# Avoid Complacency with Familiar Tools: Exploring The Shellshock Vulnerability

Joe Kamibeppu

December 14, 2016
Mentor: Professor Ming Chow
COMP116: Introduction to Computer Security

**Abstract**

Command-line shell interface users will run many tools during a given session. In addition to the default tools for basic tasks such as changing directories, transferring files, and creating new files, users run tools supplied by external packages. Users often install a package without verifying its safety and security. Being complacent about the security of familiar tools will make users less alert. The Shellshock vulnerability found in Bash is a prime example of a vulnerability in a familiar tool. This paper will provide an overview of Shellshock, and provide solutions on how to avoid becoming complacent with digital tools.

If an existing tool is replaced by a copy of the same tool with added malicious content, it may be very difficult for users to detect malicious activity caused by that tool. This paper will examine this concern using the `ssh-keygen` tool as an example. `ssh-keygen` is a tool found in the OpenSSH suite on base UNIX systems. It generates a public and private RSA key, and saves the keys on the user's system. As a proof-of-concept, I will provide a demonstration of `ssh-keygen` that executes malicious deeds instead of generating RSA keys.

**Supporting Material**

Supporting material for this paper is located in the GitHub repository below.

https://github.com/joekamibeppu/ssh-keygen-impostor

**Introduction**

We all have tools we use every day. In the physical world, these tools range from pencils that students use to take notes, to heavy machinery being operated at construction sites. It is often the case that as we continue to use such tools and become familiar with them, we become complacent about the safety and security of these tools. In the physical world, it is easy to detect such threats because they are often visible. For example, a bicycle with loose brakes or a flat tire posits a safety risk. We notice these risks quickly because they are obvious. Even if we were not paying attention to the status of the tool, the problem is identified and fixed quickly because we can "see" the problem. Moreover, tools with higher security risks such as heavy machinery often have a maintenance schedule to maintain safety. In the physical world, security issues are often visible, which allow early discovery and resolution to a problem. Moreover, a security issue often inhibits some functionality of the tool. For instance, if a bike has a flat tire, it will likely not work at all until it is repaired.

The software realm suffers from a lack of visibility. Since software cannot be seen as a physical entity, users are forced to treat it as an intangible entity. Users cannot see the entirety of the program, and must "trust" it to an extent. As Ken Thompson points out in "Reflections on Trusting Trust", however, a default assumption of safety creates a security gap (Thompson). It is often difficult to find all security deficiencies just by using the program. As long as the functionality is intact, users do not have much reason to suspect any issues with the program.

The user could run static and dynamic analyses to look for vulnerabilities, or compare the checksum, but it is hard for users to analyze all tools they come into contact with. Users that interact with many tools on a daily basis do not have the time to verify the security of all of the tools they use. Moreover, if the tool works as intended, there is less of a perceived need to spend extra time investigating any vulnerabilities. Unless a compiling error or some other problem that hinders functionality occurs, why would a user bother to correct or report that problem? I believe the lack of visibility downplays the severity of a given problem. It is clear that we treat physical and digital tools with different levels of urgency, even if digital tools are as important - if not more important - than some physical tools.

One common digital tool for developers is the Bash shell. Bash is perhaps one of the most common command processors today. Since Bash is the default shell on many operating systems, it is in an especially vulnerable position to be overlooked as a tool. Moreover, Bash is a shell; how many developers know how to test for a shell's security? Finally, Bash includes many libraries maintained externally, such as OpenSSH. Even if a user were to assume that the local Bash session is safe, how can Bash guarantee the safety of its external components that it does not manage itself? Bash has in fact encountered a number of security issues, with Heartbleed and Shellshock being the most significant.

This paper and its supporting material attempt give the reader a stronger intuition into the severity of vulnerabilities by providing an example of an implementation-level analysis of Shellshock bug. This paper includes a sample implementation of a vulnerability, which will have users execute code with unwanted consequences. In particular, this paper will revolve around a

command line tool that mimics the Bash `ssh-keygen` command provided by OpenSSH, that performs malicious activities.

