Eclipse Jetty

Threat Model and Code Review with Fix Review

June 13, 2023

Prepared for:
Greg Wilkins
The Eclipse Foundation
Organized by the Open Source Technology Improvement Fund, Inc.

Prepared by: **Cliff Smith, Sam Alws, Kelly Kaoudis, and Spencer Michaels**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

OSTIF engaged Trail of Bits to review the security of the Eclipse Foundation's Jetty project. From March 6 to March 30, 2023, a team of two consultants conducted a lightweight threat model of the project, and then a separate team of two consultants conducted a security review of the client-provided source code; the two reviews took a combined six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the product's source code and documentation. We performed a static code review using both automated and manual processes, supplemented by dynamic testing of the target system.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	9
Medium	7
Low	4
Informational	5
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Code Quality	2
Cryptography	1
Data Exposure	2
Data Validation	11
Denial of Service	7
Error Reporting	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-JETTY-1**
  An integer overflow could occur during the parsing of HPACK headers, which could cause excessive resource consumption. A maliciously crafted header will cause Jetty to allocate a 1.6 GB buffer while parsing a single message.

- **TOB-JETTY-3**
  An error in the quotation mark escaping algorithm used for command line arguments in the EE9 and EE10 CGI servlets enables arbitrary command execution.

- **TOB-JETTY-6**
  The WebSocket frame parser uses a 32-bit integer to represent the frame's length field, which can contain up to 64 bits. In addition to crashes, this bug can cause Jetty to mistakenly split one WebSocket frame into multiple in a manner similar to the errors that enable HTTP request smuggling attacks.

- **TOB-JETTY-19**
  The Jetty module configuration system supports Maven package downloads from `maven://` URIs. When the `maven-metadata.xml` file is parsed, document type definitions (DTDs) are parsed, which enables XML external entity (XXE) and XML entity expansion (XEE) attacks.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Jeff Braswell, Project Manager
jeff.braswell@trailofbits.com

The following engineers were associated with this project:

Kelly Kaoudis, Consultant
kelly.kaoudis@trailofbits.com

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

Cliff Smith, Consultant
cliff.smith@trailofbits.com

Sam Alws, Consultant
sam.alws@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 6, 2023	Lightweight threat model kickoff
March 7, 2023	Threat model discovery #1
March 10, 2023	Threat model discovery #2 and code review kickoff
March 15, 2023	Threat model readout meeting
March 30, 2023	Report readout meeting
May 5, 2023	Delivery of final report
June 13, 2023	Delivery of fix review

Project Goals

The engagement was scoped to provide a security assessment of Jetty. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the header and cookie parsing algorithms for HTTP/1 and HTTP/2 correct and standards-compliant?
- Are the WebSocket, HTTP/2, and HTTP/3 implementations secure and correct, including their code for handling parsing, message generation, and connection management?
- Do the Jetty Core, EE9, and EE10 packages securely serve static resources from the web server's filesystem? Can an attacker download files outside the configured root directory?
- Can attackers bypass any of the servlet security configuration settings specified in a servlet's `web.xml` file?
- Is the alias checking system implemented correctly?
- Does the application deployment system have any exploitable bugs?
- Do web application deployment and other features that extract archive files correctly validate file paths? Are any such features vulnerable to "zip slip" or other directory traversal attacks?
- Are the cryptography and key management features compliant with best practices?
- Are memory management operations, including buffer allocation and deallocation operations during request generation and parsing, correct and secure?

Project Targets

The engagement involved a review and testing of the following target.

Eclipse Jetty

Repository	https://github.com/jetty/jetty.project/tree/jetty-12.0.x
Version	`12.0.0` (`rev. bd0186c2f78fb7c87c7bfadf9b0a970657d071f3`)
Type	Java
Platform	JVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- A manual review of the parsers and protocol implementations, including HTTP/1.1, HTTP/2, HTTP/3, QUIC, HPACK, QPACK, cookies, multipart encoding, and WebSockets
- A manual review of the start, module, and deployment systems
- Dynamic testing of the module configuration and the start system
- Static analysis of the entire codebase using Semgrep and CodeQL
- Fuzzing of the parsers and protocol implementations using libfuzzer

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Our code review of the EE9 and EE10 libraries was not comprehensive.
- The protocol implementations were not compared to and validated against the applicable specifications point-by-point.

Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model, drawing from Mozilla's "Rapid Risk Assessment" methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). We began our assessment of the design of Jetty by reviewing the Eclipse Jetty 11.x and 12.x operations and programming guides and Jetty's in-progress CVE fix discussions.

Data Types

Depending on its configuration, a deployed Jetty server or client includes Jetty's implementations of standard web protocols as well as Java-specific protocols, including the following:

- HTTP/1.0, HTTP/1.1, HTTP/2 (cleartext and secure versions), and HTTP/3
- WebSocket
- FastCGI
- SOCKS4
- PROXY protocol

Jetty also surfaces TLS- and ALPN-related information to application developers through Jetty-provided callbacks connected to the underlying Java development kit (JDK) functionality.

Data Flow

Network Data Flow

The following diagram shows an example of a distributed deployment of Jetty.

Note that the stack of boxes labeled "Jetty Server Instance" represents a cluster of several Jetty instances serving the same application logic, each deployed on its own Java virtual machine (JVM), managed by an orchestration system such as Kubernetes.

Also note that each box labeled "Jetty" in the diagram represents a server coupled with the Jetty client component. The client component makes outbound requests on the server's behalf to other servers.

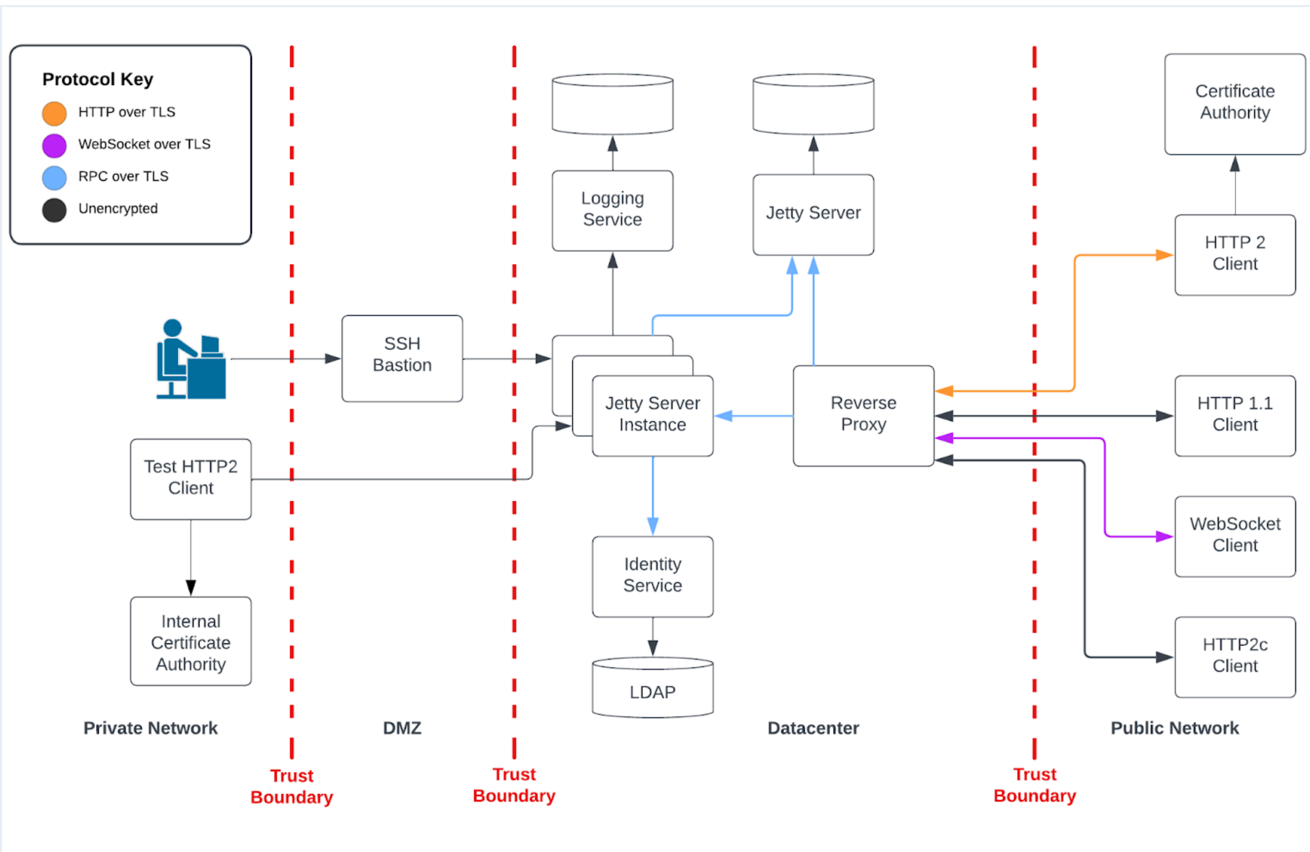

Figure 1: Example network data flows in a distributed deployment of Jetty

Embedded Data Flow

The following diagram shows an example deployment of Jetty as the embedded servlet container for another Java framework—in this case, Spring Boot. In this example, Spring Boot starts Jetty. Then, at runtime, requests pass through Jetty first and then through Spring components (here, a security filter and a request filter) before reaching the endpoint business logic.

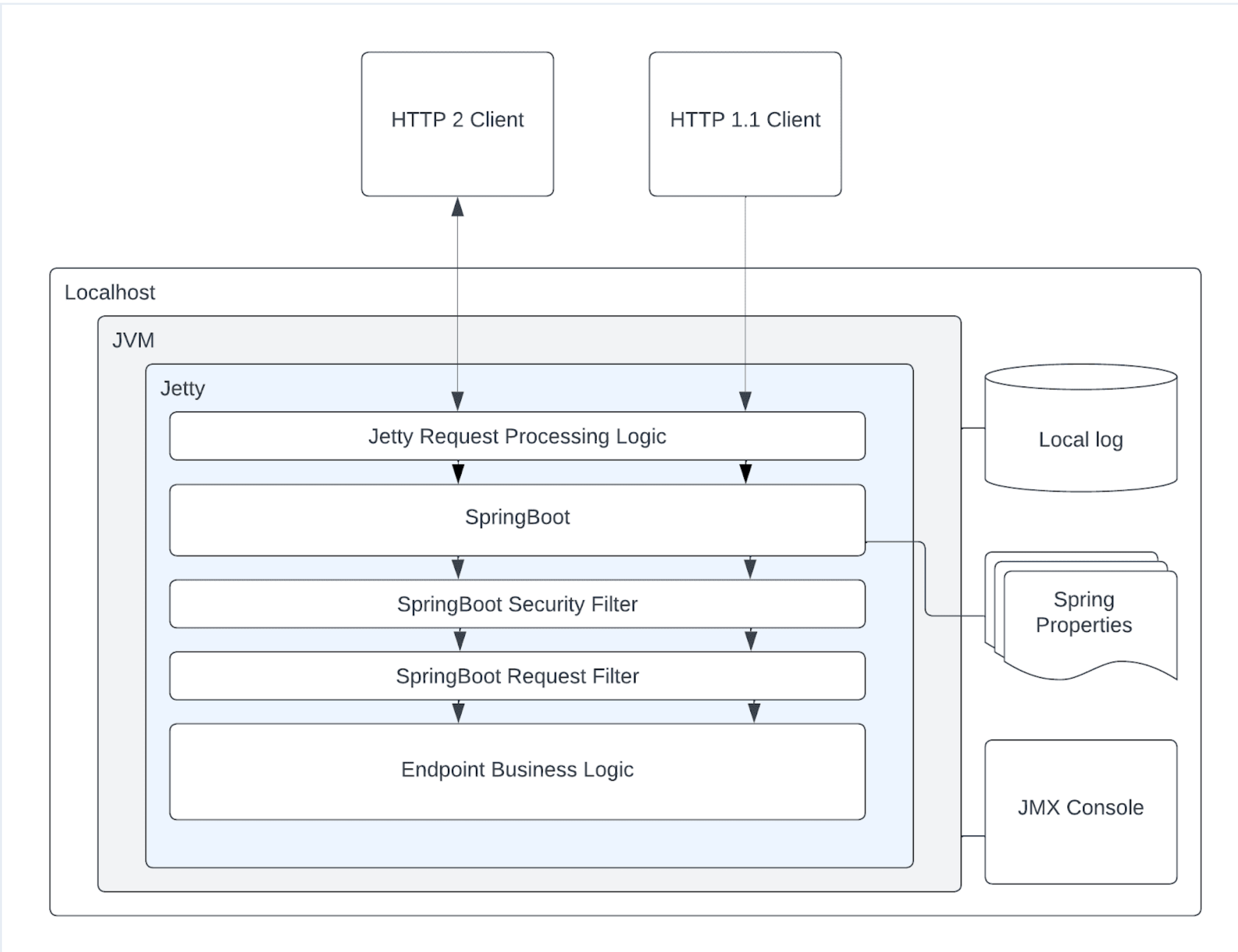

Figure 2: Example data flows where Jetty is the embedded servlet container for Spring Boot

Component Tree

The following diagram shows an example component tree of beans that a typical developer might use, such as client request filters that accept or reject connections before Jetty passes them to the served web applications, various connection factories that create and manage client connections, a login service to protect a particular `ConnectionFactory`, and several types of logging and monitoring mechanisms, the most common of which is Java Management Extensions (JMX)-based. Note that each bean must implicitly trust its registered parent.

