Assignment 4 Group
Student names and numbers:

-
-
-
-

SSH Brute Force

To protect the Ethical VM against brute force attacks on SSH passwords, we can implement several countermeasures to hinder external attackers.
The first countermeasure we have to implement is an **account lockout policy**. This allows us to lock user accounts after a certain number of failed login attempts. This helps to prevent brute force attacks by temporarily disabling the account after a specified number of failed login attempts within a given time frame. There are multiple ways to implement the above solution since there are multiple tools that allow the monitoring of login attempts (some versions of OPNsense itself can do exactly that) but we will use Fail2ban. Fail2ban scans log files (e.g. */var/log/auth.log*) and bans IPs that show any malicious signs -- too many password failures, seeking for exploits, etc. Generally Fail2Ban is then used to update firewall rules to reject the IP addresses for a specified amount of time, although any arbitrary other action (e.g. sending an email) could also be configured. Fail2Ban comes with filters for various services (apache, courier, ssh, etc) so we can easily configure it for ssh monitoring. The steps to set it up are the following:

1. We first need to install the Fail2ban service

```
sudo apt install fail2ban
```

2. The fail2ban service keeps its configuration files in the */etc/fail2ban* directory. There is a file with defaults called *jail.conf*. As the jail.conf file warns us: “you should not modify this file, but provide customizations in jail.local file, or separate .conf files under jail.d/ directory”. So we will do exactly that by copying the original file and applying our modification in the copy

```
sudo cp jail.conf jail.local
sudo nano jail.local
```

3. The settings we added were the following

```
[sshd]
enabled = true
port = ssh
filter = sshd
logpath = /var/log/auth.log
maxretry = 5
findtime = 3m
bantime = 10m
action = $(action_)s
```

The *maxretry* variable sets the number of tries a client has to authenticate within a window of time defined by *findtime*, before being banned. With the above settings, the Fail2ban service will ban a client that unsuccessfully attempts to log in 5 times within a 3 minute window. It then will be banned for 10 minutes. The ban action is the default one (called with the parameter *$(action_)s*) and it is executed automatically by Fail2ban as soon as it is triggered, by simply adding a rule in the *ufw* that blacklists the source IP of the ssh failed connections.

4. At this point, we can enable our Fail2ban service so that it will run automatically and then start it with the following commands

```
sudo systemctl enable fail2ban
sudo systemctl start fail2ban
```

Another countermeasure we have already implemented in the previous assignments is to **limit SSH access**. This consists in restricting SSH access to only authorized users and IP addresses. In fact we configured the firewalls of our ACME network to allow SSH connections only from the hosts of the CLIENTS subnet (ssh request in assignment 2). This reduces the attack surface and limits the potential for unauthorized access because if an attacker would execute an ssh brute force against the Ethical VM, the login requests must have a source IP in the CLIENTS subnet; this make the attack harder since the malicious actor must already have a local access inside the CLIENTS network or he must execute an IP spoofing attack too and we already took countermeasure for that kind of attacks (ip spoofing assignment 2).

Note that we already have implemented in our ACME network a fairly efficient **logging system** with the Graylog server that will alert us as soon as there are some consecutives failed login attempts. To include the Ethical VM inside the logging mechanism, we can simply follow the steps explained in the assignment 3 to configure the vulnerable VM to send all his logs to Graylog. In this way we will have both Fail2ban acting directly against the bruteforce and the Graylog server monitoring all the logs on the machine itself (and give us a more general view on what's happening).

Drupal 7.26 vulnerability

The fastest countermeasure against a website vulnerability is a **Web application firewall** (WAF): it is a service that can inspect and filter incoming web traffic, detect and block SQL injection attempts and other malicious requests, providing an additional layer of protection.
To set up a Web Application Firewall (WAF) on the Ethical VM, we can use a popular open-source WAF called ModSecurity along with the Apache HTTP Server. Here's the path we can follow to set it up:

1. Begin by installing the Apache HTTP Server if it's not already installed. We can do this by running the following command in the terminal:

```
sudo apt install apache2
```

2. Install ModSecurity, which is a WAF module for Apache, by running the following command:

```
sudo apt install libapache2-mod-security2
```

3. Enable the ModSecurity module by running the following

```
sudo a2enmod security2
```

4. Modify the ModSecurity configuration file to customize the rules and settings. The configuration file is located at */etc/modsecurity/modsecurity.conf*.

