Internship report

Scanning Modern Web Applications with OWASPZAP

Xavier Maso

May 2018 - August 2018

Acknowledgements

I would like to express my gratitude to Mozilla for having me this summer to learn about and work on the rather new field of "client-side web security"; it was a great chance to do research and development on this subject.

I take this opportunity to express my deepest thanks to my mentor Simon Bennetts, whom, despite being busy with his duties, has always been present to listen, help and guide me; his time, his patience and his advices take a huge part in the success of my internship.

I am grateful to the Firefox Operations Security and ZAP teams, for trusting me to be part of their group, their efforts to introduce me to their work and their will to make me learn as much as possible.

I express my special thanks to Thomas Wisniewski and Mark Goodwin, from respectively the Web Compatibility and the Crypto Engineering teams, whom without their work and their help, I would never have been able to go as far as I did.

At last, I am very grateful and consider myself lucky to have been surrounded by wonderful and supportive people at all times, in particular in the London office; all Mozillians play a huge role in what makes this company such a great place to work.

Hope to meet all of you again in the future,

Contents

Introduction

In the last couple of years, the landscape of web development has drastically changed. Web applications used to rely heavily on the back-end side, web servers, to host their logic and data. Nowadays, it is common to see the front-end side, web browsers, perform some of the tasks the former exclusively did.

As a result, new kind of vulnerabilities have made their apparition and have already had some dramatic consequences. Old techniques and tools to do security testing of web applications are not as relevant as they used to be, and need an update to carry the work they used to accomplish.

And that is exactly why I was working at Mozilla for the summer: enhance the Zed Attack Proxy (ZAP), an open source security tool they use internally to test the security of some of their applications. Our main goal was to help ZAP handle better what we called "modern" web applications.

In this report, I will present the work that has been carried out during these twelve weeks of internship to achieve this.

Firstly, I will talk about ZAP, the tool that we (me and the team) have been working on. I will take some time to give a bit of background on how it is used in the wild and present basic concepts that we leveraged when building the new features that we expected to bring to the field. Then, I will discuss our definition of "modern" web applications, and talk about the problems they introduce in terms of security testing, particularly what make them so difficult to test with ZAP (or any other security tools) as for now.

In the second chapter, I will present the first big milestone of our effort: coming up with an innovative way for testers to run checks in the client-side using our proxy. We proposed a solution for them to write scripts, that ZAP will inject in the targeted application's web pages as those ones transit from the server to the web browser. Hopefully, our design is flexible enough to ease later development, especially the additions of features to allow more automation.

At last, I will take a look at a library that we wrote, to help analyze modern applications. Despite being one of the critical piece of the work I present in the second chapter, it is meant to be a standalone component that could be used by other tools.

1 ZAP and "modern" web applications

1.1 OWASP ZAP

The Open Web Application Security Project (OWASP)[1] is an online community producing tools, tutorials and guides in the field of web application security.

The Zed Attack Proxy (ZAP)[2] is an open-source project written in Java related to this community. Actively developed by a core team of five members, it has had more than a hundred and twenty contributors[3].

As its name suggests, ZAP is a web application proxy, and it is meant to help the security testing of such applications.

1.1.1 Usage

The "traditional" way of using a web application proxy is to make it sit between a web browser and a server hosting the application that is targeted.

By doing so, the proxy can then eavesdrop on the HTTP traffic generated when the user interacts with the application. Furthermore, it even allows him to tamper with requests and responses or replay them.

These features enable ZAP users to perform different kind of testing, aiming to ensure that the transiting content is secure, but also to try to exploit vulnerabilities on the server as well as on the client-side.

1.1.2 Concepts

To validate the security of the watched content, ZAP performs different checks on it. Those checks usually consist of trying to recognize patterns and signatures aiming to detect the use of insecure parameters or known vulnerabilities in the HTTP traffic. Either such content comes from "legitimate" snooped traffic, or comes from after ZAP initiated it itself (by sending crafted requests for example).

This process of validating security and looking for known vulnerabilities in an automated way is called **scanning**.

The rules describing what to look for and how to do it during the scanning of web applications are expressed via **scripts**.

[1] `https://www.owasp.org/`

[2] `https://www.owasp.org/index.php/ZAP`

[3] Refer to `https://www.openhub.net/p/zaproxy/contributors/summary` for more details.

By default, ZAP supports the following two languages to express scripts: JavaScript (using the Nashorn engine) and Zest[4], but it can as well support others (Ruby and Python for example) via add-ons installation.

