dependency	vulnerability	solution
golang.org/x/net 0.16.0	**HTTP/2 rapid reset can cause excessive work in net/http** A malicious HTTP/2 client which rapidly creates requests and immediately resets them can cause excessive server resource consumption. While the total number of requests is bounded by the http2.Server.MaxConcurrentStreams setting, resetting an in-progress request allows the attacker to create a new request while the existing one is still executing.	1. Updating the dependency 2. Configuring Server.MaxConcurrentStreams (Explicitly setting the Server.MaxConcurrentStreams setting in the HTTP/2 server configuration. This will limit the number of simultaneously executing handler goroutines, preventing excessive resource consumption in case of rapid resets.) 3. Monitoring and adjusting (Regularly monitoring the application's performance and adjusting the Server.MaxConcurrentStreams setting as needed based on the specific use case.)
	HTTP/2 Stream Cancellation Attack The HTTP/2 protocol allows clients to indicate to the server that a previous stream should be canceled by sending a RST_STREAM frame. The protocol does not require the client and server to coordinate the cancellation in any way, the client may do it unilaterally. The client may also assume that the cancellation will take effect immediately when the server receives the RST_STREAM frame, before any other data from that TCP connection is processed.	1. Updating the dependency 2. Implementing Stream Reset Counter (Implementing a mechanism to track and limit the number of stream resets that can occur in a given window of time. This prevents a malicious client from overwhelming the server with rapid stream resets.) 3. Applying Swift-NIO-HTTP2 Remediation Techniques (Applying a reset counter with a sliding window to limit the number of stream resets in a given time frame, preventing the server from committing to excessive work that will be discarded.)
	Improper rendering of text nodes in golang.org/x/net/html Text nodes not in the HTML namespace are incorrectly literally rendered, causing text which should be escaped to not be. This could lead to an XSS attack.	1. Updating the dependency 2. Reviewing and sanitizing text content (Escaping special characters to prevent unintended HTML or JavaScript execution; using functions like html.EscapeString to ensure that text content is properly sanitized.) 3. Implementing Content Security Policies (Configuring the server to send appropriate headers, such as the Content-Security-Policy header, to restrict the sources from which content, including scripts, can be loaded.)

golang.org/x/crypto 0.16.0	**Prefix Truncation Attack against ChaCha20-Poly1305 and Encrypt-then-MAC aka Terrapin** Terrapin is a prefix truncation attack targeting the SSH protocol. More precisely, Terrapin breaks the integrity of SSH's secure channel. By carefully adjusting the sequence numbers during the handshake, an attacker can remove an arbitrary amount of messages sent by the client or server at the beginning of the secure channel without the client or server noticing it.	1. Updating the dependency 2. Implementing "Strict Kex" Countermeasure (Altering the SSH handshake to ensure that a Man-in-the-Middle attacker cannot introduce unauthenticated messages and manipulate sequence numbers across handshakes.) 3. Disabling Affected Algorithms Temporarily (Using unaffected alternatives like AES-GCM until patches are available.)
follow-redirects 1.15.3	**Follow Redirects improperly handles URLs in the url.parse() function** Versions of the package follow-redirects before 1.15.4 are vulnerable to Improper Input Validation due to the improper handling of URLs by the url.parse() function. When new URL() throws an error, it can be manipulated to misinterpret the hostname. An attacker could exploit this weakness to redirect traffic to a malicious site, potentially leading to information disclosure, phishing attacks, or other security breaches.	1. Updating the dependency 2. Validating and sanitizing input URLs (Using a robust URL validation library, such as UrlSerializer, or implementing custom validation to ensure that the URLs are well-formed and do not contain malicious elements.) 3. Implementing Content Security Policies (Configuring the server to send appropriate headers, such as the Content-Security-Policy header, to restrict the sources from which content, including scripts, can be loaded.)