The Hidden Dangers of Public Serverless Repositories: An Empirical Security Assessment

Eduard Marin[1(✉)], Jinwoo Kim[2], Alessio Pavoni[1], Mauro Conti[3,4], and Roberto Di Pietro[5]

[1] Telefonica Research, Spain
{eduard.marinfabregas, alessio.pavoni}@telefonica.com
[2] Kwangwoon University, Republic of Korea
jinwookim@kw.ac.kr
[3] University of Padua, Italy
mauro.conti@unipd.it
[4] Örebro University, Sweden
[5] King Abdullah University of Science and Technology, Saudi Arabia
roberto.dipietro@kaust.edu.sa

Abstract. Serverless computing has rapidly emerged as a prominent cloud paradigm, enabling developers to focus solely on application logic without the burden of managing servers or underlying infrastructure. Public serverless repositories have become key to accelerating the development of serverless applications. However, their growing popularity makes them attractive targets for adversaries. Despite this, the security posture of these repositories remains largely unexplored, exposing developers and organizations to potential risks. In this paper, we present the first comprehensive analysis of the security landscape of serverless components hosted in public repositories. We analyse 2,758 serverless components from five widely used public repositories popular among developers and enterprises, and 125,936 Infrastructure as Code (IaC) templates across three widely used IaC frameworks. Our analysis reveals systemic vulnerabilities including outdated software packages, misuse of sensitive parameters, exploitable deployment configurations, susceptibility to typo-squatting attacks and opportunities to embed malicious behaviour within compressed serverless components. Finally, we provide practical recommendations to mitigate these threats.

1 Introduction

Serverless computing has become a highly compelling cloud paradigm that abstracts infrastructure management tasks (e.g., load balancing and scaling) from tenants, allowing them to focus entirely on application development [53,41,48,52]. In serverless architectures, applications are implemented as a set of small, interdependent *functions*, each designed to perform a specific task. These serverless functions can communicate with one another and integrate with cloud services like event triggers, message queues or object storage to support a wide range of applications. Serverless computing offers automatic scaling in response to workload demands and follows a pure pay-per-use pricing model, where tenants are billed only for the resources consumed during execution. Due to these advantages, major cloud providers, such as AWS [5], Microsoft [8], Google [18], IBM [20] and Alibaba [2], have incorporated serverless computing into their service offerings.

As serverless adoption has grown, numerous *public serverless repositories* have emerged, enabling developers to share serverless components. These repositories host a wide range of serverless components, many of which have been downloaded thousands to millions of times. However, their increasing popularity has also made them attractive targets for adversaries [36,31]. A key problem is the lack of transparency surrounding the security practices in these repositories. Most repositories provide little to no information about the security checks performed, the approval policies enforced or how security responsibilities are divided among contributing developers, the users who download serverless components and the repository administrators who publish them. For instance, Red Hat Quay claims "*to continuously scan containers for vulnerabilities*" [22], while AWS states "*all applications published by AWS are reviewed to ensure license compliance and code quality*" [33]. Although these claims suggest some level of scrutiny, they are highly generic and offer little detail about the scope, rigour or consistency of the security checks applied. Crucially, we argue that such claims may create a false sense of security, leading users to believe that the serverless components they download have been thoroughly vetted and are safe to integrate without further verification, thus increasing the risk of supply chain attacks [49].

To the best of our knowledge, we present the first comprehensive study of the security state surrounding public serverless repositories. We focus on two fundamental yet previously under-explored research questions: (i) Do public serverless repositories introduce application-level security risks? (**RQ1**); and, (ii) Does the dynamic configuration and deployment model of serverless computing give rise to novel attack vectors? (**RQ2**). To address RQ1, we analyse the prevalence of outdated third-party libraries with known vulnerabilities in serverless components and investigate the potential for embedding malicious behaviour in components distributed as compressed archives. To address RQ2, we conduct a detailed analysis of Infrastructure as Code (IaC) templates to identify possible misconfigurations. Additionally, we discover three sensitive parameters in Docker run commands that can be exploited for malicious purposes and evaluate these repositories' susceptibility to typo-squatting attacks.

To this end, we collect and analyse 2,758 serverless components from five widely-used public repositories: (i) Docker Hub [14]; (ii) GitHub [17]; (iii) AWS Serverless Application Repository (SAR) [7]; (iv) Serverless Framework [25]; and, (v) Red Hat Quay [22]. Our selection includes one repository dedicated to *serverless plugins* and four hosting *serverless functions*, spanning both well-maintained platforms (e.g., AWS SAR and Red Hat Quay) and highly popular but less regulated ecosystems such as Docker Hub and GitHub. Additionally, we analyse 125,936 IaC templates from three widely used frameworks: Terraform [26], AWS CloudFormation [4] and AWS Serverless Application Model (SAM) [6].

Contributions. We summarize our key contributions as follows:

– We conduct a large-scale security analysis of 2,758 serverless components from major public repositories, including AWS SAR, Docker Hub, GitHub, Red Hat Quay and Serverless Framework, along with 125,936 IaC templates across Terraform, AWS CloudFormation and AWS SAM.
– We reveal two new attack vectors: the insertion of malicious behaviour into compressed serverless components and the presence of misconfigurations within IaC templates. Additionally, we discover three previously undocumented sensitive parameters in Docker run commands.
– We provide a set of actionable recommendations aimed at improving the security posture of public serverless repositories and guiding best practices for repository administrators, developers and users.

Responsible disclosure. We notified the maintainers of the affected repositories, providing detailed descriptions of the issues and recommendations for mitigation.

2 Background and Motivation

2.1 Serverless Deployment Models

Serverless functions can be deployed using various methods, including: (i) packaging the function as a Docker container image [12,16,10], (ii) uploading a pre-packaged ZIP folder containing the functions' code and dependencies [30,11,13], (iii) writing code directly in the cloud provider's console editor [3], (iv) using YAML templates or configuration files that specify the deployment details, resources and permissions of the serverless functions to be deployed and (v) using Infrastructure as Code (IaC) frameworks, such as AWS CloudFormation, to automate the deployment and management of serverless functions. Regardless of the method, developers remain responsible for providing the application code. In recent years, it has become increasingly common to accelerate development by integrating components from public repositories. However, this introduces significant security risks, as discussed in the next section.

2.2 Attack Vectors in Public Serverless Repositories

We identify five primary attack vectors (V1–V5) that pose significant security risks to public serverless repositories. It is important to note that some attack vectors are relevant only to specific deployment methods (see Table 1).

