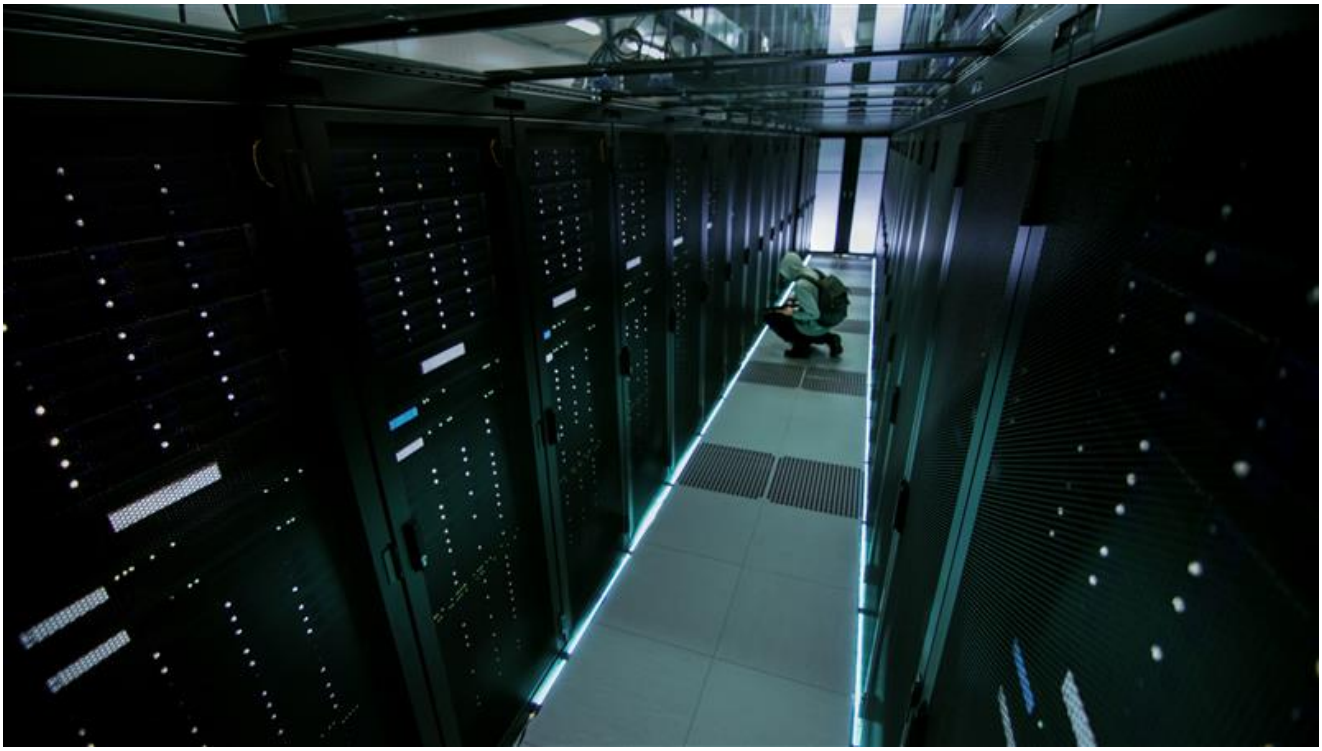
Security Breaks: TeamTNT's DockerHub Credentials Leak

trendmicro.com/en_us/research/22/i/security-breaks-teamtnts-dockerhub-credentials-leak.html

September 12, 2022



Content added to Folio

Cloud

One of our honeypots based on exposed Docker REST APIs showed cybercriminal group TeamTNT's potential attack scenario and leak of container registry credentials for docker-abuse malware. The full version of this research will be presented at the c0c0n XV Hacking and Cyber Security Conference in September 2022.

By: Nitesh Surana September 12, 2022 Read time: (words)

We constantly deploy and study our honeypots to get a view of actively exploited vulnerabilities and misconfigurations on platforms and services that pose cloud security risks. One of these honeypots is based on exposed Docker REST API for analysis from cloud

services providers' and users' perspectives. Upon analyzing the samples, we realized and were able to understand the threat actors' use of container registry features for Docker malware and tactics, techniques, and procedures (TTPs).

Our honeypots showed threat actor TeamTNT were leaking credentials from at least two of their attacker-controlled DockerHub accounts, namely *alpineos* (with over 150,000 pulls) and *sandeep078* (with 200 pulls). We have notified Docker about these accounts.