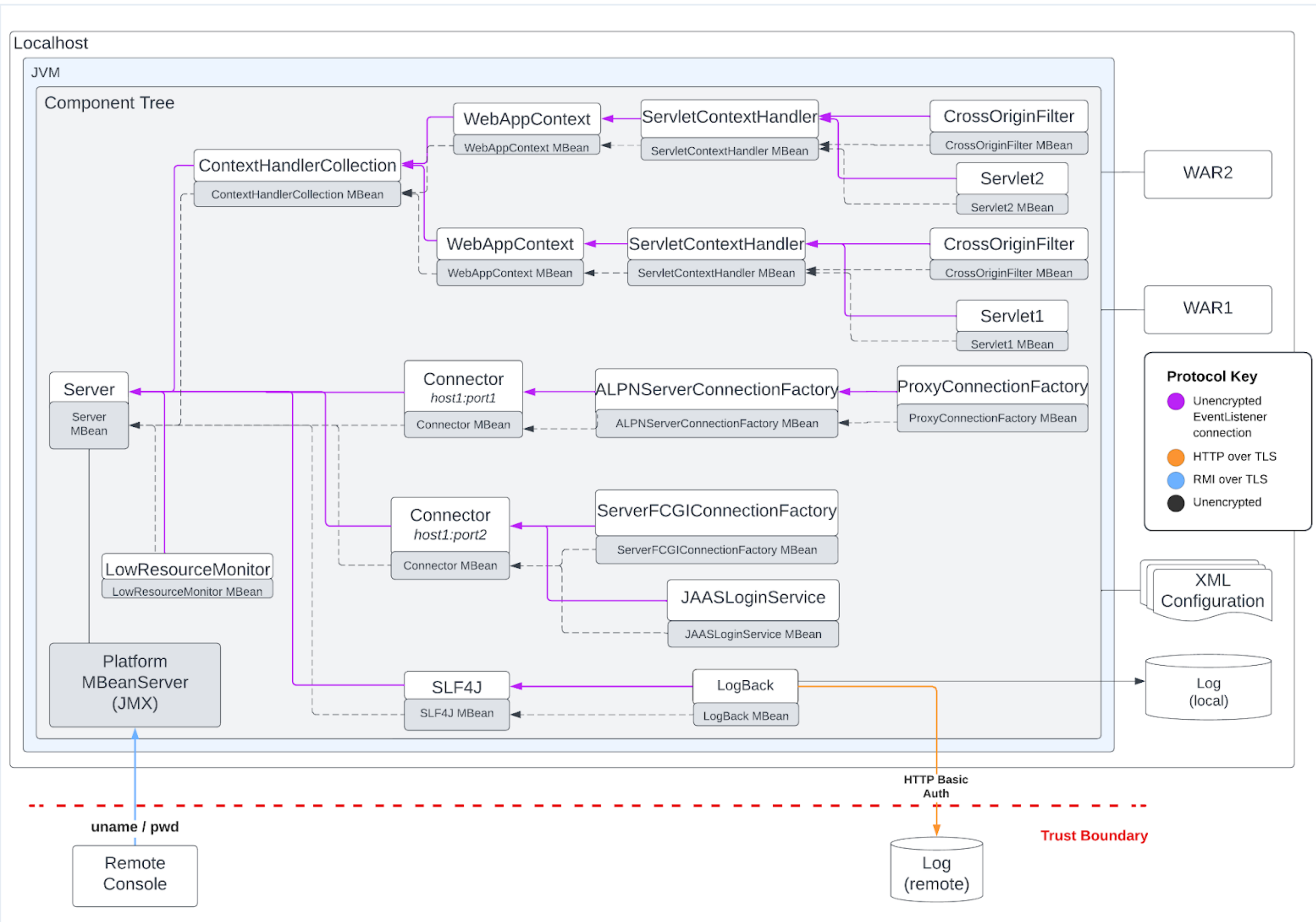

Figure 3: An example Jetty component tree

Components

The following table describes each Jetty component and dependency identified for our analysis. It also indicates whether the component or dependency is *not* in scope; an asterisk (*) next a component's name indicates that it was out of scope for this assessment. We explored the implications of threats involving out-of-scope components that directly affect in-scope components, but we did not consider threats to the out-of-scope components themselves.

Component	Description
Source Control	Source control includes the infrastructure that provides version control, hosts the Jetty codebase, facilitates the submission of pull requests and issues, and allows maintainers to release Jetty JARs and security advisories.
Client Side	Components and services on the client side initiate connections and requests.
Jetty Client (*)	A client requests data from a Jetty server or from a server built with Jetty libraries. Client-side Jetty libraries may optionally be used to handle client network connections and parsing. This component is out of scope.
Client-Side Component Libraries	Key client-side components include `ClientConnector`, `HttpClient`, and `HttpClientTransport`. The deployer or administrator can add client-side component libraries to the Jetty server to form a microservice that can both receive and initiate connections and requests.
JMX Console (*)	The JMX console is a console application (e.g., JMC, Nagios) that can connect to the JMX API to consume information regarding the server-side JVM, Jetty server, Jetty server components, and potentially also application logic. It may run remotely or on the same host as the Jetty server. This component is out of scope.
Server Side	Components on the server side receive and handle connections and requests.
Application-Specific Logic	Developer-provided business logic connects with Jetty (and clients) via the application logic base APIs. This component is out of scope.

Application Logic Base APIs	Handler APIs	APIs connect application-specific business logic to Jetty; they are an alternative to servlet APIs.
	Servlet APIs	Servlet APIs are an alternative to the Jetty handler APIs; they expose more in-depth functionality, including session management.
JMX API (*)		The `MBeanServer` platform (if included in a deployment) exposes an API to access and monitor the JVM, Jetty components, and application-specific components. Registering a bean with the JMX server creates a corresponding MBean and surfaces its status and other metadata via the API. This component is out of scope.
Server-Side Component Libraries		Server-side component libraries are used to build Jetty-based web servers. These component libraries provide server-side connection and request handling and parsing support for protocols such as HTTP/1.1, HTTP/2, HTTP/3, WebSocket, and FastCGI.
Bean		A bean is a serializable class instance at runtime, registered as part of the Jetty server's component tree. Beans added to a component tree must inherit functionality for event listening and life cycle handling. Beans in a component tree can communicate via `EventListener` APIs. Each bean in a component tree trusts its parent and any other beans with which it can communicate via `EventListener` events. A bean's parent can optionally manage its activity (start and stop it via `LifeCycle`).
Reverse Proxy (*)		The reverse proxy is a server that advertises the location or name of an application served via Jetty. The reverse proxy handles the conveyance and distribution of client requests across instances of the Jetty-served application, "fronting" the Jetty-served application so that multiple Jetty instances can handle requests directed to the same endpoint and so that no Jetty instance needs be exposed to a public network directly. The reverse proxy can also handle TLS termination on behalf of a Jetty-served application. This component is out of scope.

Trust Zones

Trust zones capture logical boundaries where controls should or could be enforced by the system, and allow developers to implement controls and policies between zones.

Zone	Description	Included Components
Public Network	The public network is the wider external-facing internet zone.	• Clients • Certificate authority

Application Network	The application network is the (private) datacenter network in which one or more clusters of Jetty server instances (or standalone Jetty servers) and additional related services reside.	• Jetty server instances
		• Reverse proxy
		• Non-Jetty services
		  ○ Logging
		  ○ Data stores
		  ○ LDAP or other identity stores
		  ○ Jetty cluster management (e.g., Kubernetes)

Private Network	The private network is an intranet or internal network that is inaccessible from the public network and has access to the application network. It is generally administrative in nature.	• Administrators
		  ○ Server administrator
		  ○ Server deployer
		• Clients
		• Remote JMX console application (JMC, Nagios, etc., potentially accessed via SSH bastion)

Localhost	The localhost is the host or container within which the JVM (running the Jetty server) runs.	• JVM • Local JMX console application
JVM	This is the local Java runtime.	• Jetty instance • JDK • Jakarta EE • Java ME (embedded deployments) • Spring Boot

Trust Zone Connections

This table describes the connections that occur between trust zones.

Originating Zone	Destination Zone	Description	Connection Types	Authentication Types
Public Network	Public Network	A client on the internet makes a network request to a public endpoint of the application served by Jetty. In this case, Jetty can also be the embedded servlet container for another framework, such as Spring Boot.	• HTTP • FastCGI • WebSocket	• Stateless; delegated to application logic • Stateful (connection based); delegated to JDK (e.g., TLS 1.2, TLS 1.3) • None
Public Network	Application Network	A client on the public network connects to a reverse proxy fronting an application served by Jetty. This reverse proxy may handle TLS termination.	• HTTP • WebSocket • FastCGI	• TLS 1.2 • TLS 1.3 • None
Application Network	Public Network	A Jetty server is configured to export logs or JMX API information to a remote service with a public endpoint (e.g., Datadog).	• HTTP • RMI	• Varies
Public Network	Application Network	The host of a Jetty server is (perhaps accidentally)	• RMI • RMI over TLS	• Username and password • None

		configured to allow public access to the JMX API port.		
Application Network	Application Network	A Jetty server instance makes a connection to an internal service (e.g., an LDAP data store or another microservice).	• LDAP • HTTP • Custom protocol (e.g., RPC)	• TLS • Application-specific request authentication • None
Application Network	Application Network	A reverse proxy forwards a request to a Jetty server instance.	• RPC • HTTP	• TLS • Application-specific request authentication • None
Private Network	Application Network	A test client connects to a hard-coded (IP or DNS) instance that is part of a cluster. All cluster instances serve the same application via Jetty.	• HTTP	• None
Private Network	Application Network	An administrator connects via SSH to the machine on which Jetty is running.	• SSH	• Username and password • Public key
Localhost	JVM	A local user makes changes to the JVM's configuration or environment or sends signals to a running JVM process.	• Filesystem • UNIX sockets • IPC signals • Java reflection	• System user authentication and access controls

Threat Actors

When conducting a threat model, we define the types of actors that could threaten the security of the system. We also define other "users" of the system who may be impacted by or induced to undertake an attack. Establishing the types of actors that use and/or could threaten the system is useful in determining which protections, if any, are necessary to mitigate or remediate vulnerabilities.

Actor	Description
External Attacker	An external attacker is an attacker on the public network (internet) from which at least one Jetty instance is accessible. This attacker can observe and analyze Jetty source commits as they land in the public repository for exploitable features.
Internal Attacker	This refers to an attacker on a private or application network from which at least one Jetty instance is accessible.
Client	"Client" refers to either a client of a Jetty server instance that can integrate the Jetty client libraries or a wholly distinct networked application.
Local Attacker	A local attacker is an attacker who controls a process or user account on the same host as the Jetty instance and can affect the system environment, including the filesystem.
Jetty Contributor	This refers to a non-maintainer Jetty contributor.
Jetty Maintainer	This refers to a core Jetty contributor. Maintainers must review and approve pull requests prior to merging them.
Application Developer	An application developer creates, maintains, and updates applications deployed via Jetty.
Server Administrator	A server administrator administers a networked application that is either built with Jetty components, served via a Jetty instance embedded as a servlet container in another framework, or served via a standalone Jetty instance.
Server Deployer	A server deployer releases an application served via Jetty or built with Jetty components into the running environment. The deployer

	may not be a separate individual from the server administrator and application developer.

