<?php

$usr="root";

$pwd="root";

if(isset($\_POST['username']) && !empty($\_POST['username']) && isset($\_POST['password']) && !empty($\_POST['password']) ){

$username=$\_POST['username'];

$password=$\_POST['password'];

if(($username==$usr) && ($password==$pwd) ){

echo '<br>login successfull';

}else{

echo '<br>login unsuccessfull';

}

}else{

echo "<br>Connot be left empty!";

}

?>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*html code for user authentication script

<html>

<head>

<title>Login</title>

</head>

<body>

<form action="login.php" method="POST" align="center">

<br>

Username:<input type="text" name="username"><br><br><br>

Password :<input type="text" name="password"><br><br>

<input type="Submit" value="Submit">

</form>

</body>

</html>

Now, Goto browser->Type <http://localhost/login.html>and run Insert Username and password as **root**.

I am assuming you have basic knowledge of php, if not go through it, its is very easy and also read about HTTP requests

* GET
* POST
* 1down vote
* You can't really have a secure authentication system using JavaScript and HTML alone.
* I would suggest [Basic HTTP authentication](https://en.wikipedia.org/wiki/Basic_access_authentication) on your server instead, as it is much more secure (not perfect by any means, but at least employs a standard server-side method of access control).
* If you must implement something in JavaScript, you could do a password only scheme based on the name of a hidden directory. Something like the following (note this is untested so will need some tweaks):
* (Code borrowed and adapted from [this question](https://stackoverflow.com/a/8571617/413180))
* <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.1/jquery.min.js"></script>
* <script>
* $(function() {
* var url = "some\_url";
* $.ajax(url,
* {
* if (statusCode == 200) {
* document.location.href = url;
* }
* else {
* alert('Incorrect password')
* }
* }
* });
* });
* </script>
* <input type="password" /><a href="javascript:void(0)">Login</a>

The code should be finished so that the function is called when the button is clicked. So if the password is foo, you set a directory on your website called foo, and if the JQuery JavaScript detects that the entered password matches a directory name (e.g. /foo/), then the user is redirected there. Therefore you'd create a /foo/index.html in order to take care of the user's logged in session.

Note that this is the most secure thing you can do with JavaScript and HTML alone and it suffers from the following vulnerabilities.

* It requires that the URL be kept secret, although this can be leaked by the referer header, by browser history and server/proxy logs.
* Once a user is logged in, they are always logged in (they could bookmark the logged in pages)
* There is no easy way to revoke a password.
* There is only one password.
* Anyone with access to view files on the server could view the directory structure and learn the password.
* The URL may be leaked by analytics tools.
* Assumes directory browsing on your server is disabled (or that there's a default page in the private page's parent directory).

In any case, always protect your server with TLS/SSL. My recommendation is to properly create a user authentication system using the advice from [OWASP](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet). The above shows only what's achievable in basic HTML (not much). However, it is better than exposing the password within client-side files.

<html>

<head>

<title>Login paget</title>

</head>

<script type="text/javascript">

function display(form){

if (form.username.value=="root") {

if (form.password.value=="root") {

location="page2.html"

} else {

alert("Invalid Password")

}

} else { alert("Invalid Username")

}

}

</script>

<body >

<form >

<input type="text" name="username" /><br><br>

<input type="password" name="password"/><br><br>

<input type="button" value="Login" onClick="display(this.form)"/>

</form>

</body>

</html>

Hello I have created a login page for you using html and Javascript. The Username and password are **root**. You see if you input correct username and password then the page directs to page2.html and this will show you

This webpage is not found

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*just try out this code -

function validate(){

var username = document.getElementById("username").value;

var password = document.getElementById("password").value;

if ( username == "username1" && password == "password1"){

alert ("Login successfully");

}

else{

alert("Invalid username or password");

}

return false;

