Web Vulnerabilities

And The People Who Love Them

Me

- TomNomNom
- Occasional bug hunter
- Lover of analogies
- Lover of questions

Insecure Direct Object References (IDOR)

- Often one of the most simple vulnerabilities to exploit
- Ever seen a URL like this?: /account?userId=314159
- Change the userId value and you might get someone else's details
- The root cause is often a misunderstanding of authentication vs. authorisation
  - Authentication: is this person who they say they are?
  - Authorisation: is this person allowed to do that?
- Possible Impacts:
  - Information disclosure
  - Changing data you don't own
  - Privilege escalation
- Mitigations:
  - *Always* check for authorisation; if possible build it into your Data Access layer

IDOR!

- A real example: https://hackerone.com/reports/227522
- In this case the ID was in the POST data; don't forget about POST data!
  - The request path, cookies, and headers too!

Open Redirects

- When you can force target.com to redirect you to evil.com
- The only really useful vectors are through the query string, path, or fragment
- There's a 'DOM' variety, where the redirect is done by client-side JS
  - Classic vulnerable code: document.location = document.location.hash.substr(1)
- Root cause is often a misunderstanding of what passes for an absolute URL
- Impact:
  - Super-convincing links for phishing campaigns
  - Bypassing whitelists for SSRF attacks etc (more on those later)
  - Information leakage - especially where authentication is involved!
- Mitigations:
  - Don't use redirects ¯_(ツ)_/¯
  - Use a (good) URL parser instead of, say, regex to validate URL inputs

Valid URLs (As Far As Chrome Is Concerned)

- http(s)://evil.com
- http(s):\\evil.com
- //evil.com
- ///evil.com
- /\evil.com
- \/evil.com
- /\/evil.com
- \\evil.com
- \/\evil.com
- /	/evil.com (that’s a tab / ASCII 0x09)
- \	\evil.com (another tab)

Auth Flows And Open Redirects

- A real example: https://hackerone.com/reports/239503
- Auth token was sent to an arbitrary site!

Information Disclosure

- A pretty broad category, but here I just mean the simple stuff
  - Accidentally exposed config files
  - Verbose error messages
  - An exposed .git directory is a goldmine
- Root cause is almost always misconfiguration in this case
  - Set your docroot to someone’s home directory? Now their SSH keys are compromised
- Mitigations:
  - Don’t put sensitive files in your docroot at all
  - Always run production with error and debug messages turned off

Files To Look Out For

- .git/config
- .travis.yml
- Makefile
- Dockerfile
- package.json
- gulpfile.js
- composer.json
- web.config
- .env
- ...too many to list...

What A Git

- A real example: https://hackerone.com/reports/248693
- .git folders can be grabbed with https://github.com/kost/dvcs-ripper

Local File Inclusion (LFI)