The account alpineos was used in exploitation attempts on our honeypots three times, from mid-September to early October 2021, and we tracked the deployments' IP addresses to their location in Germany. The threat actors were logged in to their accounts on the DockerHub registry and probably forgot to log out. Unless a user is not logged out manually, the header "X-Registry-Auth" stores the credentials.

These DockerHub profiles were actively used to deploy malicious images containing the following:

1. Rootkits
2. Docker escape kits
3. XMRig Monero miners
4. Credential stealers
5. Kinsing malware
6. Kubernetes exploit kits

In July 2021, we published our research on TeamTNT's malicious activities and found evidence of the group infiltrating via the Docker API. As a result, we found 26 unique DockerHub accounts that are either compromised or malicious. Of the two we identified here, the most interesting account for study was the alpineos account, which hosted malicious container images with over 150,000 pulls.

Container registries and Docker daemon

Docker is a container services platform that helps developers follow a write-once-run-anywhere (WORA) practice. It's simple to use and is favored by developers, as a user can write services and deploy applications at great speed. Most importantly, Docker works with any platform.

Container registries are storage and distribution platforms for container images, similar to how codes or programs are hosted on repositories like GitHub. With the right authorization context, one can simply "pull" an image, create a container based on it, and deploy applications. Many container registries such as DockerHub, Amazon Elastic Container Registry (ECR), and Alibaba Container Registry, to name a few, host container images.

When you create a container, the container daemon looks up the image from the container registry by default. In our analysis, we use DockerHub as an example.

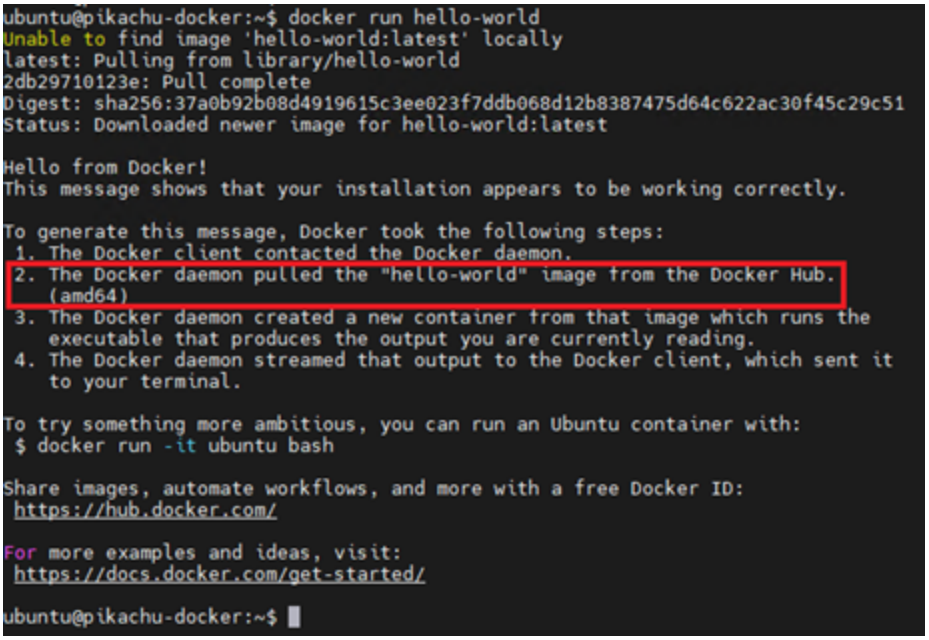


Figure 1. A “Hello World” example of the Docker daemon (dockerd) pulling the image from Docker Hub

If we don’t specify the registry, DockerHub is considered by default. Docker provides a feature for developers to create containers on a remote host when the Docker daemon (on the server) is configured to listen over the TCP port, which is port 2375 by default. This makes remote development and deployment easy for developers as it provides an interface to various Docker services like images, containers, networks, and volumes using tools like curl, wget, and docker-cli.

Docker REST API for container creation

Consider a scenario where a new container with an alpine image base (Alpine Linux, a distribution based on musl libc library and BusyBox utilities) is created on remote server 172[.]31[.]42[.]11 via docker REST API. The remote server has the dockerd exposed over TCP port 2375.

