Executing Root Commands in Web Applications While Maintaining Security Best Practices

Cox Vincent, Cooman Nico, Jans Stijn and Vermeulen Gustaaf

Abstract—**These guidelines allow a developer to configure a web application that can securely execute root commands. An overview of the disadvantages and dangers of running a web application as root provides insight into the importance of this topic. This document will provide a secure solution via a daemon, along with additional security measures to optimize a developer's web application environment.**

Index Terms—**Security, web application, root.**

I. Introduction

MOST web applications are connected to the Internet and can be accessed by anyone, including attackers. Unfortunately, a vast majority of web applications are prone to attack vectors like SQL injection, XSS and various other techniques. Running the web application as root aggravates the impact of a breach and gives the attacker full control over the server as a root user. This does not only affect the web application itself but also the entire server, including other services on the machine. This is why security professionals always prefer to run a web server as a regular user instead of as a root user. However, running as a regular user limits some functionality like execution of scripts or root commands via a web application. This paper shall address these limitations by discussing some options to execute privileged commands in a web application as a regular user.

II. Advantages of not running a web server as root

A. Extra layer of security

A lot of web applications and websites suffer from the vulnerabilities listed in the OWASP Top Ten [3]. The Open Web Application Security Project (OWASP) [4] is a worldwide not-for-profit charitable organization that focuses on improving the security of software. Their mission is to make software security transparent, so that individuals and organizations across the world can take informed decisions about how to counter software security risks. The OWASP Top Ten provides a broad consensus about the most critical web application security flaws. A few examples from this list are SQL injection, XSS and broken session management.
While a breach through any of the above techniques can be very dangerous on its own, the risk escalates drastically when the web application is running with root privileges. If an attacker manages to upload a web shell in a web application that is running as root, he or she can browse the entire system of the server and modify files including system files, databases and passwords.
This would also affect all the other services and web applications, because the attacker would be able to access and edit everything.
Running as a regular user would limit the impact of a breach to the web application; the entire system with all its services and applications would not be impacted.

B. Limiting collateral damage caused by bugs

A bug or misconfiguration in an application could possibly delete folders, even those outside the application folder, when the application is running as root. This is not necessarily caused by attackers. The fault can be attributed to a programmer who made a mistake despite having good intentions.
Running as a regular user with the right permissions would prevent deletion or modification of files outside the web application folder.

C. Limiting the impact of zero-day exploits or state hackers

A zero-day vulnerability (also known as zero-hour or 0-day) is a previously undisclosed vulner-

N. Cooman and S. Jans are with The Security Factory, Schelle, Belgium, e-mail: info@thesecurityfactory.be.
G. Vermeulen is with the Department of Electrical Engineering (ESAT) TC, KU Leuven, Technology Campus Geel, Belgium, e-mail: staf.vermeulen@kuleuven.be

ability in computer software that hackers can exploit to adversely affect computer programs, data, additional computers or even a whole network.
It is known as a ‘zero-day’ because once the flaw becomes known, the software’s writer has zero days to plan or suggest any mitigation against its exploitation (for example, by recommending workarounds or by issuing patches).
Especially advanced hackers and state-sponsored hackers have several zero-day exploits in their arsenal. If a service suffers from a zero-day while running as root, the entire machine can be controlled.

III. Executing root commands in a secure and controlled way using a daemon

The most effective and secure method in such a situation is to use a daemon running as root. The web server triggers the daemon to do certain predefined actions as root. This paper focuses on a bash daemon, but the principles are applicable for all languages.

A. Security measurements

The following guidelines must be followed for the daemon concept to be fruitful.

1) Hardcode commands that can be executed: Never get the input from a web application (or normal user account) to be executed as a command via a root account. It may be a tempting option because one does not have to hard code each command. However, if an attacker breaches the web application, he or she can inject commands which will be executed as root. This would make the use of a daemon pointless.

2) Use a case function in the daemon: A case function is a great way to let the daemon know what command needs to be run. This improves the security because commands are limited to a controlled and predefined set of commands.

3) Use escaping and sanitization: A great example demonstrating the importance of this problem involves variables used without quotes in bash [6]. Assume that a developer wants to echo the following string:

```
testString="home/*"
```

To achieve this, the developer uses the following command:

```
echo "$testString"
```

This will output the intended and original string. However, the output is different when the developer uses the same command without quotes:

```
echo $testString
```

This yields a very interesting result:

```
"/home/vincentcox"
```

It displays the user account, which is not what the developer expects. Always use escaping and sanitizing on passed variables in the daemon (and applications in general).

