A Different Kind of Remote Scripting

Published in **2600**, Autumn 2008

by Atom Smasher
atom@smasher.org
pgp = 762A 3B98 A3C3 96C9 C6B7 582A B88D 52E4 D9F5 7808

Don't trust anyone who smokes marijuana and votes for Bush. I've had friends who have smoked plenty of pot, and I've known a few people who voted for Bush that I can get along with, but anyone who does both is bad news. Stay away.

Before I go into too much detail, there are four pieces of this puzzle that we should get familiar with. If you've been using *nix for any length of time you may already be familiar with them. Alone, they allow some neat tricks, but together they can be used for a different kind of remote scripting.

For extreme convenience, it's highly recommended that these techniques are used with ssh public key authentication. If you use this from a cron job it's a necessity. Do a web-search if you're not familiar with it; there are a lot of tutorials on the web so I won't go into detail here. I will point out that not only is public key authentication more convenient than typing a pass-phrase, it's more secure against certain attacks.

In the following examples the assumption is that you're logged into a box running *nix, and have ssh access to a second box running *nix. For those on a tight budget, or otherwise restricted in access to resources, you can play along with two dumpstered computers (or two Windoze boxes, after hours), some Ubuntu CDs and a switch, router or hub. If you don't have a switch/router/hub, a crossover ethernet cable can work.

Trick 1 - SSH can execute remote commands

On my laptop I can execute the 'uptime' command and see output like this:

```
% uptime
 9:54PM up 2:07, 0 users, load averages: 0.08, 0.14, 0.10
```

But what if I want to quickly run 'uptime' on a remote server? Of course I could log in via ssh and type "uptime". Another way to do this is to specify 'uptime' as an argument to ssh:

```
% ssh atom@suspicious.org uptime
 9:54pm up 77 days 11:29, 7 users, load average: 0.00, 0.00, 0.00
```

To quote from the [OpenSSH] man page for ssh, "If command is specified, it is executed on the remote host instead of a login shell." That's a neat trick. The example above shows a simple command, without options or arguments, but with proper quoting this can also be used for complex commands (pipelines, lists, control operators, etc) or scripts.

Trick 2 - Command interpreters (shells) can read commands from standard input

```
% echo uptime | sh
 9:55PM up 2:08, 0 users, load averages: 0.05, 0.12, 0.08
```

We can pipe things into sh and they will be executed by the shell. You don't think that's exciting? OK, maybe it isn't, in that example. This can also be done with other shells (bash, zsh, ksh) and applications that can be used as command interpreters (perl, php, python, etc). In the following four examples, we can use a command line interface to pipe simple commands into different applications' standard input. The output of all four of these examples is "Hello World."

```
% echo 'puts "Hello world."' | ruby
% echo 'print "Hello World."' | python
% echo 'print "Hello World.\n";' | perl
% echo '<? echo("Hello World.\n"); ?>' | php
```

While the rest of the examples all use Bourne shell, the above examples demonstrate that you can use these techniques with just about any interpreted language.

Trick 3 - SSH can pass data through standard input, output and error

```
% echo uptime | ssh atom@suspicious.org sh
 9:57pm up 77 days 11:32, 7 users, load average: 0.00, 0.00, 0.00
```

What's actually happening there is the local machine is echoing 'uptime' into a pipe, the pipe is passing it to the standard input of the 'ssh' command, which passes it to 'sh' (again, on standard input) on the remote machine where it is executed (by 'sh', on the remote machine) and the standard output is displayed locally. Actually, if we're piping into ssh we don't have to specify the 'sh' as a command; if we leave it out, the input (the 'uptime' command, in the above example) will be executed by the default login shell. The two reasons I include the 'sh' explicitly are 1) it's unambiguous and 2) I tend to write scripts that are specific to different shells, so it's good to specify which shell to use.

```
% ssh atom@suspicious.org 'echo foobar' | rot13
sbbone
```

In the above example, the remote machine echos "foobar" to standard output. That output comes to the local machine, where it's piped through rot13. What I see on my console is the output "foobar" (from the remote machine) which is rot13 encoded on the local machine.

```
% ssh atom@suspicious.org 'echo foobar 1>&2' | rot13
foobar
```

In this example, the remote machine echos "foobar" to standard error. Although the output is still being piped into rot13, the output is not encoded because rot13 is only encoding standard output from the ssh command; the ssh command here outputs "foobar" on standard error, and it is not encoded.

