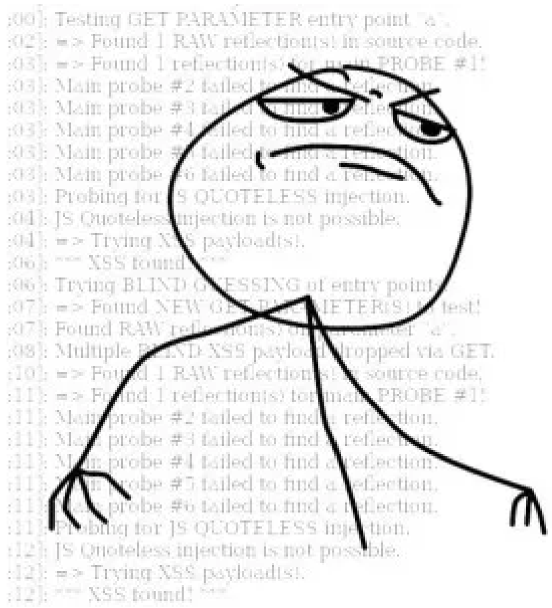
# Testing for XSS (Like a KNOXSS)

November 28, 2019 Brute The Art of XSS Payload Building

brave Proud to support a faster, safer web.



```
:00]: Testing GET PARAMETER entry point "a".
:02]: => Found 1 RAW reflection(s) in source code.
:03]: => Found 1 reflection(s) for main PROBE #1!
:03]: Main probe #2 failed to find a reflection.
:03]: Main probe #3 failed to find a reflection.
:03]: Main probe #4 failed to find a reflection.
:03]: Main probe #5 failed to find a reflection.
:03]: Main probe #6 failed to find a reflection.
:03]: Probing for JS QUOTELESS injection.
:04]: JS Quoteless injection is not possible.
:04]: => Trying XSS payload(s).
:06]: *** XSS found! ***
:06]: Trying BLIND GUESSING of entry points.
:07]: => Found NEW GET PARAMETER(S) to test!
:07]: Found RAW reflection(s) of parameter "a".
:08]: Multiple BLIND XSS payloads dropped via GET.
:10]: => Found 1 RAW reflection(s) in source code.
:11]: => Found 1 reflection(s) for main PROBE #1!
:11]: Main probe #2 failed to find a reflection.
:11]: Main probe #3 failed to find a reflection.
:11]: Main probe #4 failed to find a reflection.
:11]: Main probe #5 failed to find a reflection.
:11]: Main probe #6 failed to find a reflection.
:11]: Probing for JS QUOTELESS injection.
:12]: JS Quoteless injection is not possible.
:12]: => Trying XSS payload(s).
:12]: *** XSS found! ***
```

Testing for Cross-Site Scripting (XSS) might seem easy at first sight, with several hacking tools automating this process. But regardless of how tests to find a XSS are performed, automated or manually, here we will see a step-by-step procedure to try to find most of the XSS cases out there.

For that we will use the same approach employed by KNOXSS, our online XSS PoC tool. Although without details of its own implementation and intermediary steps needed to make its decisions (which will be done by ourselves if we follow the tests manually), this will cover pretty much what is done by this unique tool.

Translate »

## Test Procedure

**1.** First of all, we need to look for a **reflection** of our **input** in SOURCE CODE or in DOM.

*(It must be clear here the difference between those 2: source code is what we see by pressing Ctrl+U keys in browser for a given page while DOM is what we see in the Elements Tab of Browser Developer Tools by pressing F12 key).*

Some XSS vulnerabilities are a Stored type and those usually also reflect right in the response so we can treat them in the same way regarding testing. If not, they can be a Blind type (we will see later how to proceed) or simply trigger in another page. This last one makes a little harder to check the results of our attempts and although manually it can be done, in an automated way it greatly increases the complexity of a tool.