V1 **Vulnerable third-party libraries.** Serverless components typically rely on third-party libraries, many of which contain known vulnerabilities that introduce security risks [39]. Even when patches are available, outdated libraries often remain in use for extended periods, offering adversaries ample opportunities to exploit those weaknesses [49]. The rapid development cycles inherent to serverless applications make them particularly vulnerable to these risks.

Table 1: The applicability of attack vectors by deployment model
(● Applicable, ○ Not applicable)

Attack Vectors	**Deployment Model**				
	Container Image	Pre-packaged Zip	Console Editor	YAML Templates Conf. Files	IaC Frameworks
V1 Vulnerable third-party libraries	●	●	●	●	●
V2 Malicious serverless components	●	●	○	○	○
V3 Sensitive parameters in Docker run commands	●	○	○	○	○
V4 Misconfigurations in IaC templates	○	○	○	●	●
V5 Typo-squatting attacks	●	●	●	●	●

V2 **Malicious serverless components.** Many repositories allow anyone to upload serverless components after a simple registration process, enabling adversaries to easily distribute malicious components [46,43]. Most serverless platforms also accept pre-packaged compressed archives bundling code and dependencies, which can facilitate more advanced attacks. These formats can further obfuscate malicious behaviour, making detection by traditional security tools significantly more difficult.

V3 **Sensitive parameters in Docker run commands.** Some repositories (e.g., Docker Hub) allow contributors to provide execution instructions specifying how their components should be run. Adversaries can exploit this feature and include malicious `Docker run commands` that contain risky parameters (e.g., `-privileged` or `-pid=host`). Users who download these components are likely to follow the provided (malicious) `Docker run commands`, potentially compromising container isolation and jeopardizing the security of the underlying host.

V4 **Misconfigurations in IaC templates.** IaC tools (e.g., AWS CloudFormation) are widely used to specify and deploy serverless functions in production environments. They enable developers to declaratively define the serverless functions and their associated resources, configurations, permissions and policies in a structured and repeatable manner through templates. It is common practice for developers to contribute their IaC templates and reuse those shared by others. However, to date, no systematic study has investigated whether these templates contain misconfigurations or assessed the practical security consequences of such misconfigurations.

V5 **Typo-squatting attacks.** Various naming conventions such as Docker's Fully Qualified Image Identification (FQID) [50] and AWS's Amazon Resource Names (ARNs) [21] are used to uniquely identify serverless components within repositories. Because developers often enter these names manually (e.g., via terminal or editor), typographical errors are common. Adversaries can exploit this by registering malicious components with names that closely resemble those of popular or trusted ones. This typo-squatting technique leverages human error to surreptitiously distribute malicious serverless components, increasing the likelihood of accidental installation and execution by unsuspecting users.

2.3 Threat Model

We consider adversaries capable of uploading vulnerable (V1) or malicious components (V2) to these repositories, posing significant risks to unsuspecting users who use them. In doing so, adversaries can also supply execution instructions that include `Docker run commands` with sensitive parameters (e.g., `-privileged`) (V3), exploiting the tendency of many users to follow the provided instructions [51]. Similarly, adversaries can upload IaC templates with dangerous configurations to public repositories, causing any developer who uses them to unknowingly misconfigure their serverless applications and inadvertently expose critical information (V4). Finally, when uploading components to these repositories, adversaries can select component names that closely resemble popular entries in the repository (V5), aiming to exploit typographical errors made by developers when retrieving serverless components [50].

3 Security Analysis Framework for Public Serverless Repositories

In this section, we describe the process we used to discover and retrieve serverless components from public repositories, and we evaluate their susceptibility to the five attack vectors considered in this paper.

① **Data collection.** To automate the extraction of serverless component data from the selected repositories, we developed custom web scrapers using frameworks such as BeautifulSoup [9] and Selenium [24]. Using these scrapers, we collected key metadata, including: (i) the *component name*, (ii) the associated *pull command* or *GitHub URL* and (iii) the recommended *execution instructions* (when available). In some cases, this process required performing authenticated queries and adhering to repository-imposed request limits. For repositories hosting both serverless and non-serverless components, we applied a two-step filtering mechanism to eliminate non-relevant images and keep only the serverless components. We first configured our crawlers to perform queries using the keyword 'serverless' and then examined the metadata associated with each component to detect the presence of a 'serverless.yml' file. This methodology was inspired by the approach used by Eskandani et al. [45].

② **Vulnerability analysis.** Next, we performed a security analysis of the libraries included in the retrieved serverless components using Trivy [28] and Grype [19], two widely used and open-source vulnerability scanners. Both tools extract metadata, package information and libraries from container images or source code, and cross-reference them against multiple vulnerability databases [32]. The identified vulnerabilities are classified into five severity levels according to their CVSS 3.1 scores: (i) *Critical* (9.0–10.0), (ii) *High* (7.0–8.9), (iii) *Medium* (4.0–6.9), (iv) *Low* (0.1–3.9) and (v) *Unknown* (excluded from the analysis). Each serverless component was scanned separately with both tools. Using multiple scanners helps account for tool variability, as each may rely on different vulnerability databases and detection heuristics, potentially identifying distinct sets of vulnerabilities.

③ Hiding malicious behaviour in compressed serverless components. To investigate whether compression can be exploited to conceal malicious behaviour in serverless components, we utilised VirusTotal [29], a widely used platform that aggregates results from numerous antivirus engines. We hypothesized that adversaries could leverage common compression formats to evade detection [30,11,13]. To test this, we created serverless components compressed with popular compression formats, including both benign samples and variants embedded with malware. Subsequently, we submitted these samples to VirusTotal and analysed the detection rates reported by its integrated antivirus engines.

④ Identification of sensitive parameters in Docker run commands. We analysed the execution instructions provided by component owners to identify `Docker run commands` containing sensitive parameters that could be exploited in security attacks. First, we examined serverless components hosted on Docker Hub for sensitive parameters previously documented in the literature [51]. Next, we extended our analysis and discovered three previously undocumented sensitive parameters that can pose significant security risks. These newly identified parameters could enable adversaries to escalate privileges, bypass security controls or gain unauthorized access, thus increasing the potential impact of compromised serverless components distributed via public repositories.

⑤ Finding misconfigurations in IaC templates. We examined a large corpus of IaC templates to assess whether they contained insecure configurations that adversaries could exploit to compromise serverless applications. Using Trivy, we scanned these templates [27] for misconfigurations across widely used frameworks, including Terraform, AWS CloudFormation and AWS SAM, and then classified the identified issues into four severity levels. For parameters frequently associated with security risks, we conducted an in-depth analysis to evaluate their potential impact and trace their underlying root causes. Additionally, we manually reviewed the IaC templates to discover previously undocumented misconfigurations that may not be detected through Trivy's automated analysis.

