AIO Web App Pentesting Checklist

Cristi Vlad · Follow

11 min read · Oct 26, 2024

I'm testing the capabilities of *NotebookLM* for extracting valuable information from sources and I believe it can often do a much better job than *ChatGPT* especially because it's more grounded in truth by default. So, you don't have to prompt it heavily to stay grounded. On top of that, it doesn't stray from the subject at hand into hallucination land.

In this post, I've tested ***NotebookLM*** to extract an compile an inclusive web app pentesting checklist from the following three sources:

- *https://guide.offsecnewbie.com/web*
- *https://pentestbook.six2dez.com/others/web-checklist*
- *https://book.hacktricks.xyz/pentesting-web/web-vulnerabilities-methodology*

When it comes to being inclusive, I don't think it is all inclusive (as you shall see below), but it is quite decent. Many thanks to the creators of these 3 sources. What I did was only to run ***NotebookLM*** over them with my personal twist.

That being told, feel free to use this checklist in your engagements and if you want to go deeper, look no further than the 3 sources.

Reconnaissance

- If a certificate redirects to 443/HTTPS, check for alternate names in certificate details.
- Use **nikto and nmap** to scan for vulnerabilities on different ports.
- Download a wordlist of disallowed robots from the *RobotsDisallowed GitHub repository* and use **gobuster/feroxbuster** to check for directories.
- Use a variety of wordlists when scanning with tools like **gobuster** and **feroxbuster**. Scan recursively, but note this can be time-consuming.
- If **gobuster** returns many pages you know to be invalid, the webpage may be filtering requests by user-agent. Use the `-a` flag to change the user-agent in **gobuster**. This can also be achieved in **feroxbuster**.

- Use **WhatWeb** for banner grabbing. Use **Wappalyzer** for technologies.
- Use **nmap** and **Burp Suite** to test HTTP methods.
- Use **hydra** to brute-force authentication.
- Use **nmap** to scan for vulnerabilities (use **nmap** scripts).
- Use **SQLMap** to test for SQL injection vulnerabilities.

Tips:

- If you can register as a normal user, try to find an upload feature that can be abused.
- Use **grep** to search for usernames in any backup files downloaded; there may be passwords stored next to usernames that can be used to log into other services.

DNS Enumeration

- Use **host** to gather information about nameservers and mail servers.
- **Forward DNS Lookup Bruteforce**: To guess the names of valid servers, attempt to resolve given names. Use the IP range obtained to discover hostnames and IP addresses that belong to the target website.
- **Reverse DNS Lookup Bruteforce**: To find missing domain names in the forward lookup, probe the IP address range.
- **DNS Zone Transfer**: Zone transfers, which are like database replication acts between DNS servers, should be limited to authorized secondary DNS servers. If a DNS server is misconfigured, it may be vulnerable to a zone transfer attack, giving the attacker a complete map of the internal and external network.
- Use **dnsrecon** to test for zone transfers.

General Website Reconnaissance

- Determine if the target is protected by a web application firewall (WAF). The target may stop responding if probed too often.
- **Map the attack surface**: subdomains, IP blocks, and email addresses.
- **Tools: theHarvester**
- **Tips**: Use the *OSINT Framework* to explore the results of **theHarvester**.
- Find services, banners, and versions. Research CVEs and search Exploit-DB for vulnerabilities.
- Find the newest features and forgotten endpoints. **Tools: Wayback Machine.**
- check **robots.txt**, **crossdomain.xml**, and **clientaccesspolicy.xml**.
- Find hosts. **Tools: dig.**
- Use **Google** to search for data leakage.
- Map the infrastructure: middleware, programming languages, backends, and services. **Tools: Wappalyzer.**
- Find hidden folders and files. **Tools: dirb**, **nikto, feroxbuster, gobuster, Burp Suite.**
- Spider/map all application functionality.
- Identify data entry points and the technologies used.
- Read the web application's client code, paying attention to the JavaScript libraries used, messy code, and sinks.
- Check comments in the source code.
- Generate an error page, which may be vulnerable to cross-site scripting (XSS).

