lagalopex' crackme cm1 - solution

"Get a working key/keygen.
Allowed are only GPLed-tools.
Patching/Hijacking prohibited ;)"

I – Preview of the binary

```
saudade ~ $ file ./cm1
./cm1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.6.9, dynamically linked (uses shared libs), not stripped
```

First, we can see the executable is a non-stripped dynamically linked ELF binary (its author noticed he wrote it in C/C++). It seems not to be packed.

```
saudade ~ $ ./cm1
Hello saudade, lets see what you've done so far...
```

No keyboard stroke is needed.

However:

```
saudade ~ $ strace ./cm1
...
open("/home/saudade/.key_saudade", O_RDONLY) = -1 ENOENT (No such
file or directory)
exit_group(1)                           = ?
```

It is quite clear that it needs a keyfile in the directory where it has been run, which name is clearly like : `.key_$USERNAME`
We can guess for a validation routine based on this keyfile.

II – Static binary analysis

```
saudade ~ $ gdb -q ./cm1
(no debugging symbols found)
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disass main
```

Let's slowly understand the main() function body

```
Dump of assembler code for function main:
0x080484d4 <main+0>:    lea    0x4(%esp),%ecx
0x080484d8 <main+4>:    and    $0xfffffff0,%esp
0x080484db <main+7>:    pushl  0xfffffffc(%ecx)
0x080484de <main+10>:   push   %ebp
0x080484df <main+11>:   mov    %esp,%ebp
0x080484e1 <main+13>:   push   %edi
0x080484e2 <main+14>:   push   %esi
0x080484e3 <main+15>:   push   %ebx
0x080484e4 <main+16>:   push   %ecx
0x080484e5 <main+17>:   sub    $0x1028,%esp
0x080484eb <main+23>:   call   0x80483e0 <getuid@plt>
0x080484f0 <main+28>:   sub    $0xc,%esp
0x080484f3 <main+31>:   push   %eax
0x080484f4 <main+32>:   call   0x80483a0 <getpwuid@plt>
0x080484f9 <main+37>:   mov    %eax,0xfffefd8(%ebp)
```

We get from these lines a pointer to the user's passwd structure.

```
struct passwd {
    char   *pw_name;       /* user name */
    char   *pw_passwd;     /* user password */
    uid_t   pw_uid;        /* user ID */
    gid_t   pw_gid;        /* group ID */
    char   *pw_gecos;      /* real name */
    char   *pw_dir;        /* home directory */
    char   *pw_shell;      /* shell program */
};
```

The first dword of this structure holds the pointer to the username string used bu the program.

This structure pointer is saved in `0xffffefd8(%ebp)` *ie* `[EBP-1028h]`

```
0x080484ff <main+43>:   pop    %esi
0x08048500 <main+44>:   pop    %edi
0x08048501 <main+45>:   pushl  (%eax)
0x08048503 <main+47>:   xor    %esi,%esi
0x08048505 <main+49>:   push   $0x8048750
0x0804850a <main+54>:   call   0x80483d0 <printf@plt> ; on affiche le message d'accueil
0x0804850f <main+59>:   mov    0xffffefd8(%ebp),%eax
0x08048515 <main+65>:   pop    %ebx
0x08048516 <main+66>:   lea    0xffffefee(%ebp),%ebx
0x0804851c <main+72>:   pushl  (%eax)
0x0804851e <main+74>:   pushl  0x14(%eax)
0x08048521 <main+77>:   push   $0x804877f
0x08048526 <main+82>:   push   $0x1001
0x0804852b <main+87>:   push   %ebx
0x0804852c <main+88>:   call   0x80483f0 <snprintf@plt> ; on génère le nom de fichier
0x08048531 <main+93>:   add    $0x18,%esp
0x08048534 <main+96>:   push   $0x0
0x08048536 <main+98>:   push   %ebx     ; ".key_saudade"
0x08048537 <main+99>:   call   0x8048400 <open@plt>
0x0804853c <main+104>:  add    $0x10,%esp
0x0804853f <main+107>:  cmp    $0xffffffff,%eax ; no file?
0x08048542 <main+110>:  mov    %eax,%edi
0x08048544 <main+112>:  mov    $0x1,%edx
0x08048549 <main+117>:  movb   $0x20,0xffffefd3(%ebp)
0x08048550 <main+124>:  movl   $0x0,0xffffefd4(%ebp)
0x0804855a <main+134>:  jne    0x804863c <main+360>
0x08048560 <main+140>:  jmp    0x8048682 <main+430>
```