5. Enable the OWASP ModSecurity Core Rule Set (CRS): The OWASP ModSecurity CRS provides a set of predefined rules to protect against common web application vulnerabilities. Enable the CRS by running the following commands:

```
sudo apt install -y git
sudo git clone https://github.com/coreruleset/coreruleset
/etc/modsecurity/crs
sudo mv /etc/modsecurity/crs/crs-setup.conf.example
/etc/modsecurity/crs/crs-setup.conf
```

6. Include the ModSecurity configuration in the Apache server configuration by editing the Apache configuration file located at */etc/apache2/apache2.conf*. Add the following line at the end of the file:

```
Include /etc/modsecurity/modsecurity.conf
```

7. Restart the Apache service to apply the changes:

```
sudo service apache2 restart
```

Once these steps are completed, ModSecurity will be active as a WAF protecting the Ethical VM web applications.

Sambacry

The most direct action to block this attack is to **prevent the SMB anonymous access** even though it is enabled in the SMB service. To do that we can use Fail2ban again (since we have already installed it) to immediately block the IP of the host connected as an anonymous user in SMB.

1. We need to configure a Samba Jail. Take the configuration file /etc/fail2ban/jail.local that we already edited for the SSH brute force attack and add the following configuration:

```
[samba]
enabled = true
port = 139,445
```

```
filter = samba
logpath = /var/log/samba/log.%m
maxretry = 1
```

This configuration sets up a jail named "samba" that monitors ports 139 and 445 used by Samba. It uses the "samba" filter, which is responsible for parsing the Samba logs. Since the maxretry value is set to 1 then a single authentication request as anonymous is enough to immediately block the source IP of the connection.

2. Create or modify the Samba filter file to match the log entries for anonymous access attempts. Create a file named */etc/fail2ban/filter.d/samba.conf* and add the following content:

```
[Definition]
failregex = ^.*authentication for user.*as anonymous.*
ignoreregex =
```

This filter will match log entries containing the phrase "authentication for user" and "as anonymous," indicating an anonymous access attempt.

3. After making the necessary configurations, restart the Fail2Ban service for the changes to take effect:

```
sudo service fail2ban restart
```

Fail2Ban will now monitor the Samba log files for anonymous access attempts and block IP addresses that exceed the configured maxretry value.

Buffer Overflow

Performing a buffer overflow attack on a vulnerable program can have severe security implications. It's important to take appropriate measures to mitigate the risk of such attacks.
We discovered the existence of a tool called **auditd**, which is responsible for monitoring and recording system activities. We can use auditd to monitor a single script and generate alerts if that vulnerable script is executed. By setting up an appropriate audit rule, we can track specific events related to the execution of the script so that, as soon as the script is executed, auditd will write an alert and we can set up an action through **Fail2ban** that will kill the vulnerable script completely.

1. We can install auditd using the package manager of Linux distribution.

```
sudo apt install auditd
```

2. To monitor the execution of the script and detect a potential buffer overflow, we can create an audit rule to track the execution of the script and any related system calls. The rule can be added to the */etc/audit/audit.rules* file or a separate file in the */etc/audit/rules.d/* directory. For example, we can add the following rule:

```
-a always, exit -F path=/Desktop/scripts/c_scripts/test.c. -F arch=b64
-S execve -k script_execution
```

This rule will generate an audit event whenever the specified script is executed using the *execve* system call. The *-F arch=b64* option ensures that the rule applies to 64-bit systems. Note that this rule will surely be triggered when the script is executed through the bash terminal since the bash terminal executes scripts using the execve system call.

