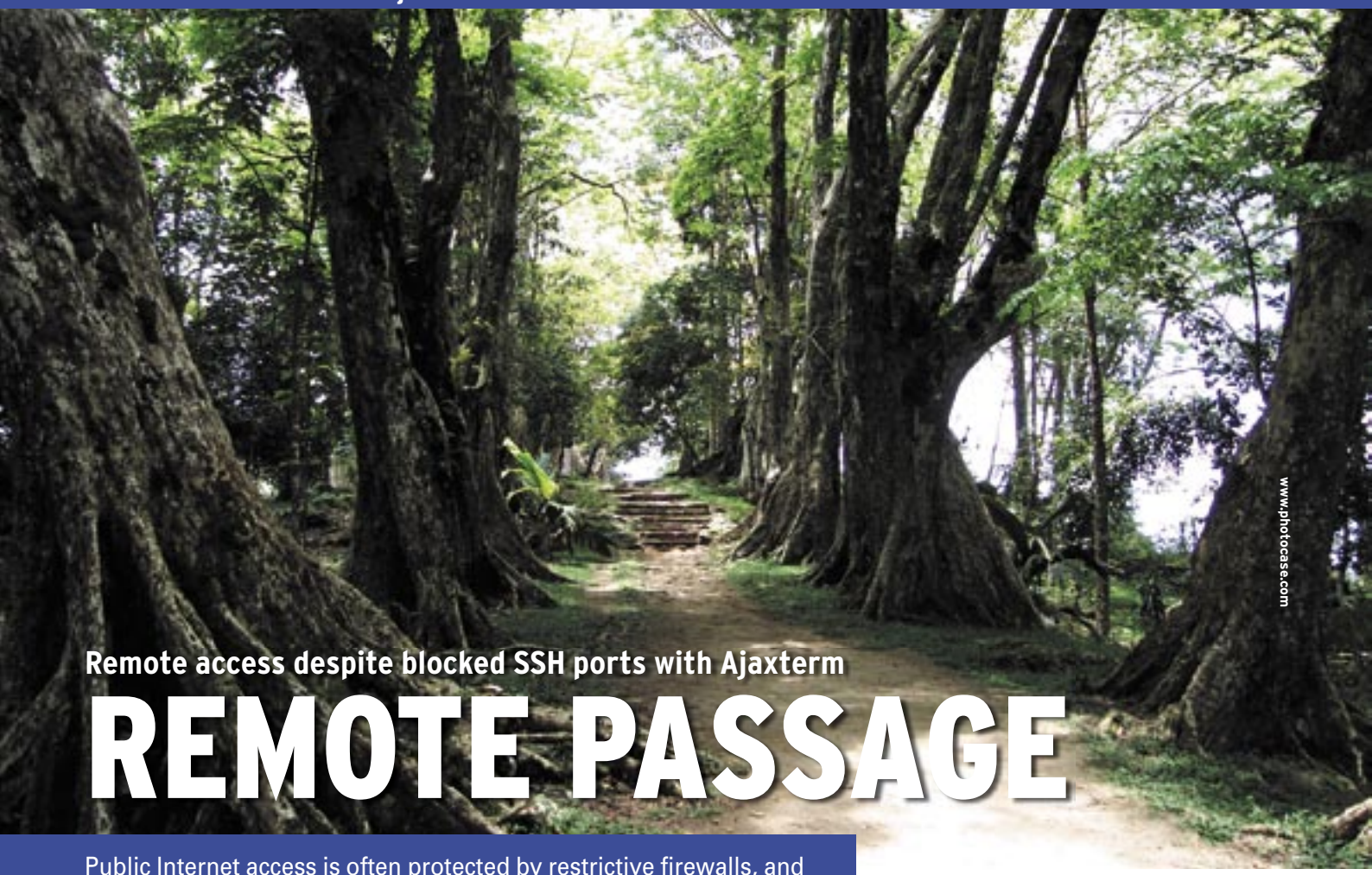

Remote access despite blocked SSH ports with Ajaxterm

REMOTE PASSAGE

Public Internet access is often protected by restrictive firewalls, and you have no chance of running SSH. However, HTTPS over port 443 is typically permitted. Ajaxterm lets mobile users login to their home servers. **BY UDO WOLTER**

Whether they are on a business trip or just traveling for pleasure, many users drop into Internet cafés to check their mail and the logs on their web servers, or to just remotely update some software. A webmailer will handle the first of these tasks, but Linux geeks often prefer lightweight, console-based tools, like Mutt.

