Обеспечение безопасности открытых проектов Python: проблема оценки потенциально разрушительного функционала

С. А. Раковский, Ш. Г. Магомедов

***Аннотация*—Открытое программное обеспечение активно используется повсеместно. К сожалению, у разработчиков до последнего времени не возникало мысли, что за безопасностью используемых ими внешних проектов необходимо следить. Программистов можно понять: хочется верить, что, если исходный код открыт, он уже проанализирован автоматическими средствами, и, если он представляет угрозу безопасности, связанный с ним пакет будет удалён. К сожалению, это не так, и вредоносный функционал может находиться в виде открытого проекта с названием, мимикрирующим под популярный пакет, месяцами. Злоумышленники, добавляющие вредоносные пакеты в общедоступные репозитории кода, преследуют разнообразные корыстные цели: кража учетных записей пользователя, шифрование файлов с требованием выкупа, несанкционированное удалённое управление устройством.**

В статье рассматривается проблема оценки потенциально разрушительного функционала в открытом программном обеспечении на языке Python. Обсуждаются различные методы оценки безопасности проектов, включая анализ кода и документации, снятие обфускации кода, репутационную проверку разработчика, отслеживание зависимостей проекта.

Внимание уделяется сложностям, связанным с определением приемлемости определенного функционала в проекте в соответствие с задачами последнего. Приводится классификация функционала, который попадает под определение небезопасного для разработчика (например, который может является каналом отправки информации злоумышленнику). Подробно разбираются существующие подходы к анализу кода, какие функциональные части можно описывать с их помощью.

***Ключевые слова*— Python, open source software, PyPI, source code analysis, malware detection.**

I. Введение

В современном мире открытое программное обеспечение (ОПО) используется повсеместно. Так, согласно компании-вендора анализа безопасности кода Synopsys, только 4% из всех проанализированных ими проектов не содержат в себе открытых библиотек [1]. Python является одним из наиболее популярных языков программирования для разработки различных проектов. Это подтверждается статистикой индекса TIOBE (оценка популярности языка по количеству поисков материалов по нему): начиная с 2008 года, язык находится в топ-10 индекса, а в 2018, 2020 и 2021 являлся самым популярным [2]. Критики TIOBE предлагают использовать индекс IEEE Spectrum (учитывает упоминания языка в поисковиках, в Twitter, Stack Overflow, Reddit, GitHub, сайтах по поиску работы), но там ситуация еще интереснее: Python занимает первое место, уже начиная с 2017 года [3]. Однако, несмотря на свою популярность, ОПО и проекты на Python в частности не являются идеальными и подвержены определенным угрозам безопасности.

Проблема заключается в том, что что открытые проекты могут содержать вредоносный код, используемый злоумышленниками для получения конфиденциальной информации, атак на информационные системы или других нежелательных действий. При этом добавлять вредоносный код в тот же Python Package Index может любой пользователь, имеющий там учетную запись [4]. Существует необходимость в оценке безопасности используемых нами проектах, чтобы убедиться, что последние не содержат разрушительного функционала.

В этой статье мы рассмотрим различные методы оценки безопасности проектов, включая анализ кода, репутационную проверку разработчиков, отслеживание зависимостей проекта. Отдельное внимание будет уделено проблеме оценки неоднозначного функционала проектов, когда определенные действия могут классифицироваться как легитимные или вредоносные в зависимости от целей проекта.

Методы, приведенные в статье, помогут специалистам, занимающимся обеспечением безопасности процесса разработки в компании, лучше понимать векторы атаки, мышление злоумышленника и натолкнуть на более детальный анализ используемой ими кодовой базы открытых проектов.

II. Оценка безопасности проекта

Первый шаг по оценке безопасности проекта является непосредственный анализ исходного кода. Определенная вредоносная активность является однозначной и очевидной, что позволяет ее легко

Статья получена 15 июня 2023.

Станислав Александрович Раковский, РТУ МИРЭА (e-mail: iam@disasm.me, rakovskiy.s.a@edu.mirea.ru).

Шамиль Гасангусейнович Магомедов, РТУ МИРЭА (e-mail: magomedov_sh@mirea.ru).