Some scripts are included by default in ZAP, while others can be installed (via these add-ons, or from other sources). Users are free to write their own as well, and define whenever (under which conditions) those should be active or not. There exist a repository of such "user defined scripts", written by the community, and made available at `https://github.com/zaproxy/community-scripts`.

Generally speaking, we can distinguish two types of scripts and thus, two types of scanning: passive and active.

"Passive scanning does not change the requests nor the responses in any way and is therefore safe to use." [5] In other words, passive scanning performs "read-only" lookups over the traffic to recognize the use of insecure parameters or vulnerabilities.

"Active scanning attempts to find potential vulnerabilities by using known attacks against the selected targets." [6] It could as well be named "proactive" scanning, with ZAP tampering with the content, at the risk of disrupting the usual behavior of the targeted application.

To notify the user when it finds something, ZAP creates **alerts**. "An alert is a potential vulnerability and is associated with a specific request"[7]; it is made of several fields, among which: its description, the evidence that the reported vulnerability is present in the application, the level of risk associated with it (how critical is the vulnerability), and the level of confidence (the likeliness for it to be a false positive or not).

At last, one way for users to extend ZAP functionalities is by using (or writing) **add-ons**[8].

They can be installed via the "marketplace"[9], which exposes the available ones from the repository: `https://github.com/zaproxy/zap-extensions`, maintained by the ZAP core team, and on which the community contributes.

For the most adventurous of us, it is possible to write such add-ons. They can make use of ZAP internals to create very powerful features, and, once passed the community review process, can be added to the "zap-extensions" repository, and be made available on the marketplace for other users to install.

[4] "Zest is an experimental specialized scripting language [...] developed by the Mozilla security team and is intended to be used in web oriented security tools.". Source: `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Zest`.

[5] Source: `https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsPscan`.

[6] Source: `https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAscan`.

[7] Source: `https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAlerts`

[8] Source: `https://github.com/zaproxy/zap-core-help/wiki/HelpStartConceptsAddons`.

[9] An element of ZAP's graphical interface.

By leveraging the concepts presented up until now, we have now a clearer understanding of what a web application penetration testing session might look like. The security tester (the ZAP user) set up ZAP to be between the browser and the server serving the application he is testing. He can make use of custom scripts and add-ons to help him look for particular vulnerabilities. If such vulnerabilities are spotted in the application, alerts will be reported and displayed for the tester to investigate.

1.2 "Modern" web applications

Traditionally, the logic of web applications (the rules describing their behavior) used to be hosted on the server side only; and the users of such applications would communicate with it by making HTTP requests to read data and perform changes.

In the last few years, mostly because of the increasing computing power of client machines, more and more of web applications started to have (and sometimes to delegate) part of their logic to the client-side. We use the term "modern" to identify such applications. They were able to do so thanks to JavaScript, a twenty-two years old programming language interpreted by web browsers, which was aimed to be a scripting language for the web, allowing for example to create dynamic web pages.

Because the browser is now hosting some of the application's logic (such as navigation, or data validation in some cases), and might contains interesting data (such as authentication credentials for example), it is necessary to be able to audit this part as well from the security point of view.

However, by proceeding how we did so far (placing our proxy between the browser and the server hosting the targeted application), we are limited in what we can achieve and find on the client-side; and thus making it difficult to scan "modern" web applications with the tools and methods available to us up until now. Indeed, analysis of the HTTP trafic only allows static research of weak points. On another hand, tampering or replaying such trafic is fine to trigger "measurable" behavior on the server side, but is most of the time inefficient when looking for client-side vulnerability, due do the environment in which the application runs in there, where it intakes several sources of input (user interactions, networking, etc.), coupled to the dynamic nature of JavaScript.

To understand better at which extent this is problematic, we need to take a look at the JavaScript language, and at the representation of web pages interactions on the client-side: the Document Object Model (or DOM).

1.2.1 JavaScript

JavaScript is the most well-known implementation of the ECMAScript specification, and is supported by all major web browsers. Any of them embed an engine to interpret

the language, amongst the most famous are SpiderMonkey, V8, Chakra, and JavaScript-Core respectively for Firefox, Google Chrome, Internet Explorer / Edge, and Safari)[10].

JavaScript is a weakly typed, prototype-based, high level, interpreted language; but more importantly, it is its dynamic nature that makes it difficult to analyze statically: in particular because that new code can be added at runtime[11]. And this is especially troublesome in an environment with a lot of external inputs: the web browser.

1.2.2 DOM

"When a web page is loaded, the browser creates a Document Object Model of the page, which is an object oriented representation of an HTML document, that acts as an interface between JavaScript and the document itself and allows the creation of dynamic web pages"[12].

