How We Escaped Docker in Azure Functions

Written by **Paul Litvak** - 27 January 2021

[Summary of Findings](#)
[What is Azure Functions?](#)
[Technical Analysis](#)
[Proof of Concept](#)
[Why Does this Matter?](#)

Summary of Findings

In previous months we identified vulnerabilities in Microsoft Azure Network Watcher and Azure App Services, leading us to investigate other types of Azure compute infrastructure. We found a new vulnerability in Azure Functions, which would **allow an attacker to escalate privileges and escape the Azure Functions Docker container to the Docker host**.

We reported the vulnerability to Microsoft's security team. They have determined the issue has no security impact on Azure Functions users. Although it is possible to escape from the function to the host, the Docker host itself is protected by a Hyper-V boundary. Based on our findings Microsoft has made changes to block /etc and /sys directories since this change has already been deployed.

Instances like this underscore that vulnerabilities are sometimes out of the cloud user's control. Attackers can find a way inside through vulnerable third-party software. While you should focus on reducing the attack surface as much as possible, you also need to prioritize the runtime environment to make sure you don't have any malicious code lurking in your systems.

What is Azure Functions?

Azure Functions is a serverless compute service that allows users to run code without having to provision or manage infrastructure. Azure Functions is Microsoft's equivalent to Amazon Web Services' well-known Lambda service.

Azure Functions can be triggered by HTTP requests and are meant to run for only a few minutes in order to handle the event. Behind the scenes, the user's code is run on an Azure-managed container and served without requiring the user to manage their own infrastructure. In other words, if the user wants to take a shortcut they can, since it's expected that Microsoft will do it for them. This code is segmented securely and is not intended to escape from its confined environment. However, we will soon demonstrate why this is not the case.

We created a demonstration of the vulnerability—mimicking an attacker having execution on Azure Functions and escalating privileges to achieve a full escape to the Docker host. Check it out below.

Technical Analysis

An Azure function requires no infrastructure management. It’s triggered by a user merely uploading their code, which enables seamlessly calling the Function. In our example, it’s invoked via HTTP: https://test11114117.azurewebsites.net

Figure 1: Example Azure Function handler code

As the user can upload any code of their choice, we abused this to gain a foothold over the Function container and further understand its internals. We wrote a reverse shell to connect to our control server once the Function was executed, so that we could operate an interactive shell.

Figure 2: Azure Function reverse shell

Once the shell was on our Function we noticed that we were running as a unprivileged 'app' user in an endpoint with a 'SandboxHost' hostname:

Figure 3: Connecting to the Function reverse shell

The environment was mostly sterile from utilities, so we added several useful tools—most notably nmap—to our Function directory and then reuploaded the new Function package.

Using nmap, we scanned localhost to familiarize ourselves with the server. As a result we spotted multiple open ports:

Figure 4: Running nmap on an Azure Function

Escalating Privileges

Since our goal was to find an elevation of privilege vulnerability, it was important that we find sockets belonging to processes associated with root. After interrogating network-related /proc files, we were able to map the ports to their corresponding processes:

Figure 5: Mapping each open port to the process that owns it

We found three privileged processes with an open port. The first was NGINX, a thoroughly tested open-source project. The local NGINX version had no known vulnerabilities so this wouldn't have helped us.

The MSI and Mesh processes offered better chances at finding potential problems as they are close-sourced, undocumented Microsoft processes. As such, we were confident that they had been less thoroughly tested.

MSI, Managed Service Identity, a feature of the serverless model, eliminates the user's need to manage identities, easing development by letting Azure handle it instead.

As for the Mesh binary, we couldn't find much information (it's unrelated to Azure's Fabric Mesh service which has a similar name).

Unfortunately, the binaries belonging to the two processes reside in root-owned directories (e.g. /root/mesh/init) and we didn't have access to them.

The Mesh process seemed to be less documented and also very relevant for our purposes, so we focused our efforts on finding out what this component does.

After searching for references to the Mesh binary in Google, we found the questioned "/root/mesh/init" path in the build log of a public Docker image in Docker Hub belonging to a Microsoft employee (we deduced this was public on purpose because it's used internally somehow).

We downloaded the image, created a container with it and extracted the Mesh Init binary. The binary was compiled from a Go codebase and conveniently for our purposes wasn't stripped.

Immediately as we opened the binary in IDA we noticed some interesting functions:

Figure 6: Mesh binary mount functions

Performing a mount is a privileged operation and should our unprivileged user access this functionality through the HTTP server, it could result in privilege escalation.

With this goal in mind and after some reverse engineering, we found the HTTP paths and variables that would allow us to invoke these functions. The server expected an HTTP variable to specify an operation to invoke:

At first we attempted to use the mount_RunCifs and mount_RunZip commands, however, we had no success as the system was lacking binaries for these functions to actually work. We had hoped that the third time would be the charm as we looked at mount_RunSquash:

Figure 7: mount_RunSquash function disassembly

The RunSquash function would simply invoke squashfuse_ll (in the init_server_pkg_mount_runSquashInternal function) to mount the given squashfs image in the path supplied by “filePath” HTTP variable onto the path specified by the “targetPath” HTTP variable.

With this information, we built our own squashfs filesystem containing only a single file that would grant our unprivileged app user root permissions using the sudoers mechanism.