Threat Scenarios

The following table describes possible threat scenarios that the system could be vulnerable to, given the design, architecture, and risk profile of Jetty.

Threat	Scenario	Actors	Components
Excessive resource consumption during parsing	Insufficient exceptional-case header or cookie parsing and exception handling in a Jetty server could allow an attacker-controlled client to cause a DoS of the Jetty server instance's other connections by sending a request containing duplicate, potentially conflicting headers; a header with an excessive number of parameters; or a header that itself contains malformed parameters crafted to pin the server to its JVM resource limits.	● Malicious client	● Jetty server ● Client
Excessive file descriptor and/or memory consumption	If a Jetty server (re)authenticates users each time a new authenticated channel opens (likely to prevent spoofing) but does not also enforce (by default) a sufficiently strict dynamic global per-user rate limit proportional to Jetty's system resource limit(s) when stateful channel-based authentication is in use, a malicious client could cause a DoS of other Jetty instance connections, especially in resource-limited or embedded use cases, by attempting to open many authenticated channels (under a mechanism such as SPNEGO).	● Malicious client	● Jetty server ● Client

Attacker-controlled application logic	The lack of served application allowlisting coupled with the lack of third-party content tracking and/or allowlisting in a Jetty server instance configured for web application "hot reloading" could allow an attacker who gains sufficient local filesystem access privileges (or who merely exploits a vulnerable servlet) to subvert that servlet or to force the Jetty server instance to serve a malicious servlet added to `$JETTY_BASE/webapps`.	• Local attacker	• Jetty server
Unsafe deserialization	The potential lack of safeguards on the deserialization of request, connection, and/or user data could allow an external attacker to exfiltrate other users' data or execute malicious code within a Jetty server process by sending a request to the Jetty server containing a payload that must be deserialized by either Jetty or the application-specific logic running on top of Jetty. The use of JPMS may reduce (but not eliminate) the impact of such an attack by reducing the accessible code in the running environment.	• Client	• Jetty server
Sensitivity to unexpected changes in the underlying implementation due to JVM or JDK "rootkits"	If a core part of the local JVM, JDK, or EE functionality called from the Jetty server is augmented or fully replaced, a local attacker could exfiltrate sensitive data from locations such as Jetty's `TrustStore` or JKS, place malicious data in the `TrustStore` or JKS, or intercept and modify sensitive data sent over (client) connections via a local user account with sufficient system privileges.	• Local attacker	• Localhost • JVM • JDK • Jakarta EE • Jetty server

Insecure default connection encryption configuration	The lack of default connection encryption (TLS) or the use of weak default cipher suites could allow a malicious intermediary with sufficient system-user permissions and access to either the client system or Jetty server instance host system to intercept and modify client (or Jetty client component) connections to the Jetty server.	• Local attacker • Remote attacker	• Jetty server • Client
Request smuggling via HTTP/2 downgrade, duplicate header allowance, or similar issues	Inconsistent header parsing and handling could allow a remote attacker to force Jetty to pass unexpected and potentially malicious additional requests to application logic or further services within the distributed system via a single crafted request. The following are examples of situations to consider mitigating where request smuggling can occur: • Improper HTTP/2-to-HTTP/1.1 downgrade header handling • Improper handling of duplicate headers in the same request (e.g., `Content-Length`) • Allowing for conflicting headers' presence in the same request (e.g., a short `Content-Length` value along with `Transfer-Encoding: chunked`)	• Remote attacker	• Jetty server

HTTP or header parsing mismatch between Jetty and Spring Boot, or similar frameworks	Potential discrepancies between protocol, header, or cookie parsing done by Spring Boot (or a similar Java framework) and by Jetty itself could allow a remote attacker to smuggle unexpected requests into the served web application when Jetty runs as the embedded servlet container within another Java framework such as Spring Boot.	• Remote attacker	• Jetty • Spring Boot
Request smuggling due to discrepancies between parsing done by other servers (e.g., a reverse proxy) and Jetty	If a Jetty instance is run in a particular compliance mode, but it is fronted by a reverse proxy whose HTTP or header parsing capabilities are not fully consistent with Jetty configured with the compliance mode in question, a remote attacker could conduct request smuggling.	• Remote attacker	• Jetty • Reverse proxy
Access to or modification of temporary data	An attacker with filesystem access to the Jetty temporary directory or an application-specific temporary directory could read sensitive data mistakenly stored there or modify files that will later be read back into the application.	• Local attacker	• Localhost • Jetty server
Security through obscurity	A remote attacker monitoring pull requests and commits to the Jetty repository could infer the presence of a vulnerability from static analysis over changes made to the codebase (or in-progress pull requests) to fix a security issue prior to its official announcement. The attacker could exploit vulnerabilities identified in this way before updates are released.	• Jetty contributor • Remote attacker	• Source control

Administrator misconfiguration of the underlying system	A misconfigured JVM that exposes the JMX API on a publicly accessible port could allow an external attacker to exfiltrate sensitive Jetty/system information or to modify the running Jetty instance or JVM (e.g., shut down the running Jetty instance—denying service to other users—or shrink resource allocations to starve legitimate connections) by connecting a JMX console application to the port.	• External attacker • Server administrator	• Jetty server • JMX API • Remote JMX console

Recommendations

- Jetty should check for a minimal set of safe(r) default security configuration practices during the server startup process.
    - Prefer the strictest default configuration overall that common Jetty use cases (such as deployment with Spring Boot and/or as part of a distributed system) can accommodate.
    - Log (likely to the user-configured Jetty error log location at the `INFO` level) brief information about any unsafe security practices in use. Consider also including links to documentation on mitigating such unsafe practices.
    - Document the safe server configurations for each of the most common types of Jetty deployments and indicate the types of attacks that such configurations will prevent. For example, configuring a Jetty server with a stricter header parsing compliance mode may decrease the likelihood of exploits of header parser differentials, such as request smuggling.
    - A Jetty instance that sources web apps from (or allows delegated web app usage from) any other system or symlinked location should log a message directing users to install web apps solely in `${jetty.base}/webapps`.
        - Also consider logging a warning if the `${jetty.base}` (or `${jetty.base}` subdirectory) access permissions are overbroad (i.e., allow read or write access from users other than the account that Jetty runs under).
    - When run with a default configuration, a Jetty instance should fail to start without a configured `TrustStore`, `JKS`, and `ssl` module.
        - The server administrator or deployer should have to purposefully set a configuration option (whose name contains the word "unsafe") to "true" or a similar setting to allow cleartext connections.
        - Throw an exception with a sufficiently explanatory name and message pointing to documentation on how to configure `TrustStore`, `JKS`, and the `ssl` module and on how to alternatively allow unsafe/cleartext connections.
    - By default, a Jetty instance should not allow `X-Forwarded-*` (e.g., `X-Forwarded-For`) headers since their directives' interpretations vary between servers, and such headers are frequently spoofed.

        - Jetty instances should use `setForwardedOnly()` by default so that Jetty administrators must explicitly configure the allowance of `X-Forwarded-*` headers; this should be documented in the programming and operations guides.

- Ensure that frameworks that can embed Jetty, such as Spring Boot, recommend and use the most up-to-date Jetty release version so that "second-degree" Jetty users can also benefit from security-related fixes.

- Check that all implementations and uses of the `Serializable` interface in Jetty both properly sanitize input prior to deserialization operations and override the `ObjectInputStream#resolveClass()` method to prevent arbitrary class deserialization in all Jetty modes of operation.

- Ensure that Jetty's default functionality for parsing headers, cookies, and request bodies received over HTTP/1.1, HTTP/2, and WebSocket is consistent with Spring Boot's functionality, as a common use case for Jetty is as the servlet container embedded within a Spring Boot deployment.

    - When Jetty is configured as the Spring Boot servlet container, prevent users from applying parsing functionality in Jetty that is not consistent with that of Spring Boot (which could result in unexpected/exploitable server-layer behavior inconsistencies).

    - If Spring Boot's default parsing behavior differs substantially from Jetty's preferred set of secure defaults, implement a Jetty "Spring Boot compliance mode" and make it the default for users configuring Jetty as a Spring Boot servlet container.

- Consider providing a default Jetty SBOM that Jetty deployers and administrators can add to as needed, and consider signing Jetty artifacts for later verification. Refer to the following resources for more information:

    - GitHub Actions: SBOM generation and usage documentation

    - GitLab: Ultimate guide to SBOMs

    - Project Sigstore, a Linux Foundation project (that Trail of Bits participates in), which maintains tooling for signing software artifacts and Git commits, as well as verification tooling that Maven Central endorses as an upcoming integration alternative to PGP

        - Sigstore blog post on using Sigstore in Java environments

        - Sigstore Maven plugin

- When remediating a CVE or other security vulnerability, do not rely on purposefully generic commit messages or vague PR discussions to try to hide code differences that patch an exploit, as they will still be findable via tools such as static analyzers and runtime data flow taint analyzers.

- Consider crawling the links between Eclipse Jetty documentation sections to ensure they are still valid. Some links to specific sections of the documentation simply redirect to the Eclipse homepage or point to unavailable prior web locations for the documentation.

- Finish the following security-related sections in the programming guide that are incomplete and marked as "TODO." Once complete, these sections will help ensure that users can set up secure Jetty instances:

    - The "Securing HTTP Server Applications" section

        - Even if it includes only simple recommendations for common web application security issues, this section could be a valuable resource for developers writing applications served via Jetty or incorporating Jetty components.

        - Use OWASP Top 10 and CWE Top 25 as a basis for the recommendations included in this section, or direct users to the CWE list and the 2017 and 2020 OWASP Top 10 lists for further reference.

        - Additionally, consider pointing users to Java-specific CWEs that capture the reason(s) for each recommended configuration setting or programming practice.

    - The "HttpClient TLS TrustStore Configuration" section

    - The "HttpClient TLS Client Certificates Configuration" section

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Tool	Description
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time
CodeQL	A code analysis engine developed by GitHub to automate security checks
CI Fuzz	A fuzzing engine used to create fuzz tests for Java applications

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase contains several arithmetic-related issues that create vulnerabilities, including the risk of an integer overflow (TOB-JETTY-1), the use of incorrect integer types (TOB-JETTY-6), and missing checks for negative input values (TOB-JETTY-7, TOB-JETTY-10).	Moderate
Auditing	The default logging level produces logs of basic system life cycle events, including server startup and application deployment events, and the debug logs provide greater detail.	Satisfactory
Authentication / Access Controls	We identified no bugs or vulnerabilities in Jetty's implementations of authentication protocols.	Strong
Complexity Management	The codebase contains a significant amount of indirection and multiple layers of abstraction, but these design choices are a reasonable way to enable code reuse and interoperation between disparate system components.	Satisfactory
Configuration	The Java XML parser is not configured to disable document type definitions when parsing Maven package metadata (TOB-JETTY-19). Additionally, the code permits some unsafe filesystem operations without checking for symbolic links (TOB-JETTY-13).	Moderate
Cryptography and Key Management	Jetty's lack of support for JDKs earlier than version 17 helps support good TLS configuration practices. However, the QUIC implementation writes the SSL certificate's private key to the filesystem in a temporary plaintext file while passing it through to the underlying quiche library (TOB-JETTY-21).	Moderate