3. Now we need to configure auditd to log events to a specific file by opening the *auditd.conf* file (*/etc/audit/auditd.conf*) and adding the following line

```
log_file = /var/log/audit/audit.log
```

4. Restart the auditd service to apply the changes

```
sudo service auditd restart
```

5. Now configure fail2ban. Open the jail.local configuration file for editing located in the usual path */etc/fail2ban/jail.local*. Add a custom jail to monitor the auditd log file for the vulnerable script execution event.

```
[vulnerable-script]
enabled = true
filter = vulnerable-script
logpath = /var/log/audit/audit.log
```

6. We now need to define the filter used above (since we are not using the default ones). Create a new file, which we called *vulnerable-script.conf*, in the filter.d directory (*/etc/fail2ban/filter.d/*). Define the regex pattern to match the log entry associated with the vulnerable script execution event.

```
[Definition]
failregex = ^.* vulnerability detected in vulnerable_script\.sh:.*
```

7. We can now create a new script called *terminate-vulnerable-script.sh* (the script that will kill the vulnerable program), inside /Desktop/scripts.

```
#!/bin/bash
pkill -f /path/to/vulnerable_script.sh
```

8. Save the file and make it executable:

```
chmod +x /Desktop/scripts/terminate-vulnerable-script.sh
```

9. Open the jail.local configuration file for editing (*/etc/fail2ban/jail.local*). Define a custom action under the [vulnerable-script] jail section.

```
[vulnerable-script]
action = %(action_)s /Desktop/scripts/terminate-vulnerable-script.sh
```

This action will simply execute the script as soon as it is triggered.

10. Restart the fail2ban service to apply the changes

```
sudo service fail2ban restart
```

This is obviously a particularly strict countermeasure since we will not be able to terminate the execution of the vulnerable script anymore (whether we are trying to exploit the script or fairly executing it for its intended purpose). However, there are multiple rules that can be implemented in auditd that can monitor not only the execution of the script but other more specific indicators of a running BOF attack, so that, the termination of the process, is executed only if the script is being exploited. Moreover, the action triggered by Fail2ban could be less strict: we could for example kill all the root shell sessions opened after the execution of the vulnerable script or simply send an alert to the IT security department.

Note that since the Ethical VM already sends all his logs to Graylog (thanks to the previous countermeasures), an eventual SOC analyst monitoring the logs could easily see that the vulnerable script is being executed.

Journalctl Weakness

We couldn't think of any way that prevents this kind of attack from happening without altering the configuration of the sudoers file so the least we could do is to limit the damages. We implemented a script scheduled in the crontab that periodically checks for open root shell sessions and closes any session that was generated by journalctl. The script is the following:

```
#!/bin/bash

# Get the list of root shell sessions
root_shell_sessions=$(ps -U root -o pid,tty,args | grep -E
"^\s*[0-9]+\s+pts" | grep -v "grep")

# Iterate over the sessions
while read -r session; do
    pid=$(echo "$session" | awk '{print $1}')
    tty=$(echo "$session" | awk '{print $2}')
    cmdline=$(echo "$session" | awk '{$1=$2=""; print $0}')

    # Check if the session is generated by journalctl
    if [[ $cmdline == *"journalctl"* ]]; then
        echo "Closing root shell session: PID=$pid TTY=$tty"
        kill "$pid"
        logger "Warning: root shell opened from journalctl"
    fi
done <<< "$root_shell_sessions"
```

This script retrieves a list of root shell sessions opened. It uses *ps* to list processes owned by the root user (*-U root*) and extracts the process ID (*pid*), terminal (*tty*) and command line arguments (*args*). The output is then filtered to select only the sessions associated with a pseudo-terminal (*pts*) and excludes any lines containing the grep command itself.

Then it starts a loop to iterate over each session in the root_shell_sessions list. For every session it extracts the process ID, the terminal tty and the command line arguments. If the cmdline contains the substring "*journalctl*" then this means that the shell has been generated from a malicious use of journalctl and kills the session.
We saved this script in /Desktop/scripts/, gave it execute permissions and scheduled it to run periodically:

```
chmod +x check_journalctl_sessions.sh
sudo crontab -e
*/2 * * * * /Desktop/scripts/check_journalctl_sessions.sh
```

So every 2 minutes this script will be run as root and will check if any root shell session is being opened from journalctl. If it finds one, it immediately closes it and writes a log in the system log file. So an eventual attacker will have a limited amount of time to execute malicious actions as root and in the meanwhile a log will be immediately sent to Graylog.