⑥ Detection of potential typo-squatting attacks. To identify potential typo-squatting attacks, we measured the similarity between component names using the Damerau-Levenshtein (DL) distance metric [50], which quantifies the minimum number of operations (insertions, deletions, substitutions, or transpositions) needed to transform one string into another. For each repository, we extracted both the username and image name associated with every serverless component and performed exhaustive pairwise comparisons to detect suspiciously similar naming patterns indicative of potential typo-squatting attacks. We focused on name pairs with low DL distances, as these indicate identical or highly similar names.

Table 2: Vulnerability statistics per repository using data collected from Trivy (blue) and Grype (red), respectively.

Repository	**# of Compo.**	**# of Vulnerabilities**				
		Mean	**Median**	**Max**	**Min**	**Std Dev**
Serverless Framework [25]	355	8/35	0/11	257/611	0/0	24/70
AWS SAR [7]	242	4/21	0/0	127/3437	0/0	13/223
GitHub [17]	712	98/119	6/23	18559/9658	0/0	838/581
Docker Hub [14]	1374	1243/1522	432/519	6897/5781	0/0	1477/1628
Red Hat Quay [22]	75	620/676	135/230	5390/5584	0/0	1057/1157

4 Application-level Security Risks

In this section, we assess the extent to which the collected serverless components are exposed to application-level security risks (RQ1). We begin by analysing the presence of known vulnerabilities in third-party libraries included in these components (V1). We then investigate the potential for concealing malicious behaviour when these components are distributed in compressed formats (V2).

4.1 Vulnerability Analysis in Third-party Libraries

Statistical analysis of vulnerabilities across repositories. To characterize the vulnerability landscape of each repository, we begin by reporting key statistical metrics—including the mean, median, minimum, maximum and standard deviation of vulnerability counts—based on data obtained from both Trivy and Grype (see Table 2). Our findings reveal a consistent discrepancy between both tools, with Grype systematically reporting higher vulnerability counts. This difference is rooted in their distinct design philosophies. Grype prioritizes sensitivity and broad detection coverage at the cost of a higher false positive rate [38], [37], while Trivy adopts a more conservative approach focused on minimizing false positives that may occasionally lead to missed vulnerabilities [35]. Among the repositories we analysed, Docker Hub and Red Hat Quay exhibit the highest mean and median vulnerability counts, as well as the largest variability among components. GitHub falls in an intermediate position, with several components exhibiting significant vulnerabilities, though generally fewer than those in Docker Hub and Red Hat Quay. Conversely, we found that the Serverless Framework and AWS SAR consistently report lower vulnerability counts.

Distribution of vulnerabilities across serverless components. To further understand the distribution of vulnerabilities within repositories, we analyse the Cumulative Distribution Function (CDF) of vulnerability counts per serverless component based on the data obtained from Trivy (see Figure 1). Our results show that approximately 80% of Docker Hub components and 60% of Red Hat Quay components have more than 100 vulnerabilities. GitHub presents a slightly better security posture: around 50% of its components contain ten or fewer vulnerabilities, while 10% have more than 100 vulnerabilities. By contrast, serverless

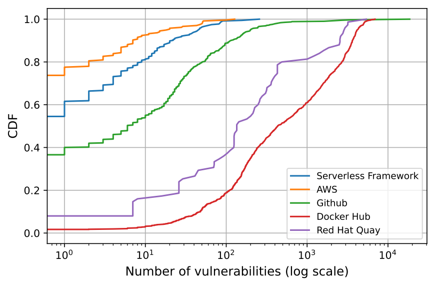


Fig. 1: CDF of vulnerability counts per serverless component.

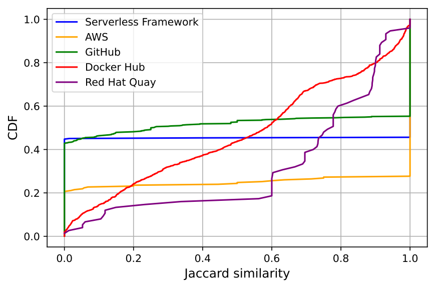


Fig. 2: Trivy vs. Grype vulnerability similarity.

components from the AWS SAR and Serverless Framework are significantly less affected. Many components have no known vulnerabilities and most affected ones contain fewer than 10, with few exceeding 100.

Comparison of Grype and Trivy detection results. Although Trivy and Grype use similar techniques for vulnerability detection, their results often differ significantly (see Table 2). To quantify this divergence, we computed the Jaccard similarity between the sets of vulnerabilities identified by each tool (see Figure 2). A score of 1 indicates complete agreement while a score of 0 indicates no overlap. For components from AWS SAR, GitHub and the Serverless Framework, many showed a Jaccard similarity of 1, suggesting identical results. Manual inspection revealed that these components frequently had no detected vulnerabilities. In contrast, components from Docker Hub and Red Hat Quay showed greater discrepancies, with fewer instances of high similarity. These findings underscore the importance of using multiple scanners to obtain a comprehensive assessment of security risks. Importantly, there is no universally accepted 'ground truth' for vulnerability detection. Some organizations prioritize precision by considering only the intersection of scanner outputs to reduce false positives, while others prioritize recall by using the union of results to maximize coverage, even at the expense of increased false positives.

Distribution of vulnerability severity across repositories. While previous analyses focused primarily on vulnerability counts, we also examined vulnerability severity, a key factor in assessing real-world security risks (see Figure 3). Although Docker Hub and Red Hat Quay report the highest total vulnerabilities, the proportion of critical and high-severity vulnerabilities in their components is relatively low, only 5% critical and 29% high in Docker Hub, and 3% critical and 23% high in Red Hat Quay. In contrast, AWS SAR and Serverless Framework, which have the lowest average vulnerability counts per component, show the highest proportions of severe vulnerabilities: 68% in AWS SAR and 57% in Serverless Framework are classified as critical or high. For details on the top 10 most commonly used packages across five platforms and their associated vulnerabilities and severities, we refer the reader to Appendix A.