To look for a reflection of our input we use any simple string like "test123" (without quotes) as our input in all of the following places (all test points are in bold):

- Values of every URL parameter and/or every POST body parameter.
  Example: param1=**value1**&param2=**value2**&param3=**value3**

- In path of the URL replacing each folder, subfolder and page, one at a time.
  Example: https://domain/**folder/subfolder1/subfolder2/page**?param=value

- URL fragment like in https://domain/path/page?param=value#**data**

**2.** For those entry points above which reflects we proceed with what we call a **probe**, a string with some needed special char designed to test if there will be some filtering or sanitizing by application.

It's just our test string with a special char attached like less than sign (<), double quote ("), single quote ('), etc. Those are the main ones, enough for a manual test and we recommend testing one at a time because by using them all you might trigger some filter or different logic from when you use just one of them.

**3.** If step 2 succeed (our probe must reflect exactly as it is or it's sanitized) it's time to check **where** it got reflected, in source code or DOM. For each one of them there are different proceedings:

3.1. Reflection in SOURCE CODE:

We check if it's outside or inside a script block.

- Outside script block: we use simple or inline HTML injection (cases #2, #3 and #4 here), closing the respective tag where injection lands if it's one of the following tags: <title>, <textarea>, <style>, <iframe>, <noscript> and <xmp>. If injection reflects in a comment section like "<!– COMMENT –>", we break out from it with "–>" or "–!>" (2 dashes, without quotes). We can also use a "javascript:alert(1)" payload (without quotes) if our input reflects inside an anchor element (<a href=HERE>) or in any other case where a source (src) or HTTP reference (href) is expected like in <iframe src=HERE>, <object src=HERE> and <embed src=HERE>. For the anchor, it will need a click to fire the XSS.

- Inside script block: we use </script> to break out from native script block and inject some HTML (case #5 here) or we inject JavaScript code straight to the script block (cases #6 and #7 here) including Quoteless Injection if applicable (check requirements in the respective post).

3.2. Reflection in DOM:

Here we will deal with our 3 sinks approach which covers the majority of DOM-based XSS cases.

- If it gets inserted into DOM elements (Document Sink), we use HTML injection like we do above with reflection in source code but with extra attention: some HTML vectors might not work like “<script>” (usually) due to the fact that scripts are the first things to load in a given page so the injected one “gets late” and does not execute. A vector like <img src onerror=alert(1)> works fine, with some other alternatives.

- If it redirects to some place with our input (Location Sink) we use  the pseudo-protocol with a payload “javascript:alert(1)” (without quotes). It’s also possible to use “data:text/html,<img/src/onerror=alert(1)>” (again without quotes) if the goal is just to control the browser (it does not run in the context of the website) or if it’s somehow used to return some data to include in DOM. For this latter possibility, we need to check if there’s a request fired with our input without necessarily performing a redirect (usually AJAX). We will fall into a variation of our Document Sink case explained above. Check “CORS Enabled XSS” for more, using the payload above with “document.domain” instead of “1” in the test page of that post to see it working.

- If there’s some string evaluation (eval function or similar) or template engine in place we have the Execution Sink case. We need to dig into script blocks and JS library calls to try to figure out how it can be vulnerable. It’s by far the most difficult one. Going blackbox to save time, we usually try some payloads like “${alert(1)}” or “alert(1)” alone, both without quotes.

**4.** Next we **drop** a Blind XSS payload for every input we test. It's just a request, regardless if it will reflect or not. We never know where this might going to end, maybe in logs displayed in SIEM-like web applications or in customer service ones (Help Desk for example). It can be a simple <script src=//domain/script.js></script> or any other XSS vector using what is explained here. To know how to create a Blind XSS script check here.

**5.** Finally we try to **guess** parameter names while also checking for reflection in the parameter name itself. Some pages come with an incomplete set of parameters, there are hidden functionalities or even forgotten parameters from a previous version. We use a list with the most common parameter names and concatenate them to save time. That list can be appended to the current set of parameters of the page or not (some might be mandatory while others can mess with your probing). Example:

**s**=test123&**search**=test123&**q**=test123&**query**=test123&**test123**=0

Notice the last "test123" is not a value of a parameter but a parameter itself whose value is 0. If we hit the source or DOM with one of those guessed parameters we proceed isolating the one which reflects and following the same steps above, starting from step 2.