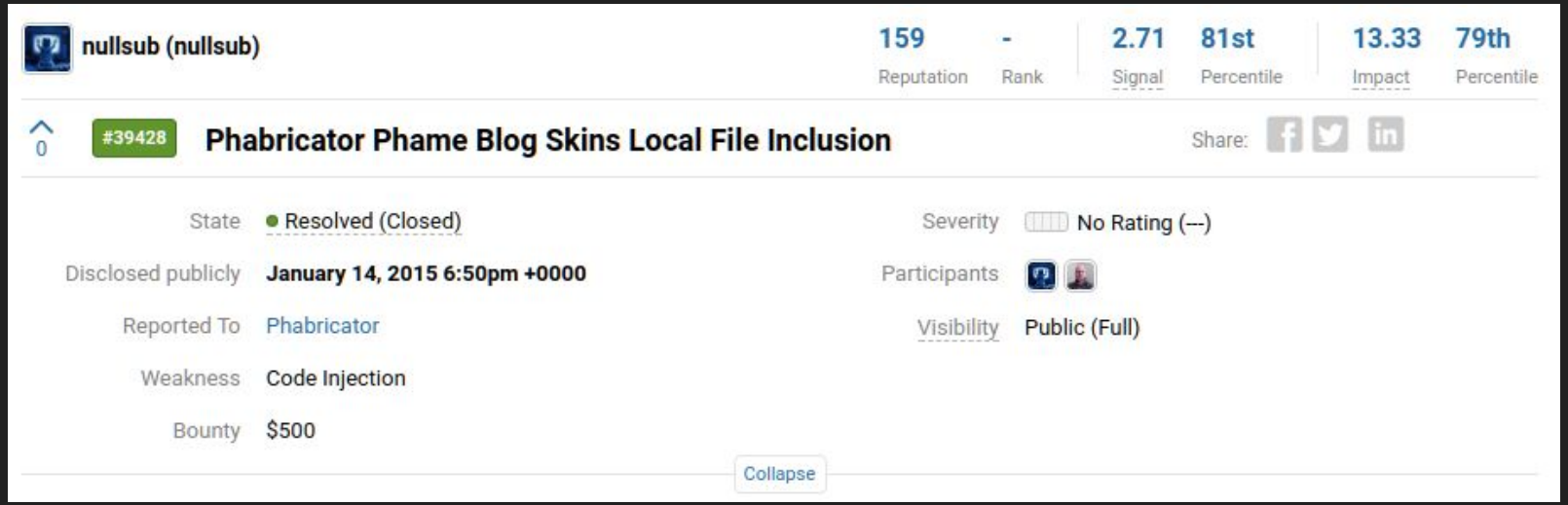
- When you have control over the filename of a server-side include
- You can get the contents of, say, /etc/passwd, source code, private keys etc
- Can sometimes be escalated to Remote Code Execution
  - If you have some control over the content of a file on disk (e.g. server logs) you might be able to execute arbitrary PHP
- The classic attack: /?template=../../../../../../../etc/passwd
- Root cause is often a failure to validate input at all
- Mitigations:
  - Validate against the *realpath* of the input (i.e. after all '..' sequences have been resolved)
  - Whitelist the filenames you want to accept
  - ...or just don't accept filenames as inputs :)

Escalating To Remote Code Execution

1. A server runs PHP
2. You've got LFI via PHP's include(): target.com/?template=../../../etc/passwd
3. Make a request to target.com/<?php echo shell_exec($_GET['c']);?>
4. The access or error log might now contain your PHP code
5. Request: target.com/?template=../../access.log&c=uname -a
6. Look smug

Local Files Included

- A real one: https://hackerone.com/reports/39428



Server Side Request Forgery (SSRF)

