3 - Secure Server Management

Used tools

- scp to get/send files into the VM over ssh
- ssh-keygen to generate the private/public key pair
- ssh-copy-id to copy the public ssh key into the VM
-

Modified NAT rules

3.1 OpenSSH Configuration

a

1. added `AllowUsers user` to the `/etc/ssh/sshd_config` file to make `user` the only account that can login through ssh
2. restarted the server with `sudo systemctl restart sshd`

b

1. generated a private/public rsa key pair with `ssh-keygen`
2. copied the public key into the VM with `ssh-copy-id` (copies the id_rsa.pub into ~/.ssh/authorized_keys)
3. uncommented and changed `# PasswordAuthentication yes` to `PasswordAuthentication no` in the `/etc/ssh/sshd_config` file to force the ssh login to happen only through rsa key mechanism
4. restarted the server with `sudo systemctl restart sshd`

For this to work, the `id_rsa.pub` file must be copied to the `/home/.ssh/authorized_keys` file.

Note: the `authorized_keys` file was changed according to the machine from which the rsa key was copied which was a DICE computer, kranjeck, in this case.

3.2 Fun with Heartbleed

a

The Heartbleed bug will allow an attacker to access memory values that he should not have access to. After establishing an SSL connection with the vulnerable server, an attacker will send a heartbeat request (this is a feature implemented to guarantee that the connection is still alive and also to avoid the connection from breaking by being idle for too long). What happens if the server is vulnerable is that upon requesting a heartbeat from the server, the user can will send a message and also the size of that message, if the attacker specifies a length greater than the real message sent, the vulnerable server will read the extra characters (max 64Kb) from memory. This heartbeat request can be performed multiple times and this will likely result in leaking different information from the server memory.

Naturally, the consequences involve data being compromised, and this data can be anything from password, private key files and so on.

To test the VM's vulnerability I took the following steps:

1. started the openssl service with `openssl s_server -key /home/user/key/key.pem -cert /home/user/key/cert.pem -accept 12345 -www`
2. used a python PoC script available on GitHub to test with the following command `python heartbleed.py localhost -p 54321` and got the confirmation

b

The steps taken to fix the bug were:

1. find and analyse the patches done by the openssl community available here
2. identify the file that implements the TLS heartbeat mechanism: `ssh/t1_lib.c`
3. identify where the bounds check is missing in the heartbeat logic that allows for heartbleed (looking for `#ifndef OPENSSL_NO_HEARTBEATS` helped). The function in question is `int tls1_process_heartbeat(SSL *s)` (line 2554)
4. introduce the bounds check to test that the length of the heartbeat request is within bounds
5. recompiled and tested again with the same tools as described in the previous section and received a `Server likely not vulnerable`message
6. created the diff for the `ssl/t1_lib.c` file

3.3

The specifications for questions `a,b,c,d` have been satisfied in the `Super Secure Digital Signature Service` web application.

The following configuration files have been changed and are available in the tarball:

- `/etc/php/php.ini` so as to enable the `openssl.so` extension
- `/etc/httpd/conf/httpd.conf` by enabling `AllowOverride All` so as to be able to use custom `.htaccess` files, namely inside the `keys` directory.

The following sections (a-d) briefly describe how the functionality was implemented.

a Sign up and login

There are 2 forms in `index.php`, namely:

- signup which will use the `register.php` endpoint to register a new account and also create the private/public key pairs (and store them in the `keys/` folder)
- login which will use the `login.php` endpoint to validate the provided credentials

After a successful register/login action the authenticated user is redirected to the `dashboard.php` page where the functionality for points `b, c and d` is implemented

b Export Public Key

In the dashboard, the user need only click the `Export Public Key` link that will trigger the `export.php` to download the public key in .pem format.

c Digital Signing

In the dashboard, the user need only input a message and click the `Sign message` link, this will redirect the user to a new page (`sign.php`) with the signed message. The signed message is encoded in base 64, which means the digital signature verification will perform the decoding.

d Digital Signature Verification

In the dashboard, the user need only input a message, paste in the signed message, choose a pem file and then click `Verify Digital Signature`, this will redirect the user to a new screen (`verify.php`) which will display the result of the verification as either `"This digital signature is valid"` or `"This digital signature is NOT valid"`.

e Security evaluation and countermeasures

Mitigation 1

Removing the `include/.admin.php` file which is a typical mistake of leaking information relating to the server. This kind of information, although relevant for debugging purposes can be used by attackers to quickly identify known vulnerabilities associated with the system or with the enable libraries. To note that this is not security by obscurity, but it is rather us following the principle of least privilege, since the application will be exposed on the internet.

BONUS: Similarly, in the original code, we had error enabling functions like `ini_set('display_errors', 'On');` that have been removed as this is no longer considered a debugging environment and these instructions could help attackers during their enterprises.

Mitigation 2

The original `login` function in the `include/function.php` file was vulnerable to SQL Injection attacks as the user input was being combined directly with the SQL query. This was fixed by rather using prepared statements.

Mitigation 3

The original mechanism to handle sessions, which involved the functions `create_token`, `check_token`, `logout`, `check_signed_in` was vulnerable due to the fact that cookies can easily be changed by the user. In fact, the session detection mechanism was based on checking that the cookie named `session` matched the hashed version of the username through the md5 algorithm. This is naturally a bad way of ensuring authentication, any user who came across this fact need only know the username of another user, calculate its md5 hash and replace the `session` cookie value.

The mitigation involved using php's default session handling mechanism that manages the session cookie without making it predictable. The aforementioned functions were updated to contemplate this and the `session_start();` command is invoked whenever we need to check if there is a valid user session.

Further effort could be put into this, namely by forcing a more recurring cookie refreshing mechanism (by default there is a 1% probability a cookie is refreshed when `session_start();` is called) so that, even if a given cookie was compromised it would be less likely that an attacker could take advantage of it. Moreover, mechanisms like associating the IP with the session could also be considered as they would have a positive impact on the user session's security.

Mitigation 4

Protecting the private (and even public) keys is a potential security risk. In the current version of the application, the keys are saved in the `keys` folder. Since the application is exposed on the web, anyone could read them. To avoid this a custom `.htaccess` file was added that prevents anyone from reading it from the client-side.

This mitigation is a good first step, but other measures could be taken like saving the files in a folder that is not inside `/srv/http` folder, encrypting the files using the user password, and more.

Mitigation 5

The initial implementation was also incurring in a very dangerous habit - that of storing the password as plain text in the database. This would mean that, if the database contents were exposed in a breach, every user's credential would be exposed and any attacker could use those to access the private keys and leading to an integrity risk in further signed messages.

To avoid this, the passwords are now being hashed with the default algorithm the current php version deems most appropriate (bcrypt, in this case), this includes a random salt for each password and also the hashes are calculated with a given cost (that could be increased to make the passwords less vulnerable to hash attacks if their hashed representations were leaked).

The authentication also uses the advised method `password_verify` that performs the password comparison in a time-safe manner.

Naturally, mechanisms like two-step authentication would bring an extra layer of security but those were a bit overkill for the case in hand 😃

Mitigation 6

The final mitigation that is being described is that the database used was both publicly accessible and publicly readable.

To address the first issue, the same technique used for isolating the `keys` folder was used, namely including a custom `.htaccess` file that prevents anyone from reading it from the client-side.

The second issue could be solved by making the sqlite database password protected.

Ideally, the database would be separated from the server code and they would only have a strictly controlled communication stream but introducing isolation in the form of, for instance, docker containers would not be adequate for this development environment.