**To The Community**

As a developer, I use many tools during a given session. I have often found myself trusting software packages without questioning their security, often being complacent with the software's popularity or ratings to justify its security. In order to make myself and the reader more aware and attempt to make a vulnerability more relatable to everyone. I selected `ssh-keygen` because it is a command that most developers have used but has many of its details abstracted away. The command is used to generate a SSH public key and private key, and save those keys to their respective files. While abstractions are helpful in maintaining a high-level workflow, they can also deter users from paying attention to the details of a command. Namely, most command line tools through user input and tool output. If correct tool output is misconceived as proper functionality, it is a problem. Moreover, pervasive issues such as Shellshock affect millions of users. The sheer scale of the problem should yield an alarm.

**Shellshock and Heartbleed**

The Heartbleed bug is a vulnerability in the OpenSSL library, which is used in Bash for data encryption. According to Synopsys, Heartbleed is not caused by a design flaw but rather by an implementation error: "[Heartbleed] is an implementation problem i.e. programming mistake in popular OpenSSL library that provides cryptographic services such as SSL/TLS to the applications and services" (Synopsys). OpenSSL Security Advisory summarizes the bug as "a

missing bounds check in the handling of the TLS heartbeat extension," which exposes memory from a client or server (OpenSSL). Heartbleed was first entered into the Common Vulnerabilities and Exposures (CVE) database on December 3rd, 2013 (CVE-2014-0160). Heartbleed continues to pose a vulnerability to OpenSSL users running certain versions.

Shellshock is another prominent vulnerability seen in Bash. Unlike Heartbleed, the cause of bug was in Bash itself. An vulnerability arises when certain characters in a Bash script are misinterpreted by Bash, and incorrectly executes arbitrary commands. The original form of Shellshock was first entered into the CVE database on September 9th, 2014 (CVE-2014-6271). Immediately after its disclosure, the Shellshock vulnerability was used to conduct distributed denial-of-service (DDoS) attacks. Akamai Technologies and the US Department of Defense were some of the first victims of Shellshock-based attacks (Saarinen).

Even after discovery, it is hard to comprehend the severity of most vulnerabilities. A high-level understanding of a bug or vulnerability can give a user an overview of the bug, but a high-level understanding can often make an issue less relatable. For significant vulnerabilities, it is important for users to understand how vulnerabilities directly relate to their lives. I believe this contributes to a perception gap between a vulnerability's perceived effect and actual effect. The following demonstration provides a low-level analysis of a sample bug.

**Demonstration of Shellshock-Like Vulnerability**

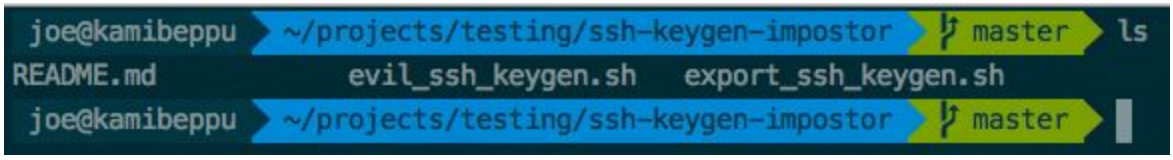
This section of the paper refers to supporting material located in a GitHub repository. In order to enhance your understanding through this section, it is recommended to follow along

using a local instance of Bash on your computer. Please follow the instructions on the repository's Readme to set up your local instance.

The Shellshock vulnerability causes users to execute arbitrary commands. This section aims to have the reader walk through an example of unintended code execution in a Bash environment. While this demonstration is not a replication of Shellshock, it is an attempt to have the reader gain intuition behind arbitrary code execution.

The OpenSSH library in Bash contains the the `ssh-keygen` command which generates public and private SSH keys for a user. In this example, the existing `ssh-keygen` command will be replaced with a malicious function of the same name. Please follow the steps below to walk through the demonstration.

