SSH Security and You - /bin/false is *not* security

Posted Wed, 28 Dec 2005

Backstory

Table of Contents

While at RIT around 2004 or 2005, I discovered that a few important machines at the datacenter allowed all students, faculty, and staff to authenticate against them via ssh. Everyone's shells appear to be set to /bin/false (or some derivative) on said machines, so the only thing you'll see after you authenticate is the login banner and your connection will close. I thought to myself, "Fine, no shell for me. I wonder if port forwarding works?"

Seems reasonable, right? Whatever sysadmin was tasked with securing these machines forgot something very important about ssh2: channels. I use them often for doing agent, x11, or port forwarding. You probably use them too, right? So what happens if we try to port forward without requesting a shell (*ssh -N*)? You might not have guessed that it allows you to do the requested port forward and keeps the connection alive. SSH stays connected because it never executes the shell, so it never gets told to die. Whoops!

ITS (RIT's "we make the network go" department) uses ssh.com's sshd on all (afaik) of their servers. I looked at the sshd_config manual for ssh.com's sshd. It clearly defines an option exactly the same as OpenSSH's sshd: AllowGroups. This allows you to restrict ssh-authenticatable users by group. What does that mean? It means you can put all the users who need to ssh to your machines into a single group and prevent unauthorized users from authenticating (getting a shell, port forwarding, etc). So what any intelligent sysadmin would do in this situation is use the AllowGroups option and fix this fairly major security issue.

RIT's ITS has fixed the issue on their own machines. Have you?

In summary: If you don't want me to have access to your machines, then don't allow me access to your machines. **/bin/false is not security**.

What is /bin/false?

Many times you will have a system where you need a user to exist in the account database (say, /etc/passwd) but don't want to give them shell access to your machine(s). A common solution to this is to set a user's shell to `/bin/false`. This has the effect of rejecting shell login attempts over ssh, telnet, or other shell-requesting protocols. It may have other side effects too, but those are beyond the scope of this article.

Simply using `/bin/false` as someone's shell does not keep them from using said account to authenticate over ssh and using non-shell tools such as port forwarding. A default configuration in sshd will often allow tunneling and other non-shell activity.

Hole #1: Potential Firewall Bypass

So, to make things more interesting, there are two obvious holes I can exploit here. The first, is firewall-bypass. ITS employs lots of ACLs limiting access to machines by IP ranges. This is a normal practice in the world. However, what if the machine I am port forwarding through is one of these trusted machines? You just gave me access to your supposedly locked-down network. Don't do that.

Hole #2: Anonymous traffic

I can make my traffic far more anonymous by using ssh's port-forward or SOCKS proxy feature. OpenSSH does not appear to log port-forward-only sessions, so chances are you can get away with using this half-secured server as a proxy. I haven't done all the research, but ssh port-forward-only sessions only seem to show up in process listings and not standard audit logs. This stuff needs to be logged if you're going to allow it.

Hole #3: Resource Starvation

The third one is less obvious, but quite easy. You setup a remote port forward (*ssh -R*) pointed at "itself" (the machine you're logging into) and then a local port forward (*ssh -L*) to the machine so you can just touch it with telnet and walk away. This creates a large problem on the end machine becuase you will eventually take up all the available file descriptors, and since unix lives on file descriptors, you just DoS'd the machine. So if some naughty person manages to guess a password of one of your 30000 users, he/she can happily perform resource starvation attacks 'till the end of the day despite your wishes that I stay off your machine. Like I said, **/bin/false is not security**.

Example DoS

I used 3 xterms for this (wow, high-tech!). I could've used one shell, but I like seeing debug output.

```
(terminal 1) whack% ssh -vN -L4141:localhost:4141 kenya
(terminal 2) whack% ssh -vN -R4141:localhost:4141 kenya
(terminal 3) whack% telnet localhost 4141
```

I use *ssh -v* because I want to see what's going on. As soon as I execute the telnet command, the other two terminals are flooded with debug information detailing new port forwarded connections happening, etc. Since you've just created a loop, you can now kill the telnet session and the loop maintains it's stability.

Simply wait a few minutes and you'll fill up the openfiles table.

Search this site
Google Custom Search

Metadata
Home
About
Resume
Code Repository
Subscribe / Feed
Twitter (@jordansissel)

Articles
ARP Security
Dynamic DNS with DHCP
OpenLDAP & Kerberos
PPP over SSH
SSH Security: /bin/false
Week of Unix Tools
Work Efficiency

Projects
logstash
eventmachine-tail
ruby minstrel
ruby-sshkeyauth
python-rightscale
s3cleaner
syslog-shipper
grok
keynav
liboverride
fex
sysadmin advent calendar
xdotool
pam_captcha
firefox tabsearch
firefox urledit
newpsm (FreeBSD)
nis2ldap
poor man's backup
Solaris audio utility
xboxproxy
xmlpresenter
xpathtool
misc scripts

Presentations
Yahoo! Hack Day '06
Yahoo! Hack Day '08
MPICH 101
Hacking/Security 101
RDF/Semantic Web
Unix Essentials
Vi/Vim Essentials
SSH Tunneling (Video)

Why does the DoS work?

How is this DoS accomplished? Well besides whoring CPU some and slowly increasing in memory usage, you chew up entries on the system's open file table. During this test, I checked the growth of the number of open file descriptors on kenya (the "target" of our DoS):

```
kenya(~) [1003] % while :; do sysctl kern.openfiles; sleep 1; done
kern.openfiles: 242
kern.openfiles: 242
kern.openfiles: 278
kern.openfiles: 652
kern.openfiles: 896
kern.openfiles: 1082
kern.openfiles: 1246
```

The number was stable at 242 before the attack began, and rose rapidly. If you don't speak bourne, the numbers are printed one per second. So the increase is something like 100-300 file descriptors per second. That is quite significant, and will very quickly hose a host. Notably, there is a rapid decelleration in the number of files opened per second, but it steadies (for me) around 20-30 per second after about 7000 open sockets.

Fixing It

The systems in question were old old Solaris 7 or Tru64 systems. Modern systems will generally have a pam module that will help you here - possibly allowing you to reject authentication requests over ssh simply because the user's shell is set to /bin/false, or /usr/sbin/nologin (or wherever that is on your system).

Other solutions include fixing your sshd config to do any of the following (assumes OpenSSH and possibly SSH.com sshd):

- Restrict which users are allowed via AllowUsers or AllowGroups
- Deny tunneling/forwarding: AllowTcpForwarding, X11Forwarding, PermitTunnel

Mentionable Notes

ssh sessions not requesting shells (*ssh -N*) do not show up in utmp, therefore are not listed in w(1) output, and not in last(1), etc. Unless there's a more in-depth audit log, you just made your traffic atleast somewhat anonymous (assuming you're actually port-forwarding). So go ahead and abuse that.

Chance are, if someone gives you an account and tries to prevent ssh access with only /bin/false, they probably don't know about ssh2 or channels, or haven't thought about how it might not prevent all ssh access. So there's a high probability that you'll get away with it forwarding traffic or DoS'ing the server.

Link to this post | posted at: 00:00