ChatGPT

Designing a Secondary Context Path Traversal Scanner in Go

Overview of Secondary Context Path Traversal

Secondary context path traversal is a class of vulnerabilities in modern web apps where one web application context (or front-end route) proxies requests to a different internal context or backend service. In other words, the application uses a route on the same origin (e.g. `/api/`) to communicate with internal APIs or microservices [1]. If user input can be inserted into that proxied path in unintended ways (such as using directory traversal patterns like `../`), an attacker might escape the intended path and access **internal endpoints** or resources not meant for external users [1]. This is what Sam Curry coined "secondary context" bugs – essentially leveraging path traversal to break out of the front-end context and reach a secondary (backend) context.

In a typical scenario, a front-end will forward requests under a certain path (like `/proxy/` or `/api/`) to an internal service. The vulnerability arises if the routing logic fails to normalize or sanitize path parameters properly. Attackers can then manipulate path parameters with traversal sequences (`../`, encoded variants, etc.) to **reach sensitive internal API paths**. For example, in Sam Curry's Starbucks case, the front-end path `/bff/proxy/stream/v1/...` could be tricked into exposing an internal search API by adding `..\` segments to climb out of the intended directory [2] [3]. The result was access to nearly 100 million customer records by querying internal endpoints that were never meant to be externally accessible [4] [5].

Choosing Go for High-Performance Scanning

Given that you want to scan very large scopes (e.g. many hosts and endpoints in bug bounty programs), **Go (Golang)** is an excellent choice for implementing this scanner. Go is known for its simplicity, strong concurrency support, and efficient memory usage, which makes it well-suited for high-performance networking tools. Key reasons to choose Go include:

- **Concurrency:** Go's goroutines and channels make it straightforward to perform many web requests in parallel, enabling fast scanning of large target lists. You can easily spawn a pool of workers to scan multiple URLs concurrently without complicated thread management.
- **Performance:** Go is a compiled language with low overhead. It handles I/O and JSON processing efficiently, which is useful when analyzing HTTP responses for subtle differences.
- **Networking libraries:** The standard library `net/http` is powerful and easy to use, with built-in support for timeouts, TLS, HTTP/2, etc. This simplifies implementation of HTTP requests and response handling. We can also easily customize headers, handle redirects, and manage cookies if needed.
- **Portability:** Go produces a single static binary. You can run the scanner on various platforms (Linux, Windows, etc.) easily, which is convenient for distributed scanning or deploying to cloud servers. One binary can handle everything ("на одной", i.e. a single self-contained tool).

- **Community and tools:** There's a rich ecosystem of Go tools for security (like ffuf, httpx, gowitness, etc.) which our scanner can take inspiration from. If needed, we can integrate community wordlists or reuse parsing logic from other Go security tools.

Overall, Go provides the speed and concurrency needed to handle **large bug bounty scopes** (hundreds of thousands of URLs) efficiently on one machine. It strikes a good balance between performance and developer productivity.

Application Architecture and Design

To build a **scalable scanner** for big scopes, we should design a robust architecture. The scanner will be a command-line application (CLI) that reads a list of target hostnames and URL paths (the "dictionary" of endpoints to test). Here's an outline of the architecture and key components:

- **Input Management:** The scanner can accept input as a file or stdin containing hostnames and paths. For example, the input might be a list of hosts and their known endpoints (from recon or a web crawler). We can support a simple format like one URL per line (e.g. `https://example.com/path`) or separate lists of hosts and path templates. The program should parse this and generate a queue of URLs to test. If the input is separated (hosts list and paths list), the scanner can combine each host with each path. This could be memory-intensive for huge lists, so we might generate combinations on the fly.

- **Concurrency Model:** Use a **worker pool** to handle scanning tasks. For instance, spawn N goroutines (threads) where each goroutine takes a URL from the job queue, performs the scan, and then takes the next. We should allow configuration of the concurrency level (threads) via CLI flag, to tune performance vs. load. A common pattern is to use a buffered channel as the job queue and have multiple worker goroutines receiving from it.

- **Rate Limiting per Host:** Since one goal is to avoid getting blocked, it's wise to control how many requests are sent to the same host in parallel. We can maintain a map of host->semaphore or use a weighted semaphore that only lets a certain number of concurrent requests to the same domain. For example, allow at most X concurrent requests per host and introduce small delays between batches. This prevents overloading one server with too many simultaneous requests.

