# Careless Retention and Management: Understanding and Detecting Data Retention Denial-of-Service Vulnerabilities in Java Web Containers

Keke Lian *Fudan University*
Lei Zhang *Fudan University*
Haoran Zhao *Fudan University*
Yinzhi Cao *Johns Hopkins University*
Yongheng Liu *Fudan University*
Fute Sun *Fudan University*
Yuan Zhang *Fudan University*
Min Yang *Fudan University*

## Abstract

Denial-of-Service (DoS) attacks have long been a major threat to the availability of the World Wide Web. While prior works have extensively studied network-layer DoS and certain types of application-layer DoS, such as Regular Expression DoS (ReDoS), little attention has been paid to memory exhaustion DoS, especially in Java Web containers. Our research target is a special type of memory exhaustion DoS vulnerabilities that retain user data in web containers, which is defined as Data Retention DoS (DRDoS) in this paper. To the best of our knowledge, there are no systematic academic studies of such DRDoS vulnerabilities of Java Web Containers except for a few manually found vulnerabilities in the Common Vulnerabilities and Exposures (CVE) database.

In this paper, we design and implement a novel static analysis approach, called DR. D, to detect and assess DRDoS vulnerabilities in Java web containers. Our key insight is to analyze the request handling process of web containers and detect whether client-controlled request data may be retained in the containers, thus leading to DRDoS vulnerabilities. We apply DR. D on four popular open-source Java web containers, discovering that all of them have DRDoS vulnerabilities. Specifically, DR. D finds 25 zero-day, exploitable vulnerabilities. We have responsibly reported all of them to corresponding developers and received confirmations. So far, we have received seventeen CVE identifiers (three of them assigned with high severity). Based on scan results from search engine, e.g., Shodan, we identify that over 1.5 million public IP addresses are hosting vulnerable versions of Java web containers potentially with DRDoS found by DR. D, demonstrating the spread of DRDoS vulnerability.

## 1 Introduction

A Denial-of-Service (DoS) attack is a cyber attack that makes a host server unavailable to its users, which poses a significant threat to the World Wide Web. Generally speaking, there are two types of DoS attacks: network- and application-layer attacks. On one hand, a network-layer attack—e.g., Distributed DoS (DDoS) [55] and TCP SYN Flood [57]—targets and exhausts network-layer resources, e.g., network bandwidth. Prior works [1, 7, 22, 23, 36, 52, 54] have extensively studied such DoS attacks and proposed many defense mechanisms [17, 18, 20, 27, 60, 61]. On the other hand, an application-layer attack [3, 9, 19, 24] targets an application vulnerability to exhaust application-layer resources, e.g., CPU and memory. Such attacks usually leverage low-bandwidth, highly-targeted, and intensive requests to overwhelm a target system, thus being more challenging to detect. The most notorious one is Regular Expression Denial-of-Service (ReDoS) [8,10,40,44,49,53], which exploits a regex vulnerability for a polynomial or exponential matching time. Other examples include memory exhaustion DoS vulnerabilities introduced by C/C++'s memory consumption bugs [16,41,47,50].

The research question of this paper is to detect a special type of application-layer DoS vulnerabilities, which we call Data Retention DoS (DRDoS), in Java web containers (e.g., Eclipse Jetty [12]), i.e., widely-used middleware that manages low-level request handling for web apps. Specifically, since web containers inevitably cache user request data for processing, such cached data—if not handled, e.g., released, properly—may be retained and accumulated in the memory, thus leading to DoS consequences. Broadly speaking, DRDoS is a type of memory exhaustion DoS [15]. However, to the best of our knowledge, there is no systematic study of detecting memory exhaustion DoS in Java let alone DRDoS. Previous works in C/C++ [16, 41, 47, 50] are not applicable here because of the differences in memory management between C/C++ and Java, making it challenging to detect retained data in Java. That said, the best that we can find for Java is a few DRDoS vulnerabilities [45, 46] that people have found in the past manually and are being documented by the Common Vulnerabilities and Exposures (CVE) database.

In this paper, we design and implement a novel static analysis approach, called DR. D (Data Retention Diagnoser), to detect and assess vulnerable data retention in Java web containers. The key idea is to analyze the request handling process and detect if client-controlled request data may be retained in the containers for a long time. While intuitively simple, it faces the following challenges:

- *Locating request handlers of web containers.* Web containers provide a multitude of request handlers for developers to customize request processing pipelines and implement application business logic. These functions are invoked according to the developer's configurations, making them challenging to analyze from low-level request reception entry points. Even worse, they are scattered throughout containers with non-standardized specifications and lack comprehensive documentation. Hence, a detection approach should understand the container's request processing and locate as many request handlers as possible.
- *Discovering long-lived objects that can retain and accumulate request data for an extended period.* While some long-lived objects may exhibit clear syntactic features, e.g., static variables, a substantial number of them lack apparent characteristics. These objects are intricately linked to container implementation and shaped by their runtime reference relationships, which are hard to model through static analysis. Furthermore, dynamic approaches like heap dump analysis are inherently limited in code coverage, and a noticeable gap exists between runtime memory objects and their counterparts in static analysis. That said, a detection approach should discover such long-lived objects through a container-independent approach.
- *Lightweight yet efficient vulnerability assessment.* The execution of request handlers in web containers often depends on intricate deployment configurations. While dynamic fuzzing techniques have been extensively studied for generating inputs that trigger vulnerabilities, the configuration constraints are hard to automate through methods like input mutation, often requiring significant manual effort. Hence, a detection approach should conduct a lightweight yet efficient analysis to confirm exploitability.

Specifically, DR. D consists of three stages, which solve the challenges above. First, DR. D locates request handlers through a request data-driven approach. It automatically identifies container-specific Java classes representing request data and then analyzes the relationships among methods that manipulate request data to locate request handlers. Second, DR. D classifies Java classes that may have a long lifecycle using a learning-based method through feature analysis. Finally, based on the results of the first two stages, DR. D employs a precise data flow analysis to detect if request data is retained in long-lived objects. It further assesses the exploitability of data retention by modeling and analyzing their exploit restrictions.

We apply DR. D on four widely-used open-source Java web containers, including Apache Tomcat, Eclipse Jetty, Red Hat Undertow, and Caucho Resin. DR. D finds all of them vulnerable to DRDoS attacks and locates 25 unique zero-day, exploitable DRDoS vulnerabilities in total. We responsibly

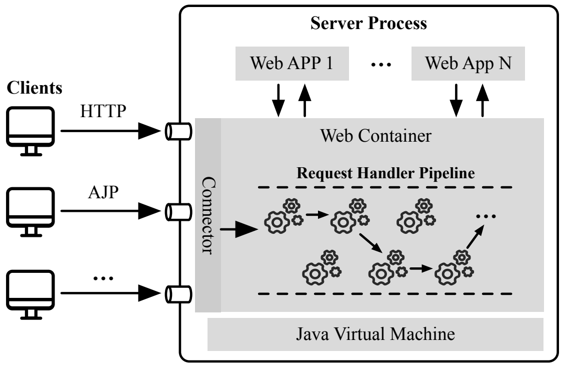


Figure 1: The Typical Architecture of Java Web Container.

disclosed our findings and vulnerabilities to respective developers and received confirmations from *all* of them. So far, we received seventeen CVE identifiers (including three of high severity) for these vulnerabilities. We also perform practical experiments in real-world environments by deploying our own vulnerable web containers and applications on popular cloud service platforms with their DoS protection enabled. The results reveal that our attacks can exhaust server memory resources successfully without triggering any alerts. Lastly, we assess possible affected IP addresses using Shodan's [38] scan results, specifically targeting web containers with vulnerable versions. Our results reveal that over 1.5 million public IPs, a huge number of victims, are hosting a vulnerable version of web containers affected by the DRDoS vulnerabilities found by DR. D.

We summarize the contributions of this paper as below:

- We conduct the first systematic study of DRDoS, an application-layer DoS attack targeting memory resources, by analyzing data retention in Java web containers. They suffer from DRDoS due to their inherent data retention requirements for specific functionalities despite Java's automated memory management mechanisms.
- We design and implement a novel static approach, named DR. D, to effectively detect vulnerable data retention in Java web containers and assess their exploitability in a lightweight way.
- We apply DR. D on 4 popular web containers to identify and confirm 25 unique zero-day DRDoS vulnerabilities, which can be easily exploited for remote DoS attacks. We responsibly disclosed them to respective developers and received their confirmations and acknowledgments.

## 2 Overview

### 2.1 Background

Java web containers, also known as servlet containers, are a core component of Java application servers, designed to manage and execute Java-based web applications. Originating from Java EE (now Jakarta EE), web containers provide a standardized environment for handling network requests and responses, managing the lifecycle of Servlets, and supporting JavaServer Pages. Unlike standalone applications, web containers serve as a runtime environment capable of hosting multiple web applications simultaneously, each with its own set of endpoints. Furthermore, web containers provide a suite of predefined request handlers for various tasks, such as user authentication and session management, allowing applications to assemble and customize tailored to their requirements.

Figure 1 illustrates the typical architecture of Java web containers, which manage requests and support various network protocols like HTTP, AJP, and custom protocols. When a client initiates a request, the containers listen on a designated port and parse the network stream to structured request objects. These requests are then processed by a series of request handlers based on deployment configurations, before being sent to the web app. Ultimately, the containers transform container-specific request objects into standardized servlet requests, adhering to the Jakarta EE specification [14], which are then routed to the appropriate web app based on the request URL. Besides, to enhance performance, some request processing tasks (e.g., loading and parsing multipart data) are deferred and only executed when required by web apps. These methods are not actively invoked by the container. Instead, they are executed on demand when web apps invoke specific request processing APIs provided by the web container.

The request handlers vary in implementation across web containers, each with unique configuration requirements. For example, Tomcat's `FormAuthenticator` implements the custom `Valve` interface to authenticate access to protected resources. As shown in Figure 2, configuring this handler involves adding the `FormAuthenticator` valve in Tomcat's *server.xml*, specifying the form-based authentication method, and defining login, error pages, and access paths for protected resources in the web app's *web.xml*. When Tomcat receives a client request, it invokes `FormAuthenticator` to check if the request targets protected resources and whether the user is authenticated. Unauthenticated users are redirected to a predefined login page, and upon successful authentication, the request continues to the protected resources.