> Important Considerations

- Inputs in Base64 format, double encoded or with strict length also have to be handled accordingly for proper testing.

- Reflections which come from error messages must be handled in a certain way to allow the correct exploitation. Those originated from Error-Based

SQL Injection, for example, demand single quote and/or backslash characters in specific ways to trigger such situations.

- Some inputs are validated but not filtered. PHP's "filter_var" function with "FILTER_VALIDATE_EMAIL" flag, for example, must be exploited using a different way to test for it like shown here.

- Some nasty JavaScript injections need syntax fixing only possible with the help of a JavaScript feature named HOISTING. Check this for more info.

- A reflected XSS triggered with POST method must have its PoC built with a HTML form. If it's not possible, it's just SELF XSS and it's useless 99% of the time.

- XML-based scenarios require strict syntax to work, so we usually try full vectors and payloads instead of testing using these steps.

There are also several tricks that can be used in countless XSS situations and some few specific XSS cases which were not possible to cover here. Our previous post is a good example.

## Final Words

All tests and information provided in this post are features or procedures employed by KNOXSS (except the anchor one) and all vectors and payloads to

Translate »

prove a XSS vulnerability manually (or not) can be found in the latest edition of our XSS Cheat Sheet.

Follow this step-by-step guide to properly test for #XSS... Like a #KNOXSS!

Tweet This

## #hack2learn

KNOXSS

#hack2EARN.

Follow Like Share Tweet Save

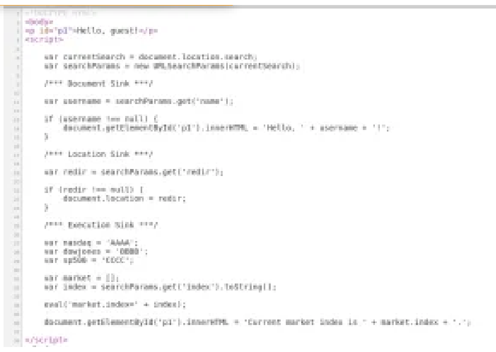
**Share this:**

Related

XSS in Limited Input Formats
March 11, 2019
In "The Art of XSS Payload Building"

XSS and RCE
May 9, 2016
In "The Art of XSS Payload Building"

Translate »



```
<!DOCTYPE html>
<body>
<p id="p1">Hello, guest!</p>
<script>

    var currentSearch = document.location.search;
    var searchParams = new URLSearchParams(currentSearch);

    /*** Document Sink ***/

    var username = searchParams.get('name');

    if (username !== null) {
        document.getElementById('p1').innerHTML = 'Hello, ' + username + '!';
    }

    /*** Location Sink ***/

    var redir = searchParams.get('redir');

    if (redir !== null) {
        document.location = redir;
    }

    /*** Execution Sink ***/

    var nasdaq = 'AAAA';
    var dowjones = 'BBBB';
    var sp500 = 'CCCC';

    var market = [];
    var index = searchParams.get('index').toString();

    eval('market.index=' + index);

    document.getElementById('p1').innerHTML = 'Current market index is ' + market.index + '.';

</script>
```

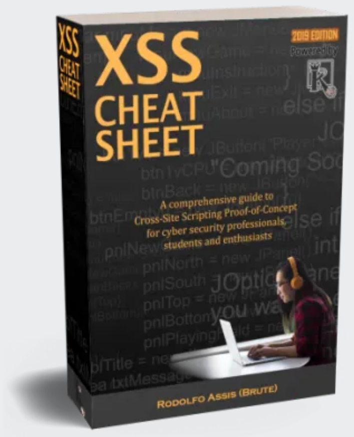
DOM-based XSS - The 3 Sinks

April 16, 2018

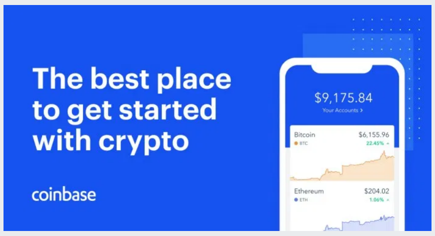
In "The Art of XSS Payload Building"