We can bet that `<main+430>` *leads to the crackme termination, ie that the validation has failed and that's confirmed:*

```
0x08048682 <main+430>:  lea    0xfffffff0(%ebp),%esp
0x08048685 <main+433>:  mov    %edx,%eax
0x08048687 <main+435>:  pop    %ecx
0x08048688 <main+436>:  pop    %ebx
0x08048689 <main+437>:  pop    %esi
0x0804868a <main+438>:  pop    %edi
0x0804868b <main+439>:  leave
0x0804868c <main+440>:  lea    0xfffffffc(%ecx),%esp
0x0804868f <main+443>:  ret
```

There are few initialisations :

- `0xffffefd3(%ebp) := 0x20 (" ")`
- `0xffffefd4(%ebp) := 0x0`
- `%esi := 0x0`

Let's continue analysis to `<main+360>` :

```
0x0804863c <main+360>:  push   %eax
0x0804863d <main+361>:  lea    0xffffffef(%ebp),%eax
0x08048640 <main+364>:  push   $0x1
0x08048642 <main+366>:  push   %eax
0x08048643 <main+367>:  push   %edi
0x08048644 <main+368>:  call   0x8048410 <read@plt>
; reading one byte from the file, loading it in 0xffffffef(%ebp)
0x08048649 <main+373>:  add    $0x10,%esp
0x0804864c <main+376>:  dec    %eax
0x0804864d <main+377>:  je     0x8048565 <main+145>
; if no byte is read (end of file), we go ahead
0x08048653 <main+383>:  mov    0xffffefd8(%ebp),%edx
0x08048659 <main+389>:  mov    0xffffefd4(%ebp),%ecx
0x0804865f <main+395>:  mov    (%edx),%eax
; %eax = (struct passwd*)->pw_name;
0x08048661 <main+397>:  xor    %edx,%edx
0x08048663 <main+399>:  cmpb   $0x0,(%eax,%ecx,1)
; if byte ptr [eax+ecx] = 0 we won
0x08048667 <main+403>:  jne    0x8048682 <main+430>
0x08048669 <main+405>:  sub    $0xc,%esp
0x0804866c <main+408>:  push   $0x804878a
0x08048671 <main+413>:  call   0x80483b0 <puts@plt>

(gdb) x/s 0x804878a
0x804878a:       "Seems you've got it ;)"
```

We now understand the byte pointed by `%eax+%ecx` *has to be null.*
But `%eax` *is pointing to the username, so* `%ecx` *has just to be equal to the username length to fall on the null terminal char.*
Initially that's not the case, because `0xffffefd4(%ebp)` *has been initialized before to* `0x0`*, so* `%ecx := 0x0`*, and as the username is never empty, we must continue the analysis, in the case where end of file was not already reached, in* `<main+145>`
The aim will be to come back later to those lines with `%ecx := strlen($USERNAME)`

We get to the beginning of the validation algorithm :

```
0x08048565 <main+145>:  mov    0xffffffef(%ebp),%dl
0x08048568 <main+148>:  mov    %edx,%eax
0x0804856a <main+150>:  mov    %dl,0xffffefdf(%ebp)
0x08048570 <main+156>:  sub    $0x2a,%eax
0x08048573 <main+159>:  cmp    $0x5,%al
0x08048575 <main+161>:  ja     0x804867d <main+425>
0x0804857b <main+167>:  movzbl %al,%eax
0x0804857e <main+170>:  jmp    *0x80487a4(,%eax,4)
```

Literraly:

`%dl := the byte read in the keyfile`

This one is saved in `0xffffefdf(%ebp)`

```
%eax := byte read
%eax := %eax - 0x2A
If ( %al > 5 ) goto exit;
else goto [0x80487a4 + %eax * 4]
```

From here, we instantly know that :

- `%eax`, *hence the byte read can only take 6 values which are :* `0x2A, 0x2B, 0x2C, 0x2D, 0x2E,0x2F`
- `0x80487a4` *holds an array of 6 memory addresses*

Actually :

```
(gdb) x/6x 0x80487a4
0x80487a4:      0x0804858f      0x08048585      0x080485bc
0x0804859b
0x80487b4:      0x0804867d      0x080485a5
```

Thus, if the byte read is :

- `0x2A ('*')` *we jump to* `0x804858f`, *ie* `<main+187>`
- `0x2B ('+')` *we jump to* `0x8048585`, *ie* `<main+177>`
- `0x2C (',')` *we jump to* `0x80485bc`, *ie* `<main+232>`
- `0x2D ('-')` *we jump to* `0x804859b`, *ie* `<main+199>`
- `0x2E ('.')` *we jump to* `0x804867d`, *ie* `<main+425>` *(program end)*