## 2.2 A Motivating Example

Figure 3 illustrates a zero-day DRDoS vulnerability uncovered by DR. D in Eclipse Jetty. This vulnerability lies in a request handler, `PushSessionCacheFilter`, used to implement the HTTP/2 Server Push feature. This feature improves web page loading performance by caching request URIs (e.g., */home*) and related resources (e.g., *script.js*), enabling the server to proactively push potentially required resources to the client. However, Jetty lacks robust limitations and effective management of the retained data, enabling a malicious client to exhaust the container's memory resources and render

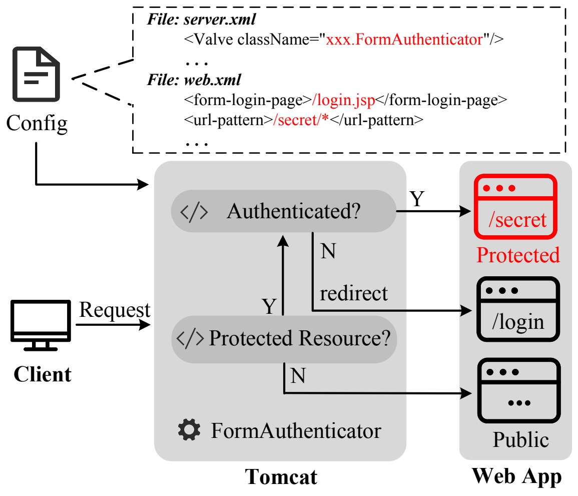

*File: server.xml*
```
<Valve className="xxx.FormAuthenticator"/>
...
```
*File: web.xml*
```
<form-login-page>/login.jsp</form-login-page>
<url-pattern>/secret/*</url-pattern>
...
```

Figure 2: A Request Handler Example in Apache Tomcat.

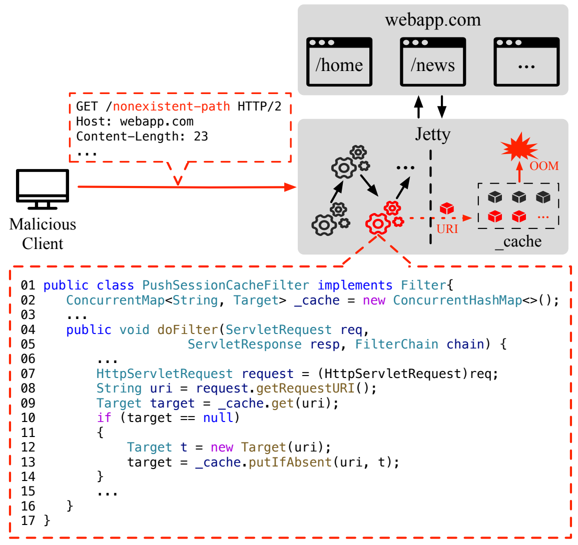

```
GET /nonexistent-path HTTP/2
Host: webapp.com
Content-Length: 23
...
```
```
01 public class PushSessionCacheFilter implements Filter{
02    ConcurrentMap<String, Target> _cache = new ConcurrentHashMap<>();
03    ...
04    public void doFilter(ServletRequest req,
05                         ServletResponse resp, FilterChain chain) {
06        ...
07        HttpServletRequest request = (HttpServletRequest)req;
08        String uri = request.getRequestURI();
09        Target target = _cache.get(uri);
10        if (target == null)
11        {
12            Target t = new Target(uri);
13            target = _cache.putIfAbsent(uri, t);
14        }
15        ...
16    }
17 }
```

Figure 3: A Motivating Example in Eclipse Jetty.

all deployed web applications unavailable to users.

Specifically, Jetty caches request URIs and related resources within a Map-type object, named `_cache` (lines 7-13), using URIs as keys to store each URI only once. Jetty also restricts requests reaching this request handler under its associated web app's host. However, this design is based on a flawed security assumption, presuming that all client-requested URIs correspond to existing web app resources. While regular users typically access a limited set of valid URIs via the web app's UI, attackers can craft a multitude of requests for non-existent URIs. This causes the size of `_cache` to grow indefinitely, consuming significant memory resources. Even worse, since the `_cache` object is managed by Jetty and not subject to Garbage Collection (GC), the retained URIs persist and accumulate in the Jetty's memory, without being released.

Here is how DR. D detects this vulnerability. First, DR. D identifies all Jetty's request classes (e.g., ServletRequest) and examines methods related to request data manipulation to locate this request handler (i.e., PushSessionCacheFilter.doFilter()). Then, DR. D identifies long-lived classes in Jetty and considers their instances as long-lived objects. Next, DR. D performs a comprehensive data flow analysis, starting from the request handler. The analysis revealed that some client-controlled request data (i.e., URI) is saved in the field (i.e., _cache) of a long-lived object, causing an increase in its size. Thus, DR. D identified this as a case of DRDoS. Lastly, DR. D conducts an assessment to determine its exploitability, which involves an examination of whether the value space of source request data (URI) is substantial enough, whether the data retention has any size limitations, and whether there are any temporal constraints on the retained data.

We responsibly disclosed this vulnerability to Jetty developers, who acknowledged it and stressed the need for a systematic analysis of similar issues within their project.

## 2.3 Threat Model

In our threat model, we consider attacks targeting remote web containers deployed in typical cloud-based or on-premises systems. Since web containers share the same Java process and memory resources with the web apps they host, the attack could render all deployed web apps in the victim container unavailable to users. We assume the defender operates as a typical web service provider that has deployed common DDoS defense mechanisms, typically including attack detection based on traffic volume and request characteristics. Moreover, we assume that deployments prioritize maintaining stable containers, with limited rotation and resetting, to optimize operational costs and efficiency.

The attacker only needs to control a single web client capable of accessing the target web container, such as a desktop computer in our experiments, without needing a large-scale botnet. This is because the attack payloads are retained and accumulated in the container's memory for an extended period after request processing is completed. The attacker can flexibly design the size and sending frequency of attack requests by comprehensively considering the exploitation conditions of the target vulnerability and the server-side defense mechanisms. Furthermore, the attacker requires no special identity or privileges as the attack exploits data management flaws in the general request processing tasks.

## 3 Methodology

Figure 4 illustrates the system architecture of DR. D, which operates in three main stages. Given a web container, DR. D first locates request handlers by automatically identifying container-specific request classes and clustering the request data manipulation methods. In the second stage, DR. D classifies long-lived Java classes via a machine learning model trained on class features. Finally, DR. D detects vulnerable data retention by analyzing whether any request data propagates from request handlers into long-lived objects through data flow analysis. Furthermore, DR. D assesses the exploitability of each vulnerability candidate by statically examining the restrictions on request data value space, data retention capacity, and the lifespan of retained data. The following sections delve into the details of each stage.

Table 1: Jakarta Servlet Request Specifications.

| Type | Name |
|---|---|
| Interface | jakarta.servlet.ServletRequest<br>jakarta.servlet.http.HttpServletRequest<br>javax.servlet.ServletRequest<br>javax.servlet.http.HttpServletRequest |
| Class | jakarta.servlet.http.HttpServletRequestWrapper<br>jakarta.servlet.ServletRequestWrapper<br>javax.servlet.http.HttpServletRequestWrapper<br>javax.servlet.ServletRequestWrapper |

## 3.1 Locating Request Handlers

In this stage, DR. D locates request handlers as entry points for analyzing request processing, as they are key functionalities of web containers and usually function as independent processing tasks. Treating them as entry points avoids the complexity of low-level network data analysis and aids in assessing and verifying vulnerability exploitability. However, this is challenging because request handlers are scattered throughout the container, lacking standardized implementation and comprehensive documentation. Even worse, many of them are not invoked by default. To address this, we propose a request data-driven approach, starting by identifying container-specific classes representing request data and then locating request handlers by analyzing the relationships between request data manipulation methods.

**Identifying Request Classes.** Regardless of the network protocols, web containers typically parse network streams into structured data objects for subsequent processing. These objects, though varied in class specifications, generally conform to the Jakarta servlet request specification for web app usage. Specifically, they either implement servlet request interfaces or can be transformed into servlet requests. Hence, DR. D leverages these characteristics to identify container-specific request classes. First, DR. D scans the web container for classes implementing the interfaces or extending the base classes derived from Jakarta docs [14], as listed in Table 1. Then, DR. D examines the constructors of these classes to identify parameters that are fully customized and not part of JDK standard classes. These parameter classes can be transformed into servlet requests. Besides, DR. D identifies classes that extend them as request classes.

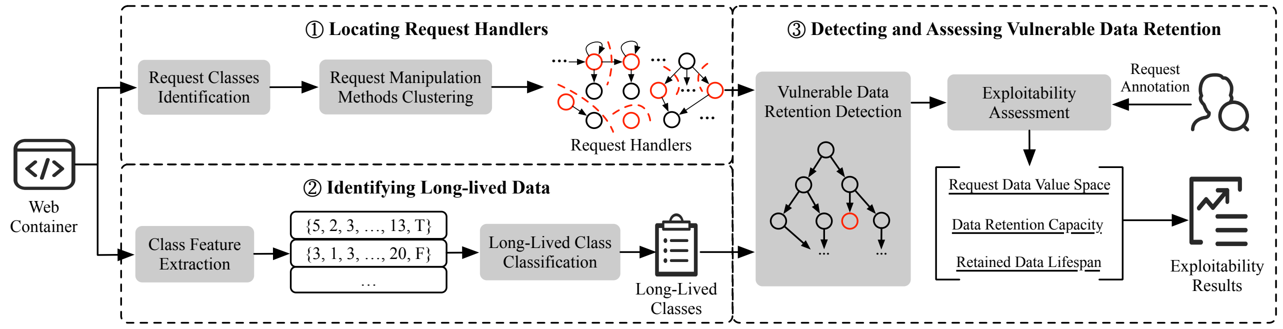

① Locating Request Handlers
Request Classes Identification
Request Manipulation Methods Clustering
Request Handlers
Web Container
② Identifying Long-lived Data
Class Feature Extraction
{5, 2, 3, …, 13, T}
{3, 1, 3, …, 20, F}
…
Long-Lived Class Classification
Long-Lived Classes
③ Detecting and Assessing Vulnerable Data Retention
Vulnerable Data Retention Detection
Exploitability Assessment
Request Annotation
Request Data Value Space
Data Retention Capacity
Retained Data Lifespan
Exploitability Results

Figure 4: The Overview Architecture of DR. D.

**Clustering Request Data Manipulation Methods.** After identifying all request classes, DR. D detects request data manipulation methods by extracting methods with parameters of the request class types. In addition to leveraging the identified request classes, DR. D also considers wrapper classes containing fields of the request class type.

Intuitively, we can analyze the invocation relationships among these methods and select methods with no callers as entry points. However, there exist intricate invocations among the request handlers due to Java polymorphism. Failing to partition these independent handlers will affect the effectiveness of data flow analysis and pose challenges for vulnerability validation. Thus, DR. D further locates and delineates request handlers among the request data manipulation methods.

The key insight here is that, despite variations in naming and functionality, request handlers exhibit common characteristics in design patterns for flexible configurability. Specifically, they are clusters of methods designed for different implementations of request processing tasks, e.g., form-based and OpenID-based authenticators for request authentication. To enable flexible configuration and unified invocation, methods in each cluster typically adhere to the same specification, e.g., implementing or overriding the same method. Additionally, the organization and management of these clusters follow common design patterns. As shown in Figure 5, we identify two management approaches by investigating how request handlers are configured and invoked. In centralized management, request handlers are stored in a queue based on developer configuration, with the manager invoking each handler sequentially (e.g., `doFilter()`). In decentralized management, the next handler is explicitly configured within the current one through its field, which invokes the subsequent handler.