}

<html>

<head>

<title>Javascript Login Form Validation</title>

<!-- Include JS File Here -->

<script src="js/login.js"></script>

</head>

<body>

<div class="container">

<form id="form\_id" method="post" name="myform">

<label>User Name :</label>

<input type="text" name="username" id="username"/>

<label>Password :</label>

<input type="password" name="password" id="password"/>

<input type="button" value="Login" id="submit" onclick="validate()"/>

</form>

</div>

</body>

</html>

Perl

**perlcompile**

* [NAME](http://perldoc.perl.org/5.8.9/perlcompile.html#NAME)
* [DESCRIPTION](http://perldoc.perl.org/5.8.9/perlcompile.html#DESCRIPTION)
  + [Layout](http://perldoc.perl.org/5.8.9/perlcompile.html#Layout)
* [Using The Back Ends](http://perldoc.perl.org/5.8.9/perlcompile.html#Using-The-Back-Ends)
  + [The Cross Referencing Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Cross-Referencing-Back-End)
  + [The Decompiling Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Decompiling-Back-End)
  + [The Lint Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Lint-Back-End)
  + [The Simple C Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Simple-C-Back-End)
  + [The Bytecode Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Bytecode-Back-End)
  + [The Optimized C Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Optimized-C-Back-End)
* [Module List for the Compiler Suite](http://perldoc.perl.org/5.8.9/perlcompile.html#Module-List-for-the-Compiler-Suite)
* [KNOWN PROBLEMS](http://perldoc.perl.org/5.8.9/perlcompile.html#KNOWN-PROBLEMS)
* [AUTHOR](http://perldoc.perl.org/5.8.9/perlcompile.html#AUTHOR)

**NAME**

perlcompile - Introduction to the Perl Compiler-Translator

**DESCRIPTION**

Perl has always had a compiler: your source is compiled into an internal form (a parse tree) which is then optimized before being run. Since version 5.005, Perl has shipped with a module capable of inspecting the optimized parse tree (B ), and this has been used to write many useful utilities, including a module that lets you turn your Perl into C source code that can be compiled into a native executable.

The B module provides access to the parse tree, and other modules ("back ends") do things with the tree. Some write it out as bytecode, C source code, or a semi-human-readable text. Another traverses the parse tree to build a cross-reference of which subroutines, formats, and variables are used where. Another checks your code for dubious constructs. Yet another back end dumps the parse tree back out as Perl source, acting as a source code beautifier or deobfuscator.

Because its original purpose was to be a way to produce C code corresponding to a Perl program, and in turn a native executable, the B module and its associated back ends are known as "the compiler", even though they don't really compile anything. Different parts of the compiler are more accurately a "translator", or an "inspector", but people want Perl to have a "compiler option" not an "inspector gadget". What can you do?

This document covers the use of the Perl compiler: which modules it comprises, how to use the most important of the back end modules, what problems there are, and how to work around them.

**Layout**

The compiler back ends are in the B:: hierarchy, and the front-end (the module that you, the user of the compiler, will sometimes interact with) is the O module. Some back ends (e.g., B::C ) have programs (e.g., *perlcc*) to hide the modules' complexity.

Here are the important back ends to know about, with their status expressed as a number from 0 (outline for later implementation) to 10 (if there's a bug in it, we're very surprised):

* **B::Bytecode**

Stores the parse tree in a machine-independent format, suitable for later reloading through the ByteLoader module. Status: 5 (some things work, some things don't, some things are untested).

* **B::C**

Creates a C source file containing code to rebuild the parse tree and resume the interpreter. Status: 6 (many things work adequately, including programs using Tk).

* **B::CC**

Creates a C source file corresponding to the run time code path in the parse tree. This is the closest to a Perl-to-C translator there is, but the code it generates is almost incomprehensible because it translates the parse tree into a giant switch structure that manipulates Perl structures. Eventual goal is to reduce (given sufficient type information in the Perl program) some of the Perl data structure manipulations into manipulations of C-level ints, floats, etc. Status: 5 (some things work, including uncomplicated Tk examples).

* **B::Lint**

Complains if it finds dubious constructs in your source code. Status: 6 (it works adequately, but only has a very limited number of areas that it checks).

* **B::Deparse**

Recreates the Perl source, making an attempt to format it coherently. Status: 8 (it works nicely, but a few obscure things are missing).

* **B::Xref**

Reports on the declaration and use of subroutines and variables. Status: 8 (it works nicely, but still has a few lingering bugs).

**Using The Back Ends**

The following sections describe how to use the various compiler back ends. They're presented roughly in order of maturity, so that the most stable and proven back ends are described first, and the most experimental and incomplete back ends are described last.

The O module automatically enabled the **-c** flag to Perl, which prevents Perl from executing your code once it has been compiled. This is why all the back ends print:

1. myperlprogram syntax OK

before producing any other output.