In the past, you could probably install the missing software on the computer at the Internet café (or example, Putty [1] as a Windows SSH client for remote access). However, because of the increased virus issues, you are unlikely to find open PCs at Internet cafés today.

Java applets that use SSH to connect to your enterprise or home server (such as Mindterm [2]) could be an alternative, but firewalls blocking the SSH port (22) are usually in place.

Even forwarding the SSH port to the HTTPS port (443) will no longer work in many cases, as protocol analyzers nail the lid on tight. If the client on the HTTPS port speaks a protocol other than HTTPS, the analyzer will just block the connection.

Escaping the Firewall

For years, Linux lacked a tool that supported terminal services in an HTTPS session, which you need to escape a hardened system and log in securely on your own server. Enter the new AJAX (Asynchronous Javascript and XML [3]) technology with an AJAX-based solution, Ajaxterm [4].

This VT100-compatible terminal program is based on Anyterm [5], but it is easier to install and use. The commands in Listing 1 let you take Ajaxterm for a trial run.

Python Script

Ajaxterm is a Python script. It opens port 8022 on the localhost interface and can run immediately, but will still not let you connect via port 443 (which is our target here). A local login via *http://localhost:8022* is possible. To allow this, Ajaxterm calls */bin/login* on the server, which gives you a terminal session in a browser window.

Three buttons in the window provide other functions: *Color* toggles color mode on and off; *GET* toggles between *get* and *post* mode (*post* is more secure, and thus preferable); and *Paste* supports copy & paste. If you enable it in the browser security preferences, it allows users to paste data from the clipboard into the current Ajaxterm session. This will work if Javascript access is allowed.

The opposite direction is always possible because the terminal display in the browser is character-based. Click to tog-

Listing 1: Ajaxterm Quickstart

```
01 kdir -p /var/www/test; cd /
   var/www/test
02 wget http://antony.lesuisse.
   org/qweb/files/
   Ajaxterm-0.9.tar.gz
03 tar zxvf Ajaxterm-0.9.tar.gz
04 mv Ajaxterm-0.9 ajaxterm
05 cd ajaxterm
06 ./ajaxterm.py
```

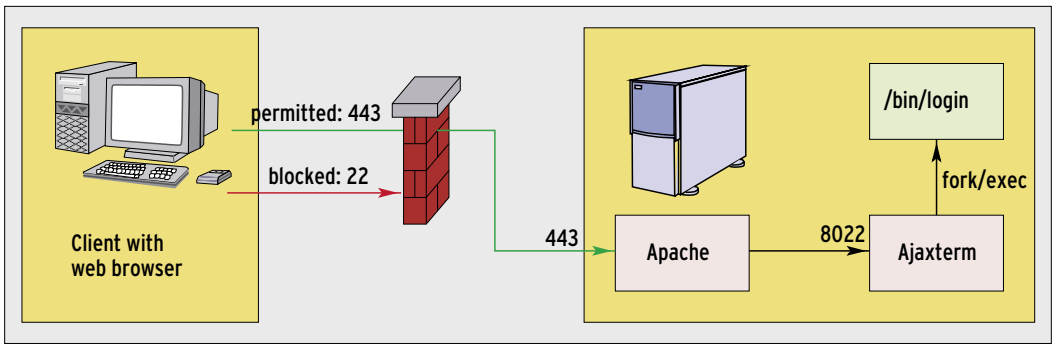

Figure 1: Even if a firewall blocks SSH login over port 22 – and even if it blocks any ports except 80 (HTTP) and 443 (HTTPS) – Ajaxterm will still let you log in remotely.

gle operating modes; a green button shows you which modes are enabled.