- **HTTP Client Configuration:** Use Go's `http.Client` with timeouts (both connect and read timeouts) to avoid hanging on slow responses. We should enable redirect following carefully – possibly disable automatic redirects for scanning (because a redirect might mask a traversal success by forwarding elsewhere). Instead, capture the redirect response for analysis. Also, set a custom `User-Agent` (perhaps mimic a common browser user-agent string by default, or allow user-specified agent) to not look like a default Go HTTP client, which some WAFs might flag.

- **Pluggable Modules (Optional):** Though not strictly necessary for a single binary, we can conceptually separate the scanner into modules:

- **Payload Generator:** Given a base path, produce the list of payload variations to try.
- **Request Sender:** Handles sending HTTP requests and returning responses.
- **Response Analyzer:** Inspects responses to detect potential vulnerabilities.

- **Result Recorder:** Logs or stores any findings. This modular approach (even if just logical separation in code) makes it easier to maintain and adjust each part (for example, updating payloads or detection logic without touching concurrency logic).

- **Data Handling and Output:** For large-scale scans, printing everything to console might be overwhelming. Instead, log only potential vulnerability findings or summary stats to console, and write detailed results to an output file (e.g. CSV or JSON). The output should include host, path, payload, response code, and any notable observations (like content snippet or headers). This way, you can sort through findings after the scan. Optionally, integrate with a database or at least allow results to be saved so you can resume scanning or avoid re-testing known vulnerable endpoints.

- **Single vs. Multi-Process:** The scanner will likely be a single process on one machine ("на одной машине"), but nothing prevents running multiple instances on different parts of the scope if needed. Given Go's efficiency, one instance should handle thousands of requests per second on decent hardware, but for extremely large scopes you could distribute the target list across multiple machines.

This architecture ensures the scanner is **monolithic** (one binary doing everything) but structured, and capable of handling a big scope without tying up too many resources or getting itself blocked immediately.

Payloads for Path Traversal Fuzzing

Choosing the right payloads is critical. We need to try a variety of **path traversal sequences** to maximize the chance of bypassing filters and triggering the vulnerability on different tech stacks. The payloads should include both straightforward traversal and encoded or obfuscated variants to evade simplistic validations (e.g. WAF rules or backend normalization). Here are some payloads and strategies:

- **Basic directory traversal:** The classic `../` sequence (dot-dot-slash) to go up one directory. Also try multiple levels like `../../` (two levels up) and so on. Many frameworks block `../` explicitly, but we still test it to gauge behavior differences [6].
- **URL-encoded traversal:** Encode special characters to bypass filters. Common examples:
- `%2e` for `.` and `%2f` for `/`. So `..%2f` is an encoded `../`. This often slips past poorly implemented checks. For instance, a researcher found `/experience/..%2f` passed a filter that blocked raw `../`, yielding a 404 from the internal server (whereas other variants gave 403) [7].
- Double URL encoding or Unicode encoding: e.g. `%252e%252e%252f` (double-encoded "../"). In some cases, double decoding by the server can lead to traversal.
- Use **backslash** encoding for Windows or alternate path parsing: `%5c` is a backslash (`\`). Some frameworks on Windows or proxies might normalize `\` differently, so payloads like `..%5c` can work. Sam Curry's initial payload list included both forward and backslash variants [6].
- **Mixed dot and slash obfuscation:** WAFs often detect straightforward `../` patterns. You can insert harmless elements to break the pattern. For example: `..../` (adding an extra dot), or `..%2f./` etc. Sam Curry bypassed a WAF by using a complex pattern: `web\..\.\..\` (mixing backslash and an extra `.\.`) which the WAF did not recognize as two consecutive directory traversals [8] [9]. Our scanner's payload list can include some of these WAF-evasion patterns (like `..\.\..\` or combining forward and backslashes) to maximize success against filters.