- Identify all parameters and document which are used for GET and POST requests.
- Identify where cookies are set, modified, and added.
- Note any strange headers.
- Use **Shodan** to find similar apps and endpoints, as well as SSH hash keys.
- Find previous vulnerabilities in the target website. **Tools**: **recon-ng**
- Read the manual for the application. Does it have a development mode? Is there a **DEBUG=TRUE** flag that can be flipped to see more information?
- Look for where data can be entered. Is it an API? Is there a paywall or sign up? Is it purely unauthenticated?
- Consider the target from the perspective of an attacker. What does it do? What is its most valuable part? Some applications will value certain features more than others. For example, a website with premium content may be more concerned about users bypassing a paywall than XSS.
- Think about the application logic. How is business conducted?
- **Tips**: If this is a bug bounty program, look for new acquisitions, code from a new team, new versions of mobile apps, new UI in the web application, and new features.

Extensions List

- Focus on the following extensions: `.asp`, `.aspx`, `.bat`, `.c`, `.cfm`, `.cgi`, `.css`, `.com`, `.config`, `.dll`, `.exe`, `.htaccess`, `.htm`, `.html`, `.inc`, `.jhtml`, `.js`, `.jsa`, `.jsp`, `.log`, `.mdb`, `.nsf`, `.pcap`, `.phar`, `.php`, `.php2`, `.php3`, `.php4`, `.php5`, `.php6`, `.php7`, `.phps`, `.phpt`, `.pht`, `.phtml`, `.pl`, `.reg`, `.sh`, `.shtml`, `.sql`, `.swf`, `.txt`, and `.xml`.
- Append `.old` or `.bak` to files.

Testing Input Validation

- Run automated scans, but remain in scope and respect the policy (I prefer Burp Suite scanner and I use it sparingly). **Tools: Burp Suite, nikto, dirb, feroxbuster.**
- Use **wpscan** to assess WordPress plugins.
- Use **cmsmap** to search for known bugs in Drupal and Joomla.
- Use a tool like **Flashbang** to decode SWF files.
- Find reflected parameters and test for XSS, HTTP parameter pollution (HPP), link manipulation, and template injection.
- Test for server-side issues like SQL injection, server-side include, OS command injection, path traversal, local and remote file inclusion (LFI/RFI), SMTP injection, SOAP injection (this also ties into XXE attacks), LDAP injection, XPath injection, code injection, deserialization attacks, and overflow attacks.
- If a parameter looks like a file, test for path traversal, RFI, and LFI.
- If a parameter looks like a URL, test for open redirection and server-side request forgery (SSRF).
- Test for injection attacks, SSRF, XPath injection, XML external entity (XXE) injection, and insecure object de-references when parsing XML, JSON, or any other markup language that the application processes.
- Look for parameters encoded in Base64 or other formats. Test again for injection attacks and insecure object de-references.
- Check for DOM-based attacks like open redirection, XSS, and client-side validation.
- Test file uploads, paying attention to SVG files, which can have embedded XML that triggers SSRF and XXE.

Tips:

- If the user has a profile and avatar, try uploading a malicious SVG file with a script.

Server Issues

- Write a script that sends every HTTPS request over HTTP.
- Test for header injections.
- Test HTTP Options, using arbitrary method names to attempt to bypass authentication pages.
- Test any client-side applet such as Flash, ActiveX, and Silverlight.
- For file uploads, look for reflected file downloads. Try uploading files with double extensions (e.g., `.php5.jpeg`) and null bytes (e.g., `.php5%00.jpeg`).
- Ensure that anti-CSRF mitigations are in place for the main functions, as well as clickjacking mitigations.

Tips:

- If there's a binary that runs as root, it should only use HTTPS and verify checksums, or use signed checks with a public key.
- Test for CAPTCHA bypassing.
- Check for frame injection and frame busting, which can still be an issue.
- Look for caching poisoning issues and sensitive data in URL parameters.
- Follow up on any information leakage.
- Look at numeric IDs, which can reveal a lot of information like the number of orders and users. Look for hashed numeric IDs.