обнаружить. Пристальное внимание нужно уделить функциям, присущим стилерам (классу вредоносного программного обеспечения (ВПО), главной задачей которого является кража пользовательской информации [5], от steal - красть) и рансомвари (ВПО, шифрующему пользовательские данные с целью потребовать выкуп [6], от ransom – вымогать): получение информации о системе, создание снимков экрана и получение содержимого буфера обмена, переписывание содержимого файлов с использованием криптографических функций, отправка данных с использованием обычно несвойственных для легитимного кода каналов: ботов Telegram и вебхук Discord, почтового протокола SMTP, IRC-клиентов. Также отдельно нужно обратить внимание на обфусцированные участки, так как они могут быть использованы для сокрытия вредоносного кода [7].

Репутационная проверка разработчика является еще одним способом определения безопасности проекта. Можно проверить, как давно пользователь зарегистрирован в качестве разработчика в Python Package Index, найти его проект на других площадках хранения проектов (например, GitHub или BitBucket). Также активные разработчики обычно имеют аккаунт на LinkedIn и Twitter. Риск того, что автором вредоносного пакета будет являться новосозданный аккаунт, выше. Также стоит проверить пакеты разработчика в базах уязвимостей (например, National Vulnerability Database – NVD [8]). Это позволит узнать, насколько адекватно разработчик реагирует на найденные в его коде уязвимости, как быстро их исправляет и занимается ли этим в принципе. Всё изложенное в этом абзаце является критериями, которые могут быть относительно легко проверены автоматически.

Однако даже если разработчик имел в прошлом безупречную репутацию, существуют как минимум два негативных сценария. Во-первых, разработчика могут банально скомпрометировать. Этого можно достигнуть путем фишинга (несанкционированное получение доступа к учетным записям через обман) или взлома его инфраструктуры. Так, в августе 2022 года в результате фишинговой атаки были скомпрометированы два популярных пакета, общее количество скачиваний которых превышает 680 тысяч [9]. Другой случай: в начале 2020 хакеры взломали компанию SolarWind, крупного американского вендора в области информационных технологий, и, оставаясь незамеченными почти год, добавляли в выпускаемые жертвой продукты свои закладки [10].

Второй негативный сценарий – разработчик может добровольно добавить в свой проект вредоносный код. Одна из мотиваций – следование протестным движениям (protestware). В рамках protestware разработчик недоволен определенной общественной ситуацией. В лучшем случае это выливается в определенное послание, которое видит пользователь или другой разработчик в рамках использования кода «бунтаря». В худшем будет как с популярной npm-библиотекой node-ipc - определение страны проживания пользователя кода на основе Geo-IP, установленных раскладок клавиатуры или системного часового пояса, с последующим затиранием всех файлов на диске «сердечками» для устройств из стран, которые не нравятся разработчику [11].

В связи с этим, репутационная проверка разработчика может снизить ожидание встретить в его коде вредоносную компоненту, но исключать проверку кода разработчиков с безупречной историей не стоит.

Отслеживание зависимостей в проекте также может помочь в оценке его безопасности. Так, в случае с уже упомянутой библиотекой node-ipc вредоносный функционал находился в импортируемом модуле, что снизит шанс обнаружения. Также для злоумышленника, желающего скомпрометировать популярную библиотеку, может оказаться проще заняться ее зависимостями, чтобы увеличить результативность кампании. Стоит понимать, что понятие «зависимость» носит вложенный характер – у зависимостей первого уровня, прямых зависимостей, могут быть свои импортируемые библиотеки, которые в свою очередь будет являться зависимостями второго уровня.

Существуют репозитории доверенных зависимостей, предоставляемые вендорами в области безопасной разработки, аналитики которых занимаются решением проблемы оценки вредоносности пакетов. Однако весомая доля пакетов ускользает от их внимания, позволяя вредоносной компоненте длительное время действовать незаметно. Это показывает известную дилемму меча и щита – атакующая сторона имеет изначальное преимущество как действующая первой.

