9 Инструмент: мониторинг сети

Раннее выявление вредоносных действий в сфере кибербезопасности повышает шанс быстро устранить эту опасность. Одним из таких методов обнаружения является мониторинг сети, определяющий появление новых или неожиданных сетевых служб (то есть открытых портов). Именно с помощью командной строки можно выявить вредоносные действия на раннем этапе.

В этой главе мы создадим инструмент, позволяющий отслеживать по всей сети изменения в открытых портах систем. Требования к инструменту следующие.

1. Прочитать файл, содержащий IP-адреса или имена хостов.
2. Выполнить для каждого упоминавшегося в файле хоста сканирование сетевых портов и определить открытые порты.
3. Сохранить вывод, полученный при сканировании портов, в файл. В имени этого файла должна быть указана текущая дата.
4. При повторном запуске сценария снова должно выполняться сканирование портов, а затем полученные результаты необходимо сравнить с ранее сохраненным последним результатом. Выявленные изменения должны выделяться на экране.
5. Автоматизировать ежедневный запуск сценария и при возникновении каких-либо изменений отправлять системному администратору сообщение по электронной почте.

Сканирование портов можно выполнить с помощью утилиты Nmap Ndiff. Но в целях обучения мы реализуем эту функцию, используя bash. Дополнительные сведения о Ndiff вы найдете по адресу https://nmap.org/ndiff.

Используемые команды

В этой главе мы рассмотрим работу с командами `crontab` и `schtasks`.

crontab

Команда `crontab` позволяет редактировать cron-таблицу в системе Linux. Таблица cron используется для планирования задач по выполнению команд в определенное время или в определенный период времени.

Общие параметры команды

- `-e` — редактировать cron-таблицу.
- `-l` — вывести текущую cron-таблицу.
- `-r` — удалить текущую cron-таблицу.

schtasks

Команда `schtasks` позволяет планировать задачи в среде Windows, запускающие выполнение необходимых команд в определенное время или промежуток времени.

Общие параметры команды

- `/Create` — запланировать новую задачу.
- `/Delete` — удалить запланированную задачу.
- `/Query` — вывести список всех запланированных задач.

Шаг 1. Создание сканера портов

В первую очередь создадим сканер портов. Для этого нужно на определенном порту создать TCP-соединение с определенным хостом. Это можно сделать с помощью файлового дескриптора bash, имя которого — `/dev/tcp`.

Для создания сканера портов сначала необходимо прочитать из файла список IP-адресов или имен хостов. Далее будет предпринята попытка подключения

к ряду портов на хостах, упоминаемых в файле. В случае успешного подключения станет понятно, что порт открыт. Если время соединения истекло или вы получили сообщение о сбросе, значит, порт закрыт. Для этого проекта мы на каждом хосте отсканируем TCP-порты с номерами от 1 до 1023 (пример 9.1).

Пример 9.1. scan.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# scan.sh
#
# Описание:
# Сканирование порта указанного хоста
#
# Использование: ./scan.sh <output file>
#   <output file> Файл, куда сохраняются результаты
#

function scan ()
{
    host=$1
    printf '%s' "$host"                                          ❶
    for ((port=1;port<1024;port++))
    do
        # порядок перенаправления важен по двум причинам
        echo >/dev/null 2>&1 < /dev/tcp/${host}/${port}          ❷
        if (($? == 0)) ; then printf ' %d' "${port}" ; fi        ❸
    done
    echo # или вывести '\n'
}

#
# основной цикл
#   читать имя каждого узла (из stdin)
#   и искать открытые порты
#   сохранить результаты в файл,
#   имя которого указано в качестве аргумента,
#   или задать имя по умолчанию на основе текущей даты
#

printf -v TODAY 'scan_%(%F)T' -1 # например, scan_2017-11-27    ❹
OUTFILE=${1:-$TODAY}                                             ❺

while read HOSTNAME
do
    scan $HOSTNAME
done > $OUTFILE                                                  ❻
```

❶ Здесь обратите внимание на команды `printf`. Ни одна из них не разбивает вывод на несколько строк, чтобы сохранить код в одной (длинной) строке.

❷ Это критический шаг в сценарии — фактически создание сетевого подключения к указанному порту. Подключение создается с помощью следующего кода:

```
echo >/dev/null 2>&1   < /dev/tcp/${host}/${port}
```

Команда `echo` здесь не имеет реальных аргументов, только перенаправления. Перенаправления обрабатываются оболочкой; команда `echo` никогда не видит эти перенаправления, но знает, что они произошли. Без аргументов `echo` в stdout будет просто напечатан символ новой строки (`\n`). Поскольку здесь о выводе мы не заботимся, и stdout, и stderr перенаправляются в `/dev/null` (фактически отбрасываются).

Ключевым моментом здесь является перенаправление stdin (через `<`). Мы перенаправляем stdin, чтобы он использовал специальное имя файла bash, `/dev/tcp/...` и некоторый номер хоста и порта. Поскольку `echo` просто выполняет вывод, команда не будет читать какие-либо входные данные из этого специального сетевого файла. Скорее, мы просто пытаемся его открыть (только для чтения), чтобы увидеть, есть ли там эти данные.

❸ Это вторая команда `printf`. Если команда `echo` выполняется успешно, значит, соединение с данным портом на указанном хосте успешно установлено. В этом случае мы выводим номер данного порта.

❹ Функция `printf` (в более новых версиях bash) поддерживает специальный формат печати значений даты и времени. Символы `%()T` — это спецификатор формата `printf`, который указывает, что это формат даты/времени. Строка в скобках содержит сведения о том, какие составляющие части даты и/или времени вы хотите показать. Здесь применены спецификаторы, которые будут использоваться в вызове системной библиотеки `strftime`. (Для более подробной информации введите `trftime`.) В этом случае `%F` означает формат «год-месяц-день» (формат даты ISO 8601). Дата/время печати определяется как `-1`, что означает «сейчас».

Параметр `-v` команды `printf` указывает, что вывод следует сохранить в переменной, а не выводить на экран. В этом случае в качестве переменной используется `TODAY`.

❺ Если пользователь в качестве первого аргумента данного сценария указывает в командной строке файл вывода, будет использован этот аргумент. Если первый аргумент отсутствует, то в качестве имени файла вывода будет использоваться строка с текущей датой, только что созданная в `TODAY`.

❻ Перенаправляя вывод в `done`, мы делаем это для всего кода внутри цикла `while`. Если бы было сделано перенаправление в самой команде сканирования, то, чтобы добавить вывод к файлу, мы должны были бы использовать символы `>>`. В противном случае при каждой итерации цикла сохраняется только один вывод команды, а предыдущий вывод блокируется. Если к файлу добавляется очередная команда, то перед началом цикла нам нужно будет этот файл обрезать. Таким образом, вы можете увидеть, что гораздо лучше просто выполнить перенаправление в цикл `while`.

Файл вывода с результатами сканирования будет отформатирован так, что разделителем будет пробел. Каждая строка начинается с IP-адреса или имени хоста, а затем перечисляются все открытые TCP-порты.

В примере 9.2 приведен вариант формата вывода, который показывает, что на хосте 192.168.0.1 открыты порты 80 и 443, а на хосте 10.0.0.5 — порт 25.

Пример 9.2. scan_2018-11-27