XSS via HTTP Headers

Select Language

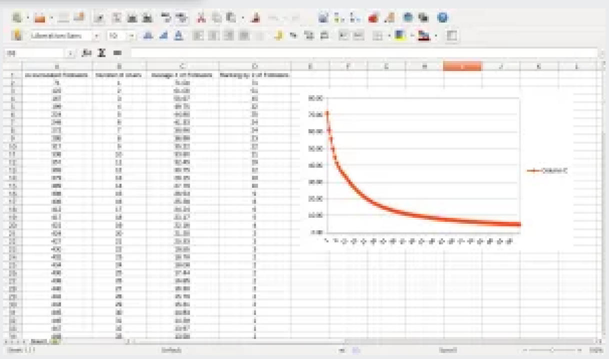

XSS
CHEAT SHEET
2019 EDITION
A comprehensive guide to Cross-Site Scripting Proof-of-Concept for cyber security professionals, students and enthusiasts
RODOLFO ASSIS (BRUTE)

GET YOURS NOW!



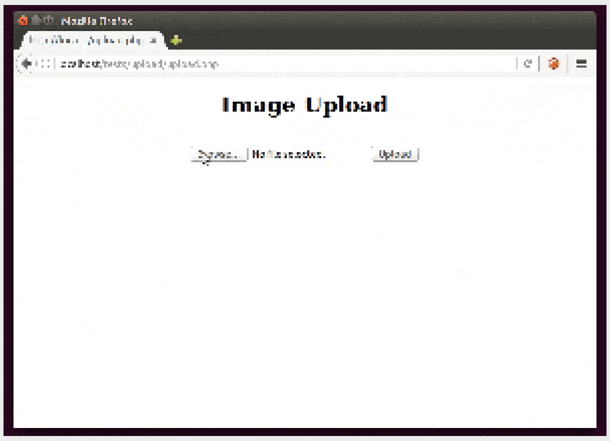
POSTS

Translate »



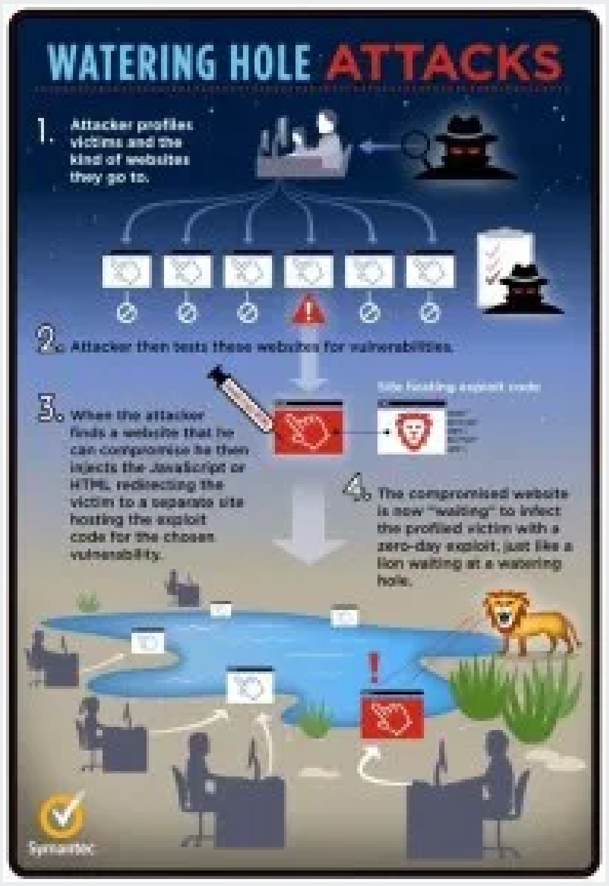
The Genesis of an XSS Worm – Part III

Be sure you have read parts I here and II here. We start the XSSbook database population […]



File Upload XSS

A file upload is a great opportunity to XSS an application. User restricted area with an […]