- 0x2F ('/') *we jump to* 0x80485a5, *ie* <main+209>

We do not care if the byte is '.', *which would lead to the validation failure, to concentrate on the 5 other possibilities.*

The most interesting case is when the byte is 0x2C (',')

```
0x080485bc <main+232>:  push   %ecx
0x080485bd <main+233>:  lea    0xffffffef(%ebp),%ecx
0x080485c0 <main+236>:  push   $0x1
0x080485c2 <main+238>:  push   %ecx
0x080485c3 <main+239>:  push   %edi
0x080485c4 <main+240>:  call   0x8048410 <read@plt>
; we read a new byte in the file
0x080485c9 <main+245>:  add    $0x10,%esp
0x080485cc <main+248>:  dec    %eax
0x080485cd <main+249>:  jne    0x804867d <main+425>
; if end of file is reached, then we failed
0x080485d3 <main+255>:  movzbl 0xffffffef(%ebp),%eax
0x080485d7 <main+259>:  cmp    %esi,%eax
0x080485d9 <main+261>:  jne    0x804867d <main+425>
; if the read byte is not equal to %esi then we failed
0x080485df <main+267>:  mov    0xffffefd8(%ebp),%eax
0x080485e5 <main+273>:  mov    0xffffefd4(%ebp),%ecx
0x080485eb <main+279>:  mov    (%eax),%edx
0x080485ed <main+281>:  movsbl (%edx,%ecx,1),%eax
0x080485f1 <main+285>:  cmp    %eax,%esi
0x080485f3 <main+287>:  jne    0x804867d <main+425>
; if the #ecx username letter is not equal to %esi then we failed :
; the byte read must be hence equal to this letter
0x080485f9 <main+293>:  inc    %ecx ; otherwise %ecx is incremented
0x080485fa <main+294>:  mov    %ecx,0xffffefd4(%ebp)
0x08048600 <main+300>:  cmpb   $0x0,(%edx,%ecx,1)
0x08048604 <main+304>:  jne    0x804861a <main+326>
0x08048606 <main+306>:  push   %edx
0x08048607 <main+307>:  lea    0xffffffef(%ebp),%eax
0x0804860a <main+310>:  push   $0x1
0x0804860c <main+312>:  push   %eax
0x0804860d <main+313>:  push   %edi
0x0804860e <main+314>:  call   0x8048410 <read@plt>
0x08048613 <main+319>:  add    $0x10,%esp
0x08048616 <main+322>:  test   %eax,%eax
0x08048618 <main+324>:  jne    0x804867d <main+425>
; if the #ecx username letter is null and if we reached end of file,
; we get back to our initial verification and we won
```

```
; on the other hand if end of file is not reached at this instant
; then we failed
0x0804861a <main+326>:  mov    $0xa,%edx
0x0804861f <main+331>:  mov    %esi,%eax
0x08048621 <main+333>:  mov    %edx,%ecx
0x08048623 <main+335>:  xor    %edx,%edx
0x08048625 <main+337>:  div    %ecx
0x08048627 <main+339>:  mov    %eax,%esi
0x08048629 <main+341>:  jmp    0x8048630 <main+348>
```

On those last lines, executed only if `%ecx` *is not equal to the username length, we have :*

`%esi := reminder of %esi / 0x0A`

We see that to increment `%ecx` *to our desired value, we must before do some operations on* `%esi` *to set it to the value of the byte read and to the value of the #ecx username letter .*

```
0x08048630 <main+348>:  mov    0xffffefdf(%ebp),%al
0x08048636 <main+354>:  mov    %al,0xffffefd3(%ebp)
0x0804863c <main+360>:  push   %eax
0x0804863d <main+361>:  lea    0xffffffef(%ebp),%eax
0x08048640 <main+364>:  push   $0x1
0x08048642 <main+366>:  push   %eax
0x08048643 <main+367>:  push   %edi
0x08048644 <main+368>:  call   0x8048410 <read@plt>
```

The value of the last read byte is saved in `0xffffefd3(%ebp)`
We then get back to the code studied before, so we can say we will loop the same way on each byte of the keyfile.