Standard output and standard error can be treated independently on one machine, even when the command is executed on another machine.

Trick 4 - SSH will return with the exit status of the last command it executes

```
% ssh atom@suspicious.org 'ls foobar'
ls: cannot access foobar: No such file or directory

% echo $?
2
```

Assuming that there isn't a file in my home directory (on the remote machine) called "foobar", the 'ls' command will exit with a non-zero return-status, and ssh will return that status to me, locally.

Note that the command argument to ssh is in single quotes. This example would work without quotes, but it's good practice to use quotes. For anything beyond simple commands it becomes necessary to use quotes. It's also worth noting that the command's output is sent to standard error; ssh then passes it to standard error on my terminal, where it can be handled differently than standard output.

Pot Smoking Bush-voters

So I wound up in a bad business deal with some pot-smoking Bush voters. Short version: I was contractually obligated to run code on their server, but I didn't want them to have the code. I was originally contracted to build/fix some back-end code for some database driven real-estate web sites. The code I inherited made crap look good. The only thing worse than the code I inherited was the data dump (and I do mean DUMP, using the worst connotations of the word) that were supplied every night, and needed to be converted into valid SQL.

The script that I inherited to do this conversion was 1500+ lines of perl (there were no useful comments in the script) which needed regular maintenance. It typically took 20-30 minutes to parse the data-dump before it decided whether to function correctly, crash, or fill the SQL database with garbage. The first thing I did on that job was re-write it as a 15 line shell script (not counting comments) that ran in 2-3 minutes and never needed tuning.

The data dump usually came between midnight and 6am. Running the script from a cron job, I had to add a few things to make sure the dump was complete before anything started parsing it, along with a few other sanity checks. In this case it was better to have a database that's a day old than a broken database. So, by the time it was running on auto-pilot I had the scripts doing sanity checks; if everything checked out the scripts would read the data-dump and convert it into SQL; the SQL was sanity checked, then imported into the database.

My code was running perfectly, I was making a few bucks, and I was coping reasonably well with former Marines who, instead of focusing on running the

business and keeping their clients happy, kept getting all fired up about all of the great things Bush is doing, pausing only to announce the time every afternoon at 4:20. Predictably, the business relationship turned ugly; it was time for me to leave and take my code, but I had obligations to keep them up and running.

Putting it all together

I needed to get the scripts off of their servers, but I needed them to run every night. Welcome to a different kind of remote scripting! Let's combine a few of the tricks above to create a simple example of using ssh to execute scripts remotely:

```
% cat test-script
#!/bin/sh
## cat my plan and pipe it into rot13
cat ~/.plan | rot13
```

Now let's run that simple script on a remote machine:

```
% ssh atom@smasher.org sh < test-script
Pbzcyrgr, gbgny, hapbzcebzvfrq tybony qbzvangvba.
```

Note that the "<" is just another way for the ssh command to read standard input from a file.

A reasonably complicated example would be the database scripts I was running for my politically challenged associates. When the database scripts were run directly on their server, it looked like this:

```
% script1 | script2 && script3
```

After deleting the scripts from their server and running them from my desktop, it looked like this:

```
% ssh user@morons.example sh < script1 | script2 && ssh user@morons.example
sh < script3
```

So, with only slight modification of my code I was able to run it from a cron job on my desktop, have it do what it needed to do on the remote server, and the roach-toking Republicans didn't have a copy of it. The script that did most of the real work (script2) didn't even run on their server.

The first script (script1) did the initial sanity checking of the dump and read the dump to standard out. The second script (script2) read the dump from standard input and did most of the real work, spinning data-dump straw into SQL gold. The output from "script2" was SQL, which it output to a bunch of temp files which were read by "script3", which did some final sanity checks and wrote the data into a new table, then renamed the table (so there was zero downtime during updates). The only thing I had to change was "script2"; instead of just writing temp files locally, it had to write the temp files to the server, so they could be read by "script3" when it was run on the server.