Tips:

- Look at SWF files, which are often vulnerable.
- Check for weak SSL ciphers.
- Test the Cross-Origin Resource Sharing (CORS) policy.

Tips:

- If CORS or **crossdomain.xml** allows subdomains, you can trick a user into executing XSS on a page by injecting an iframe on all webpages that redirects to a subdomain. For example, `sub.vulnerable.com` could be used to intercept all requests for that host and return HTML that will issue a cross-domain request to `vulnerable.es` and display it to the UI.
- Verify the Content Security Policy (CSP). Look for bypasses. Use *https://csp-evaluator.withgoogle.com*.
- Verify **HTTP Strict Transport Security (HSTS)**.
- Verify **X-XSS-Protection**.
- Verify **X-Content-Type-Options**.
- Verify **HTTP Public Key Pinning**.

Testing Authentication

- Use a **Burp Suite** extension to see what different users can see. This will test authorization.
- Brute-force basic authentication.
- Test password quality rules, such as the required length, character set (alphanumeric, upper/lowercase, and special characters). Can the

password or username be empty? Are common passwords like `123456` allowed?

- Test for username enumeration.
- Test account recovery functionality, looking for SMTP header injection.
- Does "**Remember me**" expire?
- Test removing your email address from your account. Add a new email address and make sure that the old one can no longer be used to recover the password or log in.
- Test the resilience of password brute-forcing. Does the application lock after a certain number of attempts?
- Are email verification links sent over HTTP?
- Check the scope, HttpOnly flag, and secure flag for cookies.
- Test for other strange access control methods like referral validation, which can be bypassed.

Tips:

- Does the "Remember me" function ever expire? Is there an opportunity to abuse cookies in combination with other attacks?
- Test for username uniqueness.
- Think about how logins are processed. Are they sent over HTTP? Are details sent in a POST request, or are they included in the URL? Including this information, especially the password, in the URL is bad practice.
- Test for null bytes (`%00`) in the username and password fields.
- Test for fail-open conditions, which occur when user authentication fails but results in the user gaining access to authenticated and secure

sections of the web application.

- Test for cookie poisoning. Try including the cookie names in the query string and body. Some servers may read those parameters and set them as cookies. This could be an opportunity for cookie poisoning.
- Try setting a new password by using the old password.

Testing Session Management

- Think about how well sessions are handled. Is there randomness in the session cookie? Are sessions killed in a reasonable amount of time, or do they last forever? Does the application allow multiple logins from the same user? Is this significant to the application?
- Are tokens generated predictably, or do they contain sufficiently random values? **Tools: Burp Suite Sequencer.**
- Check for the insecure transmission of tokens. Can they be accessed by JavaScript? Is this an issue?
- Check for the disclosure of tokens in logs. Are they cached server-side? Can you view this information? Can you pollute logs by setting custom tokens?
- Check the mapping of tokens to sessions. Is a token tied to a specific session, or can it be reused across different sessions?
- Check session termination.
- Check for session fixation.
- Can an attacker hijack a user's session by using the session token/cookie?
- Check for cross-site request forgery (CSRF).
- Can authenticated actions be performed within the context of the application from other websites?

- Check the cookie scope. Is the cookie scoped to the current domain, or can it be stolen? What flags are set? Is it missing the secure or HttpOnly flag? To test, trap the request in **Burp Suite** and examine the cookie.
- Understand the access control requirements. How is authentication handled in the application? Could there be any flaws here?
- Test the effectiveness of controls, using multiple accounts if possible.
- Test for insecure access control methods like request parameters and the **Referer** header.
- Test the **PUT** method.
- Test session token strength. If JWT is involved, use **jwt_tool.py**.