III. Поиск потенциально разрушительного функционала

Формирование правильного обоснования по использованию потенциально разрушительного функционала библиотекой важно для уменьшения количества недопустимых ошибок. Неточности в коде – обыденная вещь, так как код пишется человеком. Но неточность с реализации достаточно инвазивных действий может стоить пользователю его данных. Например, при реализации функционала удаления всех файлов из текущей директории, которые требовались для работы установщика, нужно учитывать, что рядом с установщиком могли лежать и другие файлы. Также разработчик может переборщить с отправляемой на сервер информацией для улучшения качества работы приложения, не проводя должное обеление от пользовательских данных, что может стоить пользователю его чувствительной информации.

Поэтому функционал, являющийся излишне широким в своих действиях или вообще находящийся в коде без должного обоснования, вызывает серьезные вопросы относительно безопасности и надёжности. В этом разделе мы рассмотрим способы поиска такого функционала относительно проектов на языке Python.

A. Классификация разрушительного функционала

Ниже приведена попытка классифицировать варианты разрушительного функционала, наблюдаемые ITW[1]:

[1] ITW, in the wild – термин, характеризующий меру распространенности объекта. ITW означает, что данная вредоносная утилита, класс такого вредоносного ПО или что-либо иное, подразумеваемое в контексте, не является теоретическим

- anti-analysis. Сюда относится попытка обфусцировать код или другим образом усложнить его анализ.
- collection. Сбор информации, которая может оказаться чувствительной для пользователя: создание скриншотов, запись нажатий клавиатуры.
- communication. Различные каналы сетевого взаимодействия (TCP, HTTP, IRC, Telegram). Их наличие будет странным в тех случаях, когда библиотеке по задумке не требуется сетевое общение и отправка куда-либо данных.
- crypto. Наличие алгоритмов хэширования, кодирования или шифрования.
- exploitation. Сюда относятся техники, связанные с повышением привилегий в системе, обращением к внутреннему функционалу системы, использованием известных уязвимостей.
- host interaction. Взаимодействие с устройством пользователя: смена обоев, вызов системных команд, создание скрытых файлов.
- persistence. Действия, служащие для долговременного закрепления в системе: добавление записей в автозагрузку или планировщик задач.
- protestware. Индикаторы политически-мотивированных лозунгов.

B. Способы описания потенциально разрушительного функционала в коде

Поиск потенциально разрушительного функционала схож с задачей поиска проблем и антипаттернов[2] в рамках контроля качества кода. Для проектов с большим количеством строк кода или сопровождаемых более чем одним человеком, контроль качества кода – естественная часть процесса поддержания качества разработки, и статический анализ кодовой базы – решение, к которому обычно приходят проекты в рамках своего жизненного цикла. В качестве инструмента по анализу кода в главную очередь используются статические анализаторы (линтеры) – инструменты, которые проверяют качество кода, руководствуясь определенными правилами.

Если мы знаем, что собой представляет вредоносный паттерн, то мы можем попробовать описать его на том же языке, на котором работает выбранный нами линтер. Для этого необходимо знать, какие возможности есть у определенного класса движков в рамках написания правил:

Линтеры на основе регулярных выражений (regex) руководствуются текстовыми поисковыми шаблонами. Это самый простой тип линтеров, так как входными данными для него является сам исходный код без информации о его контексте. Правила таких статических анализаторов позволяют находить упоминание «плохих» строк, некоторые ошибки форматирования[3]. Популярные линтеры pylint и bandit поддерживают описание правил с помощью регулярных выражений, а движок MalwareCheck в PyPI в рамках поиска вредоносных паттернов руководствуется исключительно регулярными выражениями [12]. Этот метод отличается не только простотой, но и высокой производительностью, позволяя быстро проверять большие объемы кода.

Линтеры на основе абстрактных синтаксических деревьев анализируют структуру кода, разбивая текст на токены, определяя их типы и объединяя их в логические блоки, такие как присваивание (Assign) и переопределение (AugAssign) значения переменной, объявление классов (ClassDef) и функций (FunctionDef), вызов методов (Call), управление потоком исполнения циклов (For, While, Break, Continue) и другие [13]. В итоге линтер генерирует дерево, узлами которого являются такого рода блоки. Этот тип линтеров мощнее чем regex, так как позволяет проводить расширенные проверки, такие как количество аргументов, передаваемых в известную функцию, объявленные, но неиспользуемые переменные, дублирование кода. Слабой стороной линтеров является необходимость поддержания актуальной версии – старые линтеры просто не поймут новых конструкций языка, воспринимая их за ошибку. К списку таких линтеров, помимо вышеупомянутых bandit и flake8, можно добавить isort, приводящий в структурированный вид список импортируемых модулей.