B. Triggering the daemon

”Inotifywait” efficiently waits for file changes using Linux’s ”Inotify” interface. It is very suitable for shell scripts because the package can be installed easily. It can either exit once an event occurs, or continue to execute and output events as they occur. This method gives an almost immediate response when a developer wants to trigger the daemon.
For example, one can create a directory called ‘triggerdaemon’ in the web application file structure. After this, the developer can create a daemon using Inotifywait locked on the new folder. It is possible to write a file to that folder in PHP, thus triggering the daemon. Listing 1 in the appendix shows an example code for a daemon.

IV. Alternatives

A. Using a cronjob

This method may sound very appealing because most beginners are familiar with cronjobs. However, the response time is limited because the smallest interval for a cronjob is one minute. To pass variables towards the root script, one can use a database like MySQL. Remember to escape and sanitize Bash code in the daemon script.

V. Optimisation

A. Whitelist the actions performed by the daemon

This is a very effective security measure because when an injection occurs at the daemon level, the damage is limited to the permitted commands.
To achieve this, a new user has to be created with a sudo configuration. An example with visudo:

```
peter, %operator ALL= /sbin/, /usr/sbin, /
    usr/local/apps/check.pl
```

Do not use this to loosen security by permitting all actions!

B. Chrooting

Chroot is an operation that changes the apparent root directory for the current running process and its children. A program that runs in such a modified environment cannot access files and commands outside the environmental directory tree. This modified environment is called a *chroot jail*.
A developer can limit the daemon to certain directories. This will tighten the overall security and it prevents the daemon from modifying system and configuration files.

C. DOS protection

If an attacker is able to breach the web interface, it means he or she has access to the watchfolder of the daemon and possibly also the database. If the daemon or script gets triggered at a high pace (by writing to the watchfolder or by inserting requests in the database), this would cause a Denial Of Service (DOS). For example, an attacker can trigger the deamon to let the web server restart at a high pace. Each restart command will take the server offline for a few seconds and a high pace of restart commands would take the server offline for a long time. This can be prevented by using tokens and timestamps.
The script or daemon can check if the timestamp from the trigger request is in a range from the current time and if the token is valid. This can be done by counting the trigger requests made in the last X minutes. If this count exceeds a threshold, the admin can be notified. A lot of failed checks indicate that a serious bug has occurred, or that an attacker has breached the web interface and is trying to DOS the system.

VI. Conclusion

Performing root actions triggered by a web application can be very time-consuming and difficult to implement in a secure way. In mission critical systems, these extra security measures are worth the effort because the time to recover from a root breach can be many times higher than the time taken to put these measures in place.
Moreover, name damage and leaked information about clients may also lead to a huge financial impact on a company.

Appendix

Listing 1. Example bash daemon [5].

```
#!/bin/bash
logfile="/var/log/daemon.log"
watchdir_deamon_function1="/var/www/hmtl/
    watchdir_deamon_function1"
trap process_USR1 SIGUSR1
process_USR1() {
echo 'Watchdog aborted! (USR1 SIGNAL)'>>
    $logfile
exit 0
}
deamon_function1(){
while inotifywait --outfile $logfile -e
    create $watchdir_deamon_function1; do
echo "Your function code when triggered">>
    $logfile
done
exit 0
}
me_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}
    " )" && pwd )"
me_FILE=$(basename $0)
cd /
if [ "$1" = "child" ] ; then   # 2. We are
    the child. We need to fork again.
shift
umask 0
exec setsid $me_DIR/$me_FILE refork "$@" </
    dev/null >/dev/null 2>/dev/null &
exit 0
fi
if [ "$1" != "refork" ] ; then # 1. This is
    where the original call starts.
exec $me_DIR/$me_FILE child "$@" &
exit 0
fi
# 3. We have been reforked.
shift
# Spawning the daemons
deamon_function1 &
```

References

[1] I. Ristic, *Modsecurity handbook*. Feisty Duck Limited, 2015.

[2] Cert. Web server security best practices. [Online]. Available: https://www.cert.be/docs/web-server-security-best-practices

[3] OWASP, “Owasp top ten 2013,” *The ten most critical web application security risks*, 2013.

[4] ——. The free and open software security community. [Online]. Available: https://www.owasp.org/index.php/Main_Page

[5] S. answer. Best way to make a shell script daemon? [Online]. Available: http://stackoverflow.com/a/29107686

[6] A. Robbins, *Bash pocket reference*, 2016.