**Web1**

На странице календарь, видим что число 20 является гиперссылкой, нажимаем.

На открывшейся странице видим хинт, намекающий нам посмотреть в etc/secret, а строка адреса при всем этом выглядит крайне подозрительно.

Несложно догадаться, что нужно изменить параметр file\_type так, чтобы он отдал нам не file1.txt, а etc/secret.

Итоговый вид запроса:

192.168.12.10:5001/download?file\_type=../../etc/secret

Флаг: nto{P6t9\_T77v6RsA1}

**Web2**

Открываем страницу, выданную нам в задании. На первый взгляд ничего не понятно, поэтому идем декомпилировать jar-сервер в любом попавшемся декомпиляторе (мы использовали [https://www.decompiler.com/jar/ ).](https://www.decompiler.com/jar/).)

Видим сервер на SpringBootе, по дереву находим server и идем в HelloController.

В коде видим, что у нас есть некая страница /login, которая принимает в себя параметр password.



Если достаточно внимательно проанализировать этот код, то можно заметить, что изначально переменная adminpassword = “password”, но впоследствии она перезаписывается из файла password.txt, содержимое которого нам неизвестно.

Так как перезаписывающая конструкция находится в конструкции try, понимаем, что если мы помешаем программе прочитать password.txt, переменная перезаписана не будет, и мы сможем «залогиниться» с параметром password=password.

Уже на этом этапе закрадываются мысли просто загуглить springview vuln, но покопаемся в коде еще немного.

Из общего кода, желающего нам доброго утра, немного выбивается страница /doc.



Здесь же мы видим использование шаблона.

На этом моменте в голове уже складывается небольшой пазл (шаблоны, подозрительная страница...), и можно предположить что нас ждет SSTI.

Теперь уже точно идем гуглить springview vuln.

По первой же ссылке (https://www.veracode.com/blog/secure-development/spring-view-manipulation-vulnerability) мы видим код, практически идентичный нашему, и полностью идентичную нашей страницу /doc.

Вспоминаем, что нам надо помешать программе прочитать password.txt. Берем эксплоит и подставляем в него то, что нам нужно (rm password.txt) и оборачиваем в url.

Итоговый эксплоит будет выглядеть так:

%5F%5F%24%7BT%28java%2Elang%2ERuntime%29%2EgetRuntime%28%29%2Eexec%28%22rm%20%2Fpassword%2Etxt%22%29%7D%5F%5F%3A%3A%2Ex

Выполняем его на сервере. ( ip/doc/%5F%5F%24%7BT%28java%2Elang%2ERuntime%29%2EgetRuntime%28%29%2Eexec%28%22rm%20%2Fpassword%2Etxt%22%29%7D%5F%5F%3A%3A%2Ex )

После его выполнения очень быстро (пока машина не перезагрузилась) переходим на /login, подаем ему password=password и получаем флаг.

nto{abobovichasdfas}

**Web3**

Абсолютно любой ввод в леттербокс отправляет нас на страницу /flag.

Эта страница выдает нам 403, но нехитрым перебором способов байпасса этого запрета мы обходим его, добавив еще один слэш после первого (итоговый путь: http://192.168.12.11:8001//flag)

В коде приложения видим, что подавать нужно параметр name, а само приложение написано с помощью фласка, который как свой дефолтный шаблонизатор использует jinja2.

Понимаем, что нас ждем непростая ssti, т.к самые популярные пейлоады неюзабельны из-за фильтра в коде программы.

Методом проб, ошибок и непрерывного гуглинга все ж таки пишем эксплоит, открывающий нам rce и позволяющий прочитать флаг.

Итоговый путь: http://192.168.12.11:8001//flag?name=%7B%7B%20self.\_\_init\_\_.\_\_globals\_\_.\_\_builtins\_\_.\_\_import\_\_(%27os%27).popen(%27cat%20flag.txt%27).read()%20%7D%7D