Via this interface, JavaScript can then perform different kind of manipulations to the page: adding, modifying or removing HTML elements, react to events or create new ones.

These events management, done via the Event interface, is particularly interesting: it allows scripts to execute behavior in response of things happening during the life of a web page. Some of these events are generated by user interactions: mouse, keyboard or form, while others are generated by APIs: resource, network, etc. An exhaustive list of existing DOM events is available on MDN: `https://developer.mozilla.org/en-US/docs/Web/Events`. Another interesting usage of the Event interface is the creation of custom events to tie a specific behavior to. With such a mechanism, one can write a reactive application: creating custom events to be emitted on certain conditions and trigger a specific behavior when that happen.

Getting back to our need of scanning web applications for client-side vulnerabilities, we can now see why this is troublesome for the two methods we have: scanning the HTTP response for known issues, and injecting content aiming to trigger something in the browser.

The scanned HTTP response contains HTML, as well as JavaScript (as we are talking about "modern" web applications). The first will be interpreted by web browsers to create the DOM representation they use. The latter will be interpreted as well, potentially leading to "direct" modifications of this DOM; or can create event listeners, which could lead to such DOM modifications, or at least make it difficult to keep track of what can happen in a page as it would depend on external input (user generated or not).

For example, one common use case (meant to speed up the load of a page) is to load data asynchronously, and then creates HTML elements containing this new content to be displayed.

In the end, our static analysis will miss most of the behavior that requires interactions to happen, and injecting content hoping to see an outcome in the browser would be

[10] `https://en.wikipedia.org/wiki/JavaScript`
[11] *Ibid.*
[12] `https://en.wikipedia.org/wiki/Document_Object_Model`

naively optimistic as some really complex chain of events can be required for sources of vulnerabilities to appear on the page.

1.3 Our work

We wanted to enhance ZAP to be able to scan "modern" web applications. To answer the problematics previously discussed, we articulated our work around two ideas that will be presented in the next two chapters of this document.

The first one was to provide a way for security testers to run scripts in the browser and for those to be able to report informations back to ZAP. We need these scripts to run "alongside" targeted applications, or, in other words, with the ability to manipulate the same DOM and access the same data (having the same origin) than these. Our solution for that was to develop a ZAP add-on named the **FrontEndScanner**, which inner working will be presented in the next chapter.

Secondly, we wanted to enhance the API available in the browser for the scripts to be able perform "security related" actions. Indeed, the DOM and ECMAScript specifications are meant to solve web developers' needs, but do not provide the features we expect for debugging and even less for security testing. To be fair, we should mention the WebExtensions API, available for browser extensions developers, as this API provide interesting features for our use case. However, WebExtensions still lack of cross-browser compatibility: originally based on Google Chrome Extension API, the ensuing standard specification is still a draft[13], which Firefox has not yet completely implemented. This would have forced us to write and maintain plugins for each browser we wanted to support; and worse, it would have required our users to install extensions to their browsers, which some of them simply cannot do. In the end, our solution took the form of a JavaScript library named the **front-end-tracker** to add these "missing" tools. Once injected in a web page, all scripts can take advantage of the functionalities it exposes, in particular ZAPscripts, thanks to the FrontEndScanner.

[13]It can be consulted at the following address: `https://browserext.github.io/browserext/`.

2 The FrontEndScanner

As discussed in the previous chapter, we needed a mean for our users (pentesters) to be able to look for client-side vulnerabilities by running scripts in the browser. There, we wanted those scripts to have access to the same DOM than the targeted application, and the permission to query the same data.

To answer that, we produced the FrontEndScanner ZAP add-on.

2.1 Presentation

When installed and enabled, it tampers with all HTTP responses passing through ZAP to inject a piece of JavaScript code containing custom logic, utility functions and scripts that our users wish to run in the browser. For the user scripts to be executable there, and to leverage the DOM API, we decided that they should be written in JavaScript, and injected as is into responses.

2.1.1 Same-origin policy (SOP)

"Under the policy, a web browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin. An origin is defined as a combination of URI scheme, host name, and port number."[1]

The SOP is enforced by web browsers and is implemented to prevent malicious pages to access sensitive data contained in other web pages they do not share the origin with. One common example of such sensitive data would be HTTP cookies, used to maintain authenticated user sessions via a stored secret value that only the authenticated client knows; if any other web page loaded in the browser could access this data, they would have a valid user session on the first page. For example, as an attacker, one could gain access to a user's session on `banking.com` by making them load a page under his control (let's say `evil.com`), and that would be problematic.