1. Clone and switch into the repository. The contents of the repository should be identical with the screenshot below.




2. Run this command:

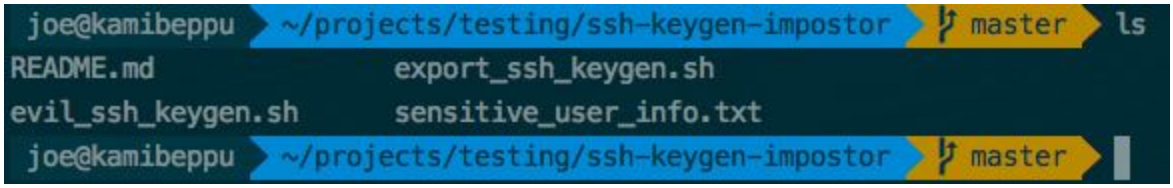
   ```
   $ ssh-keygen
   ```

   You should be prompted to create a pair of public and private SSH keys. This is the standard `ssh-keygen` command. We will replace this command with the next steps.

3. Run this command:

   ```
   $ source export_ssh_keygen.sh
   ```

The contents of your directory should now contain a new text file.

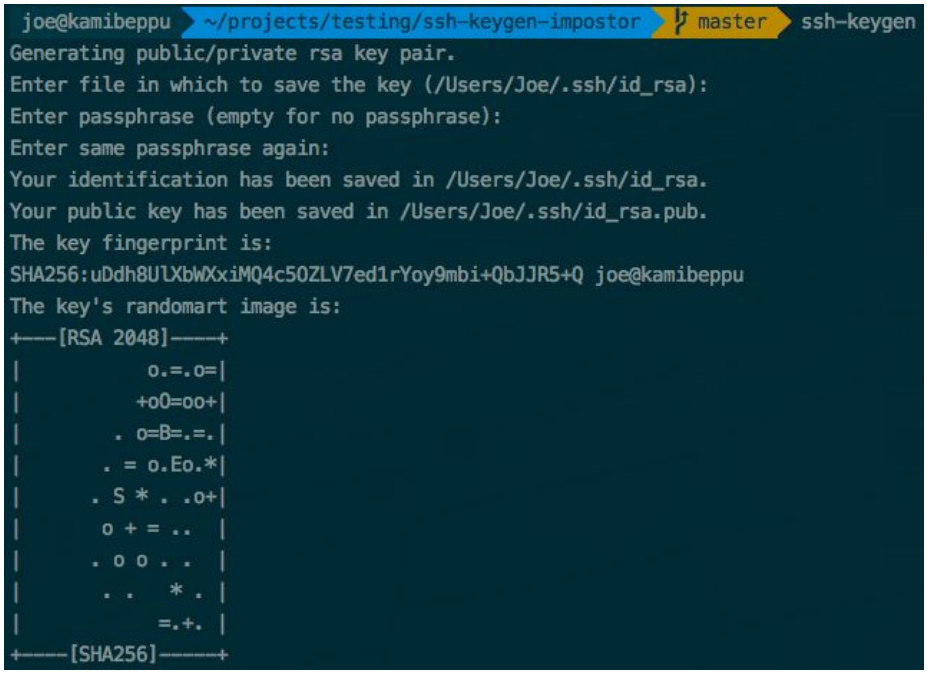



4. Run this command:

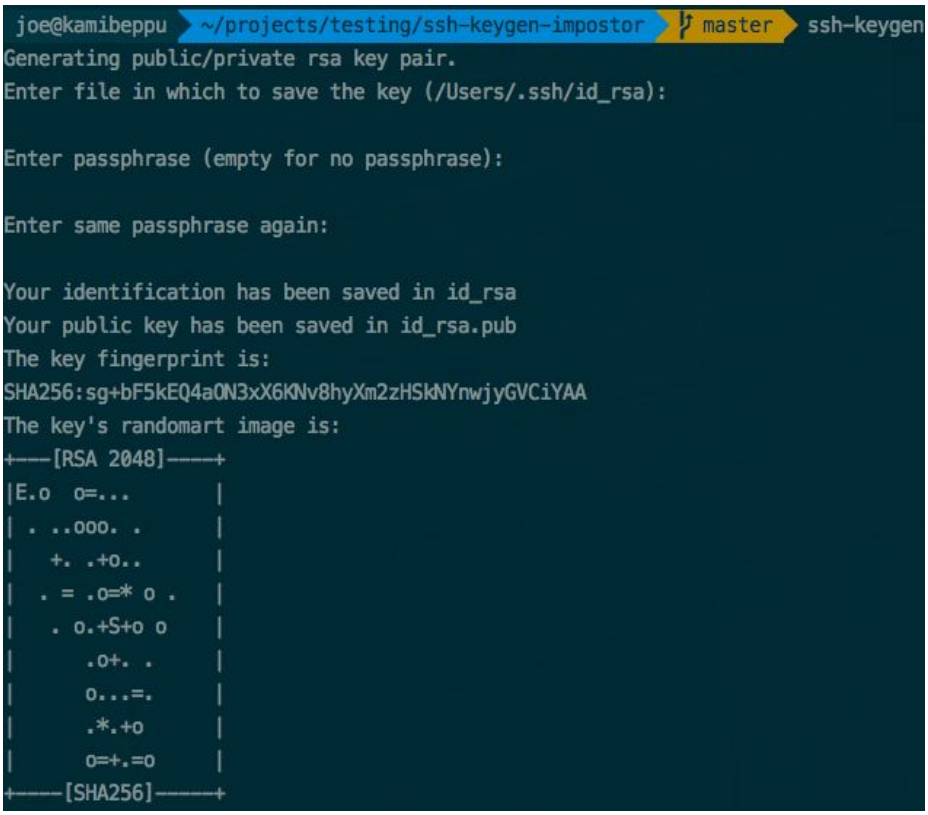
   ```
   $ ssh-keygen
   ```

   You should be prompted to create a pair of public and private SSH keys. The interaction between the user and program is almost identical with the original version of `ssh-keygen`. There may be some