**The Cross Referencing Back End**

The cross referencing back end (B::Xref) produces a report on your program, breaking down declarations and uses of subroutines and variables (and formats) by file and subroutine. For instance, here's part of the report from the *pod2man* program that comes with Perl:

1. Subroutine clear\_noremap
2. Package (lexical)
3. $ready\_to\_print i1069, 1079
4. Package main
5. $& 1086
6. $. 1086
7. $0 1086
8. $1 1087
9. $2 1085, 1085
10. $3 1085, 1085
11. $ARGV 1086
12. %HTML\_Escapes 1085, 1085

This shows the variables used in the subroutine clear\_noremap . The variable $ready\_to\_print is a my() (lexical) variable, **i**ntroduced (first declared with my()) on line 1069, and used on line 1079. The variable $& from the main package is used on 1086, and so on.

A line number may be prefixed by a single letter:

* **i**

Lexical variable introduced (declared with my()) for the first time.

* **&**

Subroutine or method call.

* **s**

Subroutine defined.

* **r**

Format defined.

The most useful option the cross referencer has is to save the report to a separate file. For instance, to save the report on *myperlprogram* to the file *report*:

1. $ perl -MO=Xref,-oreport myperlprogram

**The Decompiling Back End**

The Deparse back end turns your Perl source back into Perl source. It can reformat along the way, making it useful as a de-obfuscator. The most basic way to use it is:

1. $ perl -MO=Deparse myperlprogram

You'll notice immediately that Perl has no idea of how to paragraph your code. You'll have to separate chunks of code from each other with newlines by hand. However, watch what it will do with one-liners:

1. $ perl -MO=Deparse -e '$op=shift||die "usage: $0
2. code [...]";chomp(@ARGV=<>)unless@ARGV; for(@ARGV){$was=$\_;eval$op;
3. die$@ if$@; rename$was,$\_ unless$was eq $\_}'
4. -e syntax OK
5. $op = [**shift**](http://perldoc.perl.org/5.8.9/functions/shift.html) @ARGV || [**die**](http://perldoc.perl.org/5.8.9/functions/die.html)("usage: $0 code [...]");
6. [**chomp**](http://perldoc.perl.org/5.8.9/functions/chomp.html)(@ARGV = <ARGV>) unless @ARGV;
7. foreach $\_ (@ARGV) {
8. $was = $\_;
9. [**eval**](http://perldoc.perl.org/5.8.9/functions/eval.html) $op;
10. [**die**](http://perldoc.perl.org/5.8.9/functions/die.html) $@ if $@;
11. [**rename**](http://perldoc.perl.org/5.8.9/functions/rename.html) $was, $\_ unless $was eq $\_;
12. }

The decompiler has several options for the code it generates. For instance, you can set the size of each indent from 4 (as above) to 2 with:

1. $ perl -MO=Deparse,-si2 myperlprogram

The **-p** option adds parentheses where normally they are omitted:

1. $ perl -MO=Deparse -e 'print "Hello, world\n"'
2. -e syntax OK
3. [**print**](http://perldoc.perl.org/5.8.9/functions/print.html) "Hello, world\n";
4. $ perl -MO=Deparse,-p -e 'print "Hello, world\n"'
5. -e syntax OK
6. [**print**](http://perldoc.perl.org/5.8.9/functions/print.html)("Hello, world\n");

See [B::Deparse](http://perldoc.perl.org/5.8.9/B/Deparse.html) for more information on the formatting options.

**The Lint Back End**

The lint back end (B::Lint) inspects programs for poor style. One programmer's bad style is another programmer's useful tool, so options let you select what is complained about.

To run the style checker across your source code:

1. $ perl -MO=Lint myperlprogram

To disable context checks and undefined subroutines:

1. $ perl -MO=Lint,-context,-undefined-subs myperlprogram

See [B::Lint](http://perldoc.perl.org/5.8.9/B/Lint.html) for information on the options.

**The Simple C Back End**

This module saves the internal compiled state of your Perl program to a C source file, which can be turned into a native executable for that particular platform using a C compiler. The resulting program links against the Perl interpreter library, so it will not save you disk space (unless you build Perl with a shared library) or program size. It may, however, save you startup time.

The perlcc tool generates such executables by default.

1. perlcc myperlprogram.pl

**The Bytecode Back End**

This back end is only useful if you also have a way to load and execute the bytecode that it produces. The ByteLoader module provides this functionality.