WATERING HOLE ATTACKS
1. Attacker profiles victims and the kind of websites they go to.
2. Attacker then tests these websites for vulnerabilities.
Site hosting exploit code
3. When the attacker finds a website that he can compromise he then injects the JavaScript or HTML, redirecting the victim to a separate site hosting the exploit code for the chosen vulnerability.
4. The compromised website is now "waiting" to infect the profiled victim with a zero-day exploit, just like a lion waiting at a watering hole.
Symantec

Reflected in Watering Hole

Cross-site scripting becomes much more dangerous when used with another attack strategy. […]

Translate »

## FOLLOW ME

Tweets by @brutelogic

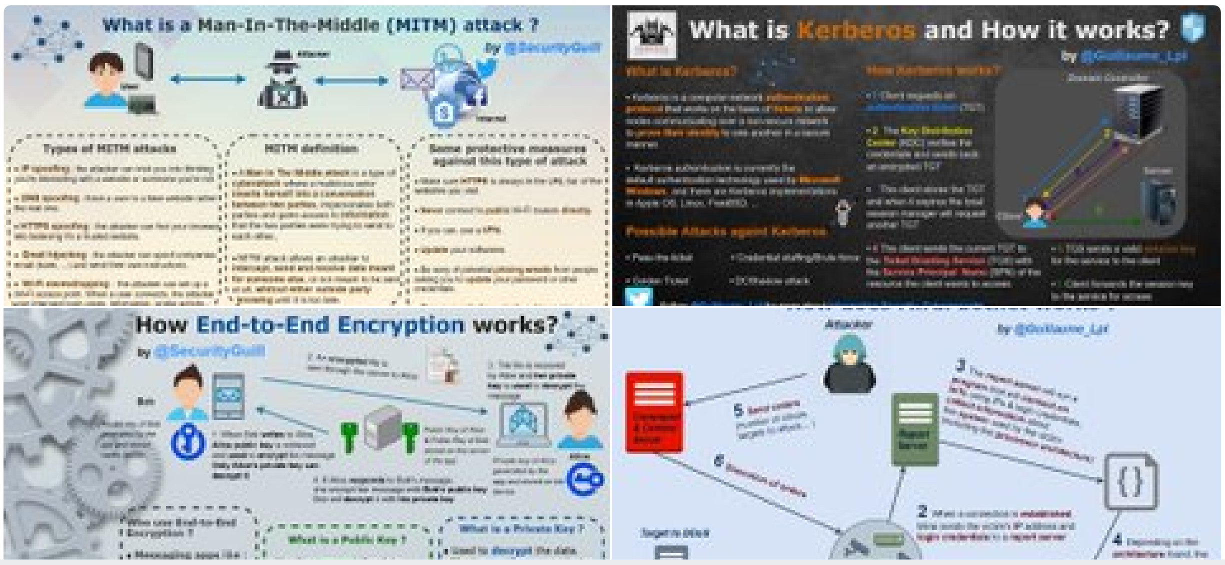
Brute Logic Retweeted

**SecurityGuill** 🌐
@SecurityGuill

This thread updated includes all my #infographics so far, they present different terms related to Information Security 🔐

It's an easy way to learn new things 📖
I hope it will be useful to the community 🌐
RT appreciated 🔁