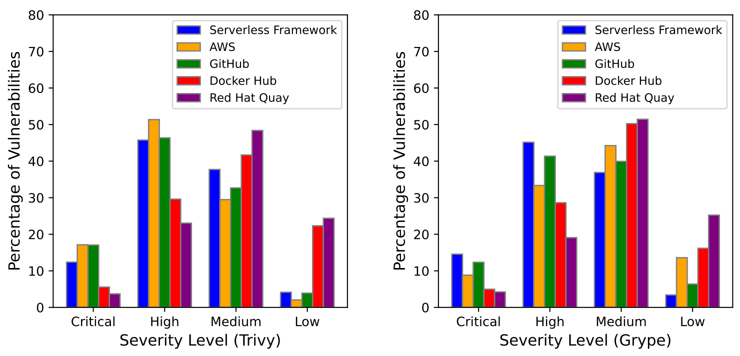


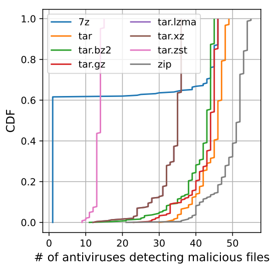
Fig. 3: Vulnerability severity using data from Trivy (left) and Grype (right).

4.2 Hiding Malicious Behaviour in Compressed Serverless Components

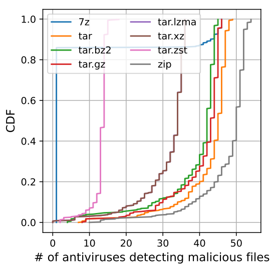
To evaluate the potential for concealing malicious behaviour within compressed serverless components, we randomly selected 1,794 components from public repositories, including 356 from AWS SAR, 657 from GitHub, 352 from the Serverless Framework, 354 from Docker Hub and 75 from Red Hat Quay. Each component was compressed in its original form using eight widely adopted formats: *7z*, *tar*, *tar.bz2*, *tar.gz*, *tar.lzma*, *tar.xz*, *tar.zst* and *zip*. We first verified that none were flagged as malicious by VirusTotal. Then, using six malicious files sourced from reputable open-source malware repositories[6], including (i) the `eicar.txt` antivirus test file, (ii) a Python remote access trojan, (iii) a Java infector, (iv) a PHP backdoor, (v) a Python backdoor and (vi) a Python trojan, we generated malicious samples by injecting one file at a time into each component and recompressing them with each compression format.

We examined how the choice of compression format affects the detection of malicious behaviour. Figure 4 presents the CDF of antivirus engines that flagged malicious components for each format. Although all injected files were detected by at least one engine, components compressed with *7z* and *tar.zst* were flagged by significantly fewer engines, indicating lower detection reliability. This is concerning because, due to trade-offs between false positives and false negatives, a component is typically considered malicious only if flagged by a minimum number of engines. In prior work, this threshold was set at five engines [51]. In our analysis, we observed several instances where malicious components compressed with certain formats fell below this threshold, suggesting that embedding malicious behaviour within compressed serverless components could be an effective evasion technique. Additionally, we confirmed that such malicious components can be successfully uploaded to public serverless repositories (see Section 6).

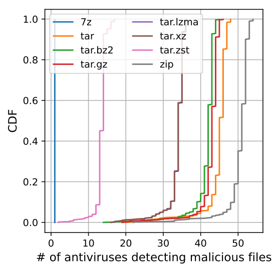
[6] For example, https://github.com/vxunderground/MalwareSourceCode and https://github.com/nijithneo/DAT



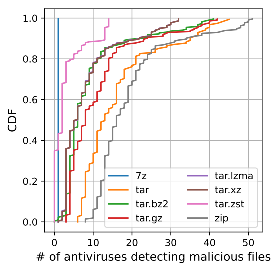
(a) AWS SAR



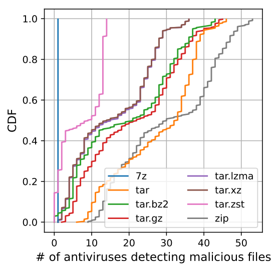
(b) GitHub



(c) Serverless Framework



(d) Docker Hub



(e) Red Hat Quay

Fig. 4: Relationship between serverless components compressed with various compression algorithms and the number of engines that successfully detect them.

5 Configuration and Deployment Security Risks

In this section, we examine the security risks in configuring and deploying serverless components (RQ2), focusing on three attack vectors: (i) sensitive parameters in Docker run commands (V3); (ii) misconfigurations in IaC templates (V4); and, (iii) potential typo-squatting attacks (V5).

5.1 Sensitive Parameter Misuse in Docker Run Commands

We analysed the Docker run commands in the execution instructions for serverless components hosted on Docker Hub (V3). We first targeted parameters previously identified as sensitive by Liu et al. [51], including `-v`, `-privileged` and `-pid` which can compromise container isolation and host security. For instance, `-v <src>:<dest>` mounts host directories into a container, potentially leaking sensitive files; `-privileged` grants unrestricted access to host resources; and `-pid=host` enables container processes to monitor and interact with all host processes, facilitating reconnaissance or privilege escalation. Our analysis found 137 uses of `-v`, two instances of `-privileged` and no occurrences of `-pid`.

Uncovering new risky parameters in Docker run commands. Beyond assessing the prevalence of known sensitive parameters in Docker run commands for serverless components hosted in public repositories, our analysis uncovered three previously undocumented parameters that pose significant security risks.

(1) Hard-coded credentials. We discovered a case where AWS access keys were hard-coded directly into a Docker run command. Executing such commands may inadvertently expose sensitive credentials, allowing adversaries to take control of the associated cloud environment. This can lead to a range of attacks, including unauthorized access to S3 buckets or launching cryptojacking campaigns [34]. To mitigate this, we recommend avoiding credential injection via command-line arguments and instead using secure methods like Docker secrets (e.g., `-secret aws_key` and `-secret aws_secret`) to manage sensitive information.

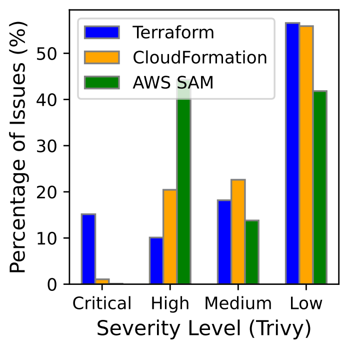
```
docker run -d --name go-serverless-aws-container
-v $PWD:/usr/src/go/src
-e AWS_KEY=JFHGUFJAKEXAMPLEJDFJHEKF
-e AWS_SECRET=AJDFUEXAMPLESDLKF
-e AWS_REGION=us-east
iamfrisbee/go-serverless-aws
```

(2) Sensitive information passed via environment variables. We found 47 instances where the `-e` parameter was used to pass sensitive information, such as credentials, tokens or keys, to containers. Adversaries with access to the Docker daemon (e.g., external adversaries who exploited a misconfiguration) can retrieve these values using commands like `docker inspect`. This exposure could lead to unauthorized access to cloud services, data exfiltration or financial abuse. As with hard-coded credentials, this risk can be mitigated by using Docker secrets.