- **Semicolon trick:** Some Java-based servers (older Tomcat versions) had vulnerabilities where `..;/` could bypass normalization. Including a payload like `..;/` or `/..;` in the path might exploit misconfigured path parsing [7]. In one case, adding a semicolon fooled the proxy's parser and allowed traversal.
- **Null byte insertion:** `..%00/` – The `%00` (null byte) can terminate strings in C-based logic, potentially truncating a path. This is less common in modern web frameworks due to safety checks, but older code in C/C++ or PHP might be vulnerable. Sam Curry tried `..%00/` and `..%0d/` (carriage return) in his attempts [10]. These can be part of the payload list for thoroughness.
- **Trailing special chars:** Adding a `?`, `#`, or other URL special characters at the end of the path segment to see if the backend parser gets confused. For example, appending `?` (`%3F`) or `#` (`%23`) after a traversal sequence. Sam Curry tested some of these: e.g. `.../%3f` or `.../%26` just to observe behavior [11]. These might reveal if the path is being concatenated or cut off at certain characters (for instance, `#` might be interpreted as fragment by front-end but passed as literal to backend, causing differences).
- **Multi-level traversal:** Once a single `../` is accepted, the scanner should attempt multiple levels (`..%2f..%2f`, etc.). However, rather than blindly trying up to some large depth (which could cause a lot of requests), a smarter approach is incremental:
- Try one `../`. If the response suggests success (or at least not an outright block), try two `../` in the next request, and so on, until we either hit a limit (say 5-7 levels) or see a distinctive error (like a backend "Bad Request" at the root). Sam Curry noted that when he finally went too many levels up, the response changed to a 400 Bad Request, indicating he reached the root of the internal API [12] [13].
- **Context-specific payloads:** If any clue suggests the backend tech, we could adapt payloads. For example, if we suspect a Node.js backend, maybe test for encoded double slashes or `%c0%af` (UTF-8 overlong encoding of slash). But since we **"don't tie to one backend stack"** (не привязываясь к одному стеку), the payloads above are generic enough for most scenarios (Java, .NET, Node, Nginx proxies, etc.).

It's a good idea to maintain a **payload list** (maybe a simple slice of strings in code or an external file) that the scanner will iterate through for each endpoint. In summary, our payload strategy is to include **classic traversal sequences and creative encoded variants**. This ensures that if any variant can slip past input validation, we will find it.

For example, a condensed payload list might include:

```
../
..%2f
..%5c
..;/
..%2f..%2f  (for multiple levels, or generate dynamically)
%2e%2e%2f   (another encoding for ../)
.%2e/       (dot-dot in another form)
..\.\..\    (WAF bypass pattern)
```

...and so on. We should also include baseline control (like requesting the normal path without modification) for comparison.

Scanning Strategy and Two Modes

To accommodate different use cases, we'll implement **two scanning modes** selectable via CLI flags (for example, `--mode=fast` vs `--mode=full`, or `-quick` vs `-deep`). This gives flexibility between quickly scanning a broad scope and deeply probing a narrower scope:

- **Fast Detection Mode:** This mode prioritizes speed and basic detection. It will:
- Use a **reduced payload set** – only the most effective few payloads (say 2-3) that have high chance to reveal a vulnerability (for instance: `../`, `..%2f`, and `..%5c..%2f` double traversal). By limiting payloads, each endpoint gets only a few requests, which is faster and less likely to trigger heavy WAF defenses.
- Maybe only try **one level** of traversal at first. The goal is to see if any traversal at all is possible. For example, append `..%2f` to the given path and see if the response is unusual [7].
- **No extensive fuzzing** of internal paths in this mode. We only check for the *existence* of a traversal bug, not enumerate all internal content.

- This mode is ideal for scanning thousands of endpoints quickly to flag which ones might be vulnerable.

- **Full/Deep Scan Mode:** This mode does comprehensive probing once you suspect an endpoint is vulnerable (or if you want to thoroughly scan a smaller scope):