На основе абстрактных синтаксических деревьях строятся продвинутые проверки.

Анализ потока данных определяет, где создаются, как передаются и где используются данные в коде. Такие проверки позволяют определить несоответствие типов переменных ожиданиям обрабатывающей стороны. Так, у переменной типа int нет метода len, поэтому синтаксис «a = 7; len(a)» вызовет исключение, и это можно обнаружить на этапе анализа потока данных. В общем виде для поиска вредоносных паттернов это не имеет назначения, но ответвление этого метода, taint analysis (“проверка на загрязненность)”, имеет практическую ценность.

Принцип **taint analysis** [14] заключается в маркировании определенных данных как небезопасные и отслеживании их использования в коде в поисках мест, где они могут вызвать уязвимости (канонические примеры, куда приводит отсутствие контроля за потоком входных данных – SQL-инъекции, инъекции команд, XSS). Обычно маркируются входные пользовательские данные, но этот метод можно экстраполизовать на чувствительные данные, утечка которых нежелательна. Так, если в коде присутствует сбор информации о системе с последующей ее отправкой в рамках на сервер, то это может быть признаком сбора телеметрии, в том числе и со злым умыслом. Также популярный паттерн, который можно отследить через загрязненные данные, это исполнение скачанного по ссылке кода.

предположением, а встречается в Интернете или на устройствах жертв.

[2] Антипаттерн – распространенный, иногда интуитивно очевидный подход к решению определенных часто встречаемых проблем, являющийся на самом деле неэффективным или вовсе опасным.

[3] Так, по руководству по написанию кода PEP 8, не рекомендуется ставить пробел между названием функции и открывающейся скобкой в рамках вызова первой. Такую ситуацию можно найти с помощью регулярного выражения “\w \(”.

Taint analysis хоть и кажется очень полезным, но это крайне ресурсозатратный способ проверки кода.

Существуют также **глубокие линтеры**, которые досконально отслеживают совместимость типов данных для функций и методов, определяют проблемы с наследованием и переопределением методов, ищут мёртвый код. К таким линтерам относятся mypy и Pyre.

Обычно в рамках поиска потенциально небезопасных паттернов используется связка регулярных выражений, абстрактных деревьев и проверки на загрязненность.

C. Поиск зависимостей

Поиск и анализ зависимостей от других проектов играет настолько же важную роль в обеспечении безопасности, как и анализ кода своего проекта. Современные приложения не обходятся без функционала, заимствованного из сторонних библиотек, и присутствует потребность убедиться, что проект не импортирует какую-нибудь библиотеку с неизвестным названием, которая на, как окажется, хранит в себе вредоносный код и используется для обхода проверок системами, которые не ищут зависимости.

К сожалению, задача автоматического поиска зависимостей у отдельного проекта является нетривиальной. Казалось бы, необходимо всего лишь изучить соглашение об описании зависимостей для пакетного менеджера отдельного языка программирования, но это оказывается нелегко в отношении Python. Язык основан в 1991 году [15], и за это время были реализованы несколько форматов описания проекта:

1. setup.py – формат описания установочного файла для установщиков setuptools и distutils, представляет собой исполняемый код. Помимо самой сути, описания проекта и его зависимостей, разработчики также реализовывают проверку, последняя ли это версия библиотеки среди доступных, ручную доустановку пакетов, компиляцию бинарных файлов в случае с проектами-обвязками над более низкоуровневыми проектами. Этот стандарт появился сам по себе и не описан строго.
2. setup.cfg – более безопасная версия setup.py, не предусматривающая исполнения кода устанавливаемого пакета (не относится к зависимостям, они могут иметь установочные скрипты в «старом» формате).
3. pyproject.toml – последняя версия описания проекта, появившаяся в 2016 году вместе со стандартом PEP 518 [16]. Является предпочтительным вариантом для описания параметров установки пакета и, как и setup.cfg, не предполагает исполнение кода.