Data Handling	There are multiple issues related to data parsing (TOB-JETTY-2) and quoting (TOB-JETTY-3, TOB-JETTY-5); the issue described in finding 3 could enable arbitrary command execution in legacy systems.	**Moderate**
Documentation	Available documentation provides thorough coverage of common use cases for system administrators and programmers, as well as available configuration options.	**Strong**
Low-Level Manipulation	The low-level packet parsing and memory buffer management routines contain bugs that result in exceptions when parsing malformed traffic (TOB-JETTY-15) and possibly DoS due to excessive resource consumption (TOB-JETTY-8).	**Moderate**
Maintenance	Some of Jetty's test cases have not been updated to match recent changes to Jetty Core (see the "Testing and Verification" section below). There are also some instances of code duplication (TOB-JETTY-22).	**Satisfactory**
Memory Safety and Error Handling	Some classes allocate buffers of excessive and incorrect sizes (TOB-JETTY-8, TOB-JETTY-11), and the HTTP/2 server fails to appropriately detect and handle errors as required by RFC 9113 (TOB-JETTY-18).	**Moderate**
Testing and Verification	Overall, tests appear to achieve reasonable coverage of major system components. However, some tests are outdated and have not been updated to account for recent changes to class interfaces. Additionally, some tests validate basic system functionality but do not cover error conditions that must be handled in ways specified by applicable standards.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Risk of integer overflow that could allow HpackDecoder to exceed maxHeaderSize	Denial of Service	**Medium**
2	Cookie parser accepts unmatched quotation marks	Error Reporting	**Informational**
3	Errant command quoting in CGI servlet	Data Validation	**High**
4	Symlink-allowed alias checker ignores protected targets list	Access Controls	**High**
5	Missing check for malformed Unicode escape sequences in QuotedStringTokenizer.unquote	Data Validation	**Low**
6	WebSocket frame length represented with 32-bit integer	Data Validation	**High**
7	WebSocket parser does not check for negative payload lengths	Data Validation	**Low**
8	WebSocket parser greedily allocates ByteBuffers for large frames	Denial of Service	**Medium**
9	Risk of integer overflow in HPACK's NBitInteger.decode	Data Validation	**Informational**
10	MetaDataBuilder.checkSize accepts headers of negative lengths	Denial of Service	**Medium**
11	Insufficient space allocated when encoding QPACK instructions and entries	Denial of Service	**Low**

12	LiteralNameEntryInstruction incorrectly encodes value length	Denial of Service	**Medium**
13	FileInitializer does not check for symlinks	Data Validation	**High**
14	FileInitializer permits downloading files via plaintext HTTP	Data Exposure	**High**
15	NullPointerException thrown by FastCGI parser on invalid frame type	Data Validation	**Medium**
16	Documentation does not specify that request contents and other user data can be exposed in debug logs	Data Exposure	**Medium**
17	HttpStreamOverFCGI internally marks all requests as plaintext HTTP	Data Validation	**High**
18	Excessively permissive and non-standards-compliant error handling in HTTP/2 implementation	Data Validation	**Low**
19	XML external entities and entity expansion in Maven package metadata parser	Data Validation	**High**
20	Use of deprecated AccessController class	Code Quality	**Informational**
21	QUIC server writes SSL private key to temporary plaintext file	Cryptography	**High**
22	Repeated code between HPACK and QPACK	Code Quality	**Informational**
23	Various exceptions in HpackDecoder.decode and QpackDecoder.decode	Denial of Service	**Informational**
24	Incorrect QPACK encoding when multi-byte characters are used	Data Validation	**Medium**

25	No limits on maximum capacity in QPACK decoder	Denial of Service	**High**

Detailed Findings

1. Risk of integer overflow that could allow HpackDecoder to exceed maxHeaderSize	
Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-JETTY-1
Target: `org.eclipse.jetty.http2.hpack.internal.MetaDataBuilder`, `org.eclipse.jetty.http2.hpack.HpackDecoder`	

Description

An integer overflow could occur in the `MetaDataBuilder.checkSize` function, which would allow HPACK header values to exceed their size limit.

`MetaDataBuilder.checkSize` determines whether a header name or value exceeds the size limit and throws an exception if the limit is exceeded:

```
291    public void checkSize(int length, boolean huffman) throws SessionException
292    {
293       // Apply a huffman fudge factor
294       if (huffman)
295           length = (length * 4) / 3;
296       if ((_size + length) > _maxSize)
297           throw new HpackException.SessionException("Header too large %d > %d",
_size + length, _maxSize);
298    }
```

Figure 1.1: `MetaDataBuilder.checkSize`

However, when the value of `length` is very large and `huffman` is `true`, the multiplication of `length` by 4 in line 295 will overflow, and `length` will become negative. This will cause the result of the sum of `_size` and `length` to be negative, and the check on line 296 will not be triggered.

Exploit Scenario

An attacker repeatedly sends HTTP messages with the HPACK header `0x00ffffffffff02`. Each time this header is decoded, the following occurs:

- `HpackDecode.decode` determines that a Huffman-coded value of length `805306494` needs to be decoded.

- `MetaDataBuilder.checkSize` approves this length.
- `Huffman.decode` allocates a 1.6 GB string array.
- `Huffman.decode` experiences a buffer overflow error, and the array is deallocated the next time garbage collection happens. (Note that this deallocation can be delayed by appending valid Huffman-coded characters to the end of the header.)

Depending on the timing of garbage collection, the number of threads, and the amount of memory available on the server, this may cause the server to run out of memory.

Recommendations

Short term, have `MetaDataBuilder.checkSize` check that `length` is below a threshold before performing the multiplication.

Long term, use fuzzing to check for similar errors; we found this issue by fuzzing `HpackDecode`.

2. Cookie parser accepts unmatched quotation marks	
Severity: **Informational**	Difficulty: **High**
Type: Error Reporting	Finding ID: TOB-JETTY-2
Target: `org.eclipse.jetty.http.RFC6265CookieParser`	

Description

The `RFC6265CookieParser.parseField` function does not check for unmatched quotation marks. For example, `parseField("\"")` will execute without raising an exception. This issue is unlikely to lead to any vulnerabilities, but it could lead to problems if users or developers expect the function to accept only valid strings.

Recommendations

Short term, modify the function to check that the state at the end of the given string is not `IN_QUOTED_VALUE`.

Long term, when using a state machine, ensure that the code always checks that the state is valid before exiting.

3. Errant command quoting in CGI servlet

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-3
Target: `org.eclipse.jetty.ee10.servlets.CGI`, `org.eclipse.jetty.ee9.servlets.CGI`	

Description

If a user sends a request to a CGI servlet for a binary with a space in its name, the servlet will escape the command by wrapping it in quotation marks. This wrapped command, plus an optional command prefix, will then be executed through a call to `Runtime.exec`. If the original binary name provided by the user contains a quotation mark followed by a space, the resulting command line will contain multiple tokens instead of one. For example, if a request references a binary called `file" name "here`, the escaping algorithm will generate the command line string `"file" name "here"`, which will invoke the binary named `file`, not the one that the user requested.

```
if (execCmd.length() > 0 && execCmd.charAt(0) != '"' && execCmd.contains(" "))
    execCmd = "\"" + execCmd + "\"";
```

Figure 3.1: *`CGI.java#L337–L338`*

Exploit Scenario

The `cgi-bin` directory contains a binary named `exec` and a subdirectory named `exec" commands`, which contains a file called `bin1`. A user sends to the CGI servlet a request for the filename `exec" commands/bin1`. This request passes the file existence check on lines 194 through 205 in `CGI.java`. The servlet adds quotation marks around this filename, resulting in the command line string `"exec" commands/bin1"`. When this string is passed to `Runtime.exec`, instead of executing the `bin1` binary, the server executes the `exec` binary with the argument `commands/bin1"`.

This behavior is incorrect and could bypass alias checks; it could also cause other unintended behaviors if a command prefix is configured. Additionally, if the `useFullPath` configuration setting is off, the command would not need to pass the existence check. Without this setting, an attacker exploiting this issue would not have to rely on a binary and subdirectory with similar names, and the attack could succeed on a much wider variety of directory structures.

Recommendations

Short term, update line 346 in `CGI.java` to replace the call to `exec(String command, String[] env, File dir)` with a call to `exec(String[] cmdarray, String[] env, File dir)` so that the quotation mark escaping algorithm does not create new tokens in the command line string.

Long term, update the quotation mark escaping algorithm so that any unescaped quotation marks in the original name of the command are properly escaped, resulting in one double-quoted token instead of multiple adjacent quoted strings. Additionally, the expression `execCmd.charAt(0) != '"'` on line 337 of `CGI.java` is intended to avoid adding additional quotation marks to an already-quoted command string. If this check is unnecessary, it should be removed. If it is necessary, it should be replaced by a more robust check that accurately detects properly formatted double-quoted strings.

4. Symlink-allowed alias checker ignores protected targets list	
Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-JETTY-4
Target: `org.eclipse.jetty.server.SymlinkAllowedResourceAliasChecker`	

Description

The class `SymlinkAllowedResourceAliasChecker` is an alias checker that permits users to access a symlink as long as the symlink is stored within an allowed directory. The following comment appears on line 76 of this class:

```
// TODO: return !getContextHandler().isProtectedTarget(realURI.toString());
```

Figure 4.1: `SymlinkAllowedResourceAliasChecker.java#L76`

As this comment suggests, the alias checker does not yet enforce the context handler's protected resource list. That is, if a symlink is contained in an allowed directory but points to a target on the protected resource list, the alias checker will return a positive match.

During our review, we found that some other modules, but not all, independently enforce the protected resource list and will decline to serve resources on the list even if the alias checker returns a positive result. But the modules that do not independently enforce the protected resource list could serve protected resources to attackers conducting symlink attacks.

Exploit Scenario

An attacker induces the creation of a symlink (or a system administrator accidentally creates one) in a web-accessible directory that points to a protected resource (e.g., a child of `WEB-INF`). By requesting this symlink through a servlet that uses the `SymlinkAllowedResourceAliasChecker` class, the attacker bypasses the protected resource list and accesses the sensitive files.

Recommendations

Short term, implement the check referenced in the comment so that the alias checker rejects symlinks that point to a protected resource or a child of a protected resource.

Long term, consider clarifying and documenting the responsibilities of different components for enforcing protected resource lists. Consider implementing redundant checks in multiple modules for purposes of layered security.

5. Missing check for malformed Unicode escape sequences in QuotedStringTokenizer.unquote	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-5
Target: `org.eclipse.jetty.util.QuotedStringTokenizer`	

Description

The `QuotedStringTokenizer` class's `unquote` method parses `\u####` Unicode escape sequences, but it does not first check that the escape sequence is properly formatted or that the string is of a sufficient length:

```
case 'u':
    b.append((char)(
            (TypeUtil.convertHexDigit((byte)s.charAt(i++)) << 24) +
                (TypeUtil.convertHexDigit((byte)s.charAt(i++)) << 16) +
                (TypeUtil.convertHexDigit((byte)s.charAt(i++)) << 8) +
                (TypeUtil.convertHexDigit((byte)s.charAt(i++)))
        )
    );
    break;
```

Figure 5.1: *`QuotedStringTokenizer.java#L547–L555`*

Any calls to this function with an argument ending in an incomplete Unicode escape sequence, such as "`str\u0`", will cause the code to throw a `java.lang.NumberFormatException` exception. The only known execution path that will cause this method to be called with a parameter ending in an invalid Unicode escape sequence is to induce the processing of an ETag `Matches` header by the `ResourceService` class, which calls `EtagUtils.matches`, which calls `QuotedStringTokenizer.unquote`.

Exploit Scenario

An attacker introduces a maliciously crafted ETag into a browser's cache. Each subsequent request for the affected resource causes a server-side exception, preventing the server from producing a valid response so long as the cached ETag remains in place.

Recommendations

Short term, add a `try-catch` block around the affected code that drops malformed escape sequences.

Long term, implement a suitable workaround for lenient mode that passes the raw bytes of the malformed escape sequence into the output.

6. WebSocket frame length represented with 32-bit integer	
Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-JETTY-6
Target: `org.eclipse.jetty.websocket.core.internal.Parser`	

Description

The WebSocket standard (RFC 6455) allows for frames with a size of up to 2^{64} bytes. However, the WebSocket parser represents the frame length with a 32-bit integer:

```
private int payloadLength;
// ...[snip]...
case PAYLOAD_LEN_BYTES:
    {
        byte b = buffer.get();
        --cursor;
        payloadLength |= (b & 0xFF) << (8 * cursor);
        // ...[snip]...
    }
```

Figure 6.1: `Parser.java`, lines 57 and 147–151

As a result, this parsing algorithm will incorrectly parse some length fields as negative integers, causing a `java.lang.IllegalArgumentException` exception to be thrown when the parser tries to set the limit of a `Buffer` object to a negative number (refer to TOB-JETTY-7). Consequently, Jetty's WebSocket implementation cannot properly process frames with certain lengths that are compliant with RFC 6455.