Based on these characteristics, DR. D locates request handlers by clustering request data manipulation methods. DR. D first groups methods sharing the same subsignature (identical function names, parameters, and return values). Then, for each group, DR. D analyzes the invocation relationships. If these methods lack direct invocations but share the same caller, which is also their common callee, they are identified as utilizing centralized management. If direct mutual invocations exist between these methods and target the same field

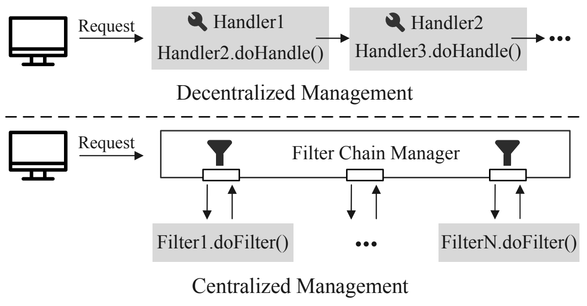


Figure 5: Two Request Handler Management Approaches.

within their class, they are identified as using decentralized management. These methods are designated as request handlers for independent request processing task analysis, and DR. D neglects the invocation edges pointing to them. Besides, DR. D collects public methods from request classes and the remaining request data manipulation methods that are not invoked by the identified request handlers, as web apps may directly invoke them to trigger implicit request processing.

## 3.2 Identifying Long-Lived Data

In this stage, DR. D identifies long-lived objects within web containers. While some long-lived objects can be identified through Java's syntactic features, such as static variables that persist in memory from class loading to program termination, many long-lived objects lack explicit characteristics. These objects are intricately linked to the web container's internal implementation, initialized during component startup, and held in reference by the container. They are immune to GC and are managed entirely by the container.

Identifying such container-managed long-lived objects poses a considerable challenge. While much research [32, 58, 62–64] has been conducted on modeling runtime object lifetimes, static analysis for heap/reference modeling in complex programs remains an open problem. This is particularly true in web containers, which involve numerous multi-threading, dynamic class loading, and reflection behaviors during startup. Although dynamic memory dump analysis can capture runtime object reference relationships, it is limited by code coverage issues, especially given the complex configurations in web containers. Additionally, mapping runtime object instances to static analysis is difficult due to the gap between data flows in the request processing and component startup. To address this challenge, DR. D employs a machine learning-based approach to classify Java classes that may exhibit long lifecycles. The key insight here is that in object-oriented languages like Java, each class is designed with a specialized responsibility, often determining whether its instances have long lifecycles during runtime. Therefore, DR. D can analyze the characteristics of a class to infer whether its instances exhibit a long lifecycle.

**Extracting Class Features.** Since there is no available long-lived class database, we manually construct a dataset for feature analysis and model training. Specifically, we collect classes in two ways. On one hand, we identify the container's startup entry point by analyzing its launch script, and then manually analyze its startup process. On the other hand, we run the web container with a basic configuration and dynamically analyze class instance information in memory using JProfiler [21]. For each class, three analysts with expertise in Java web development independently assess it as long-lived or short-lived. A label is assigned when at least two analysts have the same judgment (see detailed results in §4).

By comparing the long-lived and short-lived classes in our dataset, we make the following observations from four distinct perspectives: class instantiation, class utilization, inter-class relationships, and intra-class attributes:

- Long-lived classes tend to have fewer instances and instantiation operations are less likely to occur in frequently triggered request processing procedures.
- Long-lived class objects tend to exhibit lower mobility in function invocations.
- Long-lived classes are typically managed by the container and used to manage container resources, thus they are less referenced by other classes and have more references to other classes.
- Configurable classes are more likely to be long-lived.
- Classes that are not globally accessible are less likely to be long-lived, such as anonymous inner classes.
- Iterable classes often represent collections of data that can change frequently and are less likely to be long-lived.
- Long-lived classes are more likely to be used concurrently by multiple threads, leading to more concurrent access-protected fields and functions in the class.
- Long-lived classes tend to have a lower proportion of static methods considering isolation and concurrency.

Based on the above observations, we have identified 13 class features, detailed in Table 2. DR. D automatically extracts these features with specific details provided in Appendix A.1.

**Classifying Long-lived Classes.** Based on the above dataset and class features, we trained a binary classification model for class lifecycle prediction utilizing the widely used Random Forest [56] algorithm. This algorithm is chosen for its effectiveness in handling multiple features and reducing overfitting risks. Each class is represented as a vector of values corresponding to the 13 identified features, along with a label of '1' for long-lived classes or '0' for short-lived classes, before being fed into the model. During the analysis, DR. D automatically extracts features for each class within the container and utilizes the trained model for lifecycle prediction. For classes identified as long-lived, DR. D treats all their instances as long-lived objects, and their fields are considered as long-lived data. Additionally, DR. D identifies all static fields as long-lived, irrespective of their classes.

Table 2: List of Class Features

| No. | Class Feature Description |
|---|---|
| 1 | # of class instantiations |
| 2 | % of instances created during request processing |
| 3 | # of times as function parameter |
| 4 | # of times as function return value |
| 5 | # of times referenced by other classes |
| 6 | # of fields in class |
| 7 | Whether is configurable (True or False) |
| 8 | Whether is globally accessible (True or False) |
| 9 | Whether is iterable (True or False) |
| 10 | # of times maintained within iterable objects |
| 11 | # of fields can be accessed concurrently |
| 12 | # of methods can be accessed concurrently |
| 13 | % of static methods among all methods in class |

## 3.3 Detecting and Assessing Vulnerability

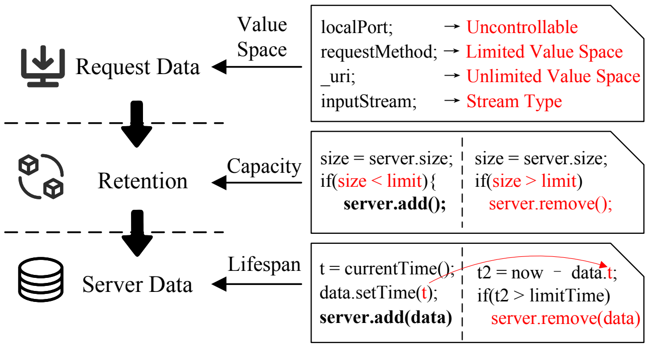
In this stage, DR. D detects vulnerable data retention and assesses their exploitability. Static analysis is effective in detecting taint-style vulnerabilities like data retention, but its high false positives introduce substantial overhead for developer verification. Fuzzing techniques can be used to validate static analysis results and reduce this overhead. However, they struggle to automatically understand and manage the intricate deployment configurations within web containers. For example, to verify a vulnerability in OpenID authentication, one must configure the module's activation within the web container, register a web app with third-party OpenID providers (e.g., Google Cloud), and define URLs that require authentication. Only after these configurations are properly set up can a client's request trigger the execution of the vulnerable code.

To address this, we propose a lightweight yet effective method to statically assess vulnerability exploitability with minimal human intervention. By modeling and analyzing the essential conditions for exploiting vulnerable data retention, DR. D categorizes and rates the exploitability of vulnerabilities through static analysis. This approach can effectively filter out unexploitable candidates, helping developers prioritize which vulnerabilities to validate.

**Detecting Vulnerable Data Retention.** Starting from the entry points identified in §3.1, DR. D performs precise context-, field-, and flow-sensitive data flow analysis with the request data as taint source. Specifically, DR. D considers two scenarios. On one hand, request data is retained in long-lived server data, leading to cumulative memory consumption. In this case, DR. D examines whether tainted data is appended to size-expandable objects referenced by long-lived classes. To identify size-expandable objects, we initially compile a list of Java native iterable classes (e.g., Map) with their addition/removal instructions and size properties from JDK documentation [34]. Then DR. D automatically identifies container-specific classes that extend or implement these native classes or interfaces through class hierarchy analysis. It determines their instructions and size properties based on the corresponding native classes or interfaces. Additionally, DR. D supports container-specific classes acting as wrappers around native classes (e.g., by using them as fields). In such cases, DR. D unfolds their function implementations to trace the underlying native operations during analysis. Furthermore, some classes implement a size property as an independent field. DR. D identifies such cases by analyzing the implementations of their element addition and removal functions to detect fields that increase or decrease correspondingly. To ascertain the longevity of a size-expandable object, DR. D scrutinizes its reference source, examining if it is a field within long-lived classes or referenced by such a field. On the other hand, request data is loaded into memory from the network stream without size limitation. Although developers often impose size limits during deserialization, we found that request parsing doesn't always happen during initial deserialization. To optimize server performance, the parsing of all request content, i.e., loading into memory, is deferred until specific request processing tasks are triggered. In this scenario, DR. D checks if tainted data can control the data loading loop. If tainted data influences a loop's termination condition, in which the tainted data is continually added to size-expandable objects, DR. D reports a potential vulnerability.

**Assessing Vulnerability Exploitability.** Data retention is exploitable only if attacker-controlled request data is retained in server data without size limitation and is not released. Hence, DR. D analyzes the exploit restrictions from three perspectives: request data value space, data retention capacity, and server data lifespan. The key insight here is that data retention is typically an essential part of request processing, so we can focus on analyzing the critical conditions that impact memory consumption statically, without grappling with runtime path and configuration constraints. Figure 6 illustrates how DR. D analyzes exploit restrictions for a specific data retention.

First, DR. D assesses the potential of request data to cause high memory consumption. Through field-sensitive data flow analysis, DR. D identifies which fields in the request class are retained. By manually annotating these fields, DR. D determines if attackers can control the retained data and if it has sufficient value space. Specifically, each field in the request classes is first annotated for client manipulability. For controllable fields, their value space is further annotated based on type and semantics. This is because request data may be retained in objects with implicit deduplication operations, such as Map keys, requiring attackers to generate diverse data for memory accumulation. Additionally, stream-type fields are annotated as they enable the container to load large amounts of data into memory. It is noteworthy that manual annotation requires only limited human effort since the number of request classes is limited and dependencies exist between fields in different request classes. In our experiments, the annotation for each container, on average, can be completed by an analyst within half an hour.



```
localPort;      → Uncontrollable
requestMethod;  → Limited Value Space
_uri;           → Unlimited Value Space
inputStream;    → Stream Type
```

```
size = server.size;  | size = server.size;
if(size < limit){    | if(size > limit)
    server.add();    |   server.remove();
```

```
t = currentTime();   | t2 = now - data.t;
data.setTime(t);     | if(t2 > limitTime)
server.add(data)     |   server.remove(data)
```

Figure 6: Exploit Restrictions Analysis For Data Retention.

Second, DR. D assesses capacity limitations for data retention operations. Specifically, for adding elements to size-expandable objects, DR. D scrutinizes the presence of size checks on data addition. Starting from a data addition instruction, DR. D first conducts backward analysis to identify all condition checks that inspect the object's size. If a condition check's alternative branch prevents the addition instruction from executing, DR. D identifies it as a capacity limit. Moreover, DR. D employs forward analysis to detect all subsequent size checks on the object. If element removal operations (e.g., remove()) occur after a condition check, DR. D also considers it as a capacity limit. For data loading within loops, DR. D analyzes tainted condition branches within the loop. If a condition check compares tainted data to an integar constant, and one of its branches results in an exception or loop exit, it is considered a capacity limit.

