УДК 004.42

Защита Python-программ от анализа и модификации алгоритмов сложных информационно-управляющих систем

© В. А. Хихлов[1], Н. С. Могилевская[2]

[1]*ОАО Радиотехнический институт им. А.Л. Минца*

[2] *Донской государственный технический университет*

Аннотация

Рассмотрены основные методы защиты Python-программ от анализа алгоритмов работы. Такая защита позволяет сохранять интеллектуальные права разработчиков программ, а также практически запрещает модификацию программного кода, что сохраняет его целостность, и тем самым обеспечивает безопасность технических систем, работающих под управлением программного обеспечения, разработанного с использованием языка Python. Для рассмотренных методов защиты проработаны основные технические детали их практической реализации. Проведен сравнительный анализ рассматриваемых методов защиты и сделан вывод о целесообразности применения этих методов в совокупности.

Ключевые слова

Python, шифрование, обфускация, модули расширения, импорт, Cython, интерпретатор, байт-код, декомпиляция, конфиденциальность, целостность, динамическая типизация, статическая типизация, AST-дерево.

Введение

Язык программирования Python обладает множеством достоинств [1], активно используется разработчиками различных программных продуктов, в том числе программных средств, обеспечивающих функционирование сложных технических систем. Основным недостатком языка Python является простота его декомпиляции и статического анализа. Например, существуют утилиты [2-3], которые позволяют восстановить байт-код Python с точностью до названия имен объектов, функций, классов и модулей. Легкость декомпиляции делает программы, написанные на этом языке незащищенными от анализа алгоритмов работы и их модификации, что является прямой угрозой целостности сложных информационно-управляющих систем, функционирование которых зависит от корректности работы программного обеспечения, разработанного на языке Python. Кроме этого легкость декомпиляции может нарушать авторские права разработчиков программ. Таким образом, задача защиты программ, написанных на языке программирования Python от анализа и модификации алгоритмов работы, является весьма актуальной.

В работе рассмотрены три основных метода защиты Python-программ от анализа и модификации алгоритмов работы: шифрование исходных кодов,

трансляция в модули расширения, обфускация. Проведен сравнительный анализ этих методов защиты, сделан вывод о целесообразности применения этих методов в совокупности. Ведется разработка специального программного средства обработки программных кодов Python-программ от анализа и модификации. Применение разрабатываемого программного средства позволит решить проблему защиты целостности программ, написанных на языке программирования Python.

Шифрование

Первый метод защиты Python-программ от анализа и модификации алгоритмов работы состоит в хранении и распространении модулей программ в зашифрованном виде. Расшифрование этих модулей происходит только при вызове их на исполнение, а после выгружаются из памяти. Для реализации механизмов шифрования используется существующий в Python протокол PEP 302 [4]. Данный протокол позволяет добавить в механизм импорта Python пользовательский набор импортеров, которые должны обеспечивать поиск и загрузку некоторых пакетов и модулей.

Рассмотрим процесс осуществления импорта зашифрованных Python-модулей. Протокол PEP 302 позволяет добавить в механизм импорта пользовательский набор импортеров, которые должны будут обеспечить новый поиск и загрузку некоторых пакетов и модулей. Импорт в Python заключается в том, что сначала ищется модуль по имени пакета, а затем, если этот модуль можно обработать, возвращается объект-загрузчик, который может загрузить этот модуль в память. Согласно протоколу импорта, класс, реализующий импорт модулей, должен иметь методы find_module(fullname, path=None) и load_module(fullname). Метод find_module осуществляет поиск модуля с именем fullname, а метод load_module осуществляет загрузку данного модуля в память интерпретатора. Таким образом, для осуществления т.н. расшифровки «на лету» необходимо, чтобы метод load_module осуществлял расшифровку файла пред его загрузкой. Ниже приведен возможный вариант реализации данного метода.

```
import sys
import imp
import  marshal
sys.path.append('C:\Python27\Lib\Projects')
class CryptoImporter(object):
    def __init__(self, root_package_path):
        self.__modules                                          =
self.__find_modules(root_package_path)

    def find_module(self, fullname, path=None):
        if fullname in self.__modules:
            return self
        return None

    def load_module(self, fullname):
        if not fullname in self.__modules:
```

```
            raise ImportError(fullname)
        imp.acquire_lock()
        try:
            #  code
            src = self.get_source(fullname)
            try:
                exec src in mod.__dict__
            except:
                del sys.modules[fullname]
                raise ImportError(filename)
        finally:
            imp.release_lock()
        return mod

    def get_source(self, filename):
        try:
            with file(filename, 'rb') as ifile:
                src = ifile.read()
            encode_src=decrypt(src)
        except IOError:
            src = ''
        return marshal.loads(encode_src)

    def __find_modules(self,root_package_path):
        modules = {}
        #  code
        return modules
```