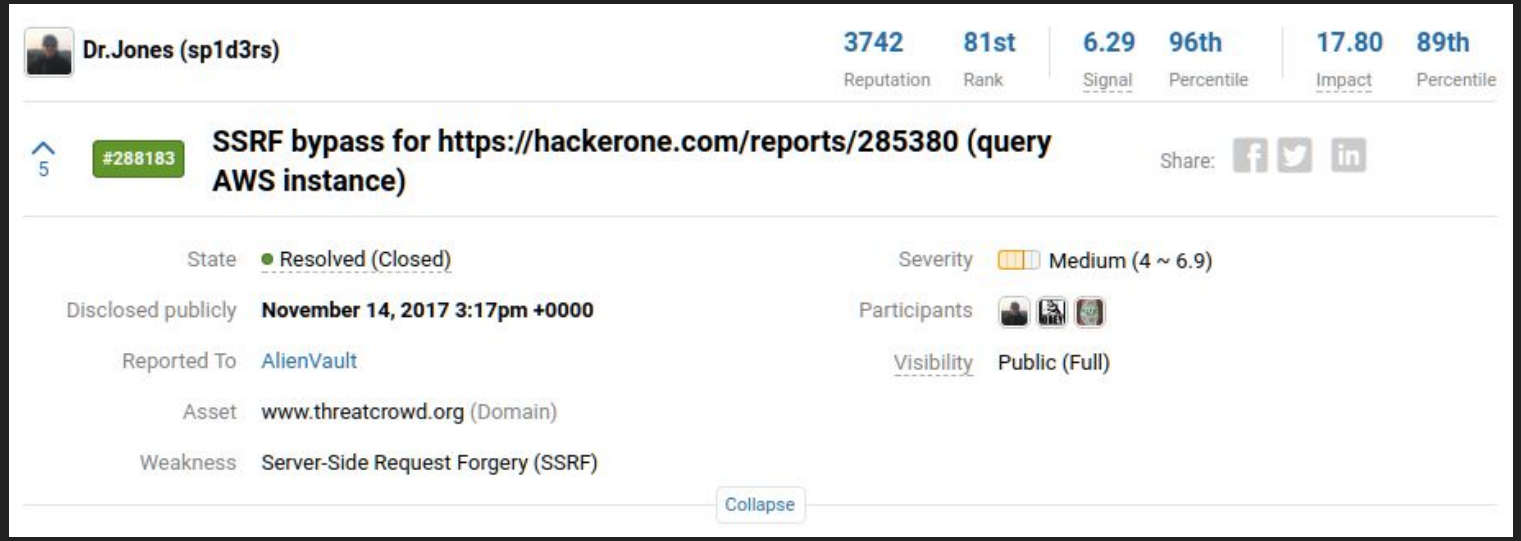
- A server fetches and (usually) returns data from an attacker-controlled URL
- A common way to pivot from external systems to internal ones
- A classic attack: /fetch?url=http://169.254.169.254/latest/meta-data/
- Root cause is often failure to validate inputs properly (a common theme)
  - Or maybe you bypassed the whitelist with an Open Redirect? :)
- Mitigations:
  - Whitelist valid URLs where possible or don't accept URLs as parameters
  - Use a good URL parser, 'resolve' the address and make sure it's not internal
  - Firewalls: don't let your web servers talk to anything sensitive
  - Don't follow redirects if possible

Some SSRF Filter Bypasses

- Previously found Open Redirects
- Public DNS records that point to internal IPs (:
- Funky IP formatting. All of these mean the same thing:
  - 127.0.0.1
  - 127.0.1
  - 127.1
  - 127.000.000.001
  - 2130706433
  - 0x7F.0x00.0x00.0x01
  - 0x7F.1
  - 0x7F000001
  - ...practically infinite variations...

Requests Forged

- In the wild: https://hackerone.com/reports/288183



Cross Site Scripting (XSS)