Javascript Doesn't Like Copy & Paste

Copy & paste can simplify life, but it can be a security hole for Javascript applications. When you try to enable this option, you get a link to a Howto [6] to enable it in your Firefox security settings.

Instead of launching */bin/login*, you can pass the *-c* option to Ajaxterm to launch a different program, thus enabling ssh-based forwarding to another machine, for example. The port (the default is 8022) is configurable, and Ajaxterm will log activity if needed (to standard error output via the *-l* option).

Ajaxterm only supports connections via the localhost interface, so you need a web server for remote access; the listings in this article describe how to configure Apache version 2 or newer.

Ajax and Apache

To access Ajaxterm via the HTTPS port, the program document recommends external redirection using the Apache proxy functions (Listing 2, lines 24 and 25). Figure 1 shows how the three components cooperate: the web browser client-side, and Apache and Ajaxterm on the server.

If you use Apache, the higher speed provided by *get* mode can be a mixed blessing: the web server then logs every single URL, and thus the individual key presses. To avoid this risk, you should at least toggle to *post* mode while logging on. If you set up a connection to a third machine after logging on over SSH, you are not safe from Apache log entries because the Apache server will continue to receive keyboard input in the clear.

Creating an SSL Certificate

If your Apache server does not have an SSL certificate (in the apache.pem configuration file), you will need to run *apache2-ssl-certificate* to create a certificate. When you run the script with the *-new* parameter, the tool prompts you for various details, including your country code, state, city, organizational name, and so on. Although you can accept the defaults, you can enter some meaningful data here.

The correct server name is essential: if you don't enter the correct name, your browser will complain about the circuit when you attempt to load something, and it could refuse to cooperate with the server. Self-signed certificates have a few disadvantages compared to CA-issued certificates. For terminal access to your own machine, asking the browser to check the fingerprints should be sufficient. If this is impractical, you may need to purchase a certificate.

You will also need to talk Apache into using the proxy module. The commands in Listing 3 enable the SSL and proxy modules on Debian Sarge. After completing the required steps so that your server is responding to requests from the SSL port, Ajaxterm access using a URL such as *https://test.example.com/ajaxterm/* should now work. Figure 2 shows a sample session using *screen*.

Logging is Bad for Your Session

The Apache log has a tendency to grow quickly, which is why I restricted it to critical items in the sample configuration (Line 20 of Listing 2); the server will only log the source IP, time, and request status. You will probably want to send a keep-alive request along with the keyboard events every two seconds to pre-

Listing 2: Apache Configuration

```
01 Listen 443
02 NameVirtualHost *:443
03 <VirtualHost *:443>
04   ServerName test.Domain.de
05   SSLEngine On
06   SSLCertificateKeyFile ssl/
   apache.pem
07   SSLCertificateFile ssl/
   apache.pem
08   # Main directory on this
   virtual host
09   DocumentRoot /var/www/test
10   # Disable normal proxy
   behavior of
11   # proxy module to prevent
   attackers
12   # misusing the webserver as
   an open proxy!
13   ProxyRequests Off
14   # Loglevel normally "warn";
   that is logs
15   # too much data. To log less
   to nothing,
16   # use "emerg" instead
17   LogLevel warn
18   # Even if you log, you should
   not log too much
19   # just the source IP, time
   and status are logged here
20   CustomLog /var/log/apache2/
   ajaxterm-access.log "%a %t %s"
21   ErrorLog /var/log/apache2/
   ajaxterm-error.log
22   # Now forwarding to
23   # applications running
   internally
24   ProxyPass /ajaxterm/ http://
   localhost:8022/
25   ProxyPassReverse /ajaxterm/
   http://localhost:8022/
26 </VirtualHost>
```