Even if no exception results, this logic error will cause the parser to incorrectly identify the sizes of WebSocket frames and the boundaries between them. If the server passes these frames to another WebSocket connection, this bug could enable attacks similar to HTTP request smuggling, resulting in bypasses of security controls.

Exploit Scenario

A Jetty WebSocket server is deployed in a reverse proxy configuration in which both Jetty and another web server parse the same stream of WebSocket frames. An attacker sends a frame with a length that the Jetty parser incorrectly truncates to a 32-bit integer. Jetty and the other server interpret the frames differently, which causes errors in the implementation of security controls, such as WAF filters.

Recommendations

Short term, change the `payloadLength` variable to use the `long` data type instead of an `int`.

Long term, audit all arithmetic operations performed on this `payloadLength` variable to ensure that it is always used as an unsigned integer instead of a signed one. The standard library's `Integer` class can provide this functionality.

7. WebSocket parser does not check for negative payload lengths

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-JETTY-7
Target: `org.eclipse.jetty.websocket.core.internal.Parser`	

Description

The WebSocket parser's `checkFrameSize` method checks for payload lengths that exceed the current configuration's maximum, but it does not check for payload lengths that are lower than zero. If the payload length is lower than zero, the code will throw an exception when the payload length is passed to a call to `buffer.limit`.

Exploit Scenario

An attacker sends a WebSocket payload with a length field that parses to a negative signed integer (refer to TOB-JETTY-6). This payload causes an exception to be thrown and possibly the server process to crash.

Recommendations

Short term, update `checkFrameSize` to throw an `org.eclipse.jetty.websocket.core.exception.ProtocolException` exception if the frame's length field is less than zero.

8. WebSocket parser greedily allocates ByteBuffers for large frames

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-JETTY-8
Target: `org.eclipse.jetty.websocket.core.internal.Parser`	

Description

When the WebSocket parser receives a partial frame in a `ByteBuffer` object and auto-fragmenting is disabled, the parser allocates a buffer of a size sufficient to store the entire frame at once:

```
if (aggregate == null)
    {
        if (available < payloadLength)
        {
            // not enough to complete this frame
            // Can we auto-fragment
            if (configuration.isAutoFragment() && isDataFrame)
                return autoFragment(buffer, available);


            // No space in the buffer, so we have to copy the partial payload
            aggregate = bufferPool.acquire(payloadLength, false);
            BufferUtil.append(aggregate.getByteBuffer(), buffer);
            return null;
            }
    //...[snip]...
}
```

Figure 8.1: `Parser.java`, lines 323–336

An attacker could send a WebSocket frame with a large payload length field, causing the server to allocate a buffer of a size equal to the specified payload length, without ever sending the entire frame contents. Therefore, an attacker can induce the consumption of gigabytes (or potentially exabytes; refer to TOB-JETTY-6) of memory by sending only hundreds or thousands of bytes over the wire.

Exploit Scenario

An attacker crafts a malicious WebSocket frame with a large payload length field but incomplete payload contents. The server then allocates a buffer of a size equal to the payload length field, causing an excessive consumption of RAM. To ensure that the connection is not promptly dropped, the attacker continues sending parts of this payload a few seconds apart, conducting a slow HTTP attack.

Recommendations

Short term, ensure that the default maximum payload size remains at a low value that is sufficient for most purposes (such as the current default of 64 KB).

Long term, to better support large WebSocket frames, update the use of `ByteBuffer` objects in the WebSocket parser so that the parser does not allocate the entire buffer as soon as it parses the first fragment. Instead, the buffer should be expanded in relatively small increments (e.g., 10 MB or 100 MB at a time) and then written to only once the data sent by the client exceeds the length of the current buffer. That way, in order to induce the consumption of a large amount of RAM, an attacker would need to send a commensurate number of bytes over the wire.

9. Risk of integer overflow in HPACK's NBitInteger.decode

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-9
Target: `org.eclipse.jetty.http2.hpack.internal.NBitInteger`	

Description

The static function `NBitInteger.decode` is used to decode bytestrings in HPACK's integer format. It should return only positive integers since HPACK's integer format is not intended to support negative numbers. However, the following loop in `NBitInteger.decode` is susceptible to integer overflows in its multiplication and addition operations:

```
public static int decode(ByteBuffer buffer, int n)
{
    if (n == 8)
    {
      // ...
    }

    int nbits = 0xFF >>> (8 - n);

    int i = buffer.get(buffer.position() - 1) & nbits;

    if (i == nbits)
    {
        int m = 1;
        int b;
        do
        {
            b = 0xff & buffer.get();
            i = i + (b & 127) * m;
            m = m * 128;
        }
        while ((b & 128) == 128);
    }
    return i;
}
```

Figure 9.1: `NBitInteger.java`, lines 105–145

For example, `NBitInteger.decode(0xFF8080FFFF0F, 7)` returns `-16257`.

Any overflow that occurs in the function would not be a problem on its own since, in general, the output of this function ought to be validated before it is used; however, when coupled with other issues (refer to TOB-JETTY-10), an overflow can cause vulnerabilities.

Recommendations

Short term, modify `NBitInteger.decode` to check that its result is nonnegative before returning it.

Long term, consider merging the QPACK and HPACK implementations for `NBitInteger`, since they perform the same functionality; the QPACK implementation of `NBitInteger` checks for overflows.

10. MetaDataBuilder.checkSize accepts headers of negative lengths

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-JETTY-10
Target: `org.eclipse.jetty.http2.hpack.internal.MetaDataBuilder`	

Description

The `MetaDataBuilder.checkSize` function accepts user-entered HPACK header values of negative sizes, which could cause a very large buffer to be allocated later when the user-entered size is multiplied by 2.

`MetaDataBuilder.checkSize` determines whether a header name or value exceeds the size limit and throws an exception if the limit is exceeded:

```
public void checkSize(int length, boolean huffman) throws SessionException
{
    // Apply a huffman fudge factor
    if (huffman)
        length = (length * 4) / 3;
    if ((_size + length) > _maxSize)
        throw new HpackException.SessionException("Header too large %d > %d", _size
+ length, _maxSize);
}
```

Figure 10.1: `MetaDataBuilder.java`, lines 291–298

However, it does not throw an exception if the size is negative.

Later, the `Huffman.decode` function multiplies the user-entered length by 2 before allocating a buffer:

```
public static String decode(ByteBuffer buffer, int length) throws
HpackException.CompressionException
{
    Utf8StringBuilder utf8 = new Utf8StringBuilder(length * 2);
// ...
```

Figure 10.2: `Huffman.java`, lines 357–359

This means that if a user provides a negative length value (or, more precisely, a length value that becomes negative when multiplied by the 4/3 fudge factor), and this length value becomes a very large positive number when multiplied by 2, then the user can cause a very large buffer to be allocated on the server.

Exploit Scenario

An attacker repeatedly sends HTTP messages with the HPACK header `0x00ff8080ffff0b`. Each time this header is decoded, the following occurs:

- `HpackDecode.decode` determines that a Huffman-coded value of length `-1073758081` needs to be decoded.
- `MetaDataBuilder.checkSize` approves this length.
- The number is multiplied by 2, resulting in `2147451134`, and `Huffman.decode` allocates a 2.1 GB string array.
- `Huffman.decode` experiences a buffer overflow error, and the array is deallocated the next time garbage collection happens. (Note that this deallocation can be delayed by adding valid Huffman-coded characters to the end of the header.)

Depending on the timing of garbage collection, the number of threads, and the amount of memory available on the server, this may cause the server to run out of memory.

Recommendations

Short term, have `MetaDataBuilder.checkSize` check that the given length is positive directly before adding it to `_size` and comparing it with `_maxSize`.

Long term, add checks for integer overflows in `Huffman.decode` and in `NBitInteger.decode` (refer to TOB-JETTY-9) for added redundancy.

11. Insufficient space allocated when encoding QPACK instructions and entries

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-JETTY-11

Target:

- `org.eclipse.jetty.http3.qpack.internal.instruction.IndexedNameEntryInstruction`
- `org.eclipse.jetty.http3.qpack.internal.instruction.LiteralNameEntryInstruction`
- `org.eclipse.jetty.http3.qpack.internal.instruction.EncodableEntry`

Description

Multiple expressions do not allocate enough buffer space when encoding QPACK instructions and entries, which could result in a buffer overflow exception.

In `IndexedNameEntry`, the following expression determines how much space to allocate when encoding the instruction:

```
int size = NBitIntegerEncoder.octetsNeeded(6, _index) + (_huffman ?
HuffmanEncoder.octetsNeeded(_value) : _value.length()) + 2;
```

Figure 11.1: `IndexedNameEntry.java`, line 58

Later, the following two lines encode the value size for Huffman-coded and non-Huffman-coded strings, respectively:

```
NBitIntegerEncoder.encode(byteBuffer, 7, HuffmanEncoder.octetsNeeded(_value));
// ...
NBitIntegerEncoder.encode(byteBuffer, 7, _value.length());
```

Figure 11.2: `IndexedNameEntry.java`, lines 71 and 77

These encodings can take up more than 1 byte if the value's length is over 126 because the number will fill up the 7 bits given to it in the first byte. However, the `int size` expression in figure 11.1 assumes that it will take up only 1 byte. Thus, if the value's length is over 126, too few bytes may be allocated for the instruction, causing a buffer overflow.

The same problem occurs in `LiteralNameEntryInstruction`:

```
int size = (_huffmanName ? HuffmanEncoder.octetsNeeded(_name) : _name.length()) +
    (_huffmanValue ? HuffmanEncoder.octetsNeeded(_value) : _value.length()) + 2;
```

Figure 11.3: `LiteralNameEntryInstruction.java`, lines 59–60

This expression assumes that the name's length will fit into 5 bits and that the value's length will fit into 7 bits. If the name's length is over 30 bytes or the value's length is over 126 bytes, these assumptions will be false and too little space may be allocated for the instruction, which could cause a buffer overflow.

A similar problem occurs in `EncodableEntry.ReferencedNameEntry`. The `getRequiredSize` method in this file calculates how much space should be allocated for its encoding:

```
public int getRequiredSize(int base)
{
    String value = getValue();
    int relativeIndex =  _nameEntry.getIndex() - base;
    int valueLength = _huffman ? HuffmanEncoder.octetsNeeded(value) :
value.length();
    return 1 + NBitIntegerEncoder.octetsNeeded(4, relativeIndex) + 1 +
NBitIntegerEncoder.octetsNeeded(7, valueLength) + valueLength;
}
```

Figure 11.4: `EncodableEntry.java`, lines 181–187

The method returns the wrong size if the value is longer than 126 bytes. Additionally, it assumes that the name will use a post-base reference rather than a normal one, which may be incorrect.

An additional problem is present in this method. It assumes that `value`'s length in bytes will be returned by `value.length()`. However, `value.length()` measures the number of *characters* in `value`, not the number of bytes, so if `value` contains multibyte characters (e.g., UTF-8), too few bytes will be allocated. The length of `value` should be calculated by using `value.getBytes()` instead of `value.length()`.

The `getRequiredSize` method in `EncodableEntry.LiteralEntry` also incorrectly uses `value.length()`:

```
public int getRequiredSize(int base)
{
    String name = getName();
    String value = getValue();
    int nameLength = _huffman ? HuffmanEncoder.octetsNeeded(name) : name.length();
    int valueLength = _huffman ? HuffmanEncoder.octetsNeeded(value) :
value.length();
    return 2 + NBitIntegerEncoder.octetsNeeded(3, nameLength) + nameLength +
NBitIntegerEncoder.octetsNeeded(7, valueLength) + valueLength;
```

```
}
```

Figure 11.5: `EncodableEntry.java`, lines 243–250

Note that `name.length()` is used to measure the byte length of `name`, and `value.length()` is used to measure the byte length of `value`.

Jetty's HTTP/3 code is still considered experimental, so this issue should not affect production code, but it should be fixed before announcing HTTP/3 support to be production-ready.

Recommendations

Short term, change the relevant expressions to account for the extra length.

Long term, build out additional test cases for QPACK and other parsers used in HTTP/3 to test for the correct handling of edge cases and identify memory handling exceptions.