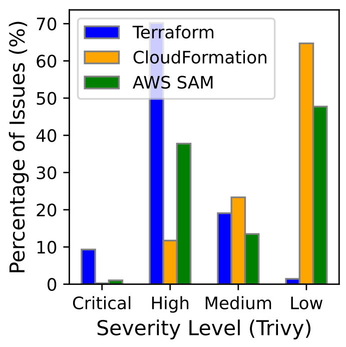
```
docker run -v $(pwd):/opt/app
-e AWS_DEFAULT_REGION
-e AWS_ACCESS_KEY_ID
-e AWS_SECRET_ACCESS_KEY
andrewoh531/docker-serverless serverless deploy
```

(3) Mounting the Docker daemon socket within containers. We identified a case where the Docker daemon socket (`/var/run/docker.sock`) was mounted directly into a container. This configuration effectively grants the container full control over the Docker daemon, allowing an adversary who compromises the container to escalate privileges and gain control over the host system. The security implications are comparable to those of the `-privileged` flag and are widely regarded as a critical misconfiguration.

```
docker run -p 8080:8080 -v
/var/run/docker.sock:/var/run/docker.sock
furikuri/serverless-to-go
```



(a) AWS



(b) GitHub

Fig. 5: Misconfigurations severity reported by Trivy for: (a) AWS Serverless Repository; and, (b) GitHub.

Table 3: Breakdown of IaC templates analyzed in our serverless repository dataset.

Repository	**Terraform** (`.tf`)	**CloudFormation** (`.yaml`, `.json`)	**AWS SAM** (`.yaml`)	**Total**
AWS	30	5,023	225	5,278
GitHub	1,764	117,860	1,034	120,658
Total	1,794	122,883	1,259	125,936

5.2 Security Analysis of IaC Templates

Given their critical role in automating the deployment of serverless applications, we conducted an in-depth analysis of IaC templates across three widely adopted frameworks: (i) Terraform; (ii) AWS CloudFormation; and, (iii) AWS Serverless Application Model (SAM) (V4). Our dataset includes IaC templates from AWS SAR and GitHub, as container-based platforms such as Docker Hub, Red Hat Quay and Serverless Framework do not include IaC templates. Table 3 provides a breakdown of the analysed IaC templates. To distinguish between frameworks, we classified templates by their extensions: `.tf` for Terraform and `.json` or `.yaml` for AWS CloudFormation. AWS SAM templates, which also use the `.yaml` format, were identified by detecting the presence of the *Transform: AWS::Serverless-2016-10-31* directive.

We scanned the retrieved IaC templates using Trivy to identify potential misconfigurations. Figure 5 shows the distribution of misconfiguration issues across four severity levels. Our analysis shows that Terraform IaC templates exhibit a significantly higher proportion of critical misconfigurations than AWS CloudFormation and AWS SAM templates. This disparity is likely due to Terraform's broader configurability across multi-cloud environments. Nevertheless, across all frameworks, the proportion of critical and high-severity issues remains substantial, underscoring the systemic risk posed by IaC misconfigurations. We also examined the five most common misconfigurations found in the templates. The most frequent issue, accounting for approximately 19% of the identified misconfigurations, involved the omission of a source ARN in Lambda permissions [15], potentially allowing unrestricted invocation of Lambda functions. Another common issue, accounting for roughly 13% of cases, was the use of default AWS-managed keys instead of customer-managed encryption keys for S3 buckets [23], which weakens control over data protection. A complete overview of the identified misconfigurations is given in Appendix B.

Cross-Origin Resource Sharing. It is well known that using a wildcard ('*') in Cross-Origin Resource Sharing (CORS) policies introduces security risks [42]. Our analysis revealed such misconfigurations in IaC templates configuring CORS for both AWS CloudFormation and AWS SAM. Through manual inspection, we identified seven instances of this issue in AWS SAR and one in GitHub, which were *not* detected by Trivy. The `CorsOrigin` attribute is used to configure CORS policies, which control cross-domain resource sharing (e.g., allowing front-end applications to access resources from AWS S3 buckets) [40]. However, we found that several templates included the configuration `Default: '*'`, which allows unrestricted access from any domain to the serverless application's API Gateway. This wildcard disables CORS protections, posing a significant risk. If adversaries compromise a serverless function, they can exploit the permissive CORS policy to exfiltrate sensitive data to untrusted or attacker-controlled domains [44]. While browsers block credentialed requests (e.g., those involving cookies or authorization headers) [1], our manual inspection of these IaC templates revealed that they provision public APIs without authentication, thereby making even unauthenticated cross-origin requests a security risk.

5.3 Typo-squatting Attacks

As the final step in our analysis, we assessed the susceptibility of public serverless repositories to typo-squatting attacks (V5), covering components from the Serverless Framework, GitHub, Red Hat Quay and Docker Hub. AWS SAR was excluded since its components are used exclusively within AWS, where the chance of typos by developers is much lower.

Figure 6 shows the CDF of DL similarities for usernames and image names. Our findings revealed two pairs of *user names* with a DL distance equal to 1: one identified on GitHub and one on Docker Hub. Additionally, we identified several *image name* pairs with a DL distance of 1, including two instances in the Serverless Framework, six in GitHub, 191 in Docker Hub and one in Red Hat Quay. These results indicate that while typo-squatting risks are present across all repositories, they are particularly pronounced in Docker Hub. Although the intent behind similar usernames or image names within the same repository is unclear, their presence suggests this vector could be exploited by adversaries.

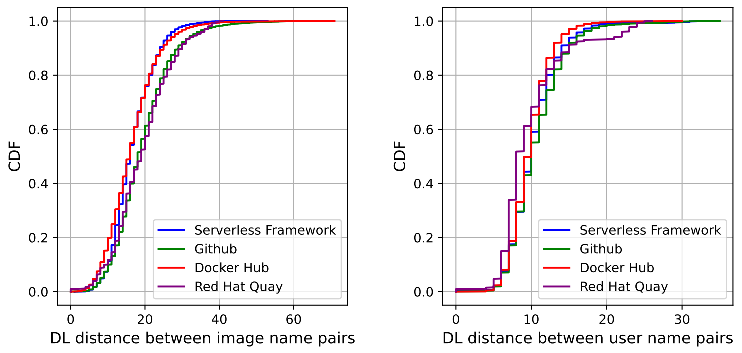


Fig. 6: Lexical analysis based on Damerau-Levenshtein distance, applied to image names (left) and user names (right)

6 Key Findings and Discussion

Our analysis reveals that public serverless repositories often lack rigorous security oversight. Repository administrators typically perform minimal (if any) security checks on the components they host. This is particularly concerning in serverless computing, where applications are rapidly built from many small, interconnected serverless functions. While this enables fine-grained security controls when properly implemented, it also requires each function to be independently configured and secured, increasing the risk of errors and misconfigurations.