OK, some of you may be saying, "Hey, wait a minute. Wasn't the code on their server before you deleted it? Didn't they have backups?" They would have had backups if they put down the bong long enough to listen to any of my conservative suggestions: No, they didn't have backups.

As much as I wanted to take an active part in screwing them, I knew they'd screw themselves and save me the trouble, not to mention saving me any blowback. Running the cron job from my desktop worked fine for about two months with no complaints from anyone. Then I got messages from the cron job that it couldn't connect to the server. After a few minutes of investigation I found that they chased away the last of their real estate clients and took the server off-line. Not only did they screw themselves, they did it right on schedule.

Security considerations & other applications

This technique is useful for running a script on a machine and making it difficult to see the code, but it is not a super-secure way to keep your code from being seen. It would be relatively easy for anyone with root access on the remote machine to "capture" the script being run over ssh, but I'll leave that for someone else to find several ways of demonstrating. In the example above, I was able to do the data processing on my desktop so even if the Buds For Bush intercepted the first and last script in the pipeline, they still wouldn't have the script that does the real work.

These techniques can also be used to hide scripts on servers that scan for executable files. Not only can the script be run remotely, but a copy of the script saved on the remote machine doesn't need to be executable to be piped into a shell and executed. The script also doesn't need to start with hash-bang (Hehe, he said hash. Yeah, hehe. Let's get stoned and vote for Bush) since it's not being invoked directly and doesn't need to provide a full path of the interpreter. If that's not enough to keep your sysadmin frustrated, you can also save an encoded copy of the script, and pipe (Hehe, he said pipe) it through a decoder before piping it into an interpreter.

And if you happen to be a sysadmin, think about using this to run a script on multiple servers, among other helpful tasks. Instead logging in to 20-30 servers to do something "quick and simple", you can run a "for loop" on your desktop that runs the script on each of the servers while you surf the web. Even for something that you normally wouldn't script, this becomes a more appealing option as the number of boxes increases. Something as simple as editing a config file on a few servers looks different when you think about it this way.

I'll leave you with a real-world example that I use regularly. It's a script to block offensive sites (ad-servers) using some of the 3rd party operating systems on a Linksys WRT-54G. I'm currently using it with DD-WRT and it's great for blocking banner ads on all computers connected to my home LAN. I call the script "adblock-wrt54g" and it's run with no arguments from my laptop and desktop. This makes it easy to update the list and instantly "protect" all of the computers on my LAN. The script that's executed on the router is a single-quoted argument to ssh; the list that it's using is piped to ssh's standard input. The router supports public key authentication so I don't have to type a pass-phrase when I run it.

I hope that you've learned something useful, and that you can go beyond my examples to create something useful. Happy hacking!

```
#!/bin/sh

## router IP address
router_ip_address=192.168.1.1

## black-listed domains
blacklist=~/blocked-domains
## the blacklist is a list of domains to block, one per line

## under normal circumstances ##
## do not edit below this line ##

## adblock-wrt54g
## (c) atom@smasher.org, 5 Sep 2006, 1 Mar 2008
## PGP = 762A 3B98 A3C3 96C9 C6B7  582A B88D 52E4 D9F5 7808
## distributed under GPL - http://www.gnu.org/copyleft/gpl.html

## make a backup copy of the original dnsmasq.conf, if a backup doesn't exist
## append the formatted black list to dnsmasq.conf
## kill and restart dnsmasq

awk '{print "address=/"$1"/127.0.0.1"}' < ${blacklist} |          \
  ssh root@${router_ip_address} '[ -f /tmp/dnsmasq.conf.orig ] && \
  cp /tmp/dnsmasq.conf.orig /tmp/dnsmasq.conf ||                  \
  cp /tmp/dnsmasq.conf /tmp/dnsmasq.conf.orig
  cat - >> /tmp/dnsmasq.conf &&                                   \
  kill -9 $(cat /var/run/dnsmasq.pid) &&                          \
  /usr/sbin/dnsmasq --conf-file /tmp/dnsmasq.conf'
```