trivial discrepancies, but the malicious version is similar enough to avoid suspicion. For comparison, here are the two user-program interactions side-by-side.

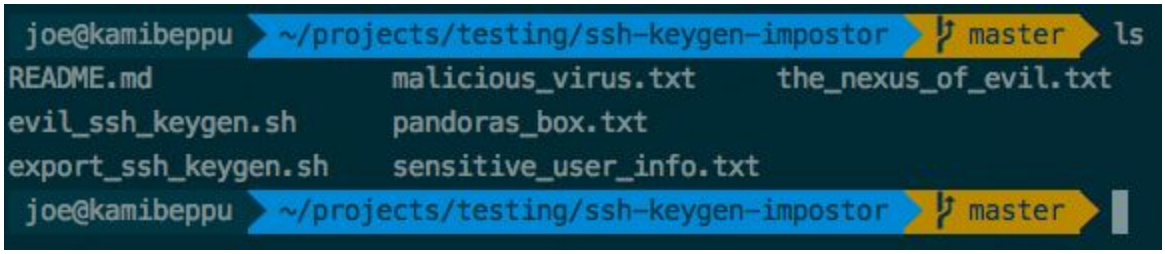
Original command




Malicious command



5. After executing the malicious command, there will be a number of new files in your current directory. These files are stand-ins for potential viruses, worms and backdoors. While they are harmless text files (with content in each one), in theory they could be replaced with harmful files.




6. Inspect the contents of `sensitive_user_info.txt`. If you specified a passphrase when "generating" your RSA keys, that passphrase has been stored in that text file. This demonstrates a large security concern because you may have used sensitive information (eg. old passwords, notable phone numbers) as your passphrase.