12. LiteralNameEntryInstruction incorrectly encodes value length

Severity: **Medium**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-JETTY-12

Target: `org.eclipse.jetty.http3.qpack.internal.instruction.LiteralNameEntryInstruction`

Description

QPACK instructions for inserting entries with literal names and non-Huffman-coded values will be encoded incorrectly when the value's length is over 30, which could cause values to be sent incorrectly or errors to occur during decoding.

The following snippet of the `LiteralNameEntryInstruction.encode` function is responsible for encoding the header value:

```
78    if (_huffmanValue)
79    {
80        byteBuffer.put((byte)(0x80));
81        NBitIntegerEncoder.encode(byteBuffer, 7,
HuffmanEncoder.octetsNeeded(_value));
82        HuffmanEncoder.encode(byteBuffer, _value);
83    }
84    else
85    {
86        byteBuffer.put((byte)(0x00));
87        NBitIntegerEncoder.encode(byteBuffer, 5, _value.length());
88        byteBuffer.put(_value.getBytes());
89    }
```

Figure 12.1: `LiteralNameEntryInstruction.java`, lines 78–89

On line 87, 5 is the second parameter to `NBitIntegerEncoder.encode`, indicating that the number will take up 5 bits in the first encoded byte; however, the second parameter should be 7 instead. This means that when `_value.length()` is over 30, it will be incorrectly encoded.

Jetty's HTTP/3 code is still considered experimental, so this issue should not affect production code, but it should be fixed before announcing HTTP/3 support to be production-ready.

Recommendations

Short term, change the second parameter of the `NBitIntegerEncoder.encode` function from 5 to 7 in order to reflect that the number will take up 7 bits.

Long term, write more tests to catch similar encoding/decoding problems.

13. FileInitializer does not check for symlinks

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-13
Target: `org.eclipse.jetty.start.FileInitializer`	

Description

Module configuration files can direct Jetty to download a remote file and save it in the local filesystem while initializing the module. During this process, the `FileInitializer` class validates the destination path and throws an `IOException` exception if the destination is outside the `${jetty.base}` directory. However, this validation routine does not check for symlinks:

```
// now on copy/download paths (be safe above all else)
if (destination != null && !destination.startsWith(_basehome.getBasePath()))
    throw new IOException("For security reasons, Jetty start is unable to process
file resource not in ${jetty.base} - " + location);
```

Figure 13.1: `FileInitializer.java`, lines 112–114

None of the subclasses of `FileInitializer` check for symlinks either. Thus, if the `${jetty.base}` directory contains a symlink, a file path in a module's .ini file beginning with the symlink name will pass the validation check, and the file will be written to a subdirectory of the symlink's destination.

Exploit Scenario

A system's `${jetty.base}` directory contains a symlink called `dir`, which points to `/etc`. The system administrator enables a Jetty module whose .ini file contains a `[files]` entry that downloads a remote file and writes it to the relative path `dir/config.conf`. The filesystem follows the symlink and writes a new configuration file to `/etc/config.conf`, which impacts the server's system configuration. Additionally, since the `FileInitializer` class uses the `REPLACE_EXISTING` flag, this behavior overwrites an existing system configuration file.

Recommendations

Short term, rewrite all path checks in `FileInitializer` and its subclasses to include a call to the `Path.toRealPath` function, which, by default, will resolve symlinks and produce the real filesystem path pointed to by the `Path` object. If this real path is outside `${jetty.base}`, the file write operation should fail.

Long term, consolidate all filesystem operations involving the `${jetty.base}` or `${jetty.home}` directories into a single centralized class that automatically performs symlink resolution and rejects operations that attempt to read from or write to an unauthorized directory. This class should catch and handle the `IOException` exception that is thrown in the event of a link loop or a large number of nested symlinks.

14. FileInitializer permits downloading files via plaintext HTTP

Severity: **High**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-JETTY-14
Target: `org.eclipse.jetty.start.FileInitializer`	

Description

Module configuration files can direct Jetty to download a remote file and save it in the local filesystem while initializing the module. If the specified URL is a plaintext HTTP URL, Jetty does not raise an error or warn the user. Transmitting files over plaintext HTTP is intrinsically unsecure and exposes sensitive data to tampering and eavesdropping in transit.

Exploit Scenario

A system administrator enables a Jetty module that downloads a remote file over plaintext HTTP during initialization. An attacker with a network intermediary position sniffs the traffic and infers sensitive information about the design and configuration of the Jetty system under configuration. Alternatively, the attacker actively tampers with the file during transmission from the remote server to the Jetty installation, which enables the attacker to alter the module's behavior and launch other attacks against the targeted system.

Recommendations

Short term, add a check to the `FileInitializer` class and its subclasses to prohibit downloads over plaintext HTTP. Additionally, add a validation check to the module .ini file parser to reject any configuration that includes a plaintext HTTP URL in the `[files]` section.

Long term, consolidate all remote file downloads conducted during module configuration operations into a single centralized class that automatically rejects plaintext HTTP URLs.

If current use cases require support of plaintext HTTP URLs, then at a minimum, have Jetty display a prominent warning message and prompt the user for manual confirmation before performing the unencrypted download.

15. NullPointerException thrown by FastCGI parser on invalid frame type

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-JETTY-15
Target: `org.eclipse.jetty.fcgi.parser.Parser`	

Description

Because of a missing `null` check, the Jetty FastCGI client's `Parser` class throws a `NullPointerException` exception when parsing a frame with an invalid frame type field. This exception occurs because the `findContentParser` function returns `null` when it does not have a `ContentParser` object matching the specified frame type, and the caller never checks the `findContentParser` return value for `null` before dereferencing it.

```
case CONTENT:
{
    ContentParser contentParser = findContentParser(headerParser.getFrameType());
    if (headerParser.getContentLength() == 0)
    {
        padding = headerParser.getPaddingLength();
        state = State.PADDING;
        if (contentParser.noContent())
            return true;
    }
    else
    {
        ContentParser.Result result = contentParser.parse(buffer);
        // ...[snip]...
    }
    break;
}
```

Figure 15.1: `Parser.java`, lines 82–114

Exploit Scenario

An attacker operates a malicious web server that supports FastCGI. A Jetty application communicates with this server by using Jetty's built-in FastCGI client. The remote server transmits a frame with an invalid frame type, causing a `NullPointerException` exception and a crash in the Jetty application.

Recommendations

Short term, add a `null` check to the `parse` function to abort the parsing process before dereferencing a `null` return value from `findContentParser`. If a `null` value is detected, `parse` should throw an appropriate exception, such as `IllegalStateException`, that Jetty can catch and handle safely.

Long term, build out a larger suite of test cases that ensures graceful handling of malformed traffic and data.

16. Documentation does not specify that request contents and other user data can be exposed in debug logs

Severity: **Medium**	Difficulty: **High**
Type: Data Exposure	Finding ID: TOB-JETTY-16
Target: Jetty 12 Operations Guide; numerous locations throughout the codebase	

Description

Over 100 times, the system calls `LOG.debug` with a parameter of the format `BufferUtil.toDetailString(buffer)`, which outputs up to 56 bytes of the buffer into the log file. Jetty's implementations of various protocols and encodings, including GZIP, WebSocket, multipart encoding, and HTTP/2, output user data received over the network to the debug log using this type of call.

An example instance from Jetty's WebSocket implementation appears in figure 16.1.

```
public Frame.Parsed parse(ByteBuffer buffer) throws WebSocketException
{
    try
    {
        // parse through
        while (buffer.hasRemaining())
        {
            if (LOG.isDebugEnabled())
                LOG.debug("{} Parsing {}", this, BufferUtil.toDetailString(buffer));
            // ...[snip]...
        }
        // ...[snip]...
    }
    // ...[snip]...
}
```

Figure 16.1: `Parser.java`, lines 88–96

Although the Jetty 12 Operations Guide does state that Jetty debugging logs can quickly consume massive amounts of disk space, it does not advise system administrators that the logs can contain sensitive user data, such as personally identifiable information. Thus, the possibility of raw traffic being captured from debug logs is undocumented.

Exploit Scenario

A Jetty system administrator turns on debug logging in a production environment. During the normal course of operation, a user sends traffic containing sensitive information, such as personally identifiable information or financial data, and this data is recorded to the debug log. An attacker who gains access to this log can then read the user data, compromising data confidentiality and the user's privacy rights.

Recommendations
Short term, update the Jetty Operations Guide to state that in addition to being extremely large, debug logs can contain sensitive user data and should be treated as sensitive.

Long term, consider moving all debugging messages that contain buffer excerpts into a high-detail debug log that is enabled only for debug builds of the application.

17. HttpStreamOverFCGI internally marks all requests as plaintext HTTP

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-17
Target: `org.eclipse.jetty.fcgi.server.internal.HttpStreamOverFCGI`	

Description

The `HttpStreamOverFCGI` class processes FastCGI messages in a format that can be processed by other system components that use the `HttpStream` interface. This class's `onHeaders` callback mistakenly marks each `MetaData.Request` object as a plaintext HTTP request, as the "TODO" comment shown in figure 17.1 indicates:

```
public void onHeaders()
{
    String pathQuery = URIUtil.addPathQuery(_path, _query);
    // TODO https?
    MetaData.Request request = new MetaData.Request(_method,
HttpScheme.HTTP.asString(), hostPort, pathQuery, HttpVersion.fromString(_version),
_headers, Long.MIN_VALUE);
    // ...[snip]...
}
```

Figure 17.1: `HttpStreamOverFCGI.java`, lines 108–119

In some configurations, other Jetty components could misinterpret a message received over FCGI as a plaintext HTTP message, which could cause a request to be incorrectly rejected, redirected in an infinite loop, or forwarded to another system over a plaintext HTTP channel instead of HTTPS.

Exploit Scenario

A Jetty instance runs an FCGI server and uses the `HttpStream` interface to process messages. The `MetaData.Request` class's `getURI` method is used to check the incoming request's URI. This method mistakenly returns a plaintext HTTP URL due to the bug in `HttpStreamOverFCGI.java`. One of the following takes place during the processing of this request:

- An application-level security control checks the incoming request's URI to ensure it was received over a TLS-encrypted channel. Since this check fails, the application rejects the request and refuses to process it, causing a denial of service.

- An application-level security control checks the incoming request's URI to ensure it was received over a TLS-encrypted channel. Since this check fails, the application attempts to redirect the user to a suitable HTTPS URL. The check fails on this redirected request as well, causing an infinite redirect loop and a denial of service.

- An application processing FCGI messages acts as a proxy, forwarding certain requests to a third HTTP server. It uses `MetaData.Request.getURI` to check the request's original URI and mistakenly sends a request over plaintext HTTP.

Recommendations

Short term, correct the bug in `HttpStreamOverFCGI.java` to generate the correct URI for the incoming request.

Long term, consider streamlining the HTTP implementation to minimize the need for different classes to generate URIs from request data.

18. Excessively permissive and non-standards-compliant error handling in HTTP/2 implementation	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-18
Target: The `org.eclipse.jetty.http2.parser` and `org.eclipse.jetty.http2.parser` packages	

Description

Jetty's HTTP/2 implementation violates RFC 9113 in that it fails to terminate a connection with an appropriate error code when the remote peer sends a frame with one of the following protocol violations:

- A `SETTINGS` frame with the `ACK` flag set and a nonzero payload length
- A `PUSH_PROMISE` frame in a stream with push disabled
- A `GOAWAY` frame with its stream ID not set to zero

None of these situations creates an exploitable vulnerability. However, noncompliant protocol implementations can create compatibility problems and could cause vulnerabilities when deployed in combination with other misconfigured systems.

Exploit Scenario

A Jetty instance connects to an HTTP/2 server, or serves a connection from an HTTP/2 client, and the remote peer sends traffic that should cause Jetty to terminate the connection. Instead, Jetty keeps the connection alive, in violation of RFC 9113. If the remote peer is programmed to handle the noncompliant traffic differently than Jetty, further problems could result, as the two implementations interpret protocol messages differently.

Recommendations

Short term, update the HTTP/2 implementation to check for the following error conditions and terminate the connection with an error code that complies with RFC 9113:

- A peer receives a `SETTINGS` frame with the `ACK` flag set and a payload length greater than zero.
- A client receives a `PUSH_PROMISE` frame after having sent, and received an acknowledgement for, a `SETTINGS` frame with `SETTINGS_ENABLE_PUSH` equal to zero.