One advantage that we automatically gain by injecting our piece JavaScript directly into the HTTP response is that we do not have to worry about the SOP. Indeed, with our add-on, our JavaScript code ends up between the `<head>` and `</head>` tags of the document, and then the browser will interpret all the new content as being part of the targeted domain. For the browser, ZAP scripts will then look like any legitimate scripts from the website, and thus have the ability to perform the same actions and to have access to the same data.

[1] Source: `https://en.wikipedia.org/wiki/Same-origin_policy`.

2.1.2 Content Security Policy (CSP)

"Content Security Policy (CSP) is a computer security standard introduced to prevent cross-site scripting (XSS), clickjacking and other code injection attacks resulting from execution of malicious content in the trusted web page context. [...] CSP provides a standard method for website owners to declare approved origins of content that browsers should be allowed to load on that website." [2]

As the previous one, this policy is enforced by web browsers. They do so by following the declaration sent by the website via the "Content-Security-Policy" header in the HTTP response. One common usage of it is to disallow inline JavaScript (contained in attributes of HTML elements or between `<script></script>` tags), to refuse the use of dynamic code evaluation (via the use of `eval` or `new Function`), and to specify from which origins to allow content from (allowing the load of resources from a CDN for example).

Unfortunately, our approach does not work straightaway for websites including a CSP that disallows inline JavaScript, because ZAP adds the code in the page between `<script></script>` tags. It is necessary for now to remove the CSP header from HTTP responses[3]. However, two issues have been raised in the ZAP repository: "Automatically manipulate CSP for it to work"[4] and "Do not inject inlined JavaScript"[5] and we hope to solve this problem with a more graceful solution.

2.2 The "Client-side JavaScript code"

So far, we have looked into what happen when HTTP responses pass through the proxy with our add-on enabled before being sent back to the browser: a piece of JavaScript code is injected inside the `<head></head>` tags. In this section, we will present a bit more in details what this JavaScript code is made of, and why.

2.2.1 The "frontEndScanner" object

The first key element to understand what is happening in the Client-side JavaScript code is the **frontEndScanner** object.

In JavaScript, objects are maps of key-value pairs. Keys are often called "properties" of the object; however, when a value associated with a key is a function, we prefer the term "method".

The current implementation of the frontEndScanner object contains two properties and one method, respectively:

[2] Source: `https://en.wikipedia.org/wiki/Content_Security_Policy`.

[3] For example, by using one of the community script, available at the following address: `https://github.com/zaproxy/community-scripts/blob/75433c6253853560fa0eca4ec3fbc26b077b2e4f/proxy/RemoveCSP.zst`.

[4] `https://github.com/zaproxy/zaproxy/issues/4893`

[5] `https://github.com/zaproxy/zaproxy/issues/4894`

- ZAPconstants to help scripts create alerts.
- The "mailbox": a "publish-subscribe" object for scripts to react to interesting events happening in the web page.
- An utility function to report alerts back to ZAP.

ZAP constants

As presented in the first chapter, each alert has a risk and a confidence property, both are mandatory for it to be valid in ZAP. Those properties are enumerated types[6], and thus have a set of predefined possible values. These values, held in the `frontEndScanner.zapAlertConstants` property, which scripts have access to, need to be used when they create alerts that are valid from ZAP's point of view.

The "mailbox"

We will dive more in the details of the inner working of the mailbox (and which problem it solves) in the next chapter. For now, let's see it as an handy object which scripts can subscribe to, allowing them to react to what would be posted on it: events that are interesting from a debugging and security perspective[7]. It is accessible via the `frontEndScanner.mailbox` variable.

Report function

At last, the frontEndScanner object exposes a way for scripts to report their findings to be registered in ZAP.

The way for the Client-side JavaScript code to communicate with ZAP is via the "ZAP callback URL", which has the following format: `<targeted domain name>/zapCallBackUrl/<random value>`. The random value is unique and generated by ZAP, and is made known to the client side by being written during the injection.

When they pass through ZAP, HTTP requests for these URLs are intercepted and not forwarded. The random value from the URL is compared to what ZAP is expecting, and the request is not processed if they do not match. This is a security mechanism to prevent malicious pages to exploit ZAP. It is important for this random value not to be leaked by being accessible from other scripts on the target domain, and we do so by enclosing it in our report function[8]. One nice side effect of this mechanism is that, from the browser perspective, requests to the "ZAP callback URL" fall completely under the SOP.

Another information enclosed in the function, and that is used to craft valid alerts from ZAP's perspective is the "historyReferenceId". ZAP relates alerts to the HTTP

[6] See: `https://en.wikipedia.org/wiki/Enumerated_type`

[7] That could be notifications when DOM events are triggered, when storages are accessed, when the DOM is mutated, and so on.