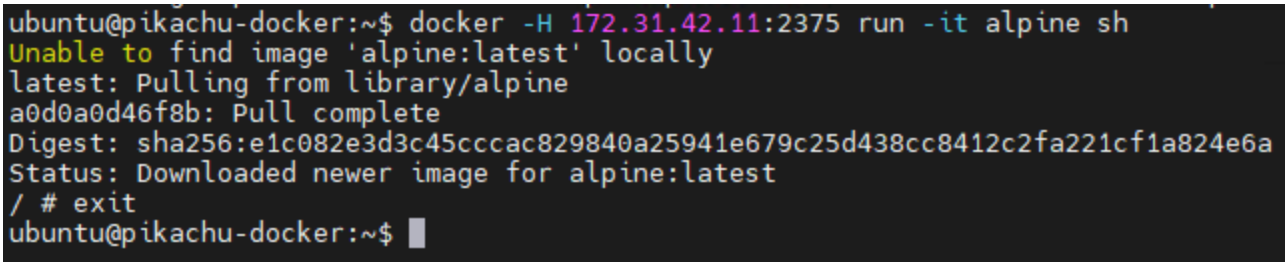
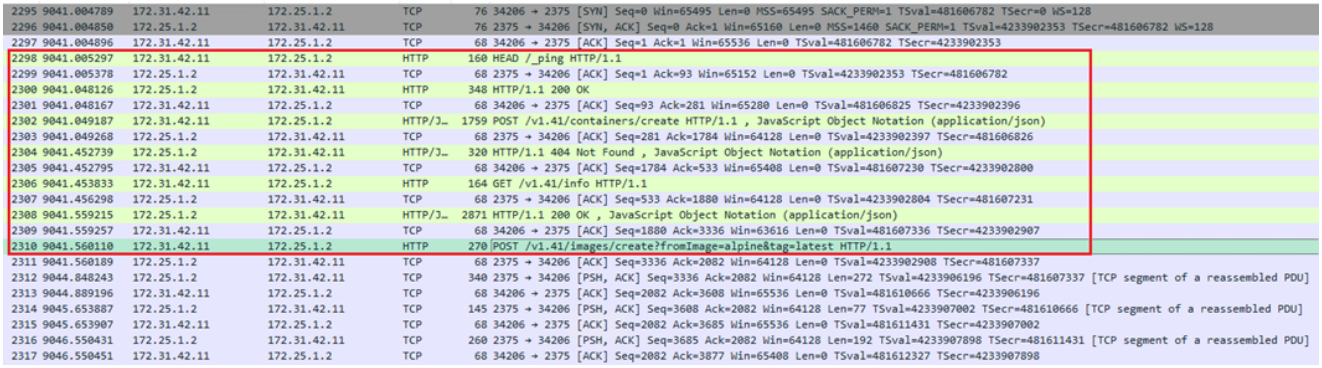