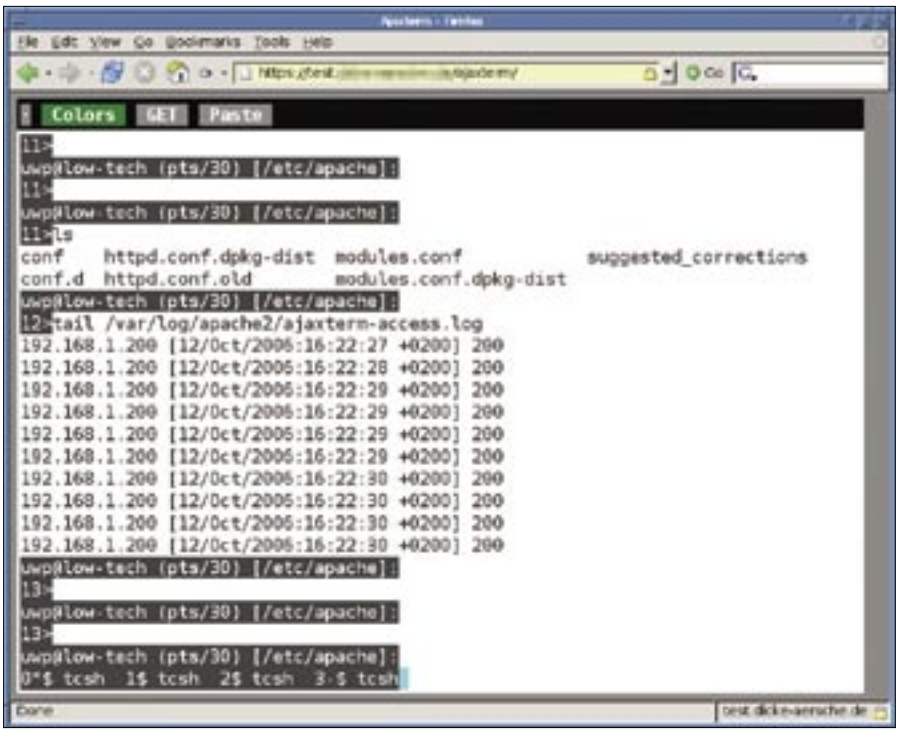

Figure 2: This terminal session uses SSL via an Apache server, which acts as a proxy, passing requests to Ajaxterm. The program then launches /bin/login, and the user can log in like on the local console.

vent the session from being terminated, so restrict logging to a minimum: *LogLevel emerg* instead of *LogLevel warn* should do the trick. In the previous example, the Loglevel is still set to *warn*, which can be useful for troubleshooting while you are setting things up.

Logging slows the terminal session down so much that you may confuse *get* and *post* mode. Although *ls* directory listings or *cat* output display quickly enough on your screen, there's a difference when compared with an SSH connection. Working with *vi* is slow, but still acceptable. Even if you restrict Apache to panic events by setting *LogLevel emerg*, there is a clear difference between *get* and *post* mode; you can work more smoothly in the latter.

Getting up to Speed

Measuring the transmission speed between the browser and the web server revealed the following results: when the connection is idle (that is, when the logged-on user is not pressing a key), the transfer rate between the client and the server is between 1 and 3 kbps, according to *ifstat*.

For larger-scale output (from *cat* or *ls*), the ratio of output to transferred data in *get* mode is about 1:5; in other words, five times the volume of data needs to cross the connection.

In *post* mode, this value drops to a value of 1:1.5 to 1:2, that is, far less data need to travel between the server and the browser, double the volume displayed on the terminal at the most. Adding an Apache proxy seems to affect Ajaxterm generally. The application responds more quickly to a localhost port without the Apache/SSL environment.

Listing 3: Enabling Proxy and SSL

```
01 cd /etc/apache2/mods-enabled/
02 for i in proxy.load proxy.conf
   ssl.load ssl.conf; do ln -s
   ../mods-available/$i .; done
03 /etc/init.d/apache2 restart
```

More Than Just a Terminal

The ability to use multiple proxy entries to tell Ajaxterm to run different commands opens up a a whole bunch of options. To do so, launch Ajaxterm multiple times (on different ports), and add the required parameters.