Testing Business Logic

- Perform this step last, after becoming familiar with the application.
- Identify the attack surface for logic flaws.
- What does the application do? What is most valuable? What would an attacker want to access?
- Test the transmission of data via the client.
- Is there a desktop or mobile application? Do the protocols used vary between those applications and the web application?
- Test for reliance on client-side input validation.
- Does the application attempt to base its logic on the client side? For example, do forms have a maximum length that is enforced client-side? Can this be modified in the browser and accepted by the application?
- Test any thick-client components like Java, ActiveX, Flash, and Silverlight. Can you download the applet and reverse engineer it?

- Test multi-stage processes for logic flaws. For example, can an attacker go from placing an order straight to delivery, bypassing payment? Can other similar processes be exploited?

Cache Attacks

- If only the path is validated, you can submit malicious queries and headers.
- Cache bad results.
- Test for race conditions.
- Can an attacker access someone else's data?
- If header injection is possible, inject a new response; the cache may store the attacker-controllable response.
- Using multiple host headers or **X-Forwarded-Host** might cause the cache to load the attacker's website and serve it. The links may be written relative to the attacker's host.
- **DNS cache poisoning**: The attacker creates a fake response for the DNS server that is cached. All users will receive the wrong response until the time to live (TTL) expires.
- **Side-channel attacks**: These exploit timing, energy consumption, noises, and electromagnetic leaks, rather than direct weaknesses in the system.
- **Offline web application cache poisoning**: The attacker loads an iframe for a victim who uses Wi-Fi. The iframe points to `facebook.com` but caches the phishing site for a few days. When the user logs in at home, the poisoned cached site is opened.

Others

Bypassing paths and forbidden commands:

- Bash substitutes `?` for any possible character in a folder name (e.g., `/usr/bin/p?ng` equals `/usr/bin/ping`).
- Bash substitutes `*` for any compatible combination in a folder name (e.g., `/usr/bin/who*mi` equals `/usr/bin/whoami`).
- Use square brackets to replace characters in a command (e.g., `/usr/bin/n[c]` equals `/usr/bin/nc`).
- Use concatenation to bypass blacklisting (e.g., `'p'i'n'g` executes the ping command).
- Use Unicode escape sequences (e.g., `\u\n\a\m\e \-\a` executes the `uname -a` command).
- Uninitialized variables equal null (i.e., nothing).
- Use an uninitialized variable between valid characters to form a command (e.g., `p${u}i${u}n${u}g` equals `ping`).
- Use an uninitialized variable without curly braces before any symbol (e.g., `cat$u /etc$u/passwd$u` displays the contents of `/etc/passwd`).
- Fake commands will execute the command while generating errors. For example, `p$(u)i$(u)n$(u)g` will execute the ping command while generating three errors, because Bash will try to execute `u` three times.
- Use the command history to bypass restrictions. For example, `!-1` will execute the last command in the history, while `!-2` will execute the second to last command in the history. Use this technique in conjunction with uninitialized variables and other techniques to bypass restrictions.

Bypassing forbidden spaces:

- Use braces to execute commands (e.g., `{cat,lol.txt}` will display the contents of `lol.txt`).

- Change the Internal Field Separator (IFS), which is a space by default. For example, change it to `]` to execute `cat]/etc/passwd`.
- Use an undefined variable as a space. For example, the command `$u $u` will be saved to the history and can be used as a space. Note that the variable `$u` is undefined in this example.
- Use an undefined variable and the history to bypass the space restriction (e.g., `uname!-1\-a` equals `uname -a`).
- Use hexadecimal encoding to bypass restrictions. For example, the following commands will display the contents of `/etc/passwd`:
- Use newlines to bypass restrictions. For example, the following commands will execute the ping command:
- Review the output of **phpinfo()**. Useful information can be extracted from the output of **phpinfo()**. Look at the settings for the loaded `php.ini` file. Pay close attention to `register_globals` and `allow_url`. If these settings are enabled, it may be possible to launch LFI and RFI attacks. Remember that you may have to add a null byte (`%00`).

I feel this should have continued, but it's decent enough. It could be expanded with the *ChatGPT o1* model, heads up ;)

Penetration Testing | Pentesting | Cybersecurity | Bug Bounty | Infosec