Application-level security risks. Although vulnerabilities in third-party libraries are a well-known threat, our findings show that such issues remain widespread even in repositories presumed to maintain strict security controls like AWS SAR. Beyond these, we uncovered new risks tied to how serverless applications are deployed. Specifically, the ability to upload pre-packaged ZIP archives bundling code and dependencies creates a blind spot for traditional scanners, which often fail to properly inspect compressed artifacts. To demonstrate this, we uploaded a malicious ZIP archive with known malware to AWS SAR via a controlled test repository. The upload completed without triggering any alerts, exposing a significant gap in current threat detection. While we deleted the component immediately to comply with ethical research standards, this experiment highlights the need for more advanced security techniques to detect malicious content within compressed serverless packages.

Configuration and deployment security risks. Our study also reveals significant risks arising from serverless application configuration and deployment practices. A common issue is the widespread use of default, overly permissive, or publicly recommended settings. Specifically, both Docker run commands and IaC templates frequently embed risky parameters, making them untrustworthy by default. Although these configuration-related threats are somewhat fewer than third-party library vulnerabilities, they are much harder to detect, as shown by our discovery of three new sensitive Docker parameters and a novel IaC misconfiguration that evaded detection by advanced scanners like Trivy. Furthermore, serverless repositories are vulnerable to typo-squatting attacks, where adversaries register components with names closely resembling popular ones. Critically, serverless automation amplifies the impact of even rare typos, since IaC templates and CI/CD pipelines may automatically fetch and deploy typo-squatted components without manual review, enabling silent integration into production.

Security model. We advocate for a security model in which public serverless repository administrators are the primary responsible for securing the components they host. To ensure transparency and accountability, they should disclose their security practices, including scanning methods, tools and the frequency of their security assessments. Administrators must implement automated, periodic vulnerability scans using a diverse set of complementary tools, alongside mandatory pre-publication compliance checks. Given the rapidly evolving serverless ecosystem, with frequent emergence of new vulnerabilities in libraries, frameworks and cloud services, continuous monitoring and automated re-evaluation of hosted components are crucial for sustained security. Additionally, repository maintainers should apply automated static analysis to configuration files and enforce blacklists of sensitive parameters (such as those identified in this and prior work) to prevent their inclusion. To mitigate typo-squatting attacks, mechanisms like similarity-based name collision alerts and stricter naming policies should be adopted. Until these security measures become standard, users and developers should assume that any serverless component from a public repository may be compromised and perform thorough independent security evaluations before use.

Possible limitations of this work. False positives are a well-known limitation of vulnerability scanners such as Trivy and Grype. These tools may classify a repository as vulnerable if it includes a package with known CVEs in its metadata files. However, if the package is not actually used in the source code, the reported vulnerability may constitute a false positive. To assess the extent of this issue, we randomly selected 30 open-source serverless components (approximately the square root of the total: 242 from AWS and 712 from GitHub). Trivy and Grype collectively reported 1,417 CVEs across these samples. We then manually verified whether the flagged packages were actually referenced in the source code. If a package appeared only in metadata files (e.g., `.lock`, `.gradle`, `.toml`, `.yml`, `.yaml`, `.json`, `.xml`, `.md`) but not in source code files (e.g., `.py`, `.js`, `.ts`, `.java`, `.go`, `.rb`, `.sh`, `.c`, `.cpp`), we classified the associated CVE as a false positive. Our analysis found that 1,275 out of 1,417 CVEs were associated with packages present in the source code, resulting in an estimated false positive rate of approximately 10%. These findings suggest that while false positives are present, the rate is within acceptable bounds and does not undermine the overall reliability of our vulnerability analysis.

7 Related work

Shu *et al.* [54] were the first to examine the security state of Docker Hub images. Wist *et al.* [55] conducted a similar study on 2,500 Docker Hub images. Liu *et al.* [51] provided a comprehensive assessment of the Docker Hub ecosystem, focusing on the detection of malicious images and the identification of exploitable parameters in Docker run commands. Other researchers have focused on analysing specific types of Docker Hub images. For instance, Zerouali *et al.* [57,56] analysed vulnerabilities and outdated packages in Debian-based and programming language-specific images [58] and Haque *et al.* [47] evaluated the exploitability of vulnerabilities in base images.

Research gap. Prior work has mainly addressed security issues in microservices distributed via Docker Hub. In contrast, our study focuses on serverless computing, which is rapidly becoming the dominant cloud application deployment model. We evaluated the vulnerability of serverless components to five distinct attack vectors, including two newly identified in this study, using components from five prominent public repositories. By analysing multiple repositories and diverse attack vectors, we provide a representative overview of current security practices in public serverless repositories.

8 Conclusion

This paper presents the first large-scale empirical analysis of security risks in public serverless repositories. Our study reveals systemic weaknesses across these repositories including (i) widespread use of vulnerable third-party dependencies; (ii) misconfigurations in IaC templates; (iii) sensitive parameters in Docker run commands; (iv) the ability to conceal malicious payloads in compressed serverless components; and (v) exposure to typo-squatting attacks. Based on these findings, we offer actionable recommendations for repository maintainers, developers and users to enhance the security of public serverless repositories.

Acknowledgments. We thank the anonymous reviewers for their insightful feedback and help in improving this paper. This research received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe programme: ELASTIC (GA#101139067); Horizon Europe: FLUIDOS (GA#101070473) and LAZARUS (GA#101070303); and the UNICO I+D Cloud program funded by the Ministry of Economic Affairs and Digital Transformation and the European Union–NextGenerationEU within the framework of the Plan de Recuperación, Transformación y Resiliencia (PRTR) with the CLOUDLESS project. This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. Additionally, this work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00457937, Design and implementation of security layers for secure WebAssembly-based serverless environments). The content of this article does not reflect the official opinion of the EU. Responsibility for the information and views expressed lies entirely with the authors.