Figure 8: Creating the sudoers file on our own server

Figure 9: Creating the squashfs image on our own server for the exploit

We included this file in our new Function image and instructed the server to mount our evil squashfs image over */etc/sudoers.d*. This granted root to our unprivileged user:

Figure 10: Escalating to root

Escaping Docker

We were able to escalate to root! However, we were still confined to our container. This new freedom was still somewhat limiting but nonetheless an upgrade to a bigger "cage."

Escalating to root within a container is a remarkable achievement, yet escalating privileges within them much more control, allowing them to break away from the container which might be monitored and moving to the Docker host which is often neglected in terms of security. Containers are often scraped for unnecessary items which the attacker might find interesting, so escalating to the Docker host could allow them to gather more compromising leads to incite further damage.

It's a known bad practice to host containers with the *––privileged* flag, or to grant them non-default capabilities, since this nullifies Docker's security features. Seeing as Azure Functions' core is its container, the first thing we did once we had execution over the Function was to check what capabilities our container had been granted. This can be achieved by reading a process's status file in the */proc* directory:

Figure 11: Azure Function process capabilities

The Cap fields relate to a Linux capability mechanism. We won't go into detail but decoding the Cap bitmap allows us to list the process's capabilities, which all processes in the container share:

Figure 12: Decoding Function process capabilities

We were very surprised to discover that Azure Functions ran with several extra capabilities. With these extra capabilities it was clear that the container was run with the ––privileged flag.

This by itself would not have helped us initially, since we only had access to an unprivileged user, and the Docker escape techniques available in this scenario required root. This all changed once we found the Privilege Escalation vulnerability.

Figure 13: Running `ps` on the Docker Host

In a nutshell, the technique we used—discovered by Felix Wilhem—abuses a feature within cgroups and allows calling a binary on the Docker host (only with the SYS_ADMIN capability as given by the **--**privileged flag). In our PoC, we instructed the system to run the 'ps' command and redirect its output to our containerized filesystem.

Once we have achieved execution on the Docker host we reported our findings to Microsoft. After assessment they have decided not to fix the bug, as they claim it does not impact security. The reason for this is because the Docker host is not the final host by itself. This "host" was managed by HyperV (Virtual Machine Manager) and protected by its sandbox, therefore our container was essentially a box within a box. This Docker host only contains our own Docker container, and it's this real host that manages shared infrastructure between different Azure Functions belonging to various Azure customers, which we were not able to access.

Proof of Concept

To make reproduction easier for those who would like to probe the Docker host environment, we've created an easy to run PoC. It contains instructions on how to upload an Azure Function with a reverse shell so that you can probe the Docker host yourself and perhaps find some use out of it. It's available on GitHub.

Why Does this Matter?

No matter how hard you work to secure your own code, sometimes vulnerabilities are out of your control. It's critical that you have protection measures in place to detect and terminate when the attacker executes unauthorized code in your production environment. This Zero Trust mentality is even echoed by Microsoft.

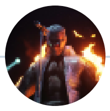
Try our Free Community Edition

Cloud Workload Protection Platforms (CWPP) like Intezer Protect monitor the runtime environment to detect and terminate any unauthorized code execution following a vulnerability exploitation or other attack vector.

Intezer Protect defends the cloud-native stack—including VMs, containers and container orchestration platforms—against the latest threats. You'll want to know what code is running in your production environments at all times. The community edition is a quick way to get this visibility.

Get Started for Free

If you're not ready to deploy, we also have a lab environment where you can simulate attacks such as backdoors, malware, and Living off the Land (LotL) threats. Contact us to access this environment.



Paul Litvak

Paul is a malware analyst and reverse engineer at Intezer. He previously served as a developer in the Israel Defense Force (IDF) Intelligence Corps for three years.

AZURE AZURE FUNCTIONS CLOUD SECURITY DOCKER DOCKER SECURITY RESEARCH RUNTIME PROTECTION VULNERABILITY

Previous Article
Swat Away Pesky Linux Cryptomine...

Next Article
Fix Your Misconfigured Docker API ...

Recommended Articles

25 MIN READ

Technical Analysis of a Novel IMEEX Framework

The IMEEX framework is a newly discovered, custom-built malware designed to target Windows systems....

10 October 2024

12 MIN READ

There’s Something About CryptBot: Yet Another Silly Stealer (YASS)

Recently Intezer was investigating a file that we came across during alert triage. This...

10 September 2024

11 MIN READ

Dissecting SSLoad Malware: A Comprehensive Technical Analysis

SSLoad is a stealthy malware that is used to infiltrate systems through phishing emails,...

10 June 2024

Subscribe to our Blog

Business Email

Subscribe

Share article

TOP BLOGS

Log In

Product

Autonomous SOC Platform

Pricing

Intezer for MSSPs

Integrations

Solutions

Reported Phishing

Endpoint Triage

SIEM Triage

SOAR Playbooks

Malware Analysis

Company

About

Contact Us

Security

Partners

News

Careers

Learn

The SecOps Automation Blog

FAQ

Documentation

Resources

YouTube Channel

Featured Resources

How Intezer’s AI-Powered Autonomous SOC Platform Works

Maximizing Incident Response Automation for Investigations

Supercharge These 3 Top Incident Response SOAR Playbooks

How Artificial Intelligence Powers the Autonomous SOC Platform