В список sys.meta_path необходимо добавить пользовательский импортер.

```
sys.meta_path.append(CryptoImporter('module_name')).
```

К достоинствам метода хранения и распространения программ в зашифрованном виде можно отнести отсутствие необходимости учитывать синтаксис языка программирования Python при шифровании, а также практическую невозможность анализа и модификации зашифрованных файлов исходных кодов программы при условии использования надежный методов шифрования. К недостаткам метода шифрования относится возможность перехвата из оперативной памяти расшифрованных модулей, необходимость решения задачи хранения и распространения ключей шифрования, а также увеличение времени работы программы.

Трансляция в модули расширения

Второй метод защиты Python-программ состоит в использовании трансляции кода в модули расширения. В Python имеется возможность подключать внешние библиотеки, написанные на других языках программирования. Если в качестве таких языков выбрать компилируемые языки программирования, например, C или C++, то появляется возможность скрыть детали реализации, т.к. подключаемая библиотека будет представлена в виде бинарного кода.

Трансляцию Python-кода в C-код с последующей компиляцией в библиотеку расширения Python можно осуществить с использованием пакета

Cython, который позволяет производить трансляцию Python-кода в C-код, с последующей компиляцией в библиотеку расширения Python. При трансляции в C-код осуществляется промежуточная трансляция в код на языке программирования Cython. Этот язык, кроме стандартного синтаксиса Python, поддерживает прямой вызов функций и методов C/C++ из кода на Cython; строгую типизацию переменных, классов, атрибутов классов [5].

Приведем пример трансляции Python-кода в C-код с последующей компиляцией в модуль расширения Python. Ниже представлен листинг файла compile.py.

```
from distutils.coreimportsetup
from distutils.extension import Extension
from Cython.Distutils import build_ext
ext_modules = [
Extension("test_mod", ["test_mod.py"])
    ]
setup(
     name='test',
     cmdclass={'build_ext': build_ext},
     ext_modules=ext_modules
)
```

В примере test_mod.py – это файл, который будет транслироваться в модуль расширения. Для трансляции необходимо выполнить следующую команду:

```
python compile.py build_ext --inplace,
```

после чего в директории с модулем test_mod появится каталог, который содержит в себе модуль расширения. В случае Linux-систем он будет иметь расширение .so, в случае Windows-систем расширение .pyd.

Достоинствами метода трансляции исходных кодов Python-модулей в исходный код модулей расширения является, во-первых, отсутствие необходимости учитывать синтаксис языка программирования Python, во-вторых, хранение файлов в виде динамически подключаемых библиотек, анализ алгоритмов работы которых, является весьма затруднительным и требует высокой квалификации программиста. Основными недостатками данного метода являются сложность трансляции больших многомодульных проектов и зависимость корректности работы программы от версии компилятора.

Обфускация

В качестве третьего метода защиты Python-программ рассмотрим обфускацию, т.е. запутывание кода. Обфускация позволяет привести исходный код программы к виду, сохраняющему функциональность, но затрудняющему анализ алгоритмов работы. Так как обычно имена объектов в программном коде несут в себе много информации и позволяют программисту, даже в отсутствии содержательных комментариев, понять для чего предназначен и какие функции должен выполнять тот или иной объект, то примером обфусцирующих преобразований может служить замена «говорящих» имен объектов на случайные и несодержательные. В результате анализа схемы

компиляции Python-кода в байт-код сделан вывод, что работа обфускатора наиболее целесообразна на этапе представления кода в виде AST-дерева. Это представление, во-первых, является довольно простым и понятным, а, во-вторых, его можно транслировать в Python-код без потери функциональности.

