Exploiting FaaS - AWS Lambda/Azure Functions/Logic Apps (version 1.2)

**Cloud Service Label: PaaS**

Description

Both Azure and AWS employ “serverless” code functionality that can be set to automatically trigger by other events within the cloud API. Coding for these functions is a new paradigm that isn’t well understood from a security perspective. Many of the vulnerabilities associated with other programming paradigms apply but the mitigations often don’t. There really aren’t code scanners that focus on FaaS code. It’s often not feasible to protect FaaS functions with proxies or firewalls. Restricting inputs to FaaS functions is often antithetical to their reason for existing. FaaS functions often take inputs from logs that are not controlled by an application developer. Sample code has already been released showing what is possible if vulnerable FaaS functions ingest log messages that have been cleverly formatted to take advantage of coding vulnerabilities. Exploiting FaaS functions can be used both to gain initial access to a cloud enterprise, collect intelligence on internal cloud assets, to elevate privileges during an existing intrusion, and to evade detection during intrusion.

Examples

|  |  |  |
| --- | --- | --- |
| **Name** | **Tactic** | **Description** |
| Puresec |  | Blog posts about exploiting FaaS functions |
| NetSPI | Privilege escalation | Demonstrates code to extract Bearer tokens from FaaS functions in Azure if you can insert code into an existing FaaS function. These tokens may have expansive rights though they are usually limited to a particular Azure service. |
| NetSPI | Persistence | FaaS functions can be used to maintain persistence after initial access has been made. Automation accounts can create scheduled tasks or web hook to maintain persistence. |
| NetSPI | Defense Evasion | FaaS functions can be utilized for defense evasion. There are testing functionality for automation accounts where code and other commands can be run to avoid detection since this is not logged. |
| Puma Security | Privilege escalation, persistence | Open source scripts to access FaaS credentials from AWS, Azure and GCP |
| Rhino Security Labs | Privilege escalation | FaaS functions can be manipulated to include custom versions of python libraries and malicious code. In an example demonstrated by Rhino Security Labs, the boto3 python library was customized and uploaded to a new Lambda layer to include code that steals the function’s environment variables. |

Mitigations

|  |  |  |
| --- | --- | --- |
| **Mitigation** | | **Description** |
| Audit | | Frequently check permissions on cloud storage to ensure proper permissions are set to deny open or unprivileged access to resources. Consider using automated resource checkers such as CloudSploit or Divvycloud. Frequently check API logs for lambda/function creation events. |
|  | AWS | To perform an audit via AWS it is suggested to review information such as account details (credentials, users, groups, roles, etc), mobile applications, EC2 configurations, policies, and account activity. How to audit these different factors can be found in detail at: **https://docs.aws.amazon.com/general/latest/gr/aws-security-audit-guide.html.** |
|  | Azure | To perform an audit via Azure an administrator can review the audit logs that are recorded under Azure’s monitoring for active directory. The audit logs allow for filtering, as well as looking at users, groups, and enterprise specific information. Full details on how to access this information can be found at: **https://docs.microsoft.com/en-us/azure/active-directory/reports-monitoring/concept-audit-logs.** |
|  | GCP | To perform an audit via GCP the logs can be reviewed. GCP breaks this down into three categories; admin activity, data access, and system events. The audit logs can be viewed a few different ways- the console, API, or gcloud. Full details on how to view these logs, how to export, and for how to configure the retention period can be found here: **https://cloud.google.com/logging/docs/audit.** |
| Secure Coding | | Avoid directly processing any input that can be generated directly by a user. Parse all inputs to functions carefully. |
| Limit Access | | Provision FaaS functions with the minimum absolute permissions possible. In AWS, ensure lambda:CreateFunction and lambda:UpdateFunctionConfiguration permissions are locked down. In Azure use Managed Identities for authorization and authentication since this makes such accesses easier to audit. |
| Associate function with a VNET or VPC | | This may cost more in Azure’s case, but the function can be protected by CSP network protections such as security groups to prevent credentials from being used from unexpected locations |

Detection

Detecting the presence of corrupted FaaS apps is exceedingly difficult currently. More likely side effects that result from such corruption like illicit network communications or unexplained account creations are more likely to be noticed in logging. For instance in Azure service accounts (MSI) the diagnostic logs may show a token authentication from non-Microsoft address spaces using a client that is not a Microsoft agent.

https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation-part-2/ References

1. https://github.com/puresec/sas-top-10 Retrieved Feb 20,2020
2. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8611 Retrieved May 14, 2020
3. https://blog.netspi.com/gathering-bearer-tokens-azure/#4 Retrieved June 8, 2020
4. https://pumasecurity.io/resources/blog/RSAC-2020-Defending-Serverless-Infrastructure-in-the-Cloud Retrieved June 9,2020 Accessed July 21, 2020