- A peer receives a GOAWAY frame with the stream identifier field not set to zero.

Long term, audit Jetty's implementation of HTTP/2 and other protocols to ensure that Jetty handles errors in a standards-compliant manner and terminates connections as required by the applicable specifications.

19. XML external entities and entity expansion in Maven package metadata parser	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-JETTY-19
Target: `org.eclipse.jetty.start.fileinits.MavenMetadata`	

Description

During module initialization, the `MavenMetadata` class parses `maven-metadata.xml` files when the module configuration includes a `maven://` URI in its `[files]` section. The `DocumentBuilderFactory` class is used with its default settings, meaning that document type definitions (DTD) are allowed and are applied. This behavior leaves the `MavenMetadata` class vulnerable to XML external entity (XXE) and XML entity expansion (XEE) attacks. These vulnerabilities could enable a variety of exploits, including server-side request forgery on the server's local network and arbitrary file reads from the server's filesystem.

Exploit Scenario

An attacker causes a Jetty installation to parse a maliciously crafted `maven-metadata.xml` file, such as by uploading a malicious package to a Maven repository, inducing an out-of-band download of the malicious package through social engineering, or by placing the `maven-metadata.xml` file on the server's filesystem through other means. When the XML file is parsed, the XML-based attack is launched. The attacker could leverage this vector to do any of the following:

- Induce HTTP requests to servers on the server's local network
- Read and exfiltrate arbitrary files on the server's filesystem
- Consume excessive system resources with an XEE, causing a denial of service

Recommendations

Short term, disable parsing of DTDs in `MavenMetadata` so that `maven-metadata.xml` files cannot be used as a vector for XML-based attacks. Disabling DTDs may require knowledge of the underlying XML parser implementation returned by the `DocumentBuilderFactory` class.

Long term, review default configurations and attributes supported by XML parsers that may be returned by the `DocumentBuilderFactory` to ensure that DTDs are properly disabled.

20. Use of deprecated AccessController class

Severity: **Informational**	Difficulty: **N/A**
Type: Code Quality	Finding ID: TOB-JETTY-20

Target:

- `org.eclipse.jetty.logging.JettyLoggerConfiguration`
- `org.eclipse.jetty.util.MemoryUtils`
- `org.eclipse.jetty.util.TypeUtil`
- `org.eclipse.jetty.util.thread.PrivilegedThreadFactory`
- `org.eclipse.jetty.ee10.servlet.ServletContextHandler`
- `org.eclipse.jetty.ee9.nested.ContextHandler`

Description

The classes listed in the "Target" cell above use the `java.security.AccessController` class, which is a deprecated class slated to be removed in a future Java release. The `java.security` library documentation states that the `AccessController` class "is only useful in conjunction with the Security Manager," which is also deprecated. Thus, the use of `AccessController` no longer serves any beneficial purpose.

The use of this deprecated class could impact Jetty's compatibility with future releases of the Java SDK.

Recommendations

Short term, remove all uses of the `AccessController` class.

Long term, audit the Jetty codebase for the use of classes in the `java.security` package that may not provide any value in Jetty 12, and remove all references to those classes.

21. QUIC server writes SSL private key to temporary plaintext file

Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-JETTY-21
Target: `org.eclipse.jetty.quic.server.QuicServerConnector`	

Description

Jetty's QUIC implementation uses quiche, a QUIC and HTTP/3 library maintained by Cloudflare. When the server's SSL certificate is handed off to quiche, the private key is extracted from the existing keystore and written to a temporary plaintext PEM file:

```
protected void doStart() throws Exception
    {
        // ...[snip]...
        char[] keyStorePassword =
sslContextFactory.getKeyStorePassword().toCharArray();
        String keyManagerPassword = sslContextFactory.getKeyManagerPassword();
        SSLKeyPair keyPair = new SSLKeyPair(
            sslContextFactory.getKeyStoreResource().getPath(),
            sslContextFactory.getKeyStoreType(),
            keyStorePassword,
            alias,
            keyManagerPassword == null ? keyStorePassword :
keyManagerPassword.toCharArray()
        );
        File[] pemFiles = keyPair.export(new
File(System.getProperty("java.io.tmpdir")));
        privateKeyFile = pemFiles[0];
        certificateChainFile = pemFiles[1];
    }
```

Figure 21.1: `QuicServerConnector.java`, lines 154–179

Storing the private key in this manner exposes it to increased risk of theft. Although the `QuicServerConnector` class deletes the private key file upon stopping the server, this deleted file may not be immediately removed from the physical storage medium, exposing the file to potential theft by attackers who can access the raw bytes on the disk.

A review of quiche suggests that the library's API may not support reading a DES-encrypted keyfile. If that is true, then remediating this issue would require updates to the underlying quiche library.

Exploit Scenario
An attacker gains read access to a Jetty HTTP/3 server's temporary directory while the server is running. The attacker can retrieve the temporary keyfile and read the private key without needing to obtain or guess the encryption key for the original keystore. With this private key in hand, the attacker decrypts and tampers with all TLS communications that use the associated certificate.

Recommendations
Short term, investigate the quiche library's API to determine whether it can readily support password-encrypted private keyfiles. If so, update Jetty to save the private key in a temporary password-protected file and to forward that password to quiche. Alternatively, if password-encrypted private keyfiles can be supported, have Jetty pass the unencrypted private key directly to quiche as a function argument. Either option would obviate the need to store the key in a plaintext file on the server's filesystem.

If quiche does not support either of these changes, open an issue or pull request for quiche to implement a fix for this issue.

22. Repeated code between HPACK and QPACK

Severity: **Informational**	Difficulty: **N/A**
Type: Code Quality	Finding ID: TOB-JETTY-22

Target:
- `org.eclipse.jetty.http2.hpack.internal.NBitInteger`
- `org.eclipse.jetty.http2.hpack.internal.Huffman`
- `org.eclipse.jetty.http3.qpack.internal.util.NBitIntegerParser`
- `org.eclipse.jetty.http3.qpack.internal.util.NBitIntegerEncode`
- `org.eclipse.jetty.http3.qpack.internal.util.HuffmanDecoder`
- `org.eclipse.jetty.http3.qpack.internal.util.HuffmanEncoder`

Description
Classes for dealing with n-bit integers and Huffman coding are implemented both in the `jetty-http2-hpack` and in `jetty-http3-qpack` libraries. These classes have very similar functionality but are implemented in two different places, sometimes with identical code and other times with different implementations. In some cases (TOB-JETTY-9), one implementation has a bug that the other implementation does not have. The codebase would be easier to maintain and keep secure if the implementations were merged.

Exploit Scenario
A vulnerability is found in the Huffman encoding implementation, which has identical code in HPACK and QPACK. The vulnerability is fixed in one implementation but not the other, leaving one of the implementations vulnerable.

Recommendations
Short term, merge the two implementations of n-bit integers and Huffman coding classes.

Long term, audit the Jetty codebase for other classes with very similar functionality.

23. Various exceptions in HpackDecoder.decode and QpackDecoder.decode	
Severity: **Informational**	Difficulty: **N/A**
Type: Denial of Service	Finding ID: TOB-JETTY-23
Target: `org.eclipse.jetty.http2.hpack.HpackDecoder`, `org.eclipse.jetty.http3.qpack.QpackDecoder`	

Description

The `HpackDecoder` and `QpackDecoder` classes both throw unexpected Java-level exceptions:

- `HpackDecoder.decode(0x03)` throws `BufferUnderflowException`.
- `HpackDecoder.decode(0x4800)` throws `NumberFormatException`.
- `HpackDecoder.decode(0x3fff 2e)` throws `IllegalArgumentException`.
- `HpackDecoder.decode(0x3fff 81ff ff2e)` throws `NullPointerException`.
- `HpackDecoder.decode(0xffff ffff f8ff ffff ffff ffff ffff ffff ffff ffff ffff ffff 0202 0000)` throws `ArrayIndexOutOfBoundsException`.
- `QpackDecoder.decode(..., 0x81, ...)` throws `IndexOutOfBoundsException`.
- `QpackDecoder.decode(..., 0xfff8 ffff f75b, ...)` throws `ArithmeticException`.

For both HPACK and QPACK, these exceptions appear to be caught higher up in the call chain by `catch (Throwable x)` statements every time the `decode` functions are called. However, catching them within `decode` and throwing a Jetty-level exception within the `catch` statement would result in cleaner, more robust code.

Exploit Scenario

Jetty developers refactor the codebase, moving function calls around and introducing a new point in the code where `HpackDecoder.decode` is called. Assuming that `decode` will throw only `org.jetty…` errors, they forget to wrap this call in a `catch (Throwable x)` statement. This results in a DoS vulnerability.

Recommendations

Short term, document in the code that Java-level exceptions can be thrown.

Long term, modify the `decode` functions so that they throw only Jetty-level exceptions.

24. Incorrect QPACK encoding when multi-byte characters are used

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-JETTY-24
Target: `org.eclipse.jetty.http3.qpack.internal.EncodableEntry`	

Description

Java's `string.length()` function returns the number of characters in a string, which can be different from the number of bytes returned by the `string.getBytes()` function. However, QPACK encoding methods assume that they return the same number, which could cause incorrect encodings.

In `EncodableEntry.LiteralEntry`, which is used to encode HTTP/3 header fields, the following method is used for encoding:

```
    public void encode(ByteBuffer buffer, int base)
    {
        byte allowIntermediary = 0x00; // TODO: this is 0x10 bit, when should this be set?
        String name = getName();
        String value = getValue();

        // Encode the prefix code and the name.
        if (_huffman)
        {
            buffer.put((byte)(0x28 | allowIntermediary));
            NBitIntegerEncoder.encode(buffer, 3, HuffmanEncoder.octetsNeeded(name));
            HuffmanEncoder.encode(buffer, name);
            buffer.put((byte)0x80);
            NBitIntegerEncoder.encode(buffer, 7, HuffmanEncoder.octetsNeeded(value));
            HuffmanEncoder.encode(buffer, value);
        }
        else
        {
            // TODO: What charset should we be using? (this applies to the instruction generators as well).
            buffer.put((byte)(0x20 | allowIntermediary));
            NBitIntegerEncoder.encode(buffer, 3, name.length());
            buffer.put(name.getBytes());
            buffer.put((byte)0x00);
            NBitIntegerEncoder.encode(buffer, 7, value.length());
            buffer.put(value.getBytes());
```

```
239         }
240     }
```

Figure 24.1: EncodableEntry.java, lines 214–240

Note in particular lines 232–238, which are used to encode literal (non-Huffman-coded) names and values. The value returned by `name.length()` is added to the bytestring, followed by the value returned by `name.getBytes()`. Then, the value returned by `value.length()` is added to the bytestring, followed by the value returned by `value.getBytes()`. When this bytestring is decoded, the decoder will read the name length field and then read that many *bytes* as the name. If multibyte characters were used in the name field, the decoder will read too few bytes. The rest of the bytestring will also be decoded incorrectly, since the decoder will continue reading at the wrong point in the bytestring. The same issue occurs if multibyte characters were used in the value field.

The same issue appears in `EncodableEntry.ReferencedNameEntry.encode`:

```
164     // Encode the value.
165     String value = getValue();
166     if (_huffman)
167     {
168        buffer.put((byte)0x80);
169        NBitIntegerEncoder.encode(buffer, 7, HuffmanEncoder.octetsNeeded(value));
170        HuffmanEncoder.encode(buffer, value);
171     }
172     else
173     {
174        buffer.put((byte)0x00);
175        NBitIntegerEncoder.encode(buffer, 7, value.length());
176        buffer.put(value.getBytes());
177     }
```

Figure 24.2: EncodableEntry.java, lines 164–177

If `value` has multibyte characters, it will be incorrectly encoded in lines 174–176.

Jetty's HTTP/3 code is still considered experimental, so this issue should not affect production code, but it should be fixed before announcing HTTP/3 support to be production-ready.

Exploit Scenario

A Jetty server attempts to add the `Set-Cookie` header, setting a cookie value to a UTF-8-encoded string that contains multibyte characters. This causes an incorrect cookie value to be set and the rest of the headers in this message to be parsed incorrectly.

Recommendations

Short term, have the `encode` function in both `EncodableEntry.LiteralEntry` and `EncodableEntry.ReferencedNameEntry` encode the length of the string using `string.getBytes()` rather than `string.length()`.