С последними двумя форматами нет никаких проблем – они представлены файлами в формате toml, который легко поддается разбору. В случае с setup.py не всё так радужно:

```
about = {}
here = os.path.abspath(os.path.dirname(__file__))
with open(os.path.join(here, "requests", "__version__.py"), "r", "utf-8") as f:
    exec(f.read(), about)

with open("README.md", "r", "utf-8") as f:
    readme = f.read()

setup(
    name=about["__title__"],
    version=about["__version__"],
    description=about["__description__"],
    long_description=readme,
    long_description_content_type="text/markdown",
    author=about["__author__"],
    author_email=about["__author_email__"],
    url=about["__url__"],
    packages=["requests"],
    package_data={"": ["LICENSE", "NOTICE"]},
    package_dir={"requests": "requests"},
    include_package_data=True,
    python_requires=">=3.7",
    install_requires=requires,
    license=about["__license__"],
    zip_safe=False,
```

Рисунок 1. Участок setup.py из requests 2.29.0

На рисунке 1 представлен участок кода установочного файла из популярной библиотеки requests. Здесь видно, что значения многих полей, в том числе необходимого нам «install_requires», описывающего зависимости, берётся из файла «__version__.py», который будет исполнен в контексте установки с помощью команды «exec». Данный пример демонстрирует, что обнаружение зависимостей является трудоёмкой задачей, так как исполнить этот код как есть мы не можем – не только потому, что он может быть вредоносным, но и из-за ресурсоемкости этого метода. Здесь следует пребегнуть к попытке упростить код с помощью ast-трансформеров, которые смогут отследить движение константы и подставить ее в нужное место.

D. Деобфускация кода

Сокрытие определенного функционала своего проекта не должна восприниматься как индикатор злых намерений разработчика. Так, библиотеки по машинному обучению mlcorelib, веб-интерфейсу simplepro содержат в себе обфусцированное содержимое, несмотря на то, что не содержат в себе вредоносного кода и выложены в репозиторий PyPI. Сам разработчик может прибегнуть к обфускации с целью усложнить обратную разработку своей интеллектуальной собственности или встроенных механизмов лицензирования.

Для злоумышленников желание скрыть вредоносную компоненту от обнаружения является естественным – это увеличит время пребывания кода незамеченным и, соответственно, позволит полезной нагрузке работать дольше, заразив большее число жертв.

Нам необходимо научиться обнаруживать такого рода участки кода и проводить их анализ. Часто обфускация представляет собой комбинацию алгоритмов сжатия и кодирования, за которой лежит исходный код в первоначальном исполнении. Однако существуют утилиты-протекторы, которые в рамках обфускации преображают код с потерей контекста: обезличивают названия переменных, функций, классов, «сжимают» код.

E. Проблемы автоматического обоснования

Пакеты в открытое программное обеспечение добавляются на постоянной основе: так, по отношению только к PyPI в 2023 году выходит более 5 тысяч новых релизов ежедневно. Идея предложить специалистам, которые умеют анализировать код, сидеть и просматривать такой объем пакетов ежедневно кажется недальновидным. Это приемлемо только на стадии написания правил обнаружения и не терпит никакой критики в отношении оперативности ответных действий на появление нового вредоносного кода.

С другой стороны, на попытке автоматизировать получение вердикта по определенному проекту, придется столкнуться с проблемой интерпретируемости получаемых результатов оценки вредоносного воздействия кода. Так, широкий пласт функционала может быть интерпретирован двояко. Такое действие, как сбор hardware id (идентификатора устройства) может быть использован как в логике проверки лицензии, так и в идентификации жертвы злоумышленником. Библиотека selenium, занимающаяся автоматизацией веб-действий, активно взаимодействует с браузером, однако взаимодействием с браузером занимаются и стилеры, задачей которых будет кража паролей, карт и сессионных ключей из браузера. Скачивание и запуск файла может служить функционалом обновления – либо доставки следующей стадии вредоносной нагрузки.

Здесь нужно обозначить преследуемую цель. Если вы хотите защитить разработчиков вашей организации, то технологии безопасных зеркал могут сработать – иметь периодически обновляемый список чистых версий пакетов является правильным решением. Однако если целью является сканирование всех появляющихся пакетов в режиме реального времени с целью поиска вредоносных, то здесь необходимо будет разрабатывать гибкую систему, которая будет учитывать время жизни пакета, пересечение кодовой базы, изменения кода по сравнению с прошлыми релизами и т.д.