[8] See: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures`.

requests and responses it analyzes. If one of the user scripts find a vulnerability in the web page it is analyzing, and wants to report it to ZAP, it needs to associate it with the corresponding HTTP exchange, which it cannot do without this historyReferenceId. Because it does not have the overview of the process, it needs to get this information from ZAP; and again, this is done as part of the injected Client-side JavaScript code.

In the end, our function has everything it needs to report individual alerts to ZAP, by passing them as an object parameter: `frontEndScanner.reportAlertToZap(alertData)`.

2.2.2 User defined scripts

Another big aspect of the Client-side JavaScript code is that it contains all the scripts that a user wishes to run in the targeted page.

Thanks to the "Scripts" add-on included by default in ZAP, users have access to different pre-defined scripts, and can even write their own. From the script pane, they can enable or disable them based on the actions they want to be performed. Depending on their type, scripts are run under different circumstances[9].

Because there was no way to run ZAP scripts in the browser before, there was no existing type representing ones that would. Therefore, we introduced two new types of scripts: "Client side Passive Scan scripts" and "Client side Active Scan scripts". They are similar to the existing "Passive Scan scripts" and "Active Scan scripts" types, except they are meant to be run in the browser. In that sense the "active" one designates scripts that interacts with the application and can potentially alter its content, while the "passive" type should not[10].

By default, only the enabled client-side passive scripts are injected in the HTTP response. Each of the injected script is injected into the body of a JavaScript function, which one takes the frontEndScanner object we presented earlier as a parameter; the function names are stored in an array for referencing. Thanks to this implementation, each script can make use of the ZAP constants, the mailbox and the report function we described earlier.

At last, the Client-side JavaScript code calls every one of these functions, resulting in the execution of users scripts in the targeted application. Voilà!

[9] See: `https://github.com/zaproxy/zap-core-help/wiki/HelpAddonsScriptsScripts#script-types` for more types and information.

[10] We would prefer to be writing **could** not, but so far, there is nothing implemented that enforces the differentiation between the two.

3 The front-end-tracker

The previous chapter described how we gave ZAP users the ability to run scripts in the browser, alongside the application they are testing. However, we found that the browser lacked the features for the scripts to be able to test "modern" web applications to their full extent.

3.1 "Missing" features

Here are a couple of examples of what we would like scripts users to have access to, or be able to do. How we implemented the answer to these needs will be presented in the following section.

3.1.1 DOM events

Modern web applications rely a lot on DOM events to interact with users. By creating listeners, they can hook events to functions, which will be run when the former occur.

For example, imagining that there is an HTML element in the page with the id `an-id`, the following code will end up writing a message to the console every time a click happens on it.

```
var element = document.getElementById('an-id');
element.addEventListener('click', function (event) {
    console.log(event + ' happened');
});
```

As mentioned in the first chapter, in "modern" web applications, some complex chain of interactions might be necessary for the source of a vulnerability to appear (for example, a click on a "Log In" button leading to the apparition of an `<input>` tag, subject to an SQL injection). Before being able to automatically replay those interactions, we would need first to remember them. To do so, we would like to be notified when those events are triggered during the lifetime of the application.

3.1.2 DOM mutations

Another thing that "modern" applications do a lot is mutating the DOM. Indeed, instead of resending an HTTP request to update the content from an HTTP response (which takes a lot of time), they use JavaScript to manipulate the loaded DOM to change

pieces of the HTML tree. So, they can remove, add or modify almost any (if not all) HTML elements or their attributes.

In the case of the example presented earlier, detecting the creation of the `<input>` element and recognizing it as an "interesting thing" (the apparition of a potential source of vulnerability) is a necessary step before testing any kind of injection on it. More generally, we would like to be notified when anything that takes input from users of the application appears in the page.

3.1.3 Interactions with storages

To persist data directly in the browser, web pages scripts can use the "sessionStorage" and the "localStorage". They both can be accessed via the Storage interface[1].

From a security perspective, we want to be able to detect whenever those storages are accessed, in particular when secrets are stored (such as JWT[2]).

3.1.4 XMLHttpRequest (XHR) calls

XHR[3] helps applications query data asynchronously; again, it avoids the overhead of having a full HTTP exchange when only a piece of the web page needs to be updated. Nowadays, XHR is a very common way to pull down data from a web server exposing an HTTP API.

By monitoring such exchanges, we could detect when data is transferred between the front-end and the back-end of an application. That could lead to learn secrets, or just knowledge about the structure of it: the format of data it manipulates, the features it exposes, and so on.