Основная идея реализации обфускатора в данной работе состоит в том, чтобы по исходным программным кодам получить AST-представление кода, которое затем анализировать и модифицировать с помощью изменения имен переменных, функций, модулей

Элементами AST-дерева являются объекты, которые описывают абстрактную грамматику языка, описание которой хранится в файле Parser/Python.asdl. Этими объектами могут быть: определение класса или функции (ClassDef, FunctionDef), определение различных операторов языка (For, While, If), выражения и составляющие их элементы (Expr, BinOp, UnaryOp) и т. д. Описание структуры языка программирования Python, представленное в файле Parser/Python.asdl, выполнено с помощью формального языка ASDL (Abstract Syntax Definition Language).

В Python имеется встроенная поддержка работы с AST-представлением кода. Для этого разработчиками Python создан стандартный модуль ast, который содержит в себе:

– классы NodeVisitor и NodeTransformer, которые обеспечивают удобный интерфейс для обхода и модификации AST-дерева соответственно;

– некоторые вспомогательные функции;

– импорт модуля _ast, который содержит в себе определение классов узлов и листьев AST-дерева.

Класс NodeVisitor позволяет обойти AST-дерево, вызывая для искомых объектов дерева соответствующую функцию. Этот класс содержит два метода visit(Node) и generic_visit(Node). По умолчанию, методы, объявленные в классе как self.visit_node_class_name, где node_class_name – имя класса узла, вызываются применительно к соответствующим узлам, иначе, если такого метода не существует, вызывается метод self. generic_visit, который вызывает self.visit для дочерних узлов. Класс NodeTransformer наследует класс NodeVisitor, который позволяет модифицировать узлы. Возвращаемое значение метода visit_node_class_name используется для того, чтобы либо заменить, либо удалить старый узел. Если вернуть значение None, то узел будет удален, иначе он будет заменен на возвращаемое значение.

Ниже представлен пример реализации данного класса.

```
node = ast.parse(source_code)
class Transformer(NodeTransformer):

    def visit_Name(self, node):
        return copy_location(Subscript(
            value=Name(id='data', ctx=Load()),
            slice=Index(value=Str(s=node.id)),
            ctx=node.ctx
        ), node)
new_node = Transformer().visit(node)
```

Приведенный пример показывает, что с использованием AST-дерева можно осуществлять статический анализ исходного кода и его модификацию. Кроме использования в задаче обфускации модификация и анализ AST-дерева позволяет: оптимизировать исходный код; транслировать исходные коды программ в исходные коды других языков программирования; проверять код на логические ошибки и т. д.

Обфускация обладает весомыми достоинствами. Обфуцированные файлы хранятся в виде, сохраняющем функциональность, но затрудняющем анализ. Программа является защищенной постоянно, независимо от того, загружена она в память или нет.

К сожалению, у обфускации есть ряд недостатков. При обфускации невозможно сокрыть все определения, а также невозможно обфусцировать некоторые типы файлов, которые чувствительны к модификации имен объектов и модулей. Это могут быть файлы ORM-модулей или не Python-файлы, которые содержат в себе имена, зависящие от Python-модулей. При организации обфускации необходимо учитывать синтаксис программного кода. Ряд сложностей при обфускации вызывают следующие свойства языка Python: динамическое изменение типа объектов; отсутствие контроля типа передаваемого и возвращаемого значений функции; динамическое изменение структуры объекта и т.д.

Выводы по сравнению методов

Как видно, каждый из рассматриваемых подходов имеет свои особенности, которые не позволяют качественно защитить Python-код от анализа алгоритмов работы в случае использования этих методов по отдельности. В случае с шифрованием это возможность перехвата из оперативной памяти. Обфускация применима не ко всем типам файлов. А трансляция в модули расширения может быть трудоемким и трудноосуществимым процессом в случае с большими и многомодульными программами. В таблице 1 приведена оценка рассматриваемых методов защиты согласно основным критериям.