Third, DR. D examines the lifespan restrictions on retained data. Although the retained data is referenced by long-lived server data and is not subject to GC, web containers may periodically check its last accessed or saved time to remove outdated and inactive data. To identify lifespan restrictions, DR. D examines whether a timestamp record is associated with the request data when it is incorporated into the server data. Specifically, during data flow analysis, DR. D identifies all timestamp generation operations and labels them as a distinct taint type. If both the timestamp taint and request data

Table 3: Vulnerability Exploitability Level Assessment.

| Exploitability Level | Request Data Value Space | Retention Capacity | Server Data Lifespan |
|---|---|---|---|
| Unconstrained | Unlimited | Unlimited | Unlimited |
| Time-Constrained | Unlimited | Unlimited | Limited |
| Context-Constrained | Unlimited | Limited | — |
| Unexploitable | Limted | — | — |

taint propagate to the server data, DR. D considers there exists a lifespan restriction.

Based on the above analysis, DR. D rates each vulnerability candidate into 4 exploitability levels, as shown in Table 3.

- *Unconstrained.* In this category, client-controlled request data with a large value space is retained without capacity constraints, and the resulting server data lacks lifespan management. A real case is the example in §2.2.
- *Time-Constrained.* In this category, the lifespan of retained data is limited. Attackers must exhaust the container's memory within a limited time frame. We demonstrate that this type of attack is still practical (detailed in §5.2).
- *Context-Constrained.* When capacity limits are in place for data retention operations, attackers cannot exhaust the web container's memory resources. However, the limit is a double-edged sword and may be abused for DoS attacks, as exemplified by the size-based limit in Resin's form-based authentication feature (detailed in §6.1).
- *Unexploitable.* Data retention cannot be exploited if attackers cannot control the retained request data or fail to cause large memory consumption due to limited value space.

Leveraging these assessment results, developers can filter out unexploitable candidates and swiftly identify those worth further verification by analyzing their restrictions.

## 4 Implementation.

We implemented a prototype of DR. D with over 6,400 LOC new Java code for static analysis, built on the Soot framework [39], and 135 LOC Python code for machine learning, utilizing scikit-learn [37] for the random forest algorithm. DR. D was developed with Java 1.8.0_121 and Python 3.10.11, and it processes Java bytecode files (i.e., class files) extracted from the release builds of web containers as its inputs.

**Machine Learning Model Training.** Following the approach in §3.2, we collected a dataset comprising 634 long-lived and 732 short-lived classes from Tomcat and Jetty. We randomly selected 80% of the classes for training and the remaining 20% for testing. The model achieved 85.98% precision, 92.16% recall, and an F1 score of 0.89 on the testing set. Furthermore, to assess generalizability, we manually validated 200 randomly selected classes from Undertow and Resin, achieving precisions of 85.04% and 81.53% respectively.

**Data Flow Analysis.** We have incorporated several advanced techniques to improve the accuracy and efficiency of DR. D's data flow analysis. Firstly, for achieving field-sensitive taint tracking, we employ access paths [2] to represent fine-grained field taint information, effectively handling the nested structures of Java classes. In this context, DR. D recursively generates a new abstract memory object for each embedded field upon access, maintaining the relationship between the new object and its parent object. Following state-of-the-art static analysis works [2, 43], DR. D sets the access path length threshold to 5. This approach helps alleviate over-tainting issues and significantly reduces false positives. Secondly, rather than maintaining a binary "tainted or not" state, we introduce taint tags to record and propagate multi-dimensional taint attributes. This enhancement enables DR. D to perform precise analysis by querying taint information, such as taint sources and value space. Thirdly, we implement method summary-based analysis to prevent redundant analyses and enhance efficiency. This ensures that each method is analyzed only once within the same context.

## 5 Evaluation

In this section, we evaluate DR. D on four widely-used open-source Java web containers in their latest version. Specifically, we selected the top three most popular open-source containers from GitHub, including Apache Tomcat, Eclipse Jetty, and Red Hat Undertow. Furthermore, we searched for commercial web containers to see if they offer open-source counterparts and found Caucho Resin. We chose open-source web containers due to their publicly available community and extensive source code comments, which aid in understanding and testing their request processing. In contrast, closed-source web containers often pose challenges for both vulnerability identification and validation. The difficulty in accessing their source code, combined with a lack of transparency and accessible documentation, significantly limits researchers' ability to conduct detailed analyses. Furthermore, non-paying users typically have access only to restricted versions, such as those without security patches. This, along with limited technical support during deployment and configuration, creates additional barriers to validating identified vulnerabilities. The experiments are performed on a MacBook Pro running Sonoma 14.2.1 (Intel core i5 2GHz, 16GB RAM). On average, each container's analysis is completed within 3 minutes.

### 5.1 Effectiveness of DR. D

Table 4 summarizes DR. D's overall detection results. In stage 1, DR. D identifies 61 request classes, with Resin having the most since it designs dedicated request classes for various request processing scenarios and introduces some customized protocols. Based on these request classes, DR. D analyzes request data manipulation methods and ultimately identifies

Table 4: DR. D's detection results on 4 popular open-source Java web containers. Among the identified 88 potential exploitable DRDoS vulnerabilities, 28 are successfully verified, resulting in 25 unique vulnerabilities (detailed in Table 6).

| Web Container | Version | Stage 1 | | Stage 2 | Stage 3 | | | |
|---|---|---|---|---|---|---|---|---|
| | | Request Classes | Analysis Entry Points | Long-Lived Classes | Unconstrained | Time-Constrained | Context-Constrained | Unexploitable |
| Tomcat | 10.1.10 | 10 | 107 | 911 | 7 | 4 | 5 | 17 |
| Jetty | 11.0.15 | 10 | 206 | 547 | 10 | 9 | 4 | 16 |
| Undertow | 2.3.7 | 7 | 361 | 474 | 12 | 5 | 2 | 8 |
| Resin | 4.0.66 | 34 | 445 | 3,309 | 25 | 1 | 4 | 16 |

Table 5: Breakdown of Identified Request Handlers.

| Functionality | Tomcat | Jetty | Undertow | Resin |
|---|---|---|---|---|
| Security | 8 | 16 | 28 | 16 |
| Resource Management | 28 | 35 | 86 | 31 |
| Performance Optimization | 2 | 6 | 35 | 20 |
| Lifecycle Management | 3 | 46 | 32 | 3 |
| Filtering & Forwarding | 16 | 18 | 47 | 42 |
| **Sum** | 57 | 121 | 228 | 112 |

a total of 1,119 methods as entry points for request processing analysis, which consists of 518 request handlers and 601 public methods. In stage 2, DR. D identifies that, on average, 20.07% of the classes in each web container may have a prolonged lifecycle. More long-lived classes are recognized in Resin due to its larger project size and greater number of classes. Then DR. D collects 28,601 fields from these classes and another 20,782 static fields from other classes. These data, not subject to GC, require careful management by web containers. In stage 3, DR. D detects 145 vulnerable data retention candidates through data flow analysis. It then assesses the exploitability of each candidate, categorizing them into four levels: 54 unconstrained, 19 time-constrained, 15 context-constrained, and 57 unexploitable candidates.

**Request Handler Analysis.** In total, DR. D identified 518 request handlers across the four web containers. These handlers are designed for specific request processing tasks and require customized configurations for utilization. As shown in Table 5, Undertow offers greater configurability and richer functionalities, featuring up to 228 request handlers, while Tomcat has the fewest, with only 57 handlers. We further break down their functionalities into 6 categories, which are mainly used for resource management, request filtering and forwarding, and lifecycle management. Although different web containers incorporate request handlers with similar functionalities, the methods and strategies used in their implementations vary, leading to differences in their security protection measures. An example is the aforementioned form-based authentication request handler and more details are presented in §6.1.

**Vulnerability Validation.** Based on DR. D's detection results, we can swiftly sieve out vulnerability candidates worthy of further verification. First, unexploitable-level candidates are dismissed due to uncontrolled requests and data retention. Second, for context-constrained candidates, we examine the purpose of retained data. If it is used in a sensitive context, e.g., authentication, the candidate is kept for additional analysis. Third, for time-constrained candidates, we examine the lifespan and size of retained data to filter out those with low memory consumption capability. As a result, 63 candidates are selected for subsequent validation. To validate their exploitability, we manually deploy and configure the relevant request handlers in the web containers. Then we construct requests to test if the target data retention instructions can be executed and if they lead to the retention of large volumes of attack data. As a result, 28 of them are successfully verified, resulting in 25 unique vulnerabilities (detailed in Table 6).

**False Positive Analysis.** Overall, the false positive rate is 55.6% and we argue that it is practical for real-world use considering the number of detection outcomes. We further analyze their root causes and summarize as follows. First, 15 of them execute successfully but fail to retain significant data due to existence constraints on server-side data. For example, Tomcat's `CrawlerSessionManagerValve` stores the context path of URIs in a long-lived Map, but only those paths that are pre-registered by developers, even though the URI value space is controlled by the client. Such constraints are difficult to identify as they require understanding the semantics and state of server-side data. Second, 11 are false alarms caused by long-lived class classification. Third, 5 of them stem from the over-approximation of data flow analysis. For example, if an element in a collection 'A' is tainted, all elements in the collection 'A' are considered tainted, resulting in excessive data flow propagation. Fourth, 4 of them cannot be exploited due to size constraints that are difficult to detect, such as asynchronous data-consuming operations.

**False Negative Analysis.** Since DR. D is the first systematic study of DRDoS vulnerabilities, there is no benchmark available to evaluate the false negatives. After manually analyzing 475 web container vulnerabilities on CVE, we found only two known DRDoS vulnerabilities. Both of them can be detected by applying DR. D on corresponding web containers.

## 5.2 Vulnerability Analysis

Table 6 presents the details of 25 verified unique exploitable DRDoS vulnerabilities. We further conduct some analysis to gain more insights into them.

**Retained Request Data.** We first analyze the types of request data retained within the web containers. Among the 25 vulnerabilities, 11 involve request URIs, 16 involve request bodies, and only 3 involve data from request headers. In addition, to exploit these vulnerabilities, 2 require urlencoded-type attack requests, 4 require multipart-type requests, 3 require XML-type requests, while the rest have no specific requirements.

**Long-lived Container Data.** We also analyze where the request data is stored within the containers. The results show that out of 25 vulnerabilities, only 2 of them are stored in static variables, while the rest are stored in long-lived data identified by machine learning.

**Data Retention Purposes.** We then analyze the purposes of these vulnerable data retentions, as outlined in Table 6. In detail, 5 instances are utilized for security authentication, 5 for resource management, 3 for access analytics, 3 for specialized request parsing, 2 for performance optimization, and 5 exist in on-demand request content loading. Additionally, 2 retentions intended for network-layer DoS protection can be abused to launch DRDoS attacks. They have garnered significant attention from developers, who rated them high severity (detailed in §6.2).