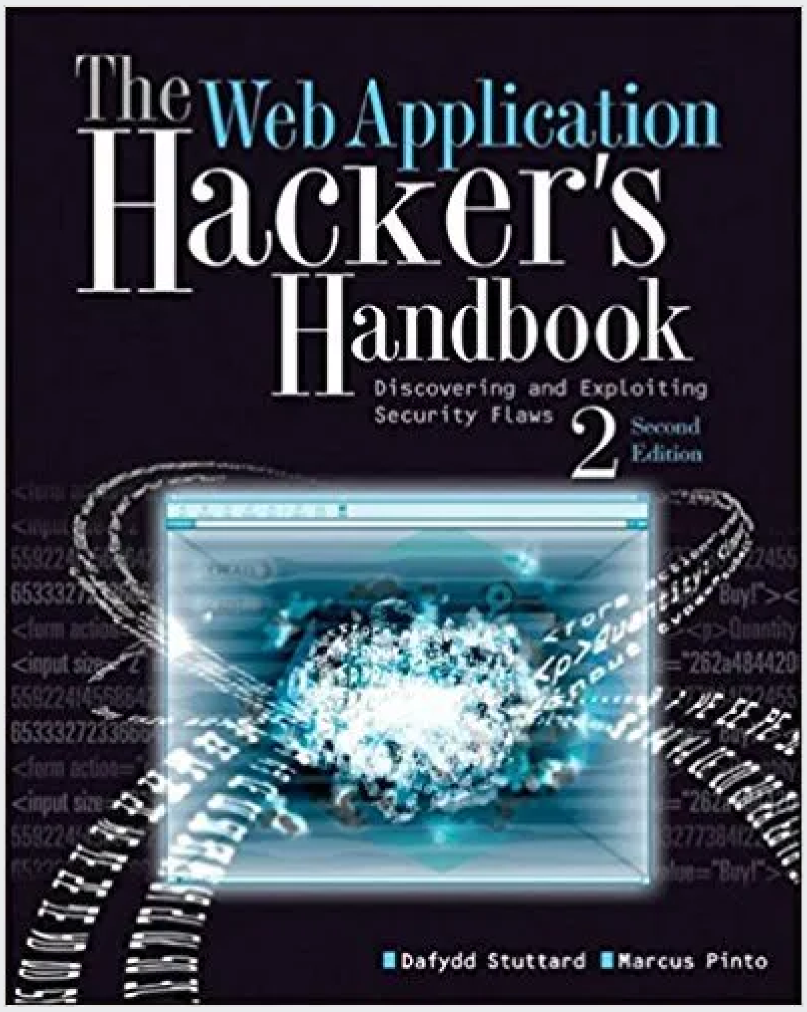
Follow me ➡️ @SecurityGuill for more about #infosec


What is a Man-In-The-Middle (MITM) attack ?
What is Kerberos and How it works?
How End-to-End Encryption works?

## ALL POSTS

Testing for XSS (Like a KNOXSS)
XSS via HTTP Headers
XSS in Limited Input Formats
Advanced JavaScript Injections
Quoteless Javascript Injections
DOM-based XSS – The 3 Sinks
Chrome XSS Auditor – SVG Bypass
The 7 Main XSS Cases Everyone Should Know
Compromising CMSes with XSS
Alternative to Javascript Pseudo-Protocol
XSS Filter Bypass With Spell Checking
XSS Challenge I
Calling Remote Script With Event Handlers
Four Horsemen of the Web Apocalypse
The Easiest Way to Bypass XSS Mitigations
XSS Authority Abuse
Reflected in Watering Hole
Bypassing Javascript Overrides
The Genesis of an XSS Worm – Part III
The Genesis of an XSS Worm – Part II
The Genesis of an XSS Worm – Part I
The Shortest Reflected XSS Attack Possible
Looking for XSS in PHP Source Code
AntiviruXSS White Paper
Avoiding XSS Detection
Blind XSS Code
XSS and RCE
CORS Enabled XSS
Chrome XSS Bypass
File Upload XSS
Leveraging Self-XSS
XSS in Mobile Devices
Cross-Origin Scripting
Transcending Context-Based Filters
vent Handlers

Multi Reflection XSS
Using XSS to Control a Browser
Source-Breaking Injections
Location Based Payloads – Part IV
Location Based Payloads – Part III
Location Based Payloads – Part II
Location Based Payloads – Part I
Probing to Find XSS
Filter Bypass Procedure
Existing Code Reuse
Agnostic Event Handlers
XSS Payload Scheme


The Web Application Hacker's Handbook
Discovering and Exploiting Security Flaws
2 Second Edition
Dafydd Stuttard
Marcus Pinto

Translate »

FREE Penetration Testing Demo
Find Your Vulnerabilities
Harden Your Defenses
Our Specialists Can Help
LEARN MORE »
ArtsSEC
INFORMATION SECURITY