25. No limits on maximum capacity in QPACK decoder	
Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-JETTY-25

Target:

- `org.eclipse.jetty.http3.qpack.QpackDecoder`
- `org.eclipse.jetty.http3.qpack.internal.parser.DecoderInstructionParser`
- `org.eclipse.jetty.http3.qpack.internal.table.DynamicTable`

Description

In QPACK, an encoder can set the dynamic table capacity of the decoder using a "Set Dynamic Table Capacity" instruction. The HTTP/3 specification requires that the capacity be no larger than the `SETTINGS_QPACK_MAX_TABLE_CAPACITY` limit chosen by the decoder. However, nowhere in the QPACK code is this limit checked for. This means that the encoder can choose whatever capacity it wants (up to Java's maximum integer value), allowing it to take up large amounts of space on the decoder's memory.

Jetty's HTTP/3 code is still considered experimental, so this issue should not affect production code, but it should be fixed before announcing HTTP/3 support to be production-ready.

Exploit Scenario

A Jetty server supporting QPACK is running. An attacker opens a connection to the server. He sends a "Set Dynamic Table Capacity" instruction, setting the dynamic table capacity to Java's maximum integer value, 2^{31-1} (2.1 GB). He then repeatedly enters very large values into the server's dynamic table using an "Insert with Literal Name" instruction until the full 2.1 GB capacity is taken up.

The attacker repeats this using multiple connections until the server runs out of memory and crashes.

Recommendations

Short term, enforce the `SETTINGS_QPACK_MAX_TABLE_CAPACITY` limit on the capacity.

Long term, audit Jetty's implementation of QPACK and other protocols to ensure that Jetty enforces limits as required by the standards.

Summary of Recommendations

Jetty is an ongoing software project with three major releases in the past three years, including Jetty 12. Trail of Bits recommends that the Eclipse Foundation address the findings detailed in this report and take the following additional steps:

- Audit protocol implementations and parsers for fields (e.g., length fields) that are defined as unsigned integers in the applicable specifications. Review the relevant code for confusion between signed and unsigned integer operations. If necessary, use the `Integer` class to ensure that such values are treated as unsigned and do not overflow to negative numbers.

- Update Jetty's tests to account for the most recent changes to Jetty Core in version 12. Expand the test cases for protocol implementations to include error conditions that must be handled in a manner specified in the relevant RFC.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Code Quality	Code antipatterns and other quality issues without security impact
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On June 5, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Jetty team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 25 issues described in this report, Jetty has resolved 20, has partially resolved two, and has not resolved the remaining three. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Risk of integer overflow that could allow HpackDecoder to exceed maxHeaderSize	Medium	Resolved
2	Cookie parser accepts unmatched quotation marks	Informational	Resolved
3	Errant command quoting in CGI servlet	High	Resolved
4	Symlink-allowed alias checker ignores protected targets list	High	Resolved
5	Missing check for malformed Unicode escape sequences in QuotedStringTokenizer.unquote	Low	Resolved
6	WebSocket frame length represented with 32-bit integer	High	Resolved
7	WebSocket parser does not check for negative payload lengths	Low	Resolved
8	WebSocket parser greedily allocates ByteBuffers for large frames	Medium	Unresolved

9	Risk of integer overflow in HPACK's NBitInteger.decode	Informational	Resolved
10	MetaDataBuilder.checkSize accepts headers of negative lengths	Medium	Resolved
11	Insufficient space allocated when encoding QPACK instructions and entries	Low	Resolved
12	LiteralNameEntryInstruction incorrectly encodes value length	Medium	Resolved
13	FileInitializer does not check for symlinks	High	Unresolved
14	FileInitializer permits downloading files via plaintext HTTP	High	Resolved
15	NullPointerException thrown by FastCGI parser on invalid frame type	Medium	Resolved
16	Documentation does not specify that request contents and other user data can be exposed in debug logs	Medium	Unresolved
17	HttpStreamOverFCGI internally marks all requests as plaintext HTTP	High	Resolved
18	Excessively permissive and non-standards-compliant error handling in HTTP/2 implementation	Low	Resolved
19	XML external entities and entity expansion in Maven package metadata parser	High	Partially Resolved
20	Use of deprecated AccessController class	Informational	Resolved
21	QUIC server writes SSL private key to temporary plaintext file	High	Partially Resolved

22	Repeated code between HPACK and QPACK	Informational	Resolved
23	Various exceptions in HpackDecoder.decode and QpackDecoder.decode	Informational	Resolved
24	Incorrect QPACK encoding when multi-byte characters are used	Medium	Resolved
25	No limits on maximum capacity in QPACK decoder	High	Resolved

Detailed Fix Review Results

TOB-JETTY-1: Risk of integer overflow that could allow HpackDecoder to exceed maxHeaderSize
Resolved in PR #9634. The decoder now checks for negative length values, allowing the decoder to detect the integer overflow condition and throw an appropriate condition.

TOB-JETTY-2: Cookie parser accepts unmatched quotation marks
Resolved in PR #9339. The cookie parsing logic has been reworked, and dynamic testing confirms that unmatched quotation marks are rejected with an appropriate error condition.

TOB-JETTY-3: Errant command quoting in CGI servlet
Resolved in PR #9516. The affected CGI servlet class has been removed.

TOB-JETTY-4: Symlink-allowed alias checker ignores protected targets list
Resolved in PR #9506. The symlink check that was previously commented out has been reinserted. Symbolic links are now appropriately checked against the protected targets list.

TOB-JETTY-5: Missing check for malformed Unicode escape sequences in QuotedStringTokenizer.unquote
Resolved in PR #9729. The string tokenizer logic has been reworked and broken into multiple classes. The logic bug leading to the mishandled Unicode escape sequences in the `QuotedStringTokenizer` and `RFC9110QuotedStringTokenizer` classes have been fixed. The `LegacyQuotedStringTokenizer` class is still vulnerable but is disabled by default. The Jetty team indicated during phone calls that this class is included for legacy support reasons only.

TOB-JETTY-6: WebSocket frame length represented with 32-bit integer and TOB-JETTY-7: WebSocket parser does not check for negative payload lengths
Resolved in PR #9741. Although the 32-bit integer data type remains in place, checks for negative payload lengths and integer overflows have been added. The WebSocket parser will no longer use a negative frame length for length comparisons, and integer overflows will cause the parser to throw an appropriate exception.

TOB-JETTY-8: WebSocket parser greedily allocates ByteBuffers for large frames
Unresolved in PR #9741. The greedy buffer allocation is unchanged. Jetty's bug tracking spreadsheet contains the following context for this finding's fix status:

> *Not an issue, added comment to explain why.*

The following comments have been added to the `org.eclipse.jetty.websocket.core.internal.Parser` class:

```
// We have already checked payload size in checkFrameSize, so we know we
can autoFragment if larger than maxFrameSize.

// The size of this allocation is limited by the maxFrameSize.
```

The default maximum frame size is set at 64 KB by the `WebSocketConstants` class.

TOB-JETTY-9: Risk of integer overflow in HPACK's NBitInteger.decode
Resolved in PR #9634. The integer decoding logic has been moved to common classes in the `jetty-http` package. The HPACK parsing code that invokes this decoding logic makes appropriate checks for negative return values, throwing an appropriate exception if a negative value is decoded.

TOB-JETTY-10: MetaDataBuilder.checkSize accepts headers of negative lengths
Resolved in PR #9634. The HPACK parsing logic has been reworked, and the affected `MetaDataBuilder.checkSize` function has been replaced with length checks in other classes. It is no longer possible for the `length` value to overflow into a very large positive integer, and the length checks are performed against the input buffer's `buffer.remaining()` value, which can never be negative.

TOB-JETTY-11: Insufficient space allocated when encoding QPACK instructions and entries
Resolved in PR #9634. Parsing is now restricted to ISO-8859-1 encoding, which uses only single-byte character encodings. Therefore, the logic bug involving multibyte character encoding has been eliminated.

TOB-JETTY-12: LiteralNameEntryInstruction incorrectly encodes value length
Resolved in PR #9634. The encoding logic has been reworked and reorganized so that the field widths are calculated in a centralized class. Field lengths appear to be correctly generated, and integers are no longer encoded using hard-coded fixed widths.

TOB-JETTY-13: FileInitializer does not check for symlinks
Unresolved in PR #9555. The `FileInitializer` class contains the following comment regarding this finding:

```
// We restrict our behavior to only modifying what exists in
${jetty.base}.
// If the user decides they want to use advanced setups, such as symlinks
to point
// to content outside of ${jetty.base}, that is their decision and we
will not
// attempt to save them from themselves.
// Note: All copy and extract steps will not replace files that already
exist.
```

TOB-JETTY-14: FileInitializer permits downloading files via plaintext HTTP
Resolved in PR #9555. The `JettyStart` class now recognizes the `--allow-insecure-http-downloads` flag, which enables file downloads over plaintext HTTP. By default, this flag is disabled, so system administrators must manually specify that they wish to enable unencrypted downloads.

TOB-JETTY-15: NullPointerException thrown by FastCGI parser on invalid frame type
Resolved in commit `e5590a`. Broader exception handling has been added to the `org.eclipse.jetty.fcgi.parser.Parser` class so that invalid frame types will invoke the normal error handling routines for malformed FastCGI traffic. No `NullPointerException` will be thrown on an invalid frame type.

TOB-JETTY-16: Documentation does not specify that request contents and other user data can be exposed in debug logs
Unresolved. No commit or pull request addressing this issue was identified, and system documentation has not undergone any relevant changes.

TOB-JETTY-17: HttpStreamOverFCGI internally marks all requests as plaintext HTTP
Resolved in PR #9733. The FastCGI HTTPS header is now checked appropriately, and each FCGI request object's HTTP scheme is set correctly.

TOB-JETTY-18: Excessively permissive and non-standards-compliant error handling in HTTP/2 implementation
Resolved in PR #9749. The HTTP/2 frame parser classes now check for each of the error conditions identified in this finding, and the error codes returned comply with the requirements of RFC 9113.

TOB-JETTY-19: XML external entities and entity expansion in Maven package metadata parser
Partially resolved in PR #9555. Jetty now invokes the XML parser's secure processing feature, which instructs the XML parser to use the most secure settings when parsing documents. However, this feature's behavior is implementation-dependent and may not be consistent across Java environments. Therefore, there may be a residual risk of XML-based attacks. To mitigate these risks even further, it may be necessary to manually check for and remove DTD declarations in the XML input or to use an XML parsing library whose behavior is known and consistent.

TOB-JETTY-20: Use of deprecated AccessController class
Resolved in PR #9616. Per documentation provided by the Jetty team, Jetty supports older Java environments that differ with respect to their support for the `SecurityManager` class. The use of reflection implemented in the PR is an effective solution to manage these requirements.

TOB-JETTY-21: QUIC server writes SSL private key to temporary plaintext file
Partially resolved. As the original finding documents, this finding reflects a weakness in the third-party quiche library and cannot be resolved by the Jetty team. However, Jetty developers have helped begin the process of resolving this finding by submitting an issue to the quiche developers.

TOB-JETTY-22: Repeated code between HPACK and QPACK
Resolved in PR #9634. The common encoding and decoding logic has been moved into the `jetty-http` directory and is reused between the HPACK and QPACK libraries.

TOB-JETTY-23: Various exceptions in HpackDecoder.decode and QpackDecoder.decode
Resolved in commit `fd913a`. The `HpackDecoder` and `QpackDecoder` classes have undergone significant rewrites with improved exception handling; by reviewing the code, we found that improved error handling will cause these classes to generate protocol-specific error conditions instead of throwing general-purpose Java exceptions.

TOB-JETTY-24: Incorrect QPACK encoding when multi-byte characters are used
Resolved in PR #9634. All QPACK encoding now uses ISO-8859-1 encoding, which is a single-byte character encoding scheme. Therefore, there are no longer any multi-byte encoding errors in the QPACK implementation.

TOB-JETTY-25: No limits on maximum capacity in QPACK decoder
Resolved in PR #9728. The `QpackDecoder` and `QpackEncoder` classes now check the maximum table capacity setting and throw an HTTP/3 protocol error if the configured capacity exceeds the configured maximum.