So we have to modify `%esi` *to success. For that, we can now take a look at the 4 other possibilities for the value of the byte read* `(0x2A, 0x2B, 0x2D, 0x2F)`

For `0x2A, %esi := %esi + 5 :`

```
0x0804858f <main+187>:  add    $0x5,%esi
0x08048592 <main+190>:  cmpb   $0x2a,0xffffefd3(%ebp)
0x08048599 <main+197>:  jmp    0x80485af <main+219>
```

For 0x2B, %esi := %esi + 1 :

```
0x08048585 <main+177>:  inc    %esi
0x08048586 <main+178>:  cmpb   $0x2b,0xffffefd3(%ebp)
0x0804858d <main+185>:  jmp    0x80485af <main+219>
```

For 0x2D, %esi := %esi - 1 :

```
0x0804859b <main+199>:  dec    %esi
0x0804859c <main+200>:  cmpb   $0x2d,0xffffefd3(%ebp)
0x080485a3 <main+207>:  jmp    0x80485af <main+219>
```

For 0x2F, %esi := %esi - 5 :

```
0x080485a5 <main+209>:  sub    $0x5,%esi
0x080485a8 <main+212>:  cmpb   $0x2f,0xffffefd3(%ebp)
```

We have then 4 arithmetic operations to modify %esi
However, each comparison between the byte read and 0xffffefd3(%ebp), *which holds the previous byte read, implies another condition :*

```
0x080485af <main+219>:  jne    0x804862b <main+343>
0x080485b1 <main+221>:  inc    %ebx
0x080485b2 <main+222>:  cmp    $0x5,%ebx
0x080485b5 <main+225>:  jbe    0x8048630 <main+348>
0x080485b7 <main+227>:  jmp    0x804867d <main+425>
...
0x0804862b <main+343>:  mov    $0x1,%ebx
0x08048630 <main+348>:  mov    0xffffefdf(%ebp),%al
0x08048636 <main+354>:  mov    %al,0xffffefd3(%ebp)
```

If we use more than 5 consecutive identical bytes in the keyfile, %ebx *will be incremented and we will failed.*

Now that we are aware of the situation, let's synthetise what to do we have to do to success.
We must force %ecx *to be equal to the username length to validate the routine.*
But the only way to modify %ecx *is to increment it when reading a* 0x2C *byte.*
However to aim at this, we must modify %esi *before so that it will be equal to the next byte read and to the #ecx username letter.*

We then have to modify `%esi` *with linear combination of* `+5,-5,+1,-1`*, then to read a* `0x2C` *byte followed by the #ecx letter of the username and to compute that for each letter of the username, knowing at every iteration* `%esi` *is divided by* `0x0A`.

Concretely, in the keyfile, for each char :

```
'*' => %esi := %esi + 5
'/' => %esi := %esi - 5
'+' => %esi := %esi + 1
'-' => %esi := %esi - 1
',' => verification of %esi then %ecx := %ecx + 1
```

The condition of the 5 consecutive identical byte is not a real problem, we just have to add the bytes `'+-'` *or* `'-+'` *or* `'/*'` *or* `'*/'` *each time it could be required. This will not affect the computation of* `%esi` *(both operations will cancel each other).*

III – Writing a keyfile generator

Here comes the C++ source of this crackme keyfile generator.

```
#include <sys/types.h>
#include <pwd.h>

#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(void)
{
        const string username = getpwuid(getuid())->pw_name;
        const string keyfilename = ".key_" + username;
```

```
        cout << "Generating " << keyfilename << endl;

        ofstream keyfile(keyfilename.c_str());
        if ( !keyfile.good() )
        {
                cerr << "error: cannot open " << keyfilename << "
for writing" << endl;
                return EXIT_FAILURE;
        }

        keyfile.seekp(0, ios_base::beg);

        string key = "";
        int n = 0;

        for ( int i = 0 ; i < username.size() ; ++i )
        {
                char c = username[i], d;
                int j,x,y,z;

                d = c - n;
                x = d/25; y = (d % 25)/5; z = (d % 25) % 5;

                for ( j = 0 ; j < x ; ++j ) key += "*****+-";
                for ( j = 0 ; j < y ; ++j ) key += '*';
                for ( j = 0 ; j < z ; ++j ) key += '+';

                key += ','; key += c;

                n = c/10;

        }

        keyfile << key.c_str();
        keyfile.close();
        cout << "Done." << endl;

        return EXIT_SUCCESS;
}
```

```
saudade ~ $ ./cm1_keygen
Generating .key_saudade
Done.
saudade ~ $ ./cm1
Hello saudade, lets see what you've done so far...
Seems you've got it ;)
```

Conclusion

*This crackme is relatively easy because of its code compacity and its absence of protections that could slow the analysis. However it is a nice practice for those who begin into the *NIX Reverse Code Engineering.*
I apologize for any english language mistakes, have fun.

DarKPhoeniX @ French Reverse Engineering Team
darkphoenix@binary-reverser.org