**Attack Performance.** We investigate the attack performance of these vulnerabilities from the perspective of data retention time and payload size. Specifically, 11 vulnerabilities have no data retention time and payload size restrictions. Furthermore, 7 vulnerabilities are designed with time-based constraints. Among them, 5 have an infinite default retention time set, 1 can be controlled by client requests with a maximum of 168 hours (#2), and the remaining 1 has a fixed data retention time of 120 minutes (#1). Additionally, 14 vulnerabilities have size limitations on the payload carried by a single request. This is because web containers set size constraints on different parts of the Request, e.g., URL and headers. In detail, 2 of them are set at 200KB, 2 at 16KB, 2 at 8KB, and the remaining 8 at 4KB. Although the individual payload size is limited, these data persistently exist in the web container without being released, making them easily exploitable by attackers.

**Data Retention Management Challenges.** We further analyze why web containers do not impose strict restrictions on these data retentions. Unfortunately, we find that balancing security and availability makes this a challenging task for web containers. Even worse, the transparency of upper-layer web apps limits the container's ability to robustly check the retained data. For example, in cases where request URIs are retained, the web container lacks complete knowledge of all valid URI information within the upper-layer web apps. Therefore, it cannot accurately determine which URIs are malicious when storing them. While some containers (e.g., Undertow) attempt to analyze the response code corresponding to a request to assess URI legitimacy, attackers can still bypass security checks by adding numerous illegal query parameters to valid URIs. Furthermore, when setting size and time limitations, selecting an appropriate threshold requires the web container to understand client usage patterns and the business requirements of web apps. This goes beyond the capabilities of web containers.

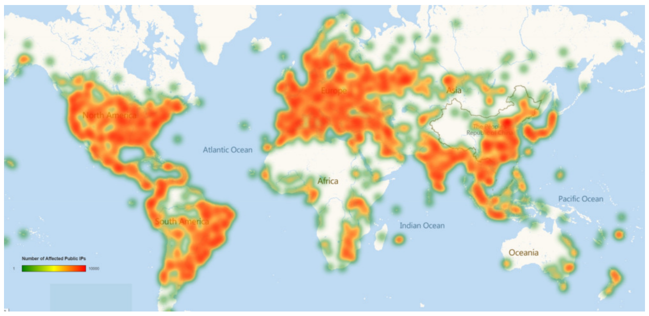


Figure 7: Geographical Distribution of Affected Public IPs.

**Real-World Affected Network Assets.** Web containers, as fundamental infrastructure components, are extensively employed in web development and are default integrations in many popular frameworks like Spring Boot. Thus, their vulnerabilities can affect a large number of web applications. To assess potentially affected network assets in the real world, we leverage Shodan's publicly available scan results (from 2024.4.20 to 2024.5.20) to identify server IPs associated with vulnerable web container versions. The results unveil that a total of over 1.5 million IPs are potentially affected by the vulnerabilities we discovered. Figure 7 presents their geographical distribution, with the majority originating from China, the United States, and Japan.

## 5.3 Cloud-based Attack Evaluation

To demonstrate the practicality of DRDoS attacks in real-world scenarios, we deployed our *own* vulnerable web containers and applications on popular cloud service platforms. Then we conducted attack tests to assess their effectiveness and the impact of the attacks under realistic conditions.

Specifically, we conducted attack experiments on four major cloud platforms: Google Cloud, Microsoft Azure, Alibaba Cloud, and Huawei Cloud. To ensure consistency across experimental environments, we selected equivalent instance types (commonly referred to as SKUs) on all platforms. Table 7 in Appendix A.2 lists the specific configurations of each virtual machine (VM) instance. Then we deployed web containers on these VMs and developed web applications to support testing various request handlers. The versions of web containers used in our experiments align with those listed in Table 4 and the memory resource limit for each container process is set to 2 GB via JVM options. Furthermore, we enabled the

Table 6: The Unique Exploitable Data Retention Vulnerabilities Exposed By DR. D.

| ID | Web Container | Method with Vulnerable Data Retention | Functionality | Exploitability Level | Payload Size[†] | Status |
|---|---|---|---|---|---|---|
| 1 | Tomcat | FormAuthenticator.saveRequest() | Form-based Authentication | Time-Constrained | 4KB | Confirmed |
| 2 | | WebdavServlet.doLock() | Resource Lock Management | Time-Constrained | 4KB | Confirmed |
| 3 | Jetty | FormAuthenticator.validateRequest() | Form-based Authentication | Time-Constrained | 200KB | CVE Assigned |
| 4 | | OpenIdAuthenticator.getChallengeUri() | OpenId Authentication | Time-Constrained | 200KB | Confirmed |
| 5 | | PushSessionCacheFilter.doFilter()-1 | Session-Scoped Server Push | Unconstrained | 4KB | Confirmed |
| 6 | | PushSessionCacheFilter.doFilter()-2 | Access Time Logging | Time-Constrained | 4KB | Confirmed |
| 7 | | DoSFilter.getRateTracker() | DoS Protection | Time-Constrained | 4KB | CVE Assigned |
| 8 | | PushSessionCacheFilter$1.requestDestroyed() | Referer Analytics | Unconstrained | 4KB | Confirmed |
| 9 | | ThreadLimitHandler.getRemote() | DoS Protection | Unconstrained | 8KB | CVE Assigned |
| 10 | | PushCacheFilter.doFilter() | HTTP/2 Server Push | Unconstrained | 4KB | Confirmed |
| 11 | Undertow | FormAuthenticationMechanism.sendChallenge() | Form-based Authentication | Time-Constrained | 16KB | CVE Assigned |
| 12 | | HttpServletRequestImpl.loadParts() | Multipart Content Loading | Unconstrained | Unlimited | CVE Assigned |
| 13 | | PushCompletionListener.exchangeEvent() | Referer Analytics | Unconstrained | 4KB | CVE Assigned |
| 14 | | FormEncodedDataParser.doParse() | Form-Encoded Data Parsing | Unconstrained | Unlimited | CVE Assigned |
| 15 | | MultiPartUploadHandler.parseBlocking() | Multipart Data Parsing | Unconstrained | Unlimited | CVE Assigned |
| 16 | Resin | FormLogin.loginChallenge() | Form-based Authentication | Context-Constrained | 8KB | Confirmed |
| 17 | | Encoding.getMimeName() | MIME Encoding Management | Unconstrained | Unlimited | CVE Assigned |
| 18 | | Encoding.getJavaName() | Java Encoding Management | Unconstrained | 16KB | CVE Assigned |
| 19 | | MemcachedConnection.handleSingleRequest() | Hmux Request Handling | Unconstrained | Unlimited | CVE Assigned |
| 20 | | DeleteCommand.execute() | Cache Delete | Unconstrained | Unlimited | CVE Assigned |
| 21 | | IncrementCommand.execute() | Cache Value Increment | Unconstrained | Unlimited | CVE Assigned |
| 22 | | MultipartFormParser.parsePostData() | Multipart Data Parsing | Unconstrained | Unlimited | CVE Assigned |
| 23 | | Form.parsePostData() | Form-Encoded Data Parsing | Unconstrained | Unlimited | CVE Assigned |
| 24 | | XmlParser.parsePITail() | XML PI Parsing | Unconstrained | Unlimited | CVE Assigned |
| 25 | | XmlParser.parseValue() | XML Attribute Value Parsing | Unconstrained | Unlimited | CVE Assigned |

[†] The payload size is largely equivalent to the size of attacker-controlled data, with minor discrepancies from essential content required for valid requests.

standard version of DDoS protection mechanisms on each cloud platform (more details are provided in Appendix A.3).

The attacks were initiated from a desktop computer equipped with an Intel Core i7 2.1GHz CPU, 16 GB of RAM, and a wired network interface card. To enhance attack efficiency, we leveraged multithreading (500 threads) for vulnerabilities with payload size limitations to increase concurrency. The same exploit was used for each vulnerability across different platforms. As a result, all attacks were successfully executed without exceeding the traffic thresholds of any platform or triggering any alerts. Figure 9 in Appendix A.4 shows the heap memory usage of web container processes over time on different platforms, with subplot IDs corresponding to vulnerability IDs in Table 6.

It can be observed that memory usage fluctuates during its increase, which is primarily due to Java's garbage collection mechanism. During each request processing, the web container generates many temporary data objects which, unlike attack data, will be actively reclaimed by the garbage collector at specific intervals. This leads to temporary reductions in memory usage. The fluctuations tend to be more pronounced in exploitations with smaller data retention, as the temporary data generated during request processing significantly exceeds the size of the retained data. Furthermore, we observed that the time required for an attack can vary significantly, even with identical payload sizes. This variation is influenced by three key factors. First, the data amplification effect. The actual size of the resident data may exceed the payload size because additional uncontrollable data can be stored during request processing. For instance, in vulnerability 1, each attack request stores not only the payload but also creates a new session object for additional client information. Second, the efficiency of server-side request processing can affect the attack request rate. For example, exploiting vulnerability 9 triggers exception handling, substantially increasing server execution overhead and thereby reducing the server's request processing throughput. Third, the network communication quality. While we have made efforts to deploy VMs in locations as geographically close as possible across platforms, they are not identical. These differences impact the network communication speed between the target server and the attacker's client. As a result, even with the same client configuration and exploit, the time required for an attack can vary significantly.

We further analyze why existing DDoS defenses are ineffective in defending against DRDoS attacks. Typically, DDoS defenses rely on pattern-based methods to identify and filter attack traffic, monitoring factors like request rates, content, and access patterns. However, these methods prove insufficient against DRDoS attacks. First, DRDoS attacks do not necessitate a continuous stream of attack requests and can be executed at a low and inconspicuous access rate. Second, DRDoS attacks exploit design flaws in data management and leverage the expected functionalities of web containers. Thus, the attack requests lack any malicious content, such as escaped characters, making them challenging to distinguish from routine business requests through content analysis. Third, each DRDoS attack request is self-contained and complete, devoid of problematic access patterns like semi-connections. Although cloud platforms have implemented absolute limits on request traffic volume and access frequency, these thresholds are inadequately set for DRDoS attacks. This is because existing limits are designed to mitigate traditional network-layer DoS attacks, which rely on a substantial volume of continuous and uninterrupted attack traffic.

## 6 Case Study

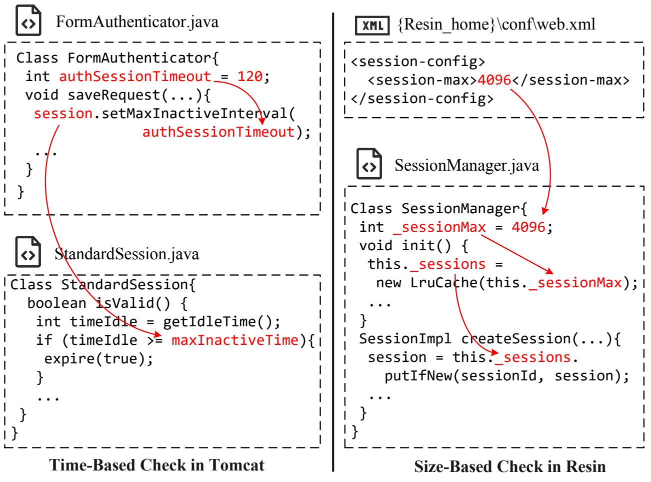
### 6.1 Exploiting Form-Based Authentication

Form-based authentication is a fundamental and widely used authentication method [31]. It utilizes an HTML form to transmit user credentials to the server. When an unauthenticated client requests a protected resource, the server issues an HTTP 302 redirect to a developer-specified login page. Then, the user inputs and submits credentials. Upon successful authentication, the server redirects back to the original URI with an authentication cookie. Our experiments show that all four web containers implement form-based authentication and all of them are vulnerable to DRDoS attacks.