7. Once you are done, close the local session to restore original settings.

This demonstration illustrated how malware could covertly execute unwanted commands in a Bash environment and collect sensitive user data. Since this demonstration relies on the assumption that the attacker is able to get such malware onto a user's Bash session in the first place, it can only stand as a proof-of-concept. However, the Shellshock bug allowed attackers to execute arbitrary commands in some vulnerable Bash environments. It is not far-fetched to say that exploiting the Shellshock bug could have enabled some attackers to force their victims to download malware.

**Shellshock Today**

The original Shellshock vulnerability and its associated bugs entailed six separate entries in the CVE database: CVE-2014-6271, CVE-2014-6277, CVE-2014-6278, CVE-2014-7169, CVE-2014-7186, and CVE-2014-7187. The original issue was reported on September 12, 2014, and the patch to address these vulnerabilities was announced on September 24, 2014 and distributed the next day. It is important to note that the significant Shellshock-based attacks on Akamai and the DoD happened on September 26 - only two days since the initial public disclosure (Saarinen). I could not find reliable resources on whether the planning for the attack began before or after the public disclosure. Regardless, it is terrifying to think that the attacker may have only needed two days to exploit the newly-discovered vulnerability.

While Shellshock has been largely resolved, the extent to which Heartbleed has been resolved is unclear. The patch to address Heartbleed was released on April 7, 2014. More than a month later the patch release, however, John Leyden of *The Register* claimed that 12,035 major websites were still vulnerable to Heartbleed (Leyden). It is possible that some neglected or abandoned websites are still vulnerable to the Heartbleed bug today.

It is important to note that both Shellshock and Heartbleed originated in implementation errors in Bash and OpenSSL. Since human error will not be going away anytime soon, we must account for similar implementation-based vulnerabilities in the future.

**Solutions to User Complacency**

First, users should exercise caution and common sense with all technologies. It is important for users to approach both new and familiar technologies with a critical mindset. One way to avoid complacency is to stay informed on recent security issues and patches. There are

many online news sources on security. I have found Hacker News, Security Week and Threat Post to be particularly informative. If time allots, looking into patch reports to understand low-level details will also be useful. I believe it is important to understand the low-level workings of a vulnerability because it will make the issue feel more relevant.

Second, users should perform checksum comparisons where appropriate. While constant checksum comparisons may be viewed as burdensome, in reality they can performed with one simple command.

**Bibliography**

Leyden, John. “AVG on Heartbleed: It’s Dangerous to Go Alone. Take This (an AVG Tool).” The Register. Accessed on December 10, 2016. http://www.theregister.co.uk/2014/05/20/heartbleed_still_prevalent/

National Cyber Awareness System. “Vulnerability Summary for CVE-2014-6271.” National Vulnerability Database. Accessed on October 31, 2016. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271

National Cyber Awareness System. “Vulnerability Summary for CVE-2014-0160.” National Vulnerability Database. Accessed on December 5, 2016. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160

OpenSSL Security Advisory. “TLS heartbeat read overrun.” OpenSSL. Access on December 8, 2016. https://www.openssl.org/news/secadv/20140407.txt

Ramey, Chet. “Bash Patch Report.” GNU.org. Accessed on October 31, 2016. http://ftp.gnu.org/gnu/bash/bash-4.3-patches/bash43-025

Reed, Thomas. “What does the Shellshock bug affect?” The Safe Mac. Accessed on October 30, 2016. http://www.thesafemac.com/what-does-the-shellshock-bug-affect/

Saarinen, Juha. “First Shellshock botnet attacks Akamai, US DoD networks.” IT News. Accessed on October 31, 2016. http://www.itnews.com.au/news/first-shellshock-botnet-attacks-akamai-us-dod-networks-396197

Synopsys. “The Heartbleed Bug.” Heartbleed. Accessed on December 8, 2016. http://heartbleed.com/