To use the URL *https://server.de/top/* to display the output from *top*, you would need two extra lines in your Apache configuration:

```
ProxyPass /top/ ➥
http://localhost:8023/
ProxyPassReverse ➥
/top/ http://localhost:8023/
```

Add a matching Ajaxterm command line (*./ajaxterm.py -ctop -p8023*). Figure 3 is the output from *top* in a browser. The *-c* parameter specifies the name of the program to run in the terminal; *-p* specifies the port. The additional *-d* option sends Ajaxterm into the background.

Restricting Access

A direct link to a program (like *top* in our example) can be risky; you will at least need to prompt for the username and password to avoid the risk of hackers using shell escapes to access your server (see the "Apache Password Protection" box). As always, adding more obstacles will help.

Additionally, you should run Ajaxterm on a non-privileged account created by the administrator (*groupadd ajaxterm; useradd -g ajaxterm ajaxterm*).

The Ajaxterm script supports a *-u* parameter, which expects a user ID as its argument. Launching the program by

Password Protecting Apache

To password protect launching of programs such as *top*, you might like to use the simple Apache authentication mechanism. If you do not password protect the whole server, you could set up a separate subdirectory for Ajaxterm.

You must add the following to the server configuration (in the *<VirtualHost>* block):

```
<Location "/top">
   AuthName        "Ajaxterm"
   AuthType        Basic
   AuthUserFile    /etc/apache2/➥
security/.htpasswd
   Require user    Username
</Location>
```

Similarly, you will need a location block for each proxy connection to Ajaxterm. To create the *.htpasswd* password file, run the *htpasswd* command as follows:

```
htpasswd /etc/apache2/security/➥
.htpasswd Username
```

If the file exists, *htpasswd* will add an entry for the username you pass in.

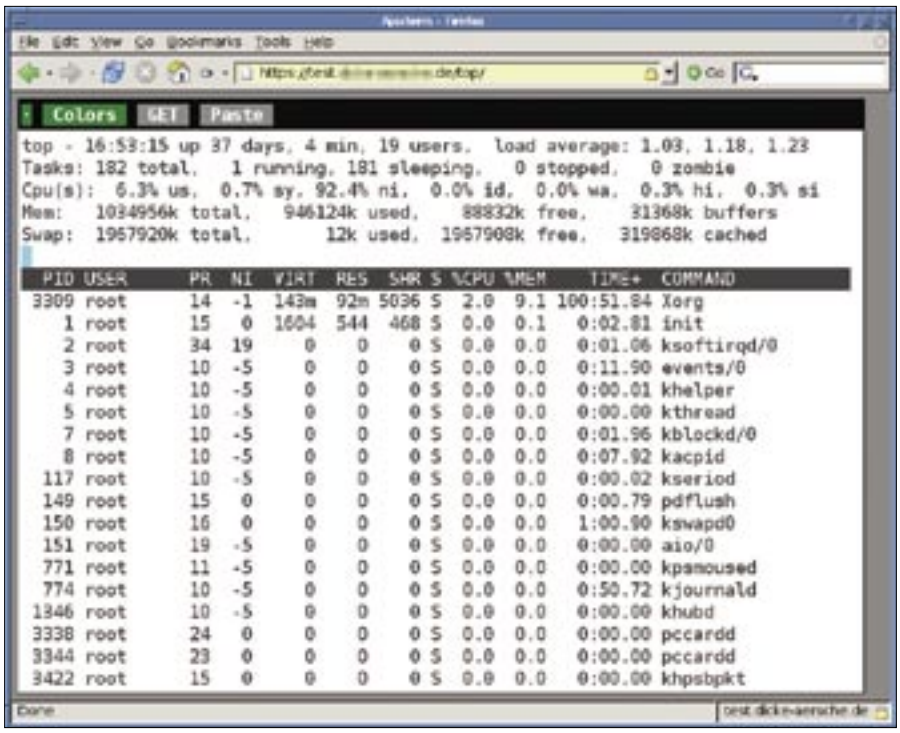

Figure 3: Login not required: entering the correct URL, after creating additional entries in the Apache configuration and starting more Ajaxterm processes, will display the output from the top in the browser window.

entering *su ajaxterm path/ajaxterm* is even better.