IV. Заключение

Поиск вредоносных проектов в репозиториях PyPI является хорошей демонстрацией творческой задачи, к которой можно найти серию подходов, «цепляющихся» за различные стороны проекта: сам код, переиспользование кодовой базы и метаинформации, репутационную статистику разработчика, поиск зависимостей проекта. К анализу каждой из этих сторон можно подойти с разной степенью детализации, от чего будет зависеть сложность задачи, так как в каждом из этих случаев есть проблемы, решение которых позволит улучшить метрики обнаруживаемости пакетов.

Однако даже применение всего этого инструментария не гарантирует полного покрытия вредоносного кода: ответом на более совершенные методы обнаружения будут всё новые способы сокрыть вредоносную деятельность. Но это определенно точно не значит, что борьба с злоумышленниками лишена смысла.

Библиография

[1] Synopsis, «[2023] Open Source Security and Risk Analysis Report». Режим доступа: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html (дата обращения: 15.02.2023).

[2] TIOBE, «TIOBE Index for February 2023». Режим доступа: https://www.tiobe.com/tiobe-index/ (дата обращения: 17.02.2023)

[3] IEEE Spectrum, «Top Programming Languages 2022». Режим доступа: https://spectrum.ieee.org/top-programming-languages-2022 (дата обращения: 17.02.2023).

[4] Kaplan B., Qian J. A survey on common threats in npm and pypi registries //Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2. – Springer International Publishing, 2021. – С. 132-156.

[5] Encyclopedia by Kaspersky, «Троянец-стилер (Trojan-PSW, Password Stealing Ware)». Режим доступа: https://encyclopedia.kaspersky.ru/glossary/trojan-psw-psw-password-stealing-ware/ (дата обращения: 17.02.2023).

[6] Kaspersky Lab, «Шифровальщики – это не про вас». Режим доступа: https://noransom.kaspersky.ru/ (дата обращения: 17.02.2023).

[7] Конференция РусКрипто, тематическая секция «Исследование и защита цифровых технологий», доклад «Технология автоматизированного подхода к реверс-инжинирингу обфускаторов-пакеров вредоносного кода». Режим доступа: https://www.ruscrypto.ru/program/sections/s_4.html (дата обращения: 20.04.2023).

[8] National Institute of Standards of Technology, «National Vulnerability Database». Режим доступа: https://nvd.nist.gov/ (дата обращения: 17.02.2023).

[9] BleepingComputer, «PyPI packages hijacked after developers fall for phishing emails». Режим доступа: https://www.bleepingcomputer.com/news/security/pypi-packages-hijacked-after-developers-fall-for-phishing-emails/ (дата обращения: 17.02.2023).

[10] Greiman V. Known Unknowns: The Inevitability of Cyber Attacks //European Conference on Cyber Warfare and Security. – 2023. – Т. 22. – №. 1. – С. 223-231.

[11] National Vulnerability Database, «CVE-2022-23812 Detail». Режим доступа: https://nvd.nist.gov/vuln/detail/CVE-2022-23812 (дата обращения: 19.02.2023).

[12] Vu D. L., Newman Z., Meyers J. S. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning.

[13] Python 3.10 Documentation, «ast – Abstract Syntax Trees». Режим доступа: https://docs.python.org/3/library/ast.html (дата обращения: 22.02.2023).

[14] Aloraini B. Towards Better Static Analysis Security Testing Methodologies. – 2020.

[15] Python Institute – Open Educational & Development Group, « Python® – the language of today and tomorrow». Режим доступа: https://pythoninstitute.org/about-python (дата обращения: 24.02.2023).

[16] Cannon B., Smith N., Stufft D. Pep 518: specifying minimum build system requirements for python projects. – 2020.