- Uses the **full payload list**, including all the exotic encodings and multi-step traversals. It will systematically try single traversal, double traversal, etc., and also include WAF bypass patterns.
- **Iterative climbing:** If one traversal attempt returns an indicator of success (like a different 404 or 400), the scanner can automatically attempt to go further (two `../`, three `../`...) until it hits the root or maximum preset depth. Essentially, it can simulate what Sam Curry did manually – climbing to the root of the internal API [12] [13].
- **Directory brute-forcing (optional):** In deep mode, after reaching what looks like the internal root (or any intermediate directory), the scanner can attempt to discover internal endpoints. This could involve sending requests to common path names (like `admin/`, `api/`, `config`, `health`, etc.) appended after the traversal sequence. For example, if `/app/proxy/..%2f` appears to put us at an internal root (giving 400 or a different error), then try `/app/proxy/..%2fadmin/` or other known paths. In one real case, fuzzing in the internal context found `application.wadl` which listed internal API endpoints [14] [15]. Another case discovered an `api/` endpoint of Nginx's management API by brute-forcing after traversal [16] [17].
- **Multiple HTTP methods:** If the mode is full, we might also try both GET and POST requests for each path. Sometimes an endpoint might only accept a certain method to trigger internal logic (e.g., a POST endpoint that is proxied). Berserker's writeup noted they specifically hunted for a **POST request** vulnerable to secondary context traversal [18]. So our scanner in full mode can attempt a POST with an empty body (or some default body) to the same URL with traversal payloads, in case the internal route is expecting POST.
- Because this mode is much heavier (lots of requests per endpoint), the CLI flag ensures the user explicitly chooses it for deeper testing on narrower targets or after identifying likely vulnerable endpoints.

Scanning process: For each target URL (host + base path): 1. Optionally, send one **baseline request** to the base path (or a slightly modified benign request) to see the normal response. This helps later to compare what changes when we inject payloads. 2. Iterate through the chosen payloads (depending on mode). For each payload: - Construct the test URL. For example, if base path is `https://example.com/app/item/42` and payload is `../`, the test URL becomes `https://example.com/app/item/42/../` (or sometimes insertion might be different, see below). - Send the HTTP request and record the HTTP status, response headers, and a snippet of the body. - If in **fast mode**, perhaps stop further payloads for this endpoint if a clear positive is found early (to save time). E.g., if `..%2f` yields an obvious difference, no need to try other payloads in fast mode. - If in **deep mode**, use logic based on responses to decide next steps (discussed in detection logic below). E.g., if single traversal yields a clue, automatically try double traversal, etc. 3. Analyze responses to decide if the target is vulnerable (see next section on detection). 4. If vulnerable, record the finding (and possibly do extra discovery in deep mode as described).

Insertion points: Typically, if the input dictionary has complete URLs or paths, we assume we append the traversal at the end of the path. However, what if the path itself contains user input segments? In some cases, it might be useful to insert traversal **in place of a parameter**. For example, if we have a URL with an ID: `/report/12345`, replacing `12345` with `../..` might traverse out of the `/report/` directory on the backend. Our scanner can identify numeric or UUID segments and attempt replacing them with traversal sequences. This can be an advanced feature: - Detect segments of the path that look variable (e.g., all digits, or an alphanumeric ID). - Try substituting those segments with `..` payloads as well. This way, we are not only appending traversal at the end, but also trying it in intermediate path positions that might be vulnerable.

Finally, ensure the scanner does **not assume any specific backend**. We are purely looking at external behavior (status codes, content) to infer vulnerability. By scanning **“все”** (everything) generically, we let the responses inform us, rather than, say, only looking for Java-specific signs. This broad approach will work whether the target is Java, .NET, Node.js, Nginx, etc., since many of the payload techniques are universal or at least harmless to try.

Detection Logic and Avoiding False Positives

Identifying a secondary context traversal vulnerability is tricky – it often doesn’t give a clean success like a clear file content dump. Instead, success may be indicated by subtle changes in the response (status code differences, error message formats, headers). Our scanner’s logic must carefully compare responses to spot these clues while filtering out normal behavior.

