PENTEST REPORT

Bash - System 2 – Root-Me Writeup

CLIENT	REPORT AUTHOR	AUTHOR ROLE	PERIOD
Root-Me	Philippe Bécué	Pen-tester	01/03/2026 - 01/03/2026

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/gQL9wDmWai0

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 2 (App-Script)

Scope Exclusions

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

06

Information Gathering

The **Bash – System 2** challenge is accessed via an SSH connection to the Root-Me infrastructure:

```
ssh -p 2222 app-script-ch12@challenge02.root-me.org
```

Once connected, the shell lands directly in the challenge directory. A detailed listing reveals the two elements at the heart of the exercise:

```
ls -la
total 16
dr-x--x--x 2 app-script-ch12-cracked app-script-ch12  4096 ... .
-rwsr-xr-x 1 app-script-ch12-cracked app-script-ch12  7452 ... ch12
-r-------- 1 app-script-ch12-cracked app-script-ch12    14 ... .passwd
```

The binary `ch12` displays the SUID bit enabled (the `s` in the `-rwsr-xr-x` permissions): regardless of the user executing it, the process will run with the privileges of its owner, `app-script-ch12-cracked`. The `.passwd` file, on the other hand, is protected for exclusive read access by the same owner. Any direct access attempt is immediately rejected:

```
cat .passwd
cat: .passwd: Permission denied
```

The challenge statement provides the complete C source code of the binary — a significant advantage compared to a real penetration test where disassembly or dynamic debugging would be required to reconstruct the program's logic.

07

OSINT

No content has been added to this section yet.

08

Enumeration

Reading the Source Code

The source code provided in the statement is brief, but each line carries crucial information for constructing the exploit:

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

int main(){
    setreuid(geteuid(), geteuid());
    system("ls -lA /challenge/app-script/ch12/.passwd");
    return 0;
}
```

Two elements immediately catch the eye. The call `setreuid(geteuid(), geteuid())` synchronizes the Real UID with the process's Effective UID — we will revisit this mechanism in detail in the Vulnerability Analysis section. The critical instruction is then `system("ls -lA /challenge/app-script/ch12/.passwd")`: the `ls` command is invoked **without an absolute path**. This detail makes the program vulnerable to a PATH Injection attack.

Checking Available Write Space

To implant our payload, we need to find a writable directory. We start with the most obvious:

```
echo "probe" > /tmp/test.txt && echo "Write to /tmp : OK"
Write to /tmp : OK
```

The `/tmp` directory is directly accessible for writing. Unlike the first challenge in the series where `/tmp` was locked and an alternative directory had to be identified with `find / -writable -type d`, here we can work immediately without any additional reconnaissance step.

09

Vulnerability Analysis

PATH Injection on system() call without absolute path

When the C function `system(command)` is called, it does not directly launch the target binary. Internally, it executes the equivalent of:

```
execve("/bin/sh", ["sh", "-c", "ls -lA /challenge/app-script/ch12/.passwd"], environ)
```

The `/bin/sh` shell receives the command as a raw string and resolves it itself. To locate `ls`, it iterates through the directories listed in the `PATH` environment variable from left to right, and executes the first file named `ls` it encounters. However, this `PATH` is **inherited from the attacker's environment** who launched the SUID binary. If the attacker places a directory containing their own executable named `ls` at the beginning of the `PATH`, it is this one that will be resolved instead of the real `/bin/ls`.

A detail that might be confusing: the command called is `ls -lA` and not a simple `ls`. This changes absolutely nothing to the exploitation. The shell first resolves the command name (`ls`), then passes the arguments (`-lA` and the path) to it. Our fake executable named `ls` will indeed receive these arguments, but it can ignore them completely — it will only execute the code we've placed in it.

The ambiguous role of setreuid

For beginners, `setreuid(geteuid(), geteuid())` might seem like a security measure. It is, in fact, the opposite: it's what makes the exploit *reliable*. Explanation. Under bash, an internal protection checks at each startup if the Real UID differs from the Effective UID. If so, bash itself calls `setuid(real_uid)` to abandon elevated privileges — a native defense against SUID binary abuse. `setreuid(geteuid(), geteuid())` neutralizes this protection by aligning the two UIDs to the Effective UID *before* the subshell is invoked. The shell will thus fully inherit the elevated privileges, without triggering any defense mechanisms.

10

Exploitation

Creating the payload in /tmp

The approach taken here is different from simply reading a file from the payload. We will use the fake `ls` to spawn an interactive shell with elevated privileges. This is a technique closer to a real privilege escalation scenario: we first obtain elevated shell access, then exploit it freely.

We create a Python script named `ls` in `/tmp`. The choice of Python illustrates a fundamental point: the nature of the language or interpreter doesn't matter. Bash, Python, Perl, Ruby, a compiled binary — any executable named `ls` would suffice. Only the filename and its execute bit matter:

```
echo '#!/usr/bin/python2' > /tmp/ls
echo 'import os' >> /tmp/ls
echo 'os.system("/bin/sh")' >> /tmp/ls
chmod +x /tmp/ls
```

We verify the content before proceeding:

```
cat /tmp/ls
#!/usr/bin/python2
import os
os.system("/bin/sh")
```

The script imports the `os` module and calls `os.system("/bin/sh")`, which forks a shell with the current process's privileges. When this process is our SUID binary `ch12`, this shell will inherit its elevated privileges.

Injecting into PATH and Triggering

We modify the `PATH` variable by placing `/tmp` at the beginning of the list, before all system directories. The keyword `export` is mandatory: without it, the modification would remain local to the current shell and would not be passed to child processes:

```
export PATH="/tmp:$PATH"
```

We confirm that bash will now resolve `ls` to our payload:

```
which ls
/tmp/ls
```

All that remains is to execute the SUID binary. The `./` prefix is essential under Linux, where the current directory is never included in the `PATH` by default:

```
./ch12
```

The execution chain unfolds automatically: `ch12` runs with the Effective UID of `app-script-ch12-cracked` → `setreuid` aligns the Real UID with the Effective UID → `system()` forks a `/bin/sh` that inherits our modified `PATH` → this shell finds `/tmp/ls` first → Python launches `/bin/sh` → we obtain an interactive shell with elevated privileges:

```
sh-4.2$ id
uid=1222(app-script-ch12-cracked) gid=1089(app-script-ch12) groups=1089(app-script-ch12)
sh-4.2$ /bin/cat .passwd
<hash>
sh-4.2$
```

We use `/bin/cat` with its absolute path to avoid any conflict with our modified `PATH` which is still active in this shell. The flag is displayed in plain text.

11

Post-Exploitation

No content has been added to this section yet.

12

Vulnerabilities Summary Table

PATH Injection on SUID binary

HIGH

APPLICATION

ch12 (SUID binary)

TYPE

Privilege Escalation

MAIN IMPACT

Execution of an interactive shell with the SUID owner's privileges via PATH variable command substitution (ls)

13

Conclusion

This challenge consolidates mastery of **PATH Injection on SUID binaries** while introducing two significant technical variations compared to its predecessor in the series. The complete attack chain can be summarized in six steps:

1. SSH connection and directory inventory → SUID binary `ch12` and inaccessible `.passwd` file
2. Source code reading → identification of `system("ls -lA ...")` with relative resolution of the `ls` binary
3. Verification that `/tmp` is writable — no further reconnaissance needed this time
4. Creation of a Python script named `ls` that spawns `/bin/sh`, made executable via `chmod +x`
5. Injection of `/tmp` at the head of the `PATH` via `export` so our fake `ls` is resolved with priority
6. Execution of `./ch12` → interactive shell with elevated privileges → reading of `.passwd` via `/bin/cat`

Two lessons are worth remembering. Firstly, since `/tmp` is available for writing, it is no longer necessary to conduct reconnaissance for writable directories as in System 1 — this illustrates that the attack surface can vary from one machine to another even with an identical vulnerability pattern. Secondly, the fact that the invoked command was `ls -lA` (with options) had no impact on the exploitation: the options are passed to the resolved binary, but our payload ignores them and executes autonomously.

14

Solution

1. Use absolute paths in all system calls

The primary fix involves replacing the relative command name with its full path. The program then becomes impervious to any manipulation of the `PATH` variable:

```
// Vulnerable
system("ls -lA /challenge/app-script/ch12/.passwd");

// Fixed
system("/bin/ls -lA /challenge/app-script/ch12/.passwd");
```

2. Redefine PATH at the beginning of an SUID program

As a defense in depth measure, any SUID program should reset the `PATH` to a minimal and trusted set before invoking `system()`. Even if the command is specified with an absolute path, this practice prevents side effects on any other present or future calls in the code:

```
putenv("PATH=/usr/local/bin:/usr/bin:/bin");
system("/bin/ls -lA /challenge/app-script/ch12/.passwd");
```

3. Prefer execve() over system()

The `system()` function systematically goes through an intermediate shell that inherits the caller's environment variables (`PATH`, `IFS`, `LD_PRELOAD`, etc.) and introduces an extended attack surface. The `execve()` function directly executes the program without invoking a shell, eliminating this surface:

```
char *args[] = {"/bin/ls", "-lA", "/challenge/app-script/ch12/.passwd", NULL};
execve("/bin/ls", args, NULL);
```

4. Apply the principle of least privilege

If the SUID bit is truly indispensable, elevated privileges should be relinquished as soon as the task requiring them is completed, via `setuid(getuid())`. This practice reduces the exposure window in case of residual vulnerability. In this specific case, the file listing operation does not justify the use of a permanent SUID — an alternative

architectural solution would be to expose this content via a dedicated service running with appropriate privileges, rather than an SUID binary.