References

1. Access-Control-Allow-Origin header - HTTP - MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Access-Control-Allow-Origin#directives
2. Alibaba Cloud Function Compute. https://www.alibabacloud.com/en/product/function-compute?_p_lc=1
3. AWS CLI Command Reference. https://awscli.amazonaws.com/v2/documentation/api/latest/reference/lambda/create-function.html
4. AWS CloudFormation. https://aws.amazon.com/cloudformation/
5. AWS Lambda. https://aws.amazon.com/lambda/
6. AWS SAM. https://aws.amazon.com/serverless/sam/?nc1=h_ls
7. AWS Serverless Application Repository, https://aws.amazon.com/serverless/serverlessrepo/
8. Azure Microsoft. https://azure.microsoft.com/en-us/solutions/serverless
9. Beautifulsoup. https://pypi.org/project/beautifulsoup4/
10. Create your first containerized functions on Azure Container Apps. https://learn.microsoft.com/en-us/azure/azure-functions/functions-deploy-container-apps
11. Deploy a Cloud Function. https://cloud.google.com/functions/docs/deploy
12. Deploy Lambda functions with container images. https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/deploy-lambda-functions-with-container-images.html
13. Deploying Lambda functions as .zip file archives. https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-zip.html
14. Docker Hub, https://hub.docker.com/
15. Ensure that lambda function permission has a source arn specified. https://aquasecurity.github.io/tfsec/v1.0.0-rc.8/checks/aws/lambda/restrict-source-arn/
16. GCP Cloud Run: Serverless Deployment. https://medium.com/@109manojsaini/serverless-deployment-cloud-run-3332c3817ef9
17. Github. https://github.com/
18. Google Cloud Serverless. https://cloud.google.com/solutions/serverless?hl=en
19. Grype Documentation. https://docs.anchore.com/current/docs/
20. IBM Cloud Functions. https://www.ibm.com/cloud/functions
21. Identify AWS resources with Amazon Resource Names (ARNs). https://docs.aws.amazon.com/IAM/latest/UserGuide/reference-arns.html
22. Red Hat Quay. https://quay.io/
23. S3 encryption should use Customer Managed Keys. https://aquasecurity.github.io/tfsec/v1.6.2/checks/aws/s3/encryption-customer-key/
24. Selenium. https://www.selenium.dev/
25. Serverless framework plugins. https://www.serverless.com/plugins/
26. Terraform. https://developer.hashicorp.com/terraform
27. Trivy - IaC. https://trivy.dev/v0.19.2/misconfiguration/iac/
28. Trivy Documentation. https://aquasecurity.github.io/trivy/v0.44/docs/
29. Virustotal. https://www.virustotal.com/
30. Zip deployment for Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/deployment-zip-push
31. Cryptojacking Invades Cloud. How Modern Containerization Trend is Exploited by Attackers. https://threatpost.com/malicious-docker-containers-earn-crypto-miners-90000/132816/ (2018)

32. Compare Trivy and Grype. https://gitlab.com/gitlab-org/gitlab/-/issues/327174 (2021)
33. AWS Serverless Application Repository – FAQs and Terms. https://aws.amazon.com/serverless/serverlessrepo/faqs/ (2023)
34. EleKtra-Leak Cryptojacking Attacks Exploit AWS IAM Credentials Exposed on GitHub. https://thehackernews.com/2023/10/elektra-leak-cryptojacking-attacks.html (2023)
35. Multiple False Positive and False Negative CVEs. https://github.com/aquasecurity/trivy/issues/3010 (2023)
36. Operation "Red Kangaroo": Industry's First Dynamic Analysis of 4M Public Docker Container Images. https://www.algosec.com/blog/operation-red-kangaroo-industrys-first-dynamic-analysis-of-4m-public-docker-container-images (2023)
37. Stemming the tide of false positive vulnerabilities. https://www.chainguard.dev/unchained/stemming-the-tide-of-false-positive-vulnerabilities (2023)
38. Why Chainguard uses Grype as its first line of defense for CVEs. https://www.chainguard.dev/unchained/why-chainguard-uses-grype-as-its-first-line-of-defense-for-cves (2023)
39. Hacking Serverless Runtimes Profiling Lambda, Azure, and more. https://www.blackhat.com/docs/us-17/wednesday/us-17-Krug-Hacking-Severless-Runtimes.pdf (2024)
40. Using cross-origin resource sharing (CORS). https://docs.aws.amazon.com/AmazonS3/latest/userguide/cors.html (2024)
41. Castro, P., Ishakian, V., Muthusamy, V., Slominski, A.: The Rise of Serverless Computing. Communications ACM **62**(12), 44–54 (2019)
42. Chen, J., Jiang, J., Duan, H., Wan, T., Chen, S., Paxson, V., Yang, M.: We still Don't have secure Cross-Domain requests: an empirical study of CORS. In: USENIX Security. pp. 1079–1093 (2018)
43. Dahlmanns, M., Sander, C., Decker, R., Wehrle, K.: Secrets Revealed in Container Images: An Internet-wide Study on Occurrence and Impact. In: ASIACCS. pp. 797–811 (2023)
44. Datta, P., Polinsky, I., Inam, M.A., Bates, A., Enck, W.: ALASTOR: Reconstructing the provenance of serverless intrusions. In: USENIX Security. pp. 2443–2460 (2022)
45. Eskandani, N., Salvaneschi, G.: The Wonderless Dataset for Serverless Computing. In: MSR. pp. 565–569 (2021)
46. Franco, J., Acar, A., Aris, A., Uluagac, S.: Forensic Analysis of Cryptojacking in Host-Based Docker Containers Using Honeypots. In: ICC. pp. 4860–4865 (2023)
47. Haque, M.U., Babar, M.A.: Well Begun is Half Done: An Empirical Study of Exploitability & Impact of Base-Image Vulnerabilities (2021), https://arxiv.org/abs/2112.12597
48. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.J., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud Programming Simplified: A Berkeley View on Serverless Computing (2019), http://arxiv.org/abs/1902.03383
49. Ladisa, P., Plate, H., Martinez, M., Barais, O.: SoK: Taxonomy of attacks on open-source software supply chains. In: S&P. pp. 1509–1526 (2023)
50. Liu, G., Gao, X., Wang, H., Sun, K.: Exploring the Unchartered Space of Container Registry Typosquatting. In: USENIX Security. pp. 35–51 (2022)