To turn a Perl program into executable byte code, you can use perlcc with the -B switch:

1. perlcc -B myperlprogram.pl

The byte code is machine independent, so once you have a compiled module or program, it is as portable as Perl source (assuming that the user of the module or program has a modern-enough Perl interpreter to decode the byte code).

See **B::Bytecode** for information on options to control the optimization and nature of the code generated by the Bytecode module.

**The Optimized C Back End**

The optimized C back end will turn your Perl program's run time code-path into an equivalent (but optimized) C program that manipulates the Perl data structures directly. The program will still link against the Perl interpreter library, to allow for eval(), [**s///e**](http://perldoc.perl.org/5.8.9/functions/s.html), [**require**](http://perldoc.perl.org/5.8.9/functions/require.html), etc.

The perlcc tool generates such executables when using the -O switch. To compile a Perl program (ending in .pl or .p):

1. perlcc -O myperlprogram.pl

To produce a shared library from a Perl module (ending in .pm):

1. perlcc -O Myperlmodule.pm

For more information, see [perlcc](http://perldoc.perl.org/5.8.9/perlcc.html) and [B::CC](http://perldoc.perl.org/5.8.9/B/CC.html).

**Module List for the Compiler Suite**

* **B**

This module is the introspective ("reflective" in Java terms) module, which allows a Perl program to inspect its innards. The back end modules all use this module to gain access to the compiled parse tree. You, the user of a back end module, will not need to interact with B.

* **O**

This module is the front-end to the compiler's back ends. Normally called something like this:

* 1. $ perl -MO=Deparse myperlprogram

This is like saying [**use**](http://perldoc.perl.org/5.8.9/functions/use.html) O 'Deparse' in your Perl program.

* **B::Asmdata**

This module is used by the B::Assembler module, which is in turn used by the B::Bytecode module, which stores a parse-tree as bytecode for later loading. It's not a back end itself, but rather a component of a back end.

* **B::Assembler**

This module turns a parse-tree into data suitable for storing and later decoding back into a parse-tree. It's not a back end itself, but rather a component of a back end. It's used by the *assemble* program that produces bytecode.

* **B::Bblock**

This module is used by the B::CC back end. It walks "basic blocks". A basic block is a series of operations which is known to execute from start to finish, with no possibility of branching or halting.

* **B::Bytecode**

This module is a back end that generates bytecode from a program's parse tree. This bytecode is written to a file, from where it can later be reconstructed back into a parse tree. The goal is to do the expensive program compilation once, save the interpreter's state into a file, and then restore the state from the file when the program is to be executed. See [The Bytecode Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Bytecode-Back-End) for details about usage.

* **B::C**

This module writes out C code corresponding to the parse tree and other interpreter internal structures. You compile the corresponding C file, and get an executable file that will restore the internal structures and the Perl interpreter will begin running the program. See [The Simple C Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Simple-C-Back-End) for details about usage.

* **B::CC**

This module writes out C code corresponding to your program's operations. Unlike the B::C module, which merely stores the interpreter and its state in a C program, the B::CC module makes a C program that does not involve the interpreter. As a consequence, programs translated into C by B::CC can execute faster than normal interpreted programs. See [The Optimized C Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Optimized-C-Back-End) for details about usage.

* **B::Concise**

This module prints a concise (but complete) version of the Perl parse tree. Its output is more customizable than the one of B::Terse or B::Debug (and it can emulate them). This module useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

* **B::Debug**

This module dumps the Perl parse tree in verbose detail to STDOUT. It's useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

* **B::Deparse**

This module produces Perl source code from the compiled parse tree. It is useful in debugging and deconstructing other people's code, also as a pretty-printer for your own source. See [The Decompiling Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Decompiling-Back-End) for details about usage.

* **B::Disassembler**

This module turns bytecode back into a parse tree. It's not a back end itself, but rather a component of a back end. It's used by the *disassemble* program that comes with the bytecode.

* **B::Lint**

This module inspects the compiled form of your source code for things which, while some people frown on them, aren't necessarily bad enough to justify a warning. For instance, use of an array in scalar context without explicitly saying [**scalar(@array)**](http://perldoc.perl.org/5.8.9/functions/scalar.html) is something that Lint can identify. See [The Lint Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Lint-Back-End) for details about usage.