Key points for detection: - **Compare to baseline:** As mentioned, capturing a baseline response for the original URL (or a safe variant) is useful. For example, request the normal path (with a valid or random parameter) and note the status and response length/type. Then if a traversal payload yields a different status or content, that deviation is a red flag. If baseline and payload both return 404 but one is HTML and one is JSON, that’s significant. In Starbucks, Sam noticed that a traversal attempt returned a JSON **"Not Found"** error, whereas a normal invalid request returned an HTML page – indicating the traversal request was handled by the internal API (which gave JSON) [19] [3] . - **Status code heuristics:** Look for unusual HTTP codes when adding traversal: - If normally a request gives 403 or 404, but with traversal you get **400 Bad Request**, it might mean you navigated out to an internal root or malformed the URL enough to hit a different parser [20] . For example, hitting the root of an API might return 400 if no resource was specified, which is a sign of success (as Blake Jacobs observed when his traversal got a 400, indicating the proxy reached an internal Nginx API root) [20] . - If other payloads were blocked (e.g. returned 403 by WAF) and one payload returns 404, that 404 could be coming from the backend (meaning the payload bypassed the filter and attempted to fetch something not present) [7] [20] . This scenario is a **strong indicator** – one variant sneaking through. - A **301/302 redirect** where normally you wouldn't get one can be a clue. Sam Curry found that requesting an internal path without a trailing slash returned a 301 redirect to add the slash (revealing the existence of that path) [21] [22] . So if a traversal request returns a redirect (especially to a path that includes the traversal or points to a new location), it might mean you successfully reached an internal endpoint that is redirecting. - Obviously, a **200 OK** on a traversal payload (when the baseline was an error) is a clear sign – it means you likely accessed a valid internal resource. For example, Blake's Nginx case: `/experience/..%2fdashboard.html` returned 200 OK (internal admin page) while the normal `/dashboard.html` was 403 externally [23] . Our scanner should treat an unexpected 200 with traversal as a likely vulnerability (and log the content or URL for user to verify). - **Content differences:** Even if status codes are the same, the **response body** might differ: - **Content-Type or format:** Perhaps the baseline 404 is an HTML page (with company branding), but the traversal 404 is a JSON or XML error. A difference in `Content-Type` header or response structure is a big hint that the response came from a different component. For instance, JSON vs HTML error message difference flagged the Starbucks issue [19] [3] . Our scanner can check if the payload response body contains keywords like `"error"` or `"Not Found"` in JSON vs. an HTML `<title>404` in the baseline. - **Headers:** The presence or change of certain headers can reveal internal services. E.g., an internal server might add a header like `X-Backend-Server: ...` or the `Server` header might change (maybe front-end is Apache but internal is Nginx, etc.). If we see new headers or different `Server` strings when using traversal, that suggests we reached a different server or context [24] [25] . For example, an internal Nginx API might respond with `Server: nginx` whereas the front-end normally hides that or uses a different value. - **Length and content details:** If traversal yields a much larger response or one containing data structures (like JSON arrays, XML, etc.) that the normal app doesn't return, it's a sign. For instance, retrieving a big JSON list when using a `/search` endpoint internally (like the Starbucks accounts list) clearly indicates a vulnerability [5] [26] . The scanner might not fully dump 100 million records (nor should it!), but even seeing the shape of the data (e.g., keys like "EmailAddress" or many entries) and the length is enough to confirm. - **Multiple payload consistency:** To avoid false positives, we should see a **pattern** of behavior. If only one weird request out of 10 gave a slight difference, we should be cautious. But if consistently, traversal payloads produce a coherent set of differences (like one yields 404 JSON, two yields 400, and maybe a known internal endpoint yields 200), then we have a solid case. The scanner can use a simple state machine: - If the first traversal attempt yields something interesting (different status or content), mark the endpoint as "suspicious". - Then perform follow-up tests (in deep mode) to confirm: e.g., try an additional traversal level or attempt to fetch a known likely file (like `..%2frobots.txt` or `..%2fapplication.wadl`). If that returns a positive result (like a 200 or different response than baseline), escalate the finding to "confirmed". - Additionally, the scanner can double-check by requesting the discovered internal path **without** traversal to see if it's accessible normally. For example, if `..%2fadmin/config` returned 200, try directly `/admin/config` on the same host. If direct access is 403 or not found, but via traversal it was accessible, that confirms a true vulnerability (internal resource exposed) [23] . This check helps eliminate false positives where maybe the resource was always public (just obscure). - **False positive scenarios:** We need to be mindful of a few cases that could mislead: - Some apps might just echo the path back or respond differently to certain malformed URLs without actually being vulnerable. For instance, a generic 400 Bad Request might occur for any weird URL (not necessarily indicating internal API). To mitigate this, we rely on *comparative differences* (we got 400 only after certain payload, whereas normal similar malformed didn't). - WAF interference: If a WAF blocks one payload but not another, the differences in response might be due solely to WAF filtering, not actual vulnerability. E.g., `../`