Флаг: nto{Ht1P\_sM088Lin6\_88Ti}

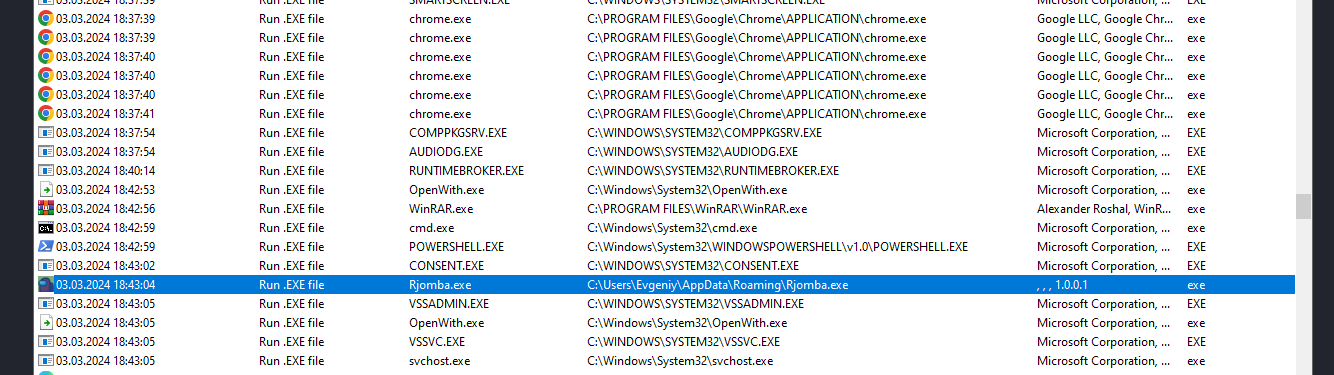
**Расследование инцидента.**

**1. Ответ: Пользователь попался на фишинг письмо**

Вообще об этом сказано еще в легенде, но нам же надо убедиться :)

Открыв машину в виртуалбоксе перемещаемся в загрузки, откуда устанавливаем хром. Непосредственно в хроме идем в историю и видим, что последняя запись истории (т.е когда система еще была жива) датируется 03.03.2024 18:41 и является Рамблером(почтой).

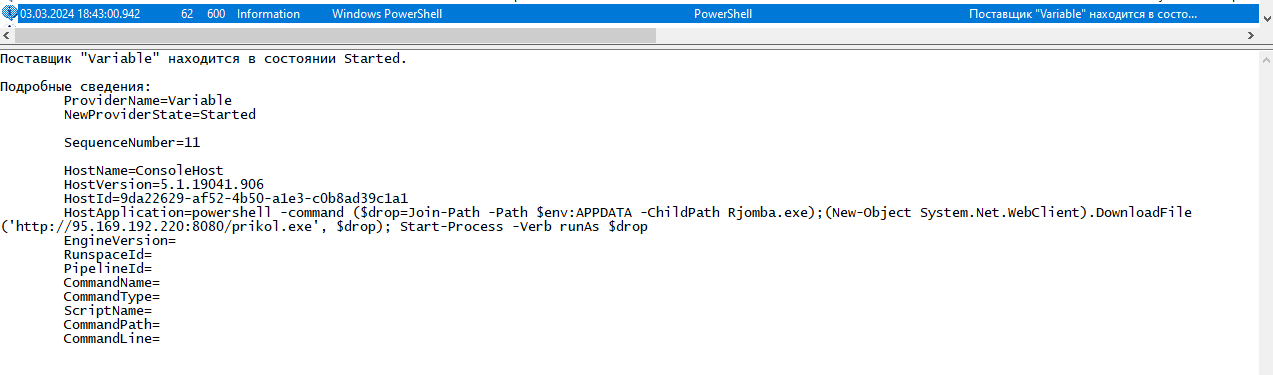
Для того, чтобы убедиться в наших словах, на гостевой машине устанавливаем программы LastActivityView и FullEventLogView, которые позволят нам просмотреть логи системы и последние запущенные приложения. Запускаем LastActivityView и перемещаемся к дате, соответствующей датам последних записей истории браузера. Среди системных и прочих процессов видим совсем нетипичную для общего поведения пользователя (здесь под общим поведением пользователя подразумевается то, что система в основном используется для работы, что видно по установленным софтам для разработки, компиляторам и т.п) программу "rjomba.exe", которая и является ВПО.



Обратим внимание на то, что было запущено перед ней. Мы видим winrar и хром, что заставляет нас вспомнить легенду и в конечном итоге наталкивает на мысли, что ВПО было получено пользователем из хрома и впоследствии через винрар (забегая вперед) развернуто в системе.

**2. Ответ: http://95.169.192.220:8080/prikol.exe**

Вернемся к программе FullEventLogView, которая нам наконец-то пригодилась. Листаем логи до времени исполнения rjomba.exe (время мы берем из программы LastActivityView) и находим интересную пачку логов, каждый из которых прямым текстом выдает нам сервер, с которого был загружен вредонос.



**3. Ответ: CVE-2023-38831: RARLAB WinRAR Code Execution Vulnerability, ПО - WinRar**

Не закрывая программу FullEventLogView продолжаем курить логи, но теперь уже в связке с LastActivityView.

Возвращаемся к последним запущенным программам. Мы видим, что сразу после винрара начинает открываться хост окна консоли и powershell, и только после этого выполняется наша rjomba.exe. Это говорит нам о том, что вредонос не был загружен с почты напрямую, а загрузился каким-то другим способом и уже после выполнения винрара. С этими мыслями возвращаемся к тем логам, что мы нашли раньше, и начинаем анализировать, что же именно исполнилось в этом самом powershelle.