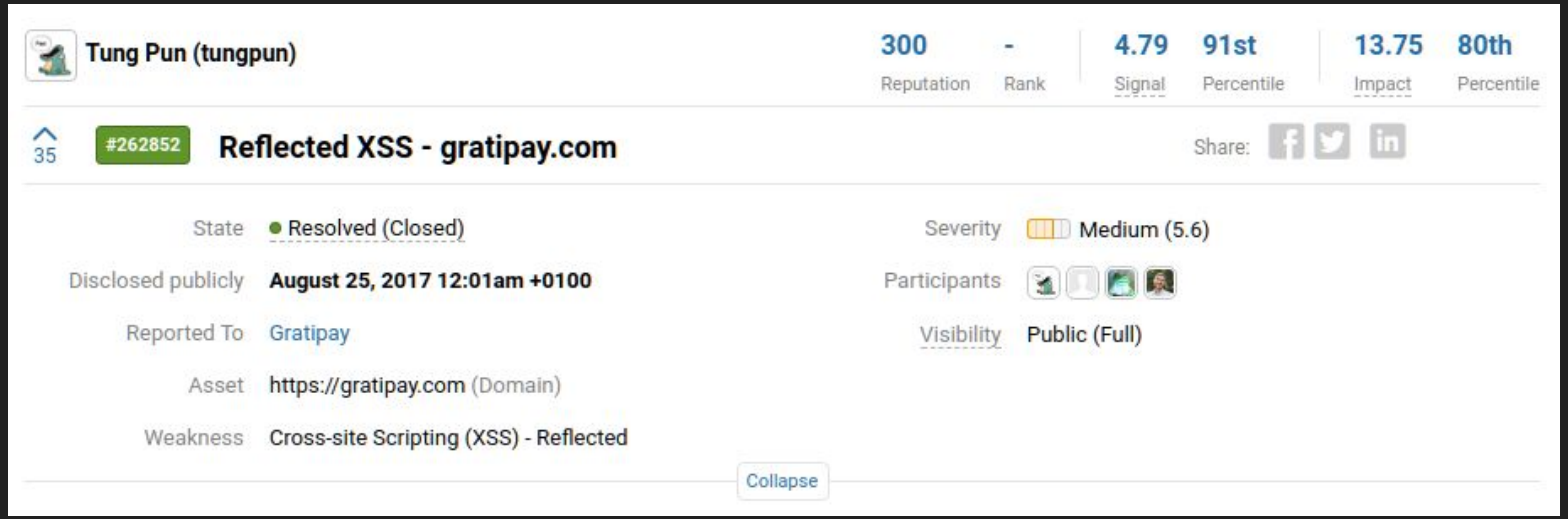
- We had to get to it sooner or later :)
- An attacker can execute arbitrary JavaScript in a victim's browser in the context of a target site
- The classic attack: target.com/?name=<script>alert('LOL XSS')</script>
- Three main kinds:
  - Reflected: input from the request is included unescaped in the page
  - Stored: input from a previous request is included unescaped in the page
  - DOM: input is improperly handled by JavaScript running in the page
- Mitigations:
  - Don't include user input in the page at all
  - Escape user input before outputting it (with a mechanism suitable for the output context!)

XSS Impact

- A lot of XSS tutorials cite stealing cookies as the main risk
  - Most sensitive cookies are set to httpOnly these days to help mitigate that
- The *real* issue is that Same Origin Policy is bypassed
  - That means anything a user can do on the target site, an attacker's JavaScript can do too
- Defence in depth:
  - Require additional user input before performing actions like change of email address, funds transfers, purchases etc
  - Partly why you're often asked to re-enter your password before doing that kind of thing
  - A good Content Security Policy can help too :)

Sites Cross Scripted

- Literally the most commonly reported vulnerability class
- A real one: https://hackerone.com/reports/262852



Cross Site Request Forgery (CSRF)

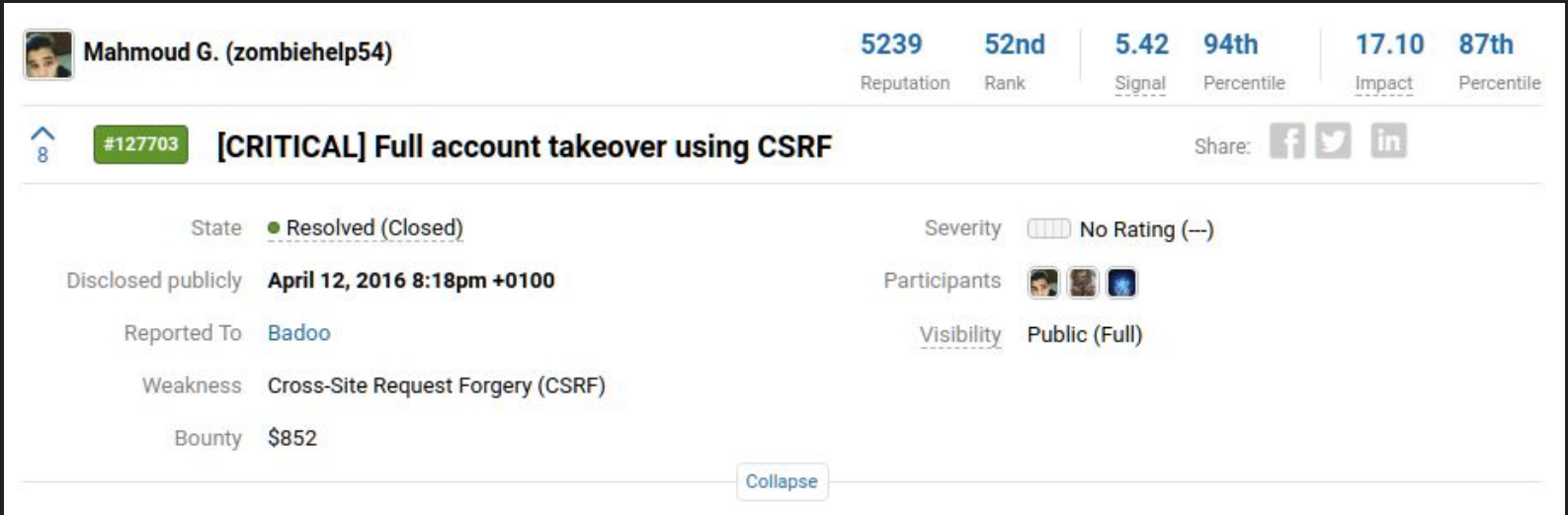
- A form on an evil.com sends data to an endpoint on target.com
  - The form is pre-filled and automatically submitted with JavaScript
- The victim is logged into target.com; their cookies are sent with the request
- An action is performed on behalf of the victim (e.g. email address change)
- Root cause: this is the way the web works by default ¯_(ツ)_/¯
- Impact:
  - Account takeover as the result of clicking a link completely unrelated to the target site
- Mitigations:
  - Use CSRF tokens...