Figure 2. Container created based on alpine image on a remote host



```
2295 9041.004789  172.31.42.11  172.25.1.2     TCP    76 34206 → 2375 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=481606782 TSecr=0 WS=128
2296 9041.004850  172.25.1.2    172.31.42.11  TCP    76 2375 → 34206 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=4233902353 TSecr=481606782 WS=128
2297 9041.004896  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=481606782 TSecr=4233902353
2298 9041.005297  172.31.42.11  172.25.1.2     HTTP   160 HEAD /_ping HTTP/1.1
2299 9041.005378  172.25.1.2    172.31.42.11  TCP    68 2375 → 34206 [ACK] Seq=1 Ack=93 Win=65152 Len=0 TSval=4233902353 TSecr=481606782
2300 9041.048126  172.25.1.2    172.31.42.11  HTTP   348 HTTP/1.1 200 OK
2301 9041.048167  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=93 Ack=281 Win=65280 Len=0 TSval=481606825 TSecr=4233902396
2302 9041.049187  172.31.42.11  172.25.1.2     HTTP/J… 1759 POST /v1.41/containers/create HTTP/1.1 , JavaScript Object Notation (application/json)
2303 9041.049268  172.25.1.2    172.31.42.11  TCP    68 2375 → 34206 [ACK] Seq=281 Ack=1784 Win=64128 Len=0 TSval=4233902397 TSecr=481606826
2304 9041.452739  172.25.1.2    172.31.42.11  HTTP/J… 320 HTTP/1.1 404 Not Found , JavaScript Object Notation (application/json)
2305 9041.452795  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=1784 Ack=533 Win=65408 Len=0 TSval=481607230 TSecr=4233902800
2306 9041.453833  172.31.42.11  172.25.1.2     HTTP   164 GET /v1.41/info HTTP/1.1
2307 9041.456298  172.25.1.2    172.31.42.11  TCP    68 2375 → 34206 [ACK] Seq=533 Ack=1880 Win=64128 Len=0 TSval=4233902804 TSecr=481607231
2308 9041.559215  172.25.1.2    172.31.42.11  HTTP/J… 2871 HTTP/1.1 200 OK , JavaScript Object Notation (application/json)
2309 9041.559257  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=1880 Ack=3336 Win=63616 Len=0 TSval=481607336 TSecr=4233902907
2310 9041.560110  172.31.42.11  172.25.1.2     HTTP   270 POST /v1.41/images/create?fromImage=alpine&tag=latest HTTP/1.1
2311 9041.560189  172.25.1.2    172.31.42.11  TCP    68 2375 → 34206 [ACK] Seq=3336 Ack=2082 Win=64128 Len=0 TSval=4233902908 TSecr=481607337
2312 9044.848243  172.25.1.2    172.31.42.11  TCP    340 2375 → 34206 [PSH, ACK] Seq=3336 Ack=2082 Win=64128 Len=272 TSval=4233906196 TSecr=481607337 [TCP segment of a reassembled PDU]
2313 9044.889196  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=2082 Ack=3608 Win=65536 Len=0 TSval=481610666 TSecr=4233906196
2314 9045.653887  172.25.1.2    172.31.42.11  TCP    145 2375 → 34206 [PSH, ACK] Seq=3608 Ack=2082 Win=64128 Len=77 TSval=4233907002 TSecr=481610666 [TCP segment of a reassembled PDU]
2315 9045.653907  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=2082 Ack=3685 Win=65536 Len=0 TSval=481611431 TSecr=4233907002
2316 9046.550431  172.25.1.2    172.31.42.11  TCP    260 2375 → 34206 [PSH, ACK] Seq=3685 Ack=2082 Win=64128 Len=192 TSval=4233907898 TSecr=481611431 [TCP segment of a reassembled PDU]
2317 9046.550451  172.31.42.11  172.25.1.2     TCP    68 34206 → 2375 [ACK] Seq=2082 Ack=3877 Win=65408 Len=0 TSval=481612327 TSecr=4233907898
```

Figure 3. Network server traffic

Looking at the server network traffic log, we can see that when a new container is requested for creation on a remote server, this is the sequence that follows:

1. The client pings the target server (packet 2298) to test if the server is accessible.
2. The server responds with the status code 200 and that it is accessible (packet 2300).
3. The client requests that the server create a container from an image named “alpine” (packet 2302).
4. If the server cannot find the “alpine” image locally, it replies with the status code 404 (packet 2304).
5. The client requests for the server information from </<version>*/info endpoint*> (packet 2306).
6. The server responds to the request with system-wide information (packet 2308).

```
POST /v1.41/containers/create HTTP/1.1
Host: 172.31.42.11:2375
User-Agent: Docker-Client/20.10.7 (linux)
Content-Length: 1527
Content-Type: application/json

{"Hostname":"","Domainname":"","User":"","AttachStdin":true,"AttachStdout":true,"AttachStderr":true,"Tty":true,"OpenStdin":true,"StdinOnce":true,"Env":null,"Cmd":
["sh"],"Image":"alpine","Volumes":{},"WorkingDir":"","Entrypoint":null,"OnBuild":null,"Labels":{},"HostConfig":{"Binds":null,"ContainerIDFile":"","LogConfig":
{"Type":"","Config":{}},"NetworkMode":"default","PortBindings":{},"RestartPolicy":{"Name":"no","MaximumRetryCount":
0},"AutoRemove":false,"VolumeDriver":"","VolumesFrom":null,"CapAdd":null,"CapDrop":null,"CgroupnsMode":"","Dns":[],"DnsOptions":[],"DnsSearch":
[],"ExtraHosts":null,"GroupAdd":null,"IpcMode":"","Cgroup":"","Links":null,"OomScoreAdj":
0,"PidMode":"","Privileged":false,"PublishAllPorts":false,"ReadonlyRootfs":false,"SecurityOpt":null,"UTSMode":"","UsernsMode":"","ShmSize":0,"ConsoleSize":
[0,0],"Isolation":"","CpuShares":0,"Memory":0,"NanoCpus":0,"CgroupParent":"","BlkioWeight":0,"BlkioWeightDevice":
[],"BlkioDeviceReadBps":null,"BlkioDeviceWriteBps":null,"BlkioDeviceReadIOps":null,"BlkioDeviceWriteIOps":null,"CpuPeriod":0,"CpuQuota":0,"CpuRealtimePeriod":
0,"CpuRealtimeRuntime":0,"CpusetCpus":"","CpusetMems":"","Devices":[],"DeviceCgroupRules":null,"DeviceRequests":null,"KernelMemory":0,"KernelMemoryTCP":
0,"MemoryReservation":0,"MemorySwap":0,"MemorySwappiness":-1,"OomKillDisable":false,"PidsLimit":0,"Ulimits":null,"CpuCount":0,"CpuPercent":0,"IOMaximumIOps":
0,"IOMaximumBandwidth":0,"MaskedPaths":null,"ReadonlyPaths":null},"NetworkingConfig":{"EndpointsConfig":{}},"Platform":null}
HTTP/1.1 404 Not Found
Api-Version: 1.41
Content-Type: application/json
Docker-Experimental: false
Ostype: linux
Server: Docker/20.10.7 (linux)
Date: Sun, 17 Oct 2021 09:42:40 GMT
Content-Length: 43

{"message":"No such image: alpine:latest"}
```