Specifically, to ensure correct access to the original resource after authentication, the web container retains the user's original request until authentication is completed. This raises security concerns: what if the client requests protected resources but does not complete the authentication? We observe that web containers store the original request in the session for an extended period. Although each session stores only one unauthenticated request, an attacker can easily create new sessions by modifying the request content. Consequently, a malicious client can craft numerous different requests, exhausting the web container's memory resources.

**Distinct Security Considerations.** These containers employ a similar design to implement the authentication functionality, i.e., storing the same request data, but with varying levels of security measures in place to manage retained data. Specifically, Jetty and Undertow do not impose dedicated restrictions on this feature. They merely apply generic constraints on the size of individual requests, set at 200KB and 16KB, respectively. In contrast, Tomcat and Resin have implemented more stringent protections. As illustrated in Figure 8, Tomcat has specifically configured a size limit (4KB) for unauthenticated requests and retention time constraints (i.e., 120 minutes). Resin goes a step further by restricting the number of unauthenticated requests allowed for retention to 4096.

**Exploitation in Real World.** Despite these mitigation measures, they are insufficient to defend against DRDoS attacks. In particular, Jetty and Undertow lack constraints on data re-



```
Class FormAuthenticator{
 int authSessionTimeout = 120;
 void saveRequest(...){
  session.setMaxInactiveInterval(
                 authSessionTimeout);
   ...
 }
}
```

```
Class StandardSession{
  boolean isValid() {
   int timeIdle = getIdleTime();
   if (timeIdle >= maxInactiveTime){
    expire(true);
   }
   ...
 }
}
```

```
<session-config>
  <session-max>4096</session-max>
</session-config>
```

```
Class SessionManager{
 int _sessionMax = 4096;
 void init() {
  this._sessions =
   new LruCache(this._sessionMax);
  ...
 }
 SessionImpl createSession(...){
  session = this._sessions.
    putIfNew(sessionId, session);
  ...
 }
}
```

Figure 8: Distinct Security Checks in Tomcat and Resin.

tention time, allowing attackers to freely send attack requests and accumulate memory consumption, leading to memory exhaustion and server unavailable to other users. Although Tomcat has set a data retention time limit (120 minutes), this threshold proves ineffective in preventing real-world attacks. According to our experiments, a malicious client can exploit this vulnerability to occupy over 180GB of server memory, significantly impacting numerous web applications in the wild. Furthermore, while Resin mitigates memory exhaustion by limiting the quantity of retained requests, it remains vulnerable to DoS attacks exploiting data semantics. Specifically, the original request is stored in the session to redirect the user to the protected resource after authentication and update the authentication status. Attackers can exploit Resin's quantity limitation by sending a substantial number of requests when a user enters credentials, causing the victim's session and original request to be removed. This prevents the user from completing authentication and accessing the target resource.

**Disclosure and Mitigation.** We responsibly disclosed the vulnerabilities to developers, who acknowledged our findings. Eclipse and Red Hat have assigned 2 CVE identifiers for them. We are actively collaborating with developers on mitigation strategies. Tomcat developers express the challenge of eliminating DRDoS risks while maintaining the desired functionality. As a patch, they reduce the data retention time from 120 minutes to 2 minutes to limit attackers' capabilities. Meanwhile, Jetty developers, recognizing the limitations of time and size-based approaches in fully mitigating DRDoS risks, are working on optimizing the form-based authentication feature to minimize retained data.

### 6.2 Exploiting DoS Defense Module

`DoSFilter` is a request handler in Jetty designed to prevent abuse from request flooding by tracking the number of requests from each client per second. If the limit is exceeded, the request will be rejected, delayed, or throttled [13]. To implement rate tracking, it uses the sessionId to identify each client and creates a RateTracker object for each client to store relevant information about their access. Although the sessionId is generated by Jetty and undergoes validation before retention, an attacker can easily generate numerous valid sessionIds by constructing different request contents, leading to Jetty creating numerous RateTracker objects. These objects are stored in a long-lived Map object and are not actively released by default. As a result, the request handler intended for network-layer DoS protection suffers from a DRDoS attack.

We have reported this vulnerability to Jetty developers, who expressed a keen interest in our findings. They promptly took corrective action, modifying the rate tracking implementation to only support client identifiers that are difficult for attackers to forge on a large scale, e.g., IP addresses. They mentioned that the `DoSFilter` was originally designed for DoS protection but introduced a new DoS attack vector. As a result, the developers assessed this vulnerability as high severity and assigned a CVE identifier for it.

## 7 Mitigation, Limitation, and Discussion

**Mitigation.** In client-server architectures, retaining client data on servers is often essential for certain functionalities. To mitigate DRDoS risks, developers should implement data retention cautiously and design robust management strategies. On one hand, during data retention implementation, it is advisable to retain data with limited value space or data that is difficult for attackers to forge on a large scale. Combining this with deduplication checks can restrict the quantity of retained data and memory usage. Additionally, it is essential to ensure end-to-end data validation and set size limits based on intended use and scenarios. On the other hand, for retained data, developers should design efficient management strategies. For non-sensitive data, such as that used for performance enhancement, developers can set capacity limits and employ a least recently used (LRU) management approach to reclaim memory resources promptly. For purpose-sensitive data, where removal could impact functionality, custom considerations are necessary. Developers should explicitly warn users of associated risks in documentation, providing flexible size- and time-based management strategies. This allows users to create tailored management policies for their specific production scenarios.

Besides, some standalone mitigation mechanisms, while unable to eliminate the root cause of memory exhaustion, can help mitigate its immediate impacts. For example, deploying OOM killers can free up resources by terminating and restarting memory-intensive web container processes. On the one hand, developers can leverage JVM-provided programming interfaces, e.g., JMX, to design independent monitoring processes that track the memory usage of the target process. When memory usage reaches the JVM limit and triggers an OOM error, the monitoring process can proactively terminate and restart the target process. On the other hand, in scenarios where the entire system runs out of memory, developers can utilize the OS-level OOM killer by configuring the *oom_score* for the target process, ensuring it is prioritized for termination.[1] Additionally, network defenses that flag clients with excessive traffic can increase the attacker's hardware or time costs, especially in vulnerability exploits with a limited attack time window. However, the thresholds must be tailored to the specific business requirements of the protected web services and the available resource capacity, balancing operational needs and security considerations.

**Limitations.** We discuss three limitations of DR. D: (i) self-developed expandable data types, (ii) potential bias of machine learning model, and (iii) manual work.

- First, although DR. D supports the automatic detection of container-customized expandable data types derived from JDK implementations, there may still be self-developed expandable classes. Nevertheless, our manual investigation revealed that over 90% of the expandable data types in these containers are inherited or dependent on JDK ones, such as `MultiMap` in Jetty and `LRUCache` in Undertow, which can be covered by DR. D. Moreover, DR. D can be easily extended to incorporate specified expandable data types.
- Second, DR. D considers only syntactic features when identifying long-lived classes. Introducing semantic information such as class names and comments might improve the identification efficiency. Nevertheless, we argue that DR. D has demonstrated good effectiveness in detecting DRDoS vulnerabilities, as evidenced by the experimental results.
- Third, DR. D requires manual effort for vulnerability assessment and verification. However, the request class annotation is worthwhile, requiring only about half an hour while enabling DR. D to filter out a significant portion (39%) of non-exploitable candidates from the analysis results. Moreover, manual vulnerability verification is common in static vulnerability detection tasks. Each request handler in web containers requires specific deployment configurations, and automating the inference and generation of these configurations remains an ongoing challenge, which is beyond the scope of this paper.

**Discussion.** This paper focuses on Java web containers due to their critical role in web development. DRDoS vulnerabilities may also exist in other web components, and DR. D can be extended to support these cases or provide design insights for targeted solutions. First, DR. D can be directly applied to other Java web containers. For instance, we applied it to the popular closed-source commercial WebLogic (restricted developer version). WebLogic includes extensive features and complex dependencies, with technical support limited

[1]Excessive swapping between RAM and disk-based swap space may occur, causing high disk I/O and thrashing, undermining the OOM Killer's effectiveness and exacerbating system-wide resource starvation.

to product purchasers, making comprehensive vulnerability validation challenging. Nevertheless, we have successfully verified one DRDoS vulnerability, which Oracle confirmed and assigned a CVE identifier. Second, DR.D can be extended to support general Java web frameworks (e.g., Spring MVC) and analyze web apps by extending the request handler recognition module to support framework-specific features, such as annotation-based request data mapping. Furthermore, DR.D's framework and design approach offer valuable insights for detecting DRDoS vulnerabilities in other web components or those built in different programming languages. This is particularly true for vulnerability modeling and exploitability assessment, given that web components often involve complex internal configurations and external dependencies.

Overall, our study highlights that, even with automated memory management languages, effective data retention management is essential for ensuring program performance and availability. This paper takes the first step in addressing this problem, and further research in this area is warranted, including but not limited to enhancing DRDoS vulnerability detection techniques and developing targeted defensive measures. We leave these for future work.

## 8 Related Work

**Network-Layer DoS Attacks.** The network-layer DoS attacks, also known as DDoS attacks, involve flooding the target system with a large volume of requests within a short time. This category of attacks has been extensively studied, yielding numerous classic attack approaches such as TearDrop [54], Ping of Death [52], and amplification attacks [1, 7, 22, 23, 36]. Fortunately for defenders, network-layer DoS attacks incur high attack costs (e.g., acquiring a large-size botnet to mount the attack) and exhibit noticeable traffic patterns. Many effective methods for detecting and mitigating them have been proposed, including CDNs [20], rule-based filters [17, 60, 61], and queue/congestion management [18, 27]. However, In this paper, we focus on specific vulnerabilities within containers during request processing, rather than at the network layer. This makes our attack requests challenging to distinguish from routine business requests. Furthermore, our attack exploits data retention within containers, allowing for flexible execution by sending attack requests at a low frequency. As a result, our attack does not exhibit problematic network-layer patterns, making existing network-layer DoS detection and defense approaches ineffective.