Таблица 1 – Сравнение методов защиты

Критерий\Метод	Шифрование	Обфускация	Трансляция в модули расширения
Возможность восстановления исходной версии программы	Есть	Нет	Нет
Возможность использования на всех типах файлов	Есть	Нет	Нет
Скорость работы	>исходной	=исходной	<=исходной
Зависимость от версии компилятора	Нет	Нет	Есть

Таблица 1 – Сравнение методов защиты (продолжение)

Критерий\Метод	Шифрование	Обфускация	Трансляция в модули расширения
Необходимость учета синтаксиса	Нет	Есть	Нет

Заключение

В работе показано, что для программ, созданных на языке Python достаточно просто получить исходные коды. В связи с этим возникает задача защиты программ, написанных на языке программирования Python от анализа алгоритмов работы. Данная задача необходима для обеспечения защиты

– целостности программ, написанных на языке программирования Python;

– авторских прав разработчиков программного обеспечения выполненного на языке программирования Python.

В работе рассмотрены основные методы защиты Python-программ от анализа алгоритмов работы. Показано, что использование этих методов защиты по отдельности не целесообразно. Предлагается использование всех рассматриваемых методов защиты в совокупности, где основным методом защиты является обфускация. На основе проведенного анализа ведется разработка по созданию программного средства, защищающего Python-программы одновременно всеми рассмотренными выше методами.

Список литературы

1. Язык программирования Python: учебник / Г. Россум, Ф. Л. Дж. Дрейк, Д. С. Откидач, М. Задка, М. Левис, С. Монтаро, Э. С. Реймонд, А. М. Кучлинг, М.-А. Лембург, К.-П. Йи, Д. Ксиллаг, Х. Г. Петрилли, Б. А. Варсав, Дж. К.Ахлстром, Дж. Роскинд, Н. Шеменор, С. Мулендер / 2001. – 454 с.
2. Декомпилятор байт-кода Python // Хостинг проектов URL: https://github.com/gstarnberger/uncompyle (дата обращения:23.08.16).
3. Декомпилятор байт-кода Python // Хостинг проектов URL: https://github.com/zrax/pycdc (дата обращения:24.08.16).
4. Протокол PEP 302 // Официальный сайт Python URL: https://www.python.org/dev/peps/pep-0302/ (дата обращения: 10.06.16).
5. Официальная страница проекта Cython URL: http://cython.org/ (дата обращения: 11.06.16).

Хихлов Владислав Алексеевич – студент, техник ОАО Радиотехнический институт им. А.Л. Минца, e-mail: hihlow.vladislav@yandex.ru

Могилевская Надежда Сергеевна – канд. техн. наук, доцент кафедры «Кибербезопасность информационных систем» Донского государственного технического университета, e-mail: 89044430127@yandex.ru

Protection Python programs from analysis and modification algorithms for complex information management systems

© V. A. Khikhlov[1], N. S. Mogilevskaya[2]

[1] *JSC Radiotechnical Institute named after academician A. L. mints, Rostov-on-Don*

[2] *Don State Technical University, Rostov-on-Don*

Abstract

Main methods of protection Python-programs from analysis of algorithms are considered. That protection allows saving intellectual property rights of software developers and practically prohibits modification of the software code that saves its integrity and thus provides security of technical systems working under control of software developed using the Python language. For the considered protection methods main technical details of realization were worked out. The comparative analysis of considered protection methods was conducted and the conclusion about the expediency of usage of these methods in conjunction was made.

Keywords

Python, encryption, obfuscation, extension modules, import, Cython, interpretato, byte-code, decompilation, confidential, confidentiality, integrity, dynamic typing, static typing, AST-tree.

Khikhlov V. A. – technician at the JSC Radiotechnical Institute named after academician A. L. Mints, student of Don State Technical University, e-mail: hihlow.vladislav@yandex.ru

Mogilevskaya N. S. – Cand. Sci. (Eng.), associate professor at Department «Cybersecurity information systems» of Don State Technical University, e-mail: 89044430127@yandex.ru