CSRF Tokens

1. Every form on target.com includes a random string as a hidden value
2. That value is stored in the user's session on the server side
3. When a user submits the form the random string is included
4. The server verifies that the string matches what's stored in the session before processing the rest of the form data
5. The random string is removed from the session and not used again
6. An attacker has no way to know what the random string should be - unless they have an XSS vulnerability already :)

Cross Site Requests Forged

- A real one: https://hackerone.com/reports/127703

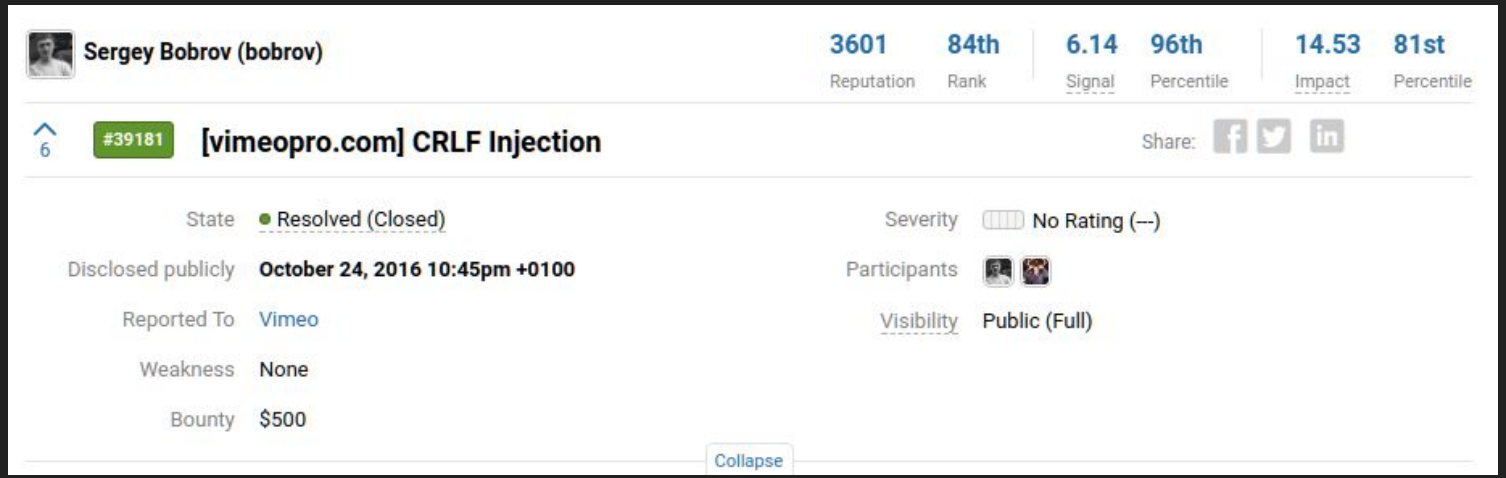


Linefeed Injection (CRLF Injection)