**Application-Layer DoS Attacks.** Application-layer DoS attacks have garnered substantial attention in recent years. They exploit specific implementation flaws within applications and utilize low-bandwidth, highly targeted, and application-specific traffic to overwhelm a target system [3, 9, 19, 24]. However, existing research primarily focuses on vulnerabilities leading to CPU resource exhaustion, such as ReDoS [26, 28, 40, 53] and algorithmic complexity (AC) vulnerabilities [25, 30, 33, 35, 48, 51], and is hardly applied to address DRDoS vulnerabilities in web containers. Specifically, some studies [11, 28, 42] focus on modeling and detecting specific types of AC vulnerabilities, which have limited application scope. Others [4, 33, 35, 48] apply general approaches such as fuzzing to obtain runtime statistics on resource (e.g., CPU time) consumption triggered by individual inputs. However, these tools are insensitive to memory consumption. Moreover, they struggle to handle the complex configuration constraints necessary for executing various request handlers within web containers. In this paper, we conduct the first systematic study of memory-oriented application-layer DoS attacks and propose an automated approach for detecting and assessing such vulnerabilities in web containers.

**Memory Leak Detection.** Many studies [5, 16, 41, 47, 50] focus on detecting memory leaks in C/C++ programs, employing static or dynamic methods to identify memory objects that are unreferenced and not explicitly released by developers. However, such vulnerabilities do not exist in Java, a memory-safe language, as Java's garbage collector will automatically reclaim unreferenced objects. Some research [29, 59] aims to detect 'memory leaks' in Java programs through runtime heap analysis. They detected vulnerabilities by locating unneeded but unreleased data in memory, which is not suitable for DRDoS vulnerabilities. This is because the request data are designed to be stored in the web container's memory for desired functionalities and future use, and therefore cannot be considered as unneeded objects.

## 9 Conclusion

In this paper, we conduct the first systematic study of DRDoS vulnerabilities in web containers and design a novel static approach, named DR.D, to detect and assess their exploitability. Our evaluation of DR.D on 4 popular Java web containers uncovers 25 unique zero-day, exploitable vulnerabilities. We responsibly disclosed them to respective developers and received their confirmations and acknowledgments. So far, we have received seventeen CVE identifiers (three assigned with high severity). We further analyze the Shodan's scan results and find over 1.5 million public IP addresses hosting these vulnerable web containers. Our study highlights that data retention management is crucial even in languages with automated memory management and warrants further research.

## Acknowledgement

We would like to thank the anonymous shepherd and reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62102093, U2436207, 62172105, 62202106, 62302101, 62172104, 62102091, 62472096, 62402114, 62402116). Min Yang is the corresponding author and a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China. Yuan Zhang was supported in part by the Shanghai Pilot Program for Basic Research - FuDan University 21TQ1400100 (21TQ012). Yinzhi Cao was supported in part by National Science Foundation (NSF) under grants CNS-21-54404 and CNS-20-46361 and a Defense Advanced Research Projects Agency (DARPA) Young Faculty Award (YFA) under Grant Agreement D22AP00137-00. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF or DARPA.

## Ethics Considerations

We discuss ethics considerations from three aspects:

**Responsible disclosure of vulnerabilities.** We responsibly disclosed all vulnerabilities to their developers and received many confirmations and even fixes. We gave developers enough time (i.e., 90 days) before making them public.

**Analysis of DoS defenses on public cloud.** The target is our *own* application deployed on web containers hosted on the public cloud. We did not attack any real-world applications or the public cloud directly.

**Public scanning of vulnerable IP addresses.** Our analysis only involves identifying the deployed web containers and their versions based on publicly available scan results from Shodan. We did not send *any* active attacks to those IP addresses.

## Open Science

To facilitate further research, we open-source the code, dataset, and pre-trained model of DR. D at `https://zenodo.org/records/14723606`.

## References

[1] M. Anagnostopoulos, G. Kambourakis, P. Kopanos, G. Louloudakis, and S. Gritzalis, "Dns amplification attack revisited," *Computers & Security*, vol. 39, pp. 475–485, 2013.

[2] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[3] U. Ben-Porat, A. Bremler-Barr, and H. Levy, "Vulnerability of network mechanisms to sophisticated ddos attacks," *IEEE Transactions on Computers*, vol. 62, no. 5, pp. 1031–1043, 2012.

[4] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele, "Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing," 2023.

[5] S. Cao, X. Sun, L. Bo, R. Wu, B. Li, and C. Tao, "Mvd: memory-related vulnerability detection based on flow-sensitive graph neural networks," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1456–1468.

[6] Caucho, "Resin watchdog system." https://www.caucho.com/resin-4.0/admin/health-watchdog.xtp, December, 2024.

[7] Cloudflare, "Dns amplification attacks," https://www.cloudflare.com/zh-cn/learning/ddos/dns-amplification-ddos-attack/, October, 2023.

[8] S. Crosby, "Denial of service through regular expressions." Washington, D.C.: USENIX Association, Aug. 2003.

[9] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[10] J. C. Davis, L. G. Michael IV, C. A. Coghlan, F. Servant, and D. Lee, "Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 443–454.

[11] J. Dietrich, K. Jezek, S. Rasheed, A. Tahir, and A. Potanin, "Evil pickles: Dos attacks based on object-graph engineering," in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[12] Eclipse, "Eclipse jetty canonical repository," https://github.com/eclipse/jetty.project, October, 2023.

[13] ——, "Jetty dosfilter," https://archive.eclipse.org/jetty/9.0.0.RC0/apidocs/org/eclipse/jetty/servlets/DoSFilter.html, September, 2023.

[14] J. EE, "Jakarta ee specification," https://jakarta.ee/specifications/, October, 2023.

[15] C. W. Enumeration, "Cwe-400: Uncontrolled resource consumption," https://cwe.mitre.org/data/definitions/400.html, September, 2023.

[16] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, and C. Zhang, "Smoke: scalable path-sensitive memory leak detection for millions of lines of code," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 72–82.

[17] P. Ferguson and D. Senie, "Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing," 1998.

[18] X. Fu and E. Modiano, "Fundamental limits of volume-based network dos attacks," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 3, pp. 1–36, 2019.

[19] M. Guirguis, A. Bestavros, I. Matta, and Y. Zhang, "Reduction of quality (roq) attacks on internet end-systems," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 2. IEEE, 2005, pp. 1362–1372.

[20] M. Jonker, A. Sperotto, R. van Rijswijk-Deij, R. Sadre, and A. Pras, "Measuring the adoption of ddos protection services," in *Proceedings of the 2016 Internet Measurement Conference*, 2016, pp. 279–285.

[21] T. JProfile, "Jprofile," https://www.ej-technologies.com/products/jprofiler/overview.html, September, 2023.

[22] G. Kambourakis, T. Moschos, D. Geneiatakis, and S. Gritzalis, "A fair solution to dns amplification attacks," in *Second International Workshop on Digital Forensics and Incident Analysis (WDFIA 2007)*. IEEE, 2007, pp. 38–47.

[23] ——, "Detecting dns amplification attacks," in *Critical Information Infrastructures Security: Second International Workshop, CRITIS 2007, Málaga, Spain, October 3-5, 2007. Revised Papers 2*. Springer, 2008, pp. 185–196.

[24] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, 2003, pp. 75–86.

[25] C. Lemieux, R. Padhye, K. Sen, and D. Song, "Perffuzz: Automatically generating pathological inputs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 254–265.

[26] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S.-C. Cheung, "Redoshunter: A combined static and dynamic approach for regular expression dos detection," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3847–3864.

[27] X. Liu, X. Yang, and Y. Xia, "Netfence: preventing internet denial of service from inside out," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, pp. 255–266, 2010.

[28] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1468–1484.

[29] C. Lou, C. Chen, P. Huang, Y. Dang, S. Qin, X. Yang, X. Li, Q. Lin, and M. Chintalapati, "{RESIN}: A holistic service for dealing with memory leaks in production cloud infrastructure," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 109–125.

[30] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 393–410.

[31] Microsoft, "Forms authentication," https://learn.microsoft.com/en-us/aspnet/web-api/overview/security/forms-authentication, September, 2023.

[32] K. Nguyen, K. Wang, Y. Bu, L. Fang, and G. Xu, "Understanding and combating memory bloat in managed data-intensive systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 26, no. 4, pp. 1–41, 2018.

[33] Y. Noller, R. Kersten, and C. S. Păsăreanu, "Badger: complexity analysis with fuzzing and symbolic execution," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 322–332.

[34] Oracle, "Java platform, standard edition & java development kit version 17 api specification," https://docs.oracle.com/en/java/javase/17/docs/api/index.html, December, 2024.

[35] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2155–2168.

[36] F. J. Ryba, M. Orlinski, M. Wählisch, C. Rossow, and T. C. Schmidt, "Amplification and drdos attack defense–a survey and new perspectives," *arXiv preprint arXiv:1505.07892*, 2015.

[37] scikit learn, "scikit-learn," https://scikit-learn.org/stable/, September, 2023.

[38] Shodan, "Shodan," https://www.shodan.io/, November, 2023.

[39] Soot, "Soot," https://soot-oss.github.io/soot/, September, 2023.

[40] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 361–376.

[41] K. Suzuki, T. Kubota, and K. Kono, "Detecting struct member-related memory leaks using error code analysis in linux kernel," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 329–335.

[42] L. D. Toffola, M. Pradel, and T. R. Gross, "Synthesizing programs that expose performance bottlenecks," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018, pp. 314–326.

[43] O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings 16*. Springer, 2013, pp. 210–225.

[44] B. Van Der Merwe, N. Weideman, and M. Berglund, "Turning evil regexes harmless," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, 2017, pp. 1–10.

[45] C. Vulnerabilities and Exposures, "Cve-2016-9589," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-9589, September, 2023.

[46] ——, "Cve-2023-26048," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26048, September, 2023.

[47] W. Wang, "Mlee: Effective detection of memory leaks on early-exit paths in os kernels," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 31–45.

[48] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 213–223.

[49] N. Weideman, B. Van Der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of nfa," in *Implementation and Application of Automata: 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings 21*. Springer, 2016, pp. 322–334.

[50] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, "Memlock: Memory usage guided fuzzing," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 765–777.

[51] Wikipedia, "Algorithmic complexity attack," https://en.wikipedia.org/wiki/Algorithmic_complexity_attack, October, 2023.

[52] ——, "Ping of death attack," https://en.wikipedia.org/wiki/Ping_of_death, October, 2023.

[53] ——, "Redos," https://en.wikipedia.org/wiki/ReDoS, October, 2023.

[54] ——, "Teardrop attack," https://en.wikipedia.org/wiki/Denial-of-service_attack#Teardrop_attacks, October, 2023.

[55] ——, "Distributed denial-of-service," https://en.wikipedia.org/wiki/Denial-of-service_attack, September, 2023.

[56] ——, "Randoeforest," https://en.wikipedia.org/wiki/Random_forest, September, 2023.

[57] ——, "Tcp syn flood," https://en.wikipedia.org/wiki/SYN_flood, September, 2023.

[58] G. Xu, "Resurrector: A tunable object lifetime profiling technique for optimizing real-world programs," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 111–130.

[59] G. Xu and A. Rountev, "Precise memory leak detection for java software using container profiling," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, pp. 1–28, 2013.