51. Liu, P., Ji, S., Fu, L., Lu, K., Zhang, X., Lee, W.H., Lu, T., Chen, W., Beyah, R.: Understanding the Security Risks of Docker Hub. In: ESORICS. pp. 257–276 (2020)
52. Marin, E., Perino, D., Di Pietro, R.: Serverless Computing: A Security Perspective. Journal of Cloud Computing **11**(1) (2022)
53. Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N.J., Popa, R.A., Gonzalez, J.E., Stoica, I., Patterson, D.A.: What serverless computing is and should become: the next phase of cloud computing. Commun. ACM pp. 76–84 (2021)
54. Shu, R., Gu, X., Enck, W.: A Study of Security Vulnerabilities on Docker Hub. In: CODASPY. pp. 269–280 (2017)
55. Wist, K., Helsem, M., Gligoroski, D.: Vulnerability Analysis of 2500 Docker Hub Images (2020), https://arxiv.org/abs/2006.02932
56. Zerouali, A., Mens, T., Decan, A., Gonzalez-Barahona, J., Robles, G.: A multi-dimensional analysis of technical lag in Debian-based Docker images. Empirical Softw. Engg. **26**(2) (2021)
57. Zerouali, A., Mens, T., Robles, G., Gonzalez-Barahona, J.M.: On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In: SANER. pp. 491–501 (2019)
58. Zerouali, A., Mens, T., Roover, C.D.: On the usage of JavaScript, Python and Ruby packages in Docker Hub images. Science of Computer Programming **207**, 102653 (2021)

A Vulnerability Composition

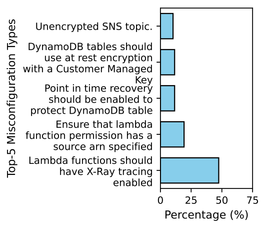
Table 4: Top 10 most commonly used packages across five platforms, along with the number and severity of associated vulnerabilities (C: *Critical*, H: *High*, M: *Medium*, L: *Low*).

AWS Serverless Repository						GitHub						Serverless Framework						Docker Hub						Red Hat Quay					
Package	**Rate**	**C**	**H**	**M**	**L**	**Package**	**Rate**	**C**	**H**	**M**	**L**	**Package**	**Rate**	**C**	**H**	**M**	**L**	**Package**	**Rate**	**C**	**H**	**M**	**L**	**Package**	**Rate**	**C**	**H**	**M**	**L**
xml2js	4.96%	0	0	1	0	semver	27.53%	0	0	2	0	xml2js	19.44%	0	0	1	0	tar	74.09%	0	7	6	2	ncurses-base	82.67%	0	1	5	21
follow-redirects	4.55%	0	1	2	0	qs	23.74%	0	4	1	0	lodash	19.15%	1	4	2	1	semver	63.97%	0	0	2	0	ncurses-libs	74.67%	0	0	5	22
axios	4.13%	0	2	2	0	follow-redirects	21.63%	0	1	2	0	semver	18.03%	0	0	1	0	curl	61.72%	25	34	47	16	libgcc	68.00%	0	0	12	14
lodash	3.72%	1	3	1	0	xml2js	21.21%	0	0	1	0	minimatch	17.46%	0	3	0	0	minimatch	57.50%	0	3	0	0	pcre2	62.67%	0	0	2	1
minimist	3.72%	1	0	1	0	minimist	21.07%	1	0	1	0	qs	13.80%	0	1	0	0	ncurses-base	57.35%	2	6	13	6	curl	56.00%	13	10	46	24
urllib3	3.72%	0	3	5	0	minimatch	20.22%	0	3	0	0	minimist	13.52%	1	0	1	0	got	57.21%	0	0	1	0	ca-certificates	54.67%	0	2	0	1
minimatch	2.89%	0	1	0	0	lodash	19.52%	1	4	2	1	aws-sdk	11.83%	0	1	0	0	qs	57.21%	0	4	1	0	libxml2	52.00%	2	1	31	7
aws-sdk	2.89%	0	1	0	0	tough-cookie	19.10%	0	1	1	0	axios	9.01%	0	2	2	0	openssl	56.77%	4	29	84	34	tar	52.00%	0	7	3	5
qs	2.48%	0	1	0	0	axios	18.96%	0	2	2	0	follow-redirects	9.01%	0	1	2	0	minimist	54.29%	1	0	1	0	glib2	48.00%	0	2	48	20
semver	2.48%	0	0	1	0	node-fetch	15.73%	0	1	1	1	async	8.45%	0	1	0	0	xml2js	53.28%	0	0	1	0	gnupg2	48.00%	0	1	4	6

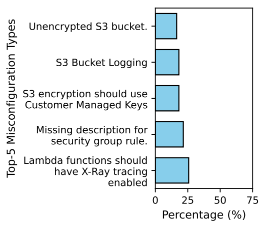
To gain deeper insight into the vulnerability landscape, we analysed the top 10 most commonly used packages in each repository (see Table 4). Notably, the *lodash* and *minimist* packages, each affected by one critical vulnerability (`CVE-2019-10744` and `CVE-2021-44906`, respectively) appear in three and four of the analysed repositories, respectively. A cross-repository comparison reveals that versions of these packages hosted in the AWS SAW consistently exhibit fewer vulnerabilities, suggesting that AWS may actively patch or curate its hosted packages. In contrast, Docker Hub and Red Hat Quay show significantly higher vulnerability counts for the same packages, likely due to frequent image reuse and less stringent update practices. Among all analysed packages, *curl* stands out for both its widespread use—particularly in Docker Hub and Red Hat Quay—and its high number of critical and high-severity vulnerabilities.

B Common Misconfigurations in IaC Templates

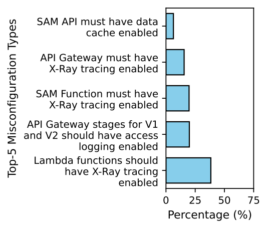
Figure 7 presents the five most frequently observed misconfigurations across the three analyzed IaC frameworks: Terraform, AWS CloudFormation, and AWS Serverless Application Model (SAM). The obtained results highlight recurring security issues that affect the security posture of serverless deployments.



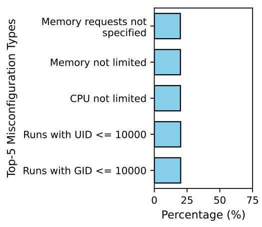
(a) Terraform in AWS



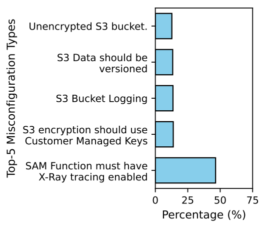
(b) Terraform in GitHub



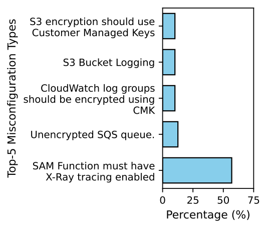
(c) CloudFormation in AWS



(d) CloudFormation in GitHub



(e) SAM in AWS



(f) SAM in GitHub

Fig. 7: Top-5 most common misconfigurations in IaC templates from AWS SAR and GitHub.