```
192.168.0.1 80 443
10.0.0.5 25
```

Шаг 2. Сравнение с предыдущим выводом

Конечная цель, которой мы хотим достичь с помощью этого инструмента, — обнаружение изменений находящегося в сети хоста. Для этого нам необходимо сохранять в файл результаты каждого сканирования. Далее — сравнить последнее сканирование с предыдущим результатом и обнаружить разницу между предыдущим и текущим состояниями. В частности, мы будем искать устройство, у которого TCP-порт открыт или закрыт. Определив состояние порта, вы можете выяснить, было ли это изменение санкционированным, или это признак злонамеренной активности.

В примере 9.3 сравниваются результаты последней проверки с сохраненными в файле предыдущими результатами и выявляются даже самые незначительные изменения.

Пример 9.3. fd2.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
```

```
# fd2.sh
#
# Описание:
# Сравнивает два результата сканирования портов для поиска изменений
# Основное предположение: оба файла имеют одинаковое количество строк,
# каждая строка с тем же адресом хоста,
# хотя перечисленные порты могут быть разными
#
# Использование: ./fd2.sh <file1> <file2>
#

# найти "$LOOKFOR" в списке аргументов для этой функции
# возвращает true (0), если его нет в списке
function NotInList ()                                          ❶
{
    for port in "$@"
    do
        if [[ $port == $LOOKFOR ]]
        then
            return 1
        fi
    done
    return 0
}

while true
do
    read aline <&4 || break     # EOF                          ❷
    read bline <&5 || break     # EOF, для симметрии           ❸

    # if [[ $aline == $bline ]] ; then continue; fi
    [[ $aline == $bline ]] && continue;                        ❹

    # есть разница, поэтому мы
    # подразделяем на хост и порты
    HOSTA=${aline%% *}                                         ❺
    PORTSA=( ${aline#* } )                                     ❻

    HOSTB=${bline%% *}
    PORTSB=( ${bline#* } )

    echo $HOSTA         # определяем хост, в котором произошли изменения

    for porta in ${PORTSA[@]}
    do              ❼
        LOOKFOR=$porta NotInList ${PORTSB[@]} && echo " closed: $porta"
    done

    for portb in ${PORTSB[@]}
```

```
    do
        LOOKFOR=$portb NotInList ${PORTSA[@]} && echo " new: $portb"
    done

done 4< ${1:-day1.data} 5< ${2:-day2.data}                    ❽
# day1.data и day2.data являются именами по умолчанию, что упрощает тестирование
```

❶ Функция `NotInList` написана так, чтобы возвращать значение, приравненное к `true` или `false`. Помните, что в оболочке (за исключением значений в двойных скобках) `0` считается истинным. (После выполнения команды возвращается `0`, когда ошибки не возникает; ненулевые возвращаемые значения обычно указывают на ошибку, поэтому считаются ложными.)

❷ «Уловка» в этом сценарии заключается в том, что можно читать из двух разных потоков ввода. Для этого в сценарии мы используем файловые дескрипторы 4 и 5. Здесь переменная `aline` заполняется данными, прочитанными из файлового дескриптора 4. Мы вскоре увидим, где дескрипторы 4 и 5 получают свои данные. Символ `&`, который находится перед дескриптором файла 4, обозначает, что это дескриптор файла 4. Без символа `&` bash будет пытаться читать из файла с именем 4. После прочтения последней строки входных данных, когда мы достигнем конца файла, команда `read` возвращает ошибку. В этом случае будет выполнена команда `break`, завершающая цикл.

❸ Аналогично `bline` будет считывать свои данные из дескриптора 5. Поскольку предполагается, что два файла имеют одинаковое количество строк (то есть одни и те же хосты), то команда `break` здесь тоже нужна, так как она выполняется и в предыдущей строке. Такая симметрия делает файл более читаемым.

❹ Если две строки идентичны, нет необходимости разбирать их на отдельные номера портов, поэтому мы сразу переходим к следующей итерации цикла.

❺ Мы изолируем имя хоста, удалив все символы, находящиеся после первого пробела (включая и сам первый пробел).

❻ И наоборот, мы можем извлечь все номера портов, удалив имя хоста и все символы из начала строки вплоть до первого пробела (включая и сам первый пробел). Обратите внимание: мы не просто присваиваем этот список переменной, а используем скобки, чтобы инициализировать ее как массив, в котором каждая запись — номер порта.

