hints, so we can provide you with the best support:

8.1. Get Support

For casual users, our support forum at SourceForge is probably best suited: http://sourceforge.net/tracker/?group_id=11118&atid=211118

All users are of course welcome to discuss their issues on the users mailing list, where the developers also hang around.

8.2. Report Bugs

Please report all bugs *only* through our bug tracker: http://sourceforge.net/tracker/?group_id=11118&atid=111118.

Before doing so, please make sure that the bug has not already been submitted and observe the additional hints at the top of the submit form.

Please try to verify that it is a Privoxy bug, and not a browser or site bug first. If unsure, try toggling off Privoxy, and see if the problem persists. The appendix of the user manual also has helpful information on action debugging. If you are using your own custom configuration, please try the stock configs to see if the problem is configuration related.

If not using the latest version, chances are that the bug has been found and fixed in the meantime. We would appreciate if you could take the time to upgrade to the latest version (or even the latest CVS snapshot) and verify your bug, but this is not required for reporting.

8.3. Request New Features

You are welcome to submit ideas on new features or other proposals for improvement through our feature request tracker at http://sourceforge.net/tracker/?atid=361118&group_id=11118.

8.4. Report Ads or Other Actions−Related Problems

Please send feedback on ads that slipped through, innocent images that were blocked, and any other problems relating to the `default.action` file through our actions feedback mechanism located at http://www.privoxy.org/actions/. On this page, you will also find a bookmark which will take you back there from any troubled site and even pre−fill the form!

New, improved `default.action` files will occasionally be made available based on your feedback. These will be announced on the ijbswa−announce list and available from our the files section of our project page.

8.5. Other

For any other issues, feel free to use the mailing lists. Technically interested users and people who wish to contribute to the project are also welcome on the developers list! You can find an overview of all Privoxy−related mailing lists, including list archives, at: http://sourceforge.net/mail/?group_id=11118.

9. Privoxy Copyright, License and History

9.1. License

9.2. History

In the beginning, there was the Internet Junkbuster, by Anonymous Coders and Junkbusters Corporation. It saved many users a lot of pain in the early days of web advertising and user tracking.

But the web, its protocols and standards, and with it, the techniques for forcing users to consume ads, give up autonomy over their browsing, and for spying on them, kept evolving. Unfortunately, the Internet Junkbuster did not. Version 2.0.2, published in 1998, was (and is) the last official release available from Junkbusters Corporation. Fortunately, it had been released under the GNU GPL, which allowed further development by others.

So Stefan Waldherr started maintaining an improved version of the software, to which eventually a number of people contributed patches. It could already replace banners with a transparent image, and had a first version of pop−up killing, but it was still very closely based on the original, with all its limitations, such as the lack of HTTP/1.1 support, flexible per−site configuration, or content modification. The last release from this effort was version 2.0.2−10, published in 2000.

Then, some developers picked up the thread, and started turning the software inside out, upside down, and then reassembled it, adding many new features along the way.

The result of this is Privoxy, whose first stable version, 3.0, was released August, 2002.

10. See also

Other references and sites of interest to Privoxy users:

http://www.privoxy.org/, the Privoxy Home page.

http://www.privoxy.org/faq/, the Privoxy FAQ.

http://sourceforge.net/projects/ijbswa/, the Project Page for Privoxy on SourceForge.

http://config.privoxy.org/, the web−based user interface. Privoxy must be running for this to work. Shortcut: http://p.p/

http://www.privoxy.org/actions/, to submit "misses" to the developers.

http://cvs.sourceforge.net/cgi−bin/viewcvs.cgi/ijbswa/contrib/, cool and fun ideas from Privoxy users.

http://www.junkbusters.com/ht/en/cookies.html, an explanation how cookies are used to track web users.

http://www.junkbusters.com/ijb.html, the original Internet Junkbuster.

http://www.waldherr.org/junkbuster/, Stefan Waldherr's version of Junkbuster, from which Privoxy was derived.

http://privacy.net/analyze/, a useful site to check what information about you is leaked while you browse the web.

http://www.squid−cache.org/, a very popular caching proxy, which is often used together with Privoxy.

http://www.privoxy.org/developer−manual/, the Privoxy developer manual.