-> blocked (403), but `..%2f` -> passed to backend (404). Here the difference *does* indicate vulnerability (bypass succeeded) [7] [20], but we should be careful that we actually reached backend (the content or headers should confirm it's a backend 404, not just a different block page). Many WAF block pages have distinctive content (like "Request blocked" text). Our scanner can watch for known WAF signatures in responses (common ones from Cloudflare, Akamai, etc.) and perhaps log those differently. If we only see WAF block pages for all payloads, then the target might be protected or no vulnerability could be confirmed. - **Static false positives:** If our input list has some path that naturally returns a 404 in JSON (maybe it's an API endpoint that always returns JSON formatted errors), and our traversal attempt also returns JSON 404, we might think it's internal when it's actually just the normal API behavior. To avoid this, the baseline vs traversal comparison is key. We should compare the *same endpoint* with and without traversal. For example, request `/api/v1/users/12345` vs `/api/v1/users/12345/..%2f`. If both return JSON 404, it might just be that the app always gives JSON errors. But if one is JSON and the other is HTML or has differences in phrasing, then it's something. - Including multiple payload variations helps ensure we differentiate between "just invalid input" and "traversal successful". If every variant yields identical results, likely no vulnerability. If one variant stands out, that's the signal.

In summary, the detection logic uses a combination of **status code analysis, content comparison, and follow-up verification** to reliably identify a secondary context path traversal. The scanner should err on the side of caution: flag anything suspicious for the user to manually verify, but try to filter obvious false positives by using the techniques above.

Handling Rate Limits and Blocking

When scanning at scale, you must be careful not to get yourself locked out or overload the target. Here are strategies to **bypass rate limits and avoid blocks**:

- **Concurrent but not too aggressive:** We mentioned implementing per-host concurrency limits. For example, restrict to maybe 5-10 parallel requests per host (configurable). Even if our tool overall is scanning 100 endpoints in parallel across many hosts, each individual server won't get slammed with more than a reasonable number of simultaneous requests. This prevents triggering anti-DoS measures on the host.
- **Throttle request rate:** In addition to parallelism limits, introduce a small **random delay or jitter** between requests to the same host. Instead of blasting requests in a tight loop, add, say, 100-500 milliseconds of random sleep between batches of requests. This makes the traffic pattern less robotic and less likely to hit strict rate-throttling. Some WAFs will block clients that make X requests in Y seconds; a bit of randomness can keep under the radar.
- **Rotate User-Agent and headers:** If you send hundreds of requests with the same user-agent, especially the default Go user-agent, you might get flagged. The scanner can cycle through a list of common browser user-agent strings for different requests. Also, consider randomizing other headers (order of headers, or add innocuous headers like `X-Request-ID` with random UUIDs) to make each request look more unique [27] [28]. Be careful though: some WAFs track by IP and session more than headers, but it can help marginally.
- **IP rotation (proxies):** For truly massive scans or very protected targets, using multiple IP addresses or proxies can bypass IP-based rate limits [27]. We could allow the user to specify a **proxy list** and cycle through them (e.g., using `http.Transport` with proxy setting). This way, requests to the same target come from different IPs. This is more complex and usually not default, but providing the option via CLI could be valuable for long-running scans where blocks are encountered.

- **Detecting blocks:** The scanner should monitor responses for signs of blocking:
- HTTP 429 Too Many Requests – clear indicator of rate limiting. If we receive 429, our code can automatically back off: pause requests to that host for a specified “cool-down” (maybe read `Retry-After` header if present) [29] .
- Sudden sequence of 403/406 errors after a certain point could mean a WAF has flagged us. If, say, everything starts returning a generic block page or CAPTCHA page, the scanner could log "Potential block detected on host X, pausing scanning for a minute".
- Implement an exponential backoff: e.g., on first block, wait 30 seconds and try again slowly. If still blocked, either escalate to using a different proxy or abort further scanning of that host to avoid causing more issues.
- **Bypassing WAF filters:** Apart from rate limiting, WAFs also block on pattern. Our varied payload approach already helps here (one encoding might bypass the filter). Additionally, if one payload triggers a block (say `../` causes an immediate IP ban), we might want to try less obvious payloads first. In practice, we could order the payloads from least likely to trigger WAF to most likely. For example, perhaps start with URL-encoded `..%2f` instead of raw `../`, since encoded might slip through quietly. By the time we use an obvious one, we might have identified a vulnerability and can skip the dangerous payload. This is a bit speculative, but ordering and gradually ramping up "aggressiveness" can help.
- **Maintain session/cookies if needed:** If the scanner has to log in or maintain a session for certain endpoints (not typical for pre-auth vulnerabilities like these, but just in case), it should reuse cookies and handle them. Logging in once and reusing the session can avoid triggering multiple authentication attempts. However, for our use-case (pre-auth path traversal), we mostly deal with unauthenticated scanning.