Проанализировав команду, найденную в логах, понимаем, что она загружает некий файл prikol.exe с сервера 95.169.192.220:8080, и сохраняет его в rjomba.exe, который впоследствии и выполнится.

После составления в голове нетрудных причинно-следственных связей и недолгого гуглинга становится понятно, что позволил запустить powershell-код именно winrar, т.к до него и после него ничего более подозрительного не выполнялось.

Открываем гугл и вбиваем "winrar vuln". По первой же ссылке в гугле находим статью от гугла, которая ссылается на CVE-2023-38831, которая, в свою очередь, является причиной исполнения столь подозрительного кода в нашей системе. Убеждаемся в этом найдя страницу с пейлоадом для этой язвы на гитхабе, где продемонстрирована работа уязвимости.

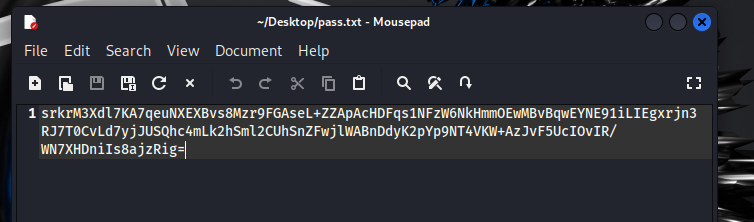
1. **Ответ: Функция isDebuggerPresent и пресечение любых попыток взаимодействия с процессом.**

Открываем вирус в ghidra и начинаем копаться. После некоторого гуглинга и усиленных попыток что-то раскопать наталкиваемся на функцию isDebuggerPresent(), которой сразу же находим объяснение в гугле: <https://learn.microsoft.com/en-us/windows/win32/api/debugapi/nf-debugapi-isdebuggerpresent>

Под пресечением попыток взаимодействия с процессом я имею ввиду закрытие таскменеджера и других программ для мониторинга процессов.

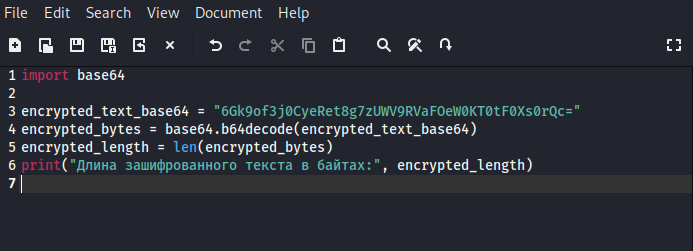
1. **Ответ: AES-256, режим CBC**

Для того, чтобы это понять, заглянем в любой из зашифрованных файлов. К примеру, pass.txt:



Мы видим base64 строку. Самый популяный шифр результатом которого является base64 строка - AES.

Длина данного сообщения равна 32 байта (для доказательства приложен файл aes.py). Длина зашифрованного текста кратна размеру блока (в AES размер блока равен 256 бит, т.е 32 байт).  
(в питоновском файле приведен другой пример содержимого шифрованного файла т.к делал другой человек, суть от этого не меняется)



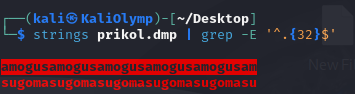
1. **Ответ: amogusamogusamogusamogusamogusam**

Здесь сначала лучше прочесть 7 пункт, т.к мы делали не по порядку :)

Используем все тот же дамп процесса, который мы сняли в 7 пункте. Перемещаем его на хост систему (в кали).

Немного подтянув матчасть становится понятно, что длина искомого ключа - 32 символа.

На кали запускаем терминал и раскладываем наш дамп на строки, задав вывод только 32-символьных строк.



А вот и наш ключ.