❼ Посмотрите на это выражение. За присвоением переменной в той же строке сразу идет команда `echo`. Для оболочки это означает, что значение переменной

действительно только на время выполнения данной команды. К своему предыдущему значению переменная возвращается сразу после выполнения команды. Вот почему мы в этой строке не повторяем `$LOOKFOR` — это не будет действительным значением. Мы бы могли разделить это выражение на две отдельные команды — присваивание переменной и вызов функции. Но тогда бы вы не узнали об этой функции в bash.

❽ Здесь демонстрируется новый вариант использования файловых дескрипторов. Файловый дескриптор 4 получает «перенаправление» для чтения входных данных из файла, указанного в первом аргументе сценария. Соответственно дескриптор 5 получает свои входные данные из второго аргумента. Если один или оба параметра не заданы, сценарий будет использовать имена, указанные по умолчанию.

Шаг 3. Автоматизация и уведомление

Хотя вы можете выполнять сценарий вручную, было бы гораздо лучше, если бы он автоматически запускался каждый день или каждые несколько дней и уведомлял вас о любых обнаруженных изменениях. Сценарий `autoscan.sh`, показанный в примере 9.4, является единственным сценарием, использующим для сканирования сети и вывода любых изменений файлы `scan.sh` и `fd2.sh`.

Пример 9.4. autoscan.sh

```
#!/bin/bash -
#
# Bash и кибербезопасность
# autoscan.sh
#
# Описание:
# Автоматическое сканирование портов (с помощью сценария scan.sh)
# Сравнение вывода с предыдущими результатами и e-mail пользователя
# Предполагается, что сценарий scan.sh находится в текущем каталоге
#
# Использование: ./autoscan.sh
#

./scan.sh < hostlist                                    ❶

FILELIST=$(ls scan_* | tail -2)                         ❷
FILES=( $FILELIST )

TMPFILE=$(tempfile)                                     ❸
```

```
./fd2.sh ${FILES[0]} ${FILES[1]} > $TMPFILE

if [[ -s $TMPFILE ]]   # не пустой                          ❹
then
    echo "mailing today's port differences to $USER"
    mail -s "today's port differences" $USER < $TMPFILE    ❺
fi
# очистка
rm -f $TMPFILE                                             ❻
```

❶ При выполнении сценария `scan.sh` будут проверены все хосты, находящиеся в файле с именем `hostlist`. Поскольку сценарию `scan.sh` имя файла в качестве аргумента мы не предоставляем, это имя сценарий сгенерирует сам. При этом используется числовой формат «год-месяц-день».

❷ Проименованные файлы, выводимые из сценария `scan.sh`, по умолчанию будут отсортированы. Команда `ls` вернет эти файлы в порядке, определенном датами их создания. При этом не потребуется указывать команде `ls` какие-либо специальные параметры. Используя команду `tail`, мы получим два последних имени из данного списка. Чтобы облегчить разделение на две части, поместим имена в массив.

❸ Создание с помощью команды `tempfile` временного имени файла — самый надежный способ убедиться, что файл не используется или не может быть записан.

❹ С помощью параметра `-s` проверяется размер файла: если он больше нуля, значит, файл не пустой. Временный файл не будет пустым, если при сравнении его размера с размером файла `fd2.sh` обнаружится разница.

❺ Для переменной `$USER` автоматически устанавливается идентификатор пользователя, однако, если адрес электронной почты отличается от идентификатора пользователя, в эту переменную может потребоваться поместить другое значение.

❻ Существуют более надежные способы убедиться, что файл удален, независимо от того, где и когда сценарий будет завершен. Но это минимум, позволяющий предотвратить накопление рабочих файлов. Сценарии захвата, использующие встроенную команду `trap`, вы найдете далее.

В операционной системе Windows запуск сценария `autoscan.sh` с заданным интервалом можно настроить с помощью команды `schtasks`. Для запуска этого сценария в Linux с заданным интервалом следует воспользоваться командой `crontab`.