- Linefeed / New-line and/or Carriage Return characters are returned in response headers unescaped
- An attacker can inject their own headers into the response and:
  - Set cookies for session-fixation attacks
  - Specify their own Cross Origin Resource Sharing policy
- Example attack: /path%0aSet-Cookie:sessid=1234
- Sometimes a response *body* can be injected also, allowing XSS etc
- Occasionally headers can be injected into the *request* due to bad handling by intermediate application-layer load balancers
  - Can sometimes be used to bypass checks, spoof CF-Connecting-IP / True-Client-IP headers
- Mitigations:
  - Always escape / URL-encode output in headers

Linefeeds Injected

- In the wild: https://hackerone.com/reports/39181

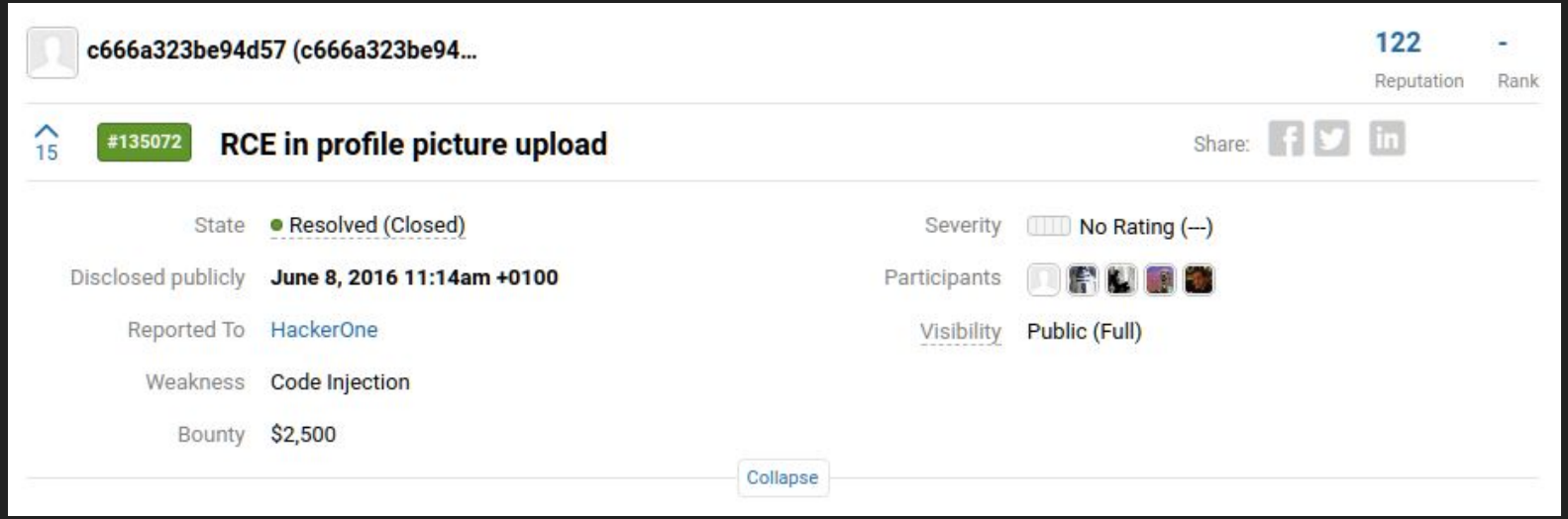


Remote Code Execution

- Some kind of code in the request is executed by the remote server
- Pretty much the holy grail of vulnerabilities
- Impact: hopefully fairly obvious ¯_(ツ)_/¯
- It could be—for example—PHP code, or it could be a user input being passed to a system exec() type call
- Example attack against an input passed to a shell: /?param=$(uname -a)
- Mitigations:
  - Don’t pass user input to system command / shell executions at all
  - Escape or whitelist inputs; e.g. with PHP’s escapeshellarg()