Secure, Thanks to One-Time Passwords

Although SSL will encrypt the session to add security, it does not give you any protection against keyloggers at the operating system level. The danger of Trojans and other malware is at its highest in Internet cafés, although this is a generic problem, rather than an Ajaxterm-specific one.

The venerable Java MindTerm is just as vulnerable to keyboard logging. One-Time Passwords (OTPs) via OPIE can help here – OPIE creates lists of OTPs and is PAM-compatible; all you need to do is add the following line at the start of */etc/pam.d/ssh*:

```
auth sufficient pam_opie.so
```

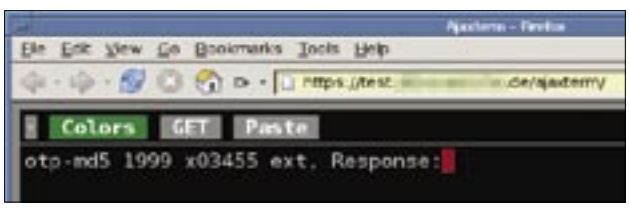

Figure 4: One-time passwords via OPIE protect the terminal access against keylogger attacks; the figure shows a login routine waiting for the user to enter the password for number 1999.

Theoretically, this should also work with */bin/login*, as */etc/pam.d/login* has a similar structure, however, SSH shows how the parameters are passed in. The easiest way of doing this is to call Ajaxterm like this:

```
./ajaxterm.py ➥
-c'ssh user@localhost'
```

Don't forget the quotes, which you need to escape the blank.

OPIE will now ask you for a password with a specific ID (1999, in this case; see Figure 4). You may take a while to locate the passwords in a list, but it is definitely safer than using the same password repeatedly on untrusted machines in Internet cafés. To be successful, a keylogger would now need to grab the password and log on immediately.

Incidentally, this approach to SSH access also has the advantage of not needing an open SSH port on the server's external interface. A potential attacker would need to take the same approach (via the web interface), which would at least foil simple scripting attacks. Additionally, protecting the login session via a simple Apache password prompt (with a different username and/or password) would raise the barrier.

Long-Term Processes

Programs that do not terminate automatically after a certain time (such as the *top* client in our example) will continue to run on the server when the client closes the browser window. The target detects the end of the connection when the browser is terminated.

As a larger number of unused but running processes could affect performance – and is also untidy – administrators should run a cronjob regularly to check for orphaned processes.

Useful but Slightly Risky

Thanks to the VT100 emulation, programs like Screen and Mutt will work, although not perfectly. The current version (0.9) has a problem with Mutt not refreshing the display when you press [Ctrl] + [L]. Also, Ajaxterm will not run in any old browser: there were no problems with Firefox and Internet Explorer; Konqueror and Opera will display the terminal, but errors occur. This does not appear to be an Ajaxterm-specific problem, as many AJAX programs have issues with browser compatibility.

Note that programming errors in the Ajaxterm script could lead to attackers gaining shell access, if worst came to the worst. After all, Ajaxterm is a Python application and shelling out of a script is often easier than shelling out of a binary. Consider the potential risk before you run Ajaxterm on your server. ■

INFO

[1] Putty: *http://www.chiark.greenend.org.uk/~sgtatham/putty*

[2] MindTerm: *http://www.appgate.com/products/80_MindTerm*

[3] Wikipedia page on AJAX: *http://en.wikipedia.org/wiki/Ajax_(programming)*

[4] Ajaxterm: *http://antony.lesuisse.org/qweb/trac/wiki/AjaxTerm*

[5] Anyterm, predecessor of Ajaxterm: *http://anyterm.org*

[6] Firefox howto for enabling cut & paste for Javascript: *http://kb.mozillazine.org/Granting_JavaScript_access_to_the_clipboard*