* **B::Showlex**

This module prints out the my() variables used in a function or a file. To get a list of the my() variables used in the subroutine mysub() defined in the file myperlprogram:

* 1. $ perl -MO=Showlex,mysub myperlprogram

To get a list of the my() variables used in the file myperlprogram:

* 1. $ perl -MO=Showlex myperlprogram

[BROKEN]

* **B::Stackobj**

This module is used by the B::CC module. It's not a back end itself, but rather a component of a back end.

* **B::Stash**

This module is used by the [perlcc](http://perldoc.perl.org/5.8.9/perlcc.html) program, which compiles a module into an executable. B::Stash prints the symbol tables in use by a program, and is used to prevent B::CC from producing C code for the B::\* and O modules. It's not a back end itself, but rather a component of a back end.

* **B::Terse**

This module prints the contents of the parse tree, but without as much information as B::Debug. For comparison, [**print**](http://perldoc.perl.org/5.8.9/functions/print.html) "Hello, world." produced 96 lines of output from B::Debug, but only 6 from B::Terse.

This module is useful for people who are writing their own back end, or who are learning about the Perl internals. It's not useful to the average programmer.

* **B::Xref**

This module prints a report on where the variables, subroutines, and formats are defined and used within a program and the modules it loads. See [The Cross Referencing Back End](http://perldoc.perl.org/5.8.9/perlcompile.html#The-Cross-Referencing-Back-End) for details about usage.

**KNOWN PROBLEMS**

The simple C backend currently only saves typeglobs with alphanumeric names.

The optimized C backend outputs code for more modules than it should (e.g., DirHandle). It also has little hope of properly handling **[goto](http://perldoc.perl.org/5.8.9/functions/goto.html)** **LABEL** outside the running subroutine (**[goto](http://perldoc.perl.org/5.8.9/functions/goto.html)** &sub is okay). **[goto](http://perldoc.perl.org/5.8.9/functions/goto.html)** **LABEL** currently does not work at all in this backend. It also creates a huge initialization function that gives C compilers headaches. Splitting the initialization function gives better results. Other problems include: unsigned math does not work correctly; some opcodes are handled incorrectly by default opcode handling mechanism.

BEGIN{} blocks are executed while compiling your code. Any external state that is initialized in BEGIN{}, such as opening files, initiating database connections etc., do not behave properly. To work around this, Perl has an INIT{} block that corresponds to code being executed before your program begins running but after your program has finished being compiled. Execution order: BEGIN{}, (possible save of state through compiler back-end), INIT{}, program runs, END{}.

#!/bin/bash

#

# Install example script and nightly cronjob

#

target\_script=example.sh

#

# Write out example script to node

#

mkdir -p /etc/config/scripts

cat << 'EOF' > "/etc/config/scripts/$target\_script"

**#!/bin/bash**

**me=$(basename $0)**

**echo "Example script ran successfully at $(date)" > "/tmp/$me.log"**

EOF

#

# Optional: install cronjob to run nightly at 1am

#

(

crontab -l | grep -v "$target\_script"

echo "**0 1 \* \* \* bash /etc/config/scripts/$target\_script**"

) | crontab -

=====================================================================================<config>

<ports>

<port1>

<label>Cisco Switch</label>

...

</port1>

...

</ports>

...

</config>

Display complete usage/help:

config -h

Display (get) the entire configuration (this does not display all possible IDs, only those that are set):

config -g config

Display a subsection of the configuration tree:

config -g config.ports.port1

Set a value for a configuration ID (validation is not enforced when using the CLI, take care to set appropriate values):

config -s config.ports.port1.label="Cisco Switch"

Delete an ID/value pair:

config -d config.ports.port1.label

Delete an entire subsection:

config -d config.ports.port1

Run a configurator to apply settings:

config -r serialconfig

Run all configurators to apply settings:

config -a

Hints & tips

– You can chain config commands, this can be significantly faster than running separate commands as config.xml is only read/written once:

config -s config.ports.port1.label="Cisco Switch" -s config.ports.port1.speed=9600 -r serialconfig

– The usual [bash escaping caveats](https://linux.die.net/Bash-Beginners-Guide/sect_03_03.html) apply when operating at the CLI, so take care to appropriate escape special characters such as space and $ with single quotes, double quotes or backslashes.