Remote Code Executed

- You know the drill: https://hackerone.com/reports/135072



XML External Entities (XXE)

- XML processed by the target has *External Entities* processed
- XML lets you define custom entities, which can have external sources:

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

- Often allows Local File Inclusion, sometimes even Remote Code Execution
- Mitigation: disable external entities for the XML library being used

Entities Externalised

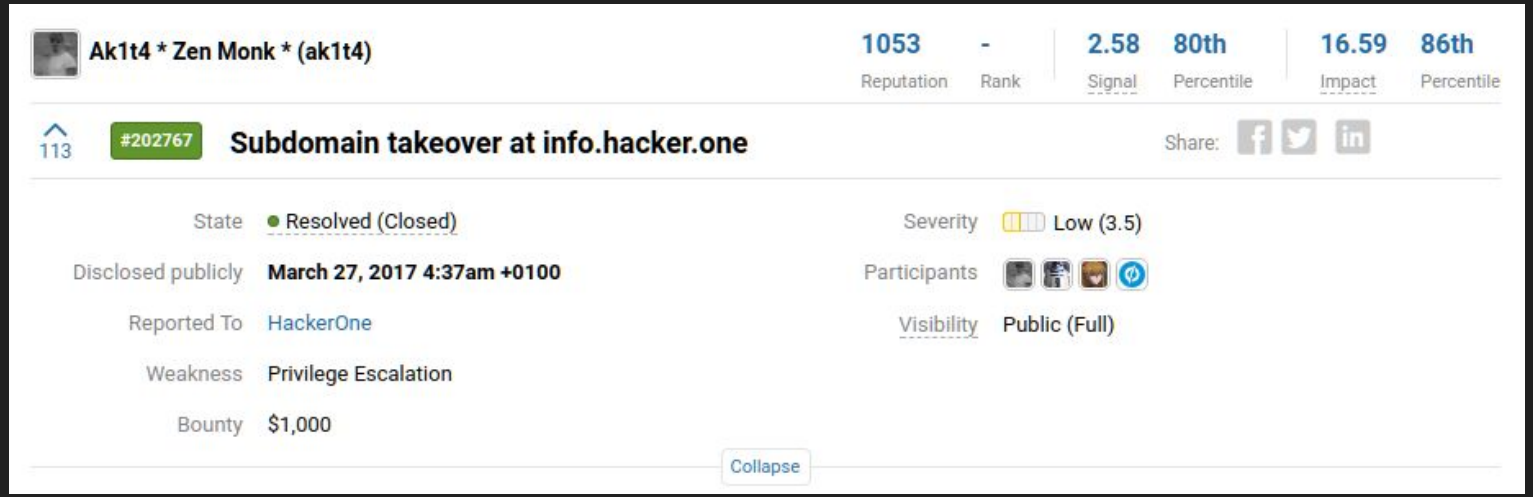
- https://hackerone.com/reports/248668

Subdomain Takeovers

- A subdomain is configured to point at a service, but the domain has not been ‘claimed’ on the service
- An attacker can claim the domain and control its content
- Often called a ‘dangling CNAME’
- Example services that lack domain ownership verification: S3, CloudFront
- A subdomain may also point to another domain that is no-longer registered
- Impact: An attacker can control the content served from sub.target.com and:
  - Make the company look bad
  - Steal cookies scoped to .target.com
  - More easily bypass whitelists (e.g. CORS configurations)
- Mitigation: don’t leave your subdomains pointed at unconfigured services or unregistered domains?