[60] A. Yaar, A. Perrig, and D. Song, "Siff: A stateless internet flow filter to mitigate ddos flooding attacks," in *IEEE*

*Symposium on Security and Privacy, 2004. Proceedings. 2004*. IEEE, 2004, pp. 130–143.

[61] ——, “Stackpi: New packet marking and filtering mechanisms for ddos and ip spoofing defense,” *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 10, pp. 1853–1863, 2006.

[62] D. Yan, G. Xu, and A. Rountev, “Uncovering performance problems in java applications with reference propagation profiling,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 134–144.

[63] D. Yan, G. Xu, S. Yang, and A. Rountev, “Leakchecker: Practical static memory leak detection for managed languages,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2014, pp. 87–97.

[64] D. Yan, S. Yang, and A. Rountev, “Systematic testing for resource leaks in android applications,” in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 411–420.

## A Appendix

### A.1 Class Feature Extraction

DR. D extracts class features through static analysis. Specifically, DR. D scans all methods in the web container to analyze the invocation times and locations (i.e., methods) of constructors for each class, facilitating the extraction of class instantiation-related features (#1 and #2). Subsequently, DR. D collects the parameter types and return value types of all methods to analyze their mobility features (#3 and #4). For inter-class reference features (#5 and #6), DR. D enumerates the number and types of fields in each class. For feature #7, DR. D considers classes containing request handlers as configurable. For feature #8, DR. D analyzes the access modifiers of each class to determine if they are globally accessible. Then, DR. D recursively examines the interfaces and parent class information implemented by each class to ascertain their iterability (feature #9). Furthermore, DR. D analyzes all methods to record the actual parameter types of all iterable objects' adding instructions, e.g., `HashMap.add()`, thereby collecting feature #10. To identify fields susceptible to concurrent access (feature #11), DR. D examines the modifiers and types of each field in the class. If a field is declared volatile or synchronized, or if its type belongs to data types in the Java concurrency library (i.e., java.util.concurrent), it is considered prone to concurrent access. Regarding methods potentially subject to concurrent access (feature #12), DR. D scrutinizes the method signatures to determine if they are synchronized and examines the presence of synchronized code blocks within the method implementations. Finally, feature #13 is addressed by directly analyzing the modifier information of each method within the class to identify static methods.

### A.2 Attack Scenario Description

Table 7 lists the hardware and software configuration details of the virtual machines created on the four cloud platforms, including CPU model, RAM size, disk configuration, network bandwidth, operating system, Java runtime version, and the Round-Trip Time (RTT) between the simulated attacker and the server. It is worth noting that although the maximum bandwidth on Google Cloud and Microsoft Azure is not customizable and exceeds 300 Mbps, we ensured that the traffic throughout the experiments remained below 300 Mbps.

### A.3 DDoS Defenses

Table 8 lists the DDoS defense strategies enabled on each platform. In our experiments, the attack requests were not detected by the DDoS defense mechanisms of any platform and did not trigger any alerts. We attempted to understand the specific implementations of each platform's DDoS defense strategies. However, these implementations are black boxes to developers. Based on an analysis of their documentation, we found that their designs primarily consist of two components: traffic volume analysis and traffic behavior analysis.

Traffic volume analysis involves monitoring the incoming traffic size of the protected instances to check whether it exceeds the platform-specified threshold. Traffic behavior analysis identifies anomalous attack traffic based on known DDoS attack patterns, using feature matching or machine learning methods. When abnormal traffic is detected, the suspicious traffic is redirected from the original network to the platform's scrubbing services. These services identify and filter out malicious traffic, reinject sanitized legitimate traffic back into the original network, and forward it to the target while generating analysis reports and alerts for users.

Regarding traffic volume thresholds, Alibaba Cloud and Huawei Cloud explicitly provided threshold values in their consoles, whereas Google Cloud and Microsoft Azure did not disclose specific numbers. According to their documentation, these thresholds are automatically generated for different instances based on proprietary machine learning algorithms. As for anomalous behavior characteristics, the platforms did not disclose the supported attack patterns and feature lists but claimed to detect common DDoS attacks, such as SYN Flood attacks, UDP reflection attacks, and HTTP Flood attacks.

### A.4 Memory Usage Visualization

Figure 9 shows the heap memory usage of web container processes over time during vulnerability exploitation on different platforms, with subplot IDs corresponding to vulnerability

Table 7: Hardware and Software Configurations of Virtual Machines Deployed on Four Cloud Platforms

| | **Google Cloud** | **Microsoft Azure** | **Alibaba Cloud** | **Huawei Cloud** |
|---|---|---|---|---|
| CPU | Intel Xeon (Cascade Lake) Gold 6268CL @2.80GHz 8vCPU | Intel Xeon (Cascade Lake) Platinum 8272CL @2.50GHz 8vCPU | Intel Xeon (Cascade Lake) Platinum 8269CY @2.50 GHz 8vCPU | Intel Xeon (Cascade Lake) Gold 6278C @2.60GHz 8vCPU |
| RAM | 8GB | 8GB | 8GB | 8GB |
| Disk (Max IOPS/Throughput) | Balanced Persistent Disk 40GB (15000 / 240MB/s) | Premium SSD 30GB (3500 / 170MB/s)) | ESSD Entry 40GB (6000 / 150MB/s) | General Purpose SSD 40G (20000 / 250MB/s) |
| Bandwith | >300Mbps | >300Mbps | 300Mbps | 300Mbps |
| OS | Ubuntu 22.04 LTS - x64 | Ubuntu 22.04 LTS - x64 | Ubuntu 22.04 LTS - x64 | Ubuntu 22.04 LTS - x64 |
| Java Runtime | Java 17 | Java 17 | Java 17 | Java 17 |
| RTT (Min/Max/Avg) | 295/302/297ms | 339/342/340ms | 9/10/9ms | 5/7/5ms |

Table 8: Summary of Enabled DDoS Defense Strategies Across Four Cloud Platforms.

| **Cloud Platform** | **Defense Name** | **Traffic Volume Analysis** | **Traffic Behavior Analysis** | **Defense Alert** |
|---|---|---|---|---|
| Google Cloud | Cloud Armor Standard | Adaptive Threshold | ✓ | Not Triggered |
| Microsoft Azure | DDoS IP Protection | Adaptive Threshold | ✓ | Not Triggered |
| Alibaba Cloud | DDoS Protection Basic | 450Mbps \| 100kpps | ✓ | Not Triggered |
| Huawei Cloud | Anti-DDoS Service Basic | 120Mbps | ✓ | Not Triggered |

IDs in Table 6. The memory usage was monitored via JMX with a sampling frequency of once per second until an OutOfMemory (OOM) error was triggered. The servers typically became unresponsive before the OOM error due to excessive garbage collection in the final stages. The vulnerability 16 is context-constrained and does not lead to memory exhaustion. Therefore, it is not included in the figure. Notably, the exploitation of vulnerabilities in Resin did not fully exhaust the container's memory resources but still caused a DoS due to Resin's watchdog security mechanism [6]. Specifically, Resin servers are started and monitored by a separate watchdog process, which continuously checks the server's health and proactively kills the Resin instance if it becomes unresponsive. While the watchdog process automatically restarts the server after a default period of two minutes, the hosted web applications cannot be accessed during this time. By repeatedly exploiting the vulnerability, attackers can induce a continuous DoS.

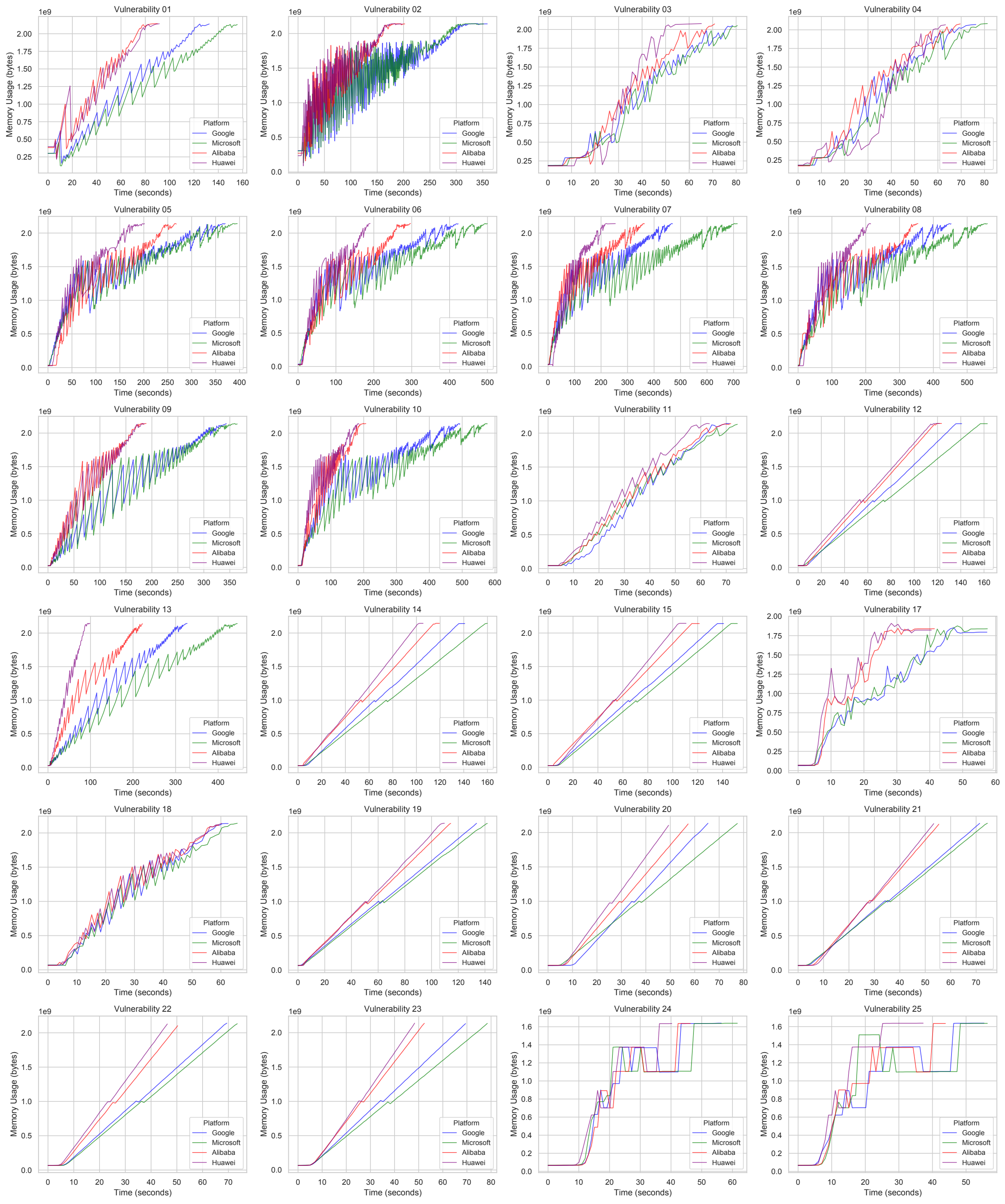


Figure 9: The Heap Memory Usage of Web Container Processes Over Time on Four Cloud Platforms