– You will find examples of various configuration tasks throughout this knowledge base, e.g. [network interface setup](https://opengear.zendesk.com/hc/en-us/articles/216371643-Configuring-the-network-interface-or-resetting-the-IP-address-from-the-command-line-) and [serial port setup](https://opengear.zendesk.com/hc/en-us/articles/216373283-Default-serial-port-settings-and-enabling-ports).  However, sometimes the simplest way to discover how to accomplish a configuration task via CLI is to try it through the web UI and see what changed:

* Save "before" config

config -g config > /tmp/config.old

* Set and apply changes via the web UI
* Save "after" config

config -g config > /tmp/config.new

* Compare before and after

diff /tmp/config.old /tmp/config.new

– You can queue all configurators to run next boot, by running:

touch /etc/config/.run\_configurators

reboot

– You can "get" configuration transformed into properly escaped "set" format using the following command:

config -g config | awk '{ printf "config -s '\''%s", $1; $1=""; gsub(/'\''/, "'\''\\'\'''\''"); printf "=%s'\''\n", substr($0, 2); }'

– Some config subsections are lists that have a "total" item, that must be updated to reflect new or deleted items.  The "List Operations" are provided to automatically handle this:

prefix=$(config --list-base config.firewall.portrules --list-add)

config -s $prefix.name="New Firewall Rule" -s $prefix.action=accept # and so on

– Standard bash *for* and *while* loops come in handy to automate repetitive configuration tasks:

# Set serial ports 2, 5, 10, 11 and 23 to Console Server mode  
for PORTNUM in 2 5 10 11 23; do config -s config.ports.port$PORTNUM.mode=portmanager; done  
config -r serialconfig

# Set all serial ports to Console Server mode  
TOTAL=$(cat /var/run/serial-ports | wc -l)

PORTNUM=0

while [[ $(( PORTNUM++ )) -lt $TOTAL ]]; do config -s config.ports.port$PORTNUM.mode=portmanager; done  
config -r serialconfig

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ssh clients article with ssh connect

 Note: This article provides basic, practical examples for various common SSH clients – for more general information on configuring and accessing the Opengear console server via SSH, refer to this article.

SSH provides a convenient and secure means to access the command line of the Opengear device itself, and its connected serial console ports.

A few popular clients covered by this article:

PuTTY – a freeware implementation for Windows

SecureCRT – a commercial implementation with a free trial for Windows, Mac and Linux

OpenSSH command line client (the ssh command) – an open source implementation typically included with MacOS and Linux distributions

PuTTY for Windows

Download, optionally install, and run the PuTTY executable

Set Connection Type to SSH

Set Host Name (or IP address) to the Opengear device's IP address

Set Port to 22

Click Open

To connect to the CLI of the Opengear itself:

Login as root or a users or admin group user, e.g.:

login as: root

Password: default

Disconnect by running: exit

To connect to a console:

Login by adding :serial to your username, e.g.:

login as: root:serial

Password: default

Type a console port number and hit <enter>

For a complete list of console access conventions including direct access using crafted username and Ports, refer to this article

Disconnect with the following escape sequence: <enter>~.

SecureCRT for Mac

Download, install and run SecureCRT

Set Protocol to SSH2

Set Hostname to the Opengear device's IP address

Set Port to 22

To connect to the CLI of the Opengear itself:

Set Username to root or a users or admin group user

Click Connect

When prompted, enter your user's Password

Disconnect by running: exit

To connect to a console:

Add :serial to your Username, e.g. root:serial

Click Connect

When prompted, enter your user's Password

Type a console port number and hit <enter>

For a complete list of console access conventions including direct access using crafted username and Ports, refer to this article

Disconnect with the following escape sequence: <enter>~.

OpenSSH for Linux

Open a terminal window

To connect to the CLI of the Opengear itself:

Run:

ssh -l root 192.168.0.1

.. where root is a users or admin group user (or root), and 192.168.0.1 is the IP address or hostname or your Opengear device

When prompted, enter your user's Password

Disconnect by running: exit

To connect to a console:

Run:

ssh -l root:serial 192.168.0.1

.. where root is a users or admin group user (or root), and 192.168.0.1 is the IP address or hostname of your Opengear device

When prompted, enter your user's Password

1: Router 4: PDU 6: ISR 8: Switch

33: Front, Upper 34: Front, Lower

Connect to port > 1

Type a console port number and hit <enter>

For a complete list of console access conventions including direct access using crafted username and TCP ports, refer to this article

Disconnect with the following escape sequence: <enter>~.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*