Subdomains Taken Over

- https://hackerone.com/reports/202767



SQL Injection (SQLi)

- User input is used in a SQL query without any kind of escaping
- Classic attack: target.com/users?id=1 union select name, hash from users
- Impact:
  - Usually access to private data in the database, or even dumping the entire database
  - Sometimes an attacker can change data - e.g. if the insertion is into an update or insert query
- Mitigation:
  - Use prepared statements...

Prepared Statements

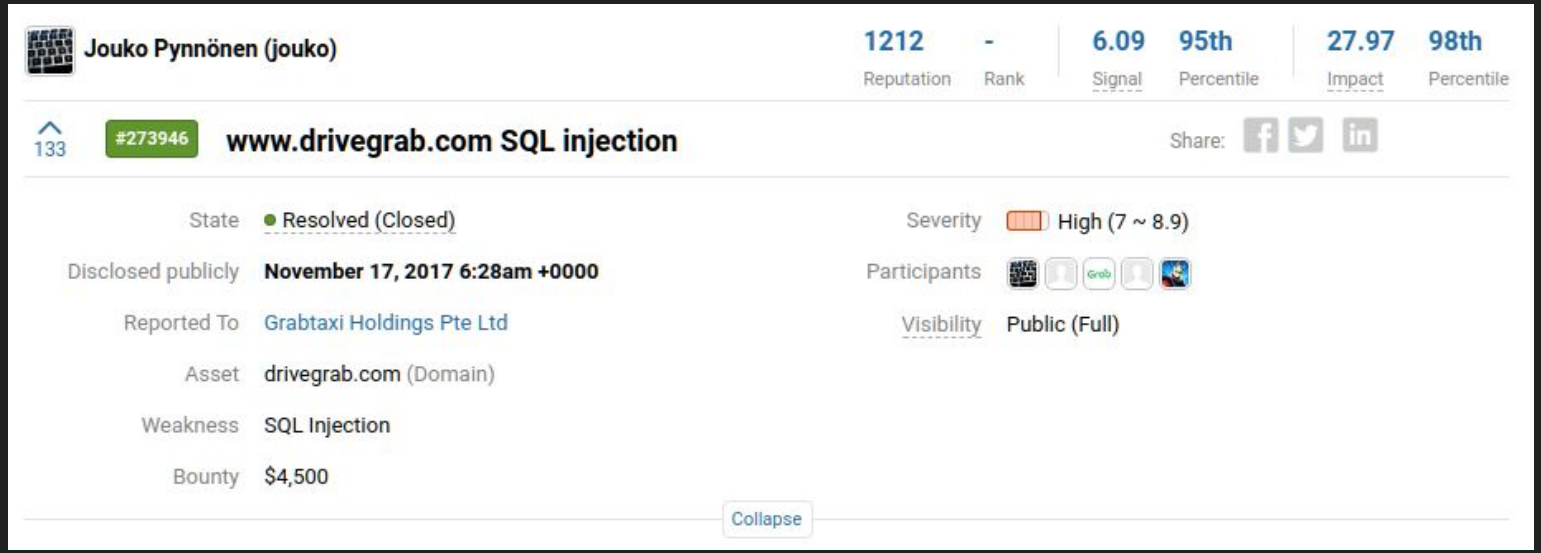
- Tell the database about the static and dynamic parts of the query separately, so it knows which is which
- Pseudo-code:

```
// Wrong!
result = query(“select name, age from users where id = $ID”)

// Right!
statement = prepare(“select name, age from users where id = :id”)
result = execute(statement, [:id => $ID])
```

SQL Injected

- https://hackerone.com/reports/273946



Get Creative

- There’s many more kinds of vulnerabilities out there
  - Including things that haven’t been thought of yet!
- Chain things together, create new things; get creative!
- The worst bugs are often just flaws in business logic
  - You often don’t need anything other than a web browser
- ...but don’t be stupid. Only target systems you have permission to target :)

Questions?

- I like questions! Ask me questions! :)