3.1.5 postMessage calls

"postMessage" is a method exposed by the "Client" interface[4]; it is a mechanism with which pages can communicate between themselves. To watch over the communications of our application, it could be interesting to inspect messages entering and leaving it.

To react to messages, a "Worker"[5], a "SharedWorker"[6] or a "Window"[7] use the "addEventListener" mechanism presented earlier to hook to the "message" event. Because of that, the reception of messages will be dealt with by dealing with DOM events in general.

In the end, we still need a way to monitor calls to "postMessage".

[1] See: `https://developer.mozilla.org/en-US/docs/Web/API/Storage`.
[2] Json Web Tokens, often used as a proof of identity to maintain an authenticated user session. See `https://jwt.io/` for more informations.
[3] See: `https://developer.mozilla.org/en-US/docs/Glossary/XHR_(XMLHttpRequest)`.
[4] See: `https://developer.mozilla.org/en-US/docs/Web/API/Client/postMessage`.
[5] See: `https://developer.mozilla.org/en-US/docs/Web/API/Worker`.
[6] See: `https://developer.mozilla.org/en-US/docs/Web/API/SharedWorker`.
[7] `See:https://developer.mozilla.org/en-US/docs/Web/API/Window`.

And what else we did not think yet about!

3.2 Implementation

Ideally, we need a way to report when these features are called, without interrupting them. By taking advantage of the functional nature of JavaScript, especially the fact that functions are first-class citizens[8], we were able to do so with what we called **hooks**.

3.2.1 Hooks

Hooks are the way by which the standard behaviors we have presented in the previous section (DOM events, Storage interactions), are wrapped into custom functions written by us. By doing so, we can intercept when those features are called, do some kind of reporting (or even modification) before triggering the expected behavior.

In the current implementation, our custom function does a couple of things:

1. It prevents the default behavior from happening[9].
2. It reports the called function and the arguments that has been passed to it by posting to the mailbox (more on that later).
3. It calls the function, with the original parameters.
4. It recreates and dispatches[10] the event in the case its default behavior had been prevented.

Here is what it would look like, for example to wrap the `getItem`[11] function from the Storage interface. For the sake of the example, the code has been simplified to be more explicit.

```
const oldGetItem = Storage.prototype.getItem;

Storage.prototype.getItem = function (...args) {
    mailbox.publish(
      'storage',
      {action: 'get', args: args}
    );
    return oldGetItem(...args);
}
```

[8] Source: `https://en.wikipedia.org/wiki/Javascript#Functional`.

[9] This is mandatory as some behaviour would break the reporting. For example clicks on `<a>...</a>` pointing to a different origin would make the window loads another page, and our asynchronous reporting would sadly never be called.

[10] See: `https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/dispatchEvent`.

[11] See: `https://developer.mozilla.org/en-US/docs/Web/API/Storage/getItem`.

The hook for DOM events wraps "addEventListener" from the "EventTarget" interface[12]. By doing so, we can make sure that every time a behavior is bond to a script, it gets changed to inject some reporting code around the expected behavior.

The hook for Storage wraps "getItem", "setItem" and "removeItem" from the "Storage" interface[13] Again, this ensures that every time a script interacts with any storage, we will know it.

Unfortunately, we did not have the time to implement all the hooks that we hoped to[14]. So far only the ones for DOM events and Storage are available, and therefore, the hooks for DOM mutation, XHR and postMessage are still missing.

However, the problems that we are trying to solve, and the way we wanted to tackle them are really similar to what the web compatibility team at Mozilla want to deal with their "tinker-tester-developer-spy" plugin[15].

Most of the heavy lifting has been done on their side, and even though integration is not trivial[16], the hooks to deal with DOM mutations and XHR calls are already working in their plugin.

3.2.2 The mailbox

The **mailbox** is the core of the reporting capability of the front-end-tracker. It is a "publish-subscribe" object: wrappers created by hooks will publish messages to it (as shown in the previous dummy example), and whichever script in a web page can subscribe to it, as it is exposed as a global variable in the window. Furthermore, the mailbox is "topic based" means each message is assigned a topic (as a String), so subscribers can be notified only when messages under a topic that interest them is posted to the mailbox.

Subscribing to the mailbox means binding a function to the reception of a message of a given topic. For example, one can log DOM events' information by subscribing to the "dom-events" topic, and calling the JavaScript printing function on the received data, as shown in the following code:

```
const topic = 'dom-events';
mailbox.subscribe(topic, function (_, data) {
  console.log(data);
});
```

[12] See: https://developer.mozilla.org/en-US/docs/Web/API/EventTarget.

[13] See: https://developer.mozilla.org/en-US/docs/Web/API/Storage.