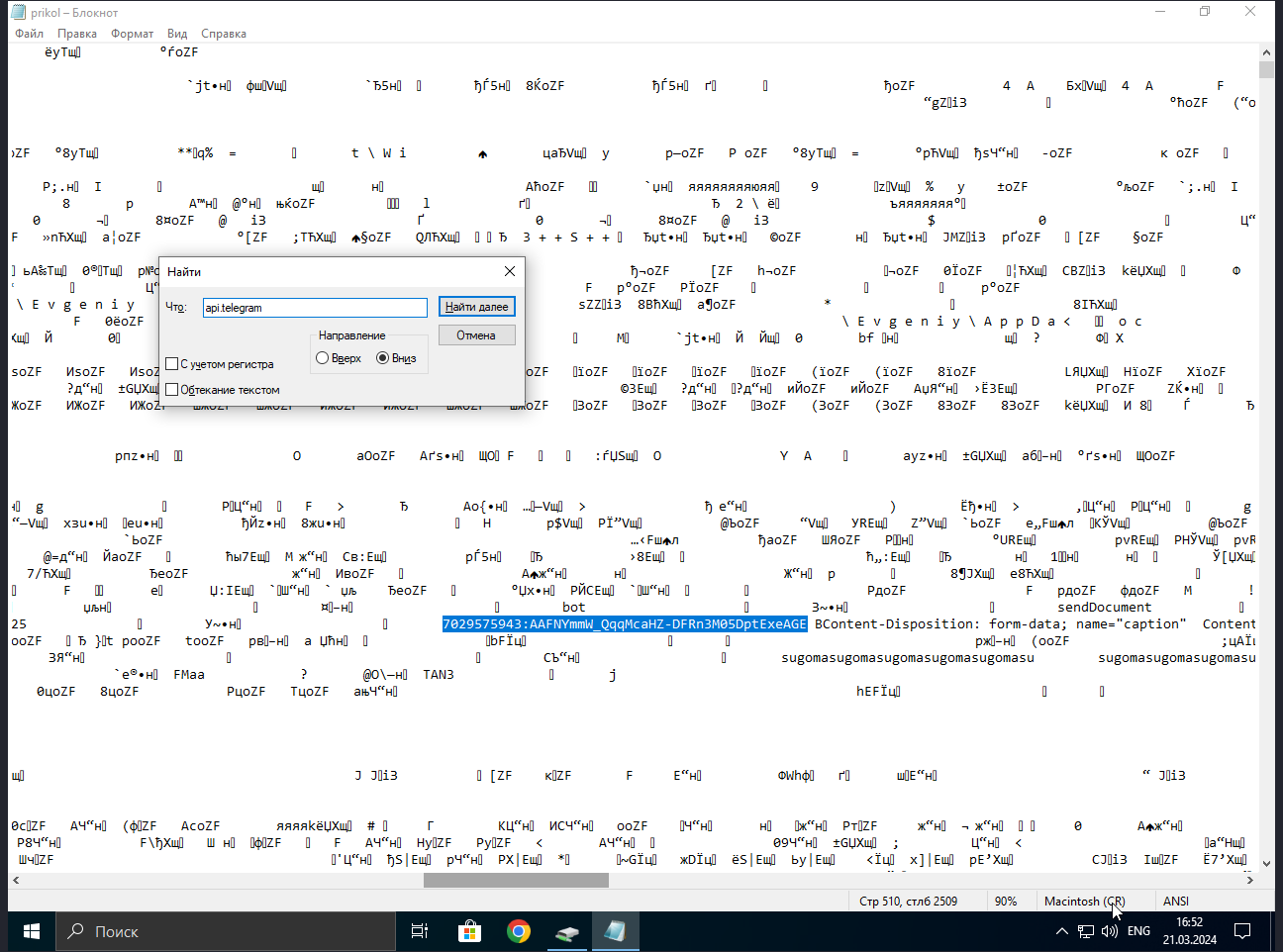
1. **Ответ: Отсылает себе через телеграм. Токен - 7029575943:AAFNYmmW\_QqqMcaHZ-DFRn3M05DptExeAGE**

В первую очередь заливаем вирусняк на вирустотал, чтобы увидеть его поведение пока что без запуска на нашей машине. На вирустотале в вкладке Community (https://www.virustotal.com/gui/file/9c8cf3de80161ea51ccf1e7d51b8643c6ee1dc2e7bef64ca9e5b4f2492e1bfd3/community) обнаруживаем отчеты по этому вирусу на триаге (https://tria.ge/240304-p4pdmabh35). Переходим туда и листаем, в процессе этого нехитрого дела в разделе взаимодействия малваря с сетью обнаруживаем запросы к telegram.

Первая часть вопроса есть, теперь наша задача - обнаружить токен, посредством которого происходит аутентификация малваря на апи телеграма.

Для того, чтобы достать его, можно воспользоваться дампом процесса. Чтобы сдампить процесс с вирусом, устанавливаем программу ProcessExplorer а также переименовываем ее, чтобы вирус не убил наш инструмент. Запускаем переименованый эксплорер, после чего запускаем вирус. После того, как нашли prikol.exe в списке процессов, замораживаем его и создаем full dump.

Далее можно было бы поступить более умно, но на первое время воспользуемся обычным блокнотом. Открываем наш дамп в нем и ищем строчку, которая включает в себя «api.telegram.org». Довольно скоро находим эту строчку, а неподалеку от нее видим и токен.



1. **Ответ: sFYZ#2z9VdUR9sm`3JRz**

Так как мы уже подтянули матчасть в пункте 6, мы знаем не только то, что для дешифрования ключа при режиме CBC помимо ключа нужен IV, но и даже длину этого самого вектора. Прекрасно!

Проделываем ты же самые манипуляции с дампом что и в пункте 6, но заменяем искомую длину с 32 на 16.



Profit!