Ensuring the Security of Open Python Projects: The Challenge of Assessing Potentially Destructive Functionality

S. Rakovsky, S. Magomedov

***Abstract*—Open source software is widely used everywhere. Unfortunately, until recently, developers have not considered the need to monitor the security of the external projects they use. One can understand programmers: they would like to believe that if the source code is open, it has already been analyzed by automated means, and if it poses a security threat, the associated package will be removed. Unfortunately, this is not the case, and malicious functionality can exist in the form of an open project named to mimic a popular package for months. Malicious actors who add harmful packages to public code repositories pursue various selfish goals: theft of user accounts, file encryption with a ransom demand, unauthorized remote control of the device.**

This article discusses the problem of assessing potentially destructive functionality in open source software in Python. Various methods for assessing project security are discussed, including code and documentation analysis, code deobfuscation, developer reputation checks, and project dependency tracking.

Attention is paid to the difficulties associated with determining the acceptability of certain functionality in a project in accordance with its tasks. A classification of functionality that falls under the definition of unsafe for the developer (for example, which can be a channel for sending information to a malicious actor) is provided. Existing approaches to code analysis are examined in detail, and the functional parts that can be described with their help are discussed.

***Keywords*—Python, open source software, PyPI, source code analysis, malware detection.**

REFERENCES

[1] Synopsis. [2023] Open Source Security and Risk Analysis Report. — online; accessed: 2023-02-15. — URL: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[2] TIOBE. TIOBE Index for February 2023. — online; accessed: 2023-02-16. — URL: https://www.tiobe.com/tiobe-index/.

[3] IEEE Spectrum. Top Programming Languages 2022. — online; accessed: 2023-02-17. — URL: https://spectrum.ieee.org/top-programming-languages-2022.

[4] Kaplan B., Qian J. A survey on common threats in npm and pypi registries //Deployable Machine Learning for Security Defense: Second International Workshop, MLHat 2021, Virtual Event, August 15, 2021, Proceedings 2. – Springer International Publishing, 2021. – С. 132-156.

[5] Encyclopedia by Kaspersky. Троянец-стилер (Trojan-PSW, Password Stealing Ware). — online; accessed: 2023-02-17. — URL: https://encyclopedia.kaspersky.ru/glossary/trojan-psw-psw-password-stealing-ware/.

[6] Kaspersky Lab. Shifroval'shhiki – jeto ne pro vas. — online; accessed: 2023-02-17. — URL: https://noransom.kaspersky.ru/.

[7] RusCrypto Conference, thematic section «Issledovanie i zashhita cifrovyh tehnologij». Tehnologija avtomatizirovannogo podhoda k revers-inzhiniringu obfuskatorov-pakerov vredonosnogo koda. — online; accessed: 2023-04-20. — URL: https://www.ruscrypto.ru/program/sections/s_4.html.l.

[8] National Institute of Standards of Technology. National Vulnerability Database. — online; accessed: 2023-02-17. — URL: https://nvd.nist.gov/.

[9] BleepingComputer. PyPI packages hijacked after developers fall for phishing emails. — online; accessed: 2023-02-17. — URL: https://www.bleepingcomputer.com/news/security/pypi-packages-hijacked-after-developers-fall-for-phishing-emails/.

[10] Greiman V. Known Unknowns: The Inevitability of Cyber Attacks //European Conference on Cyber Warfare and Security. – 2023. – Т. 22. – №. 1. – С. 223-231.

[11] National Vulnerability Database. CVE-2022-23812 Detail. — online; accessed: 2023-02-19. — URL: https://nvd.nist.gov/vuln/detail/CVE-2022-23812.

[12] Vu D. L., Newman Z., Meyers J. S. Bad Snakes: Understanding and Improving Python Package Index Malware Scanning.

[13] Python 3.10 Documentation. ast – Abstract Syntax Trees. — online; accessed: 2023-02-22. — URL: https://docs.python.org/3/library/ast.html.

[14] Aloraini B. Towards Better Static Analysis Security Testing Methodologies. – 2020.

[15] Python Institute – Open Educational & Development Group. Python® – the language of today and tomorrow. — online; accessed: 2023-02-24. — URL: https://pythoninstitute.org/about-python.

[16] Cannon B., Smith N., Stufft D. Pep 518: specifying minimum build system requirements for python projects. – 2020.

About of Authors

S. A. Rakovsky is with the MIREA – Russian Technological University, Moscow, Russia (e-mail: iam@disasm.me, rakovskiy.s.a@edu.mirea.ru).

S. G. Magomedov is with the MIREA – Russian Technological University, Moscow, Russia (e-mail: magomedov_sh@mirea.ru).