[14] You can see the current availabel hooks at the following address: https://github.com/zaproxy/front-end-tracker/tree/master/src/hooks.

[15] Meant to help web developers to debug the web applications for compatibility issues. Available at the address: https://github.com/webcompat/tinker-tester-developer-spy.

[16] As it requires a very powerful technique from a developer's arsenal: copy pasting code from a project to another. It has been documented here: https://github.com/zaproxy/front-end-tracker/wiki/Hooks#create-and-update, and an issue to find a more elegant solution has been raised: https://github.com/webcompat/tinker-tester-developer-spy/issues/13.

3.3 The library

3.3.1 Constraints

As mentioned earlier in this document, "modern" web applications rely heavily on event listeners to build reactive applications. Recent frameworks and libraries will abuse this mechanism as soon as they will be loaded in the page. To get relevant debugging capabilities, it is important for our hooks to run before these scripts, so they can inject debugging code (which role is to post relevant informations to the mailbox) when event listeners are added.

Note that in the previous chapter, I explained that the FrontEndScanner add-on injects the client-side JavaScript code in the `<head></head>` tags, on top of the page. This is exactly for this reason: to make sure that it is the first script loaded and interpreted by the browser.

ZAP lets users pick the browser of their choice when testing web applications... Or almost. The team carries the effort of supporting only "major" recent browsers, *i.e* the latest versions of Mozilla Firefox and Google Chrome. We wanted it to stay like that, so our front-end-tracker needs to be compatible with at least those two browsers.

I already mentioned in the course of the first chapter that we wanted to avoid developing browser extensions: the API did not seem mature enough, we preferred to maintain a single tool and we wanted to avoid adding an extra installation step for ZAP users. Furthermore, this would de facto eliminate all the browsers we did not write an extension for from the list of our supported platforms.

3.3.2 JavaScript modules

In the browser, the way to add script functionalities is by loading them or writing them inside `<script></script>` tags. One of the major downside of this approach is that it pollutes the global namespace, as declared functions are attached to the window scope. By doing so, it makes them "public" as well, which is less than ideal if we want the other scripts in the web page not to know some implementation details (such as a secret random value used in the way we communicate with ZAP for example[17]).

Node.js[18], has a notion of "module", which regroup a set of functions and data in an object that can be imported in an application[19] It has a nice explicit syntax to express the load of modules in a script.

```
var module = require('my-module');
```

Example of module import in Node.js

[17] More details have been provided in the section 2.2.1.

[18] "An open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser.", source: `https://en.wikipedia.org/wiki/Node.js`

[19] See: `https://www.w3schools.com/nodejs/nodejs_modules.asp`.

It is possible to simulate such a module system in the browser, by using what is called "the module pattern", which makes use of JavaScript objects and immediately-invoked function expression[20].

By using a module bundler[21], it is possible to write scripts importing modules using the Node.js `require` syntax, and have them working in the browser by transforming them upfront to use the module pattern. Note that this does not happen at runtime: it is a build step which result is then used in the browser.

3.3.3 Node Package Manager (npm)

"npm is the package manager for JavaScript"[22], and it is possible for anyone to publish their code to their registry. Are called JavaScript packages the groupings of one or more module(s) together for easier distribution. That is under this format that we decided to release our front-end-tracker: by publishing it as a package on the npm registry, at the following endpoint: `https://www.npmjs.com/package/@zaproxy/front-end-tracker`.

3.3.4 Usage

So, for an application to make use of the front-end-tracker, it just has to `require('front-end-tracker')` after having installed it via npm. However, as mentioned, because this is meant to be used in the browser, the code requiring our tool needs to be passed through a module bundler for it to be transformed into a "browser-interpretable" script, that we often call the bundled version of the script.

For users that are not familiar with JavaScript build systems, we wanted to make the front-end-tracker available on `unpkg.com`, a popular "content delivery network for everythin on npm"[23]. Unfortunately, we did not have the time to carry this task through completion, but we hope to see the related issue[24] answered in a near future.

There is a common case where one can not modify the targeted application code to inject the front-end-tracker bundled script. It is common because ZAP users and security testers in general often do not control applications they test, even less they own them.

In this situation, being able to arbitrarily inject the front-end-tracker into any web page is needed. Before developing the FrontEndScanner for example, we expected to be able to do exactly that to be able to test our front-end-tracker on popular websites.

[20] See: `https://medium.com/sungthecoder/javascript-module-module-loader-module-bundler-es6-modul 6b16`.

[21] See: `https://stackoverflow.com/questions/38864933/what-is-difference-between-module-loader-an answer-42317497`.

[22] Source: `https://www.npmjs.com/`.