Планирование задачи в Linux

Чтобы запланировать выполнение задачи в Linux, сначала нужно перечислить все существующие файлы cron:

```
$ crontab -l

no crontab for paul
```

Как вы можете убедиться, файла cron пока не существует. Для создания и редактирования нового файла cron укажите параметр `-e`:

```
$ crontab -e

no crontab for paul - using an empty one
Select an editor. To change later, run 'select-editor'.
  1. /bin/ed
  2. /bin/nano          <---- easiest
  3. /usr/bin/vim.basic
  4. /usr/bin/vim.tiny
Choose 1-4 [2]:
```

В любом редакторе добавьте в файл cron следующую строку, чтобы сценарий `autoscan.sh` запускался в 08:00 утра каждый день:

```
0 8 * * * /home/paul/autoscan.sh
```

Первые пять элементов определяют дату и время, когда будет выполняться задача, а шестой элемент — это команда или файл, которые должны быть выполнены. В табл. 9.1 описаны поля файла cron и их допустимые значения.

Для выполнения сценария autoscan.sh в качестве команды (вместо использования bash auto scan.sh) необходимо предоставить ему соответствующие полномочия. Например, с помощью строки chmod 750/home/paul/autoscan.sh владельцу файла (возможно, Paul) предоставляются права на чтение, запись и выполнение, а также разрешение на чтение и выполнение для группы и никаких других разрешений.

Таблица 9.1. Поля файла cron

Поле	Разрешенные значения	Пример	Значение
Минута	0–59	0	00 минут
Час	0–23	8	8 часов
День месяца	1–31	*	Любой день
Месяц	1–12, January — December, Jan — Dec	Mar	Март
День недели	1–7, Monday — Sunday, Mon–Sun	1	Понедельник

В примере, показанном в табл. 9.1, выполнение задачи начинается каждый понедельник марта, в 08:00 утра. В любом поле может быть установлено значение *, что эквивалентно любому значению.

Планирование задач в Windows

Запланировать автоматический запуск сценария `autoscan.sh` в Windows немного сложнее, так как изначально этот сценарий не будет работать из командной строки. Вместо этого вам нужно запланировать запуск Git Bash и в качестве аргумента указать файл `autoscan.sh`. Чтобы в системе Windows запланировать запуск сценария `autoscan.sh` каждый день в 08:00, напишите следующее:

```
schtasks //Create //TN "Network Scanner" //SC DAILY //ST 08:00
//TR "C:\Users\Paul\AppData\Local\Programs\Git\git-bash.exe
C:\Users\Paul\autoscan."
```

Обратите внимание: чтобы задача выполнялась правильно, нужно точно указать путь к Git Bash и сценарию `autoscan.sh`. При указании параметров обязательно используйте двойной слеш, так как сценарий будет выполняться не из командной строки Windows, а из Git Bash. В табл. 9.2 подробно описывается значение каждого из параметров.

Таблица 9.2. Параметры команды schtasks

Параметр	Описание
//Create	Создание новой задачи
//TN	Имя задачи
//SC	Частота расписания. Допустимые значения: минута, час, день, неделя, месяц, единожды при запуске, при входе в систему, при простое, при событии
//ST	Время запуска
//TR	Задание для выполнения

Выводы

Способность обнаруживать отклонения от установленного базового уровня является одной из самых эффективных при выявлении аномальной активности. Неожиданное открытие системой порта сервера может указывать на наличие сетевого бэкдора (backdoor). В следующей главе мы рассмотрим, как для обнаружения в локальной файловой системе подозрительной активности можно использовать определение исходного состояния этой системы.

Упражнения

Попробуйте расширить и настроить функционал инструмента мониторинга сети, добавив следующие возможности.

1. При сравнении двух отсканированных файлов следует учитывать разницу в их размерах или разницу в наборах IP-адресов/имен хостов.
2. Используйте `/dev/tcp` для создания простейшего SMPT-клиента, чтобы сценарий не требовал наличия команды `mail`.

Чтобы просмотреть дополнительные ресурсы и получить ответы на эти вопросы, зайдите на сайт https://www.rapidcyberops.com/.