Finally, always respect **politeness** in scanning: provide configurable options for max requests per second, timeouts, etc., so the user can ensure they don’t overwhelm targets unintentionally. The goal is to find vulnerabilities, not to perform a denial of service. By implementing backoff and making the scanner's pattern mimic normal user traffic (to some extent), we reduce the chance of being blocked before we find the juicy stuff.

Conclusion

Building a secondary context path traversal scanner in Go involves combining efficient design with clever attack techniques drawn from real-world research. We chose Go for its high concurrency and portability, which lets us scan big scopes in bug bounty programs swiftly. The scanner’s architecture uses concurrent workers with careful rate limiting to cover lots of endpoints without tripping defenses. We incorporated two modes: a fast scan for initial detection and a deep scan for thorough exploitation, so you can adapt to the scope size and depth needed.

Crucially, we use a diverse set of payloads – from straightforward `../` to encoded and obfuscated variants – to maximize our chances of bypassing input filters and hitting internal APIs. The logic to detect a vulnerability focuses on **differences in responses**: a different status code (e.g. 404 vs 403) or a change in response format (JSON vs HTML) when a traversal is added can be the telltale sign [19] [7] . By automating these checks and even verifying by direct access attempts, we reduce false positives and ensure confidence in our findings.

Finally, the scanner is built to be stealthy and resilient against rate limits: using concurrent-but-controlled scanning, random delays, and payload ordering to avoid WAFs. It will slow down or shuffle requests as needed to stay under the radar, and can even utilize multiple IPs if provided [27] [28].

Armed with this tool, you'll be able to work through large target scopes and sniff out those secondary context path traversal bugs that often lead to high-impact bounty findings. As demonstrated by researchers in cases like Starbucks and others, these bugs can be extremely powerful [5] [14]. Good luck, and happy hacking!

Sources:

- Sam Curry's Starbucks write-up (example of secondary context traversal and payloads) [30] [19]
- Blake Jacobs' NGINX reverse proxy traversal (noticing 404 vs 403 differences) [7] [20]
- CriticalThinking Podcast notes on secondary context & tips (overview of concept) [1] [31]
- Berserker's medium post (real-world impact of such vulnerability) [14] [18]
- Searchlight Cyber (Assetnote) on Workspace ONE UEM vuln (another example of exploitation) [32] [33]

---

[1] [31] [HackerNotes Ep. 65] Motivation and Methodology with Sam Curry (Zlz)
https://blog.criticalthinkingpodcast.io/p/motivation-and-methodology-with-sam-curry

[2] [3] [4] [5] [6] [8] [9] [10] [11] [12] [13] [19] [21] [22] [26] [30] Hacking Starbucks and Accessing Nearly 100 Million Customer Records
https://samcurry.net/hacking-starbucks

[7] [16] [17] [20] [23] [24] [25] Breaking Reverse Proxy Parser Logic
https://www.secjuice.com/breaking-parser-logic-gain-access-to-nginx-plus-api-read-write-upstreams/

[14] [15] [18] Secondary Context Leads To Company Takeover. | by Berserker | Medium
https://medium.com/@Berserker1337/secondary-context-leads-to-company-takeover-0dc88ca751ea

[27] Rate Limit in Web Scraping: How It Works and 5 Bypass Methods
https://scrape.do/blog/web-scraping-rate-limit/

[28] Bypass Rate Limit While Web Scraping Like a Pro - ZenRows
https://www.zenrows.com/blog/web-scraping-rate-limit

[29] Prevent API Overload: A Comprehensive Guide to Rate Limiting with ...
https://dev.to/arifszn/prevent-api-overload-a-comprehensive-guide-to-rate-limiting-with-bottleneck-c2p

[32] [33] Secondary Context Path Traversal in Omnissa Workspace ONE UEM › Searchlight Cyber
https://slcyber.io/assetnote-security-research-center/secondary-context-path-traversal-in-omnissa-workspace-one-uem/