[23] Source: `https://unpkg.com/`.

[24] See: `https://github.com/zaproxy/front-end-tracker/issues/5`.

For that, we advised a solution and wrote a ZAPHTTP Sender script to add custom JavaScript on top of any web page, and we made it available in the community scripts[25]. By turning it on in ZAP, and using it to load the bundled front-end-tracker from a file, one can tamper with any HTTP response passing through their proxy to add our beloved tool to the HTML it contains.

In the end, despite being a central piece of the FrontEndScanner without which it would not be as powerful, the front-end-tracker was designed and developed to be a completely standalone component. Because of that, anyone can make use of it to help them debug their own application.

[25] Available at the following address: `https://github.com/zaproxy/community-scripts/blob/master/httpsender/inject_js_in_html_page.js`.

Conclusion

During these twelve weeks of internship, I worked on improving ZAP for it to be able to handle "modern" web applications, and their new paradigm. Indeed, to improve the user experience of their application, more and more developers chose to write heavy client containing business logic and data, which was exclusively hosted on the back-end couple of years ago. What are called "modern" web applications are these heavy clients, which take the form of JavaScript applications, interpreted by web browsers.

With such a reshaping of web applications, security professionals need to update their tools to 1. find new vulnerabilities that arose because new actors are involved (in particular the web browser), 2. make sense of the logic of an application that is now distributed between the back-end and frond-end.

In such an environment which takes several form of inputs and can have a lot of side effects, predicting the flow of an application statically is impossible. Even with the current way of doing things: ZAP injecting content as the requests and responses pass by, triggering interesting behavior in the browser can be troublesome, as a complex set of conditions may be necessary for it to occur. Our proposed solution for this problem was to build a component to inject alongside applications, that will make use of the browser API to have access to front-end data, and monitor interactions happening there. This new ZAP feature has been released under the form of a ZAP add-on.

One required ability of our component was to be able to run scripts written by ZAP users, to express checks to be run in the application, all in the browser. Furthermore, we added some utilities functions exposed by our component for these scripts to be able to interact with ZAP, in particular to report back vulnerabilities that would have been found in the front-end side.

To help these scripts make sense out of the complex life of a modern web application (user interactions, fetching data over the network, and so on), we created a JavaScript library: the front-end-tracker, meant to expand the APIs available in the browser, especially to target debugging and security testing use cases. Because ZAP's component needs these functionalities for the scripts to be powerful, it is dependent on the library. However, the library can be used in other contexts as well, and we released it individually hoping this would be the case.

In the end, we made significant steps in the direction of a better tooling for web applications security testers. Our ZAP add-on and JavaScript library allow them to write security checks that would be performed directly in the browser, and that can

understand and depend on all kind of inputs (whether it be user, storage, network, etc.) by interacting with the APIs available here.

We hope our framework will be opening the door to a new generation of client-side security scripts to detect automatically the presence of vulnerabilities in front-end applications.

Even though we only have implemented the structure to run passive (non interactive) scripts, nothing prevents users to use the APIs available to tamper with data and be disruptive. So, from a technical point of view, active scripts are already a thing in our architecture; but we would prefer to enforce the distinction between the two kinds, to avoid non power-users to make dramatic mistakes while testing their (or others) applications.

Unfortunately, we lacked of time to deeply test and make an extensive use of what we produced, but we still managed to provide enough documentation for anyone to start consuming what we have done and even writing their own scripts.

Fortunately, we managed to create a proof-of-concept making use of what we produced. It is a script, injected by ZAP alongside the application, that will be executed in the browser; here, it takes advantage of the front-end-tracker to run verify every time data is written to the storages if they contain credentials.

Overall, there is room for improvement.

The front-end-tracker could integrate more hooks to deal with DOM mutations or network interactions (fetching data asynchronously is a popular pattern). With a bit of time and effort, it could surely be made easier to install and use as well.

Even though we kept in mind that we wanted our add-on to be used in a fully automated environment, our implementation so far requires users to interact manually with web pages. It would be interesting to see users scripts to run in a navigation session driven by Selenium for example.

And obviously, the whole class of active scripts is interesting: once users make scripts which interact with the application, allowing injections, tampering with data and so on, they could start to detect code injections, modification of the flow of the application, and what else.

Twelve weeks were a short time to produce everything we would have hoped to see. Even though we put a lot of effort into building production ready content for ZAP, the add-on could not be released yet; gladfully, it is not so far from being made available.

In the end, we defined and propose a new way to test the front-end side of web applications, and we provided the basic tooling to start doing it this way. We are looking forward to see how the community will react to this, the kind of use testers will make of it, and what kind of improvements they will bring.