Figure 4. Showing 404 status code as alpine image is locally unavailable

Figure 5. Returning system-wide information

7. The client requests the server to create a container from the alpine image. The “latest” tag is chosen when no tags are specified.

8. The server responds with the download progress of the alpine image from DockerHub.

Figure 6. Creating a container from the alpine image

9. The client requests the server to create a container from the now-downloaded alpine image.

Figure 7. Request to create a container; the server responds with the ID of the newly created container

Notice the value in the X-Registry-Auth header from Figure 6, where the header is with a Base64-encoded string {}. The Docker documentation in Figure 8 enumerates the authentication details:

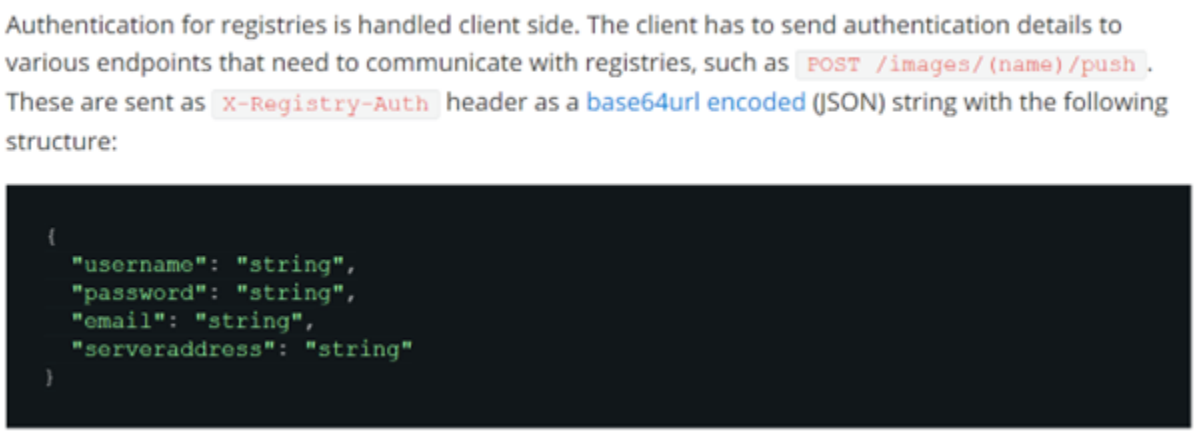


Figure 8. Docker documentation on registries' authentication

In the said scenario, the client who initiated the creation of the alpine-based container on the server did not log in to the DockerHub registry. Hence, the value of the X-Registry-Auth header is {} encoded in Base64. Using "docker login", one can authenticate to container registries and securely work on their repository.

Should an authorized user repeat the procedure wherein a legitimate user with a profile of "satoshiav0cad0" is logged in using "docker login" and looks at the same header value, the header X-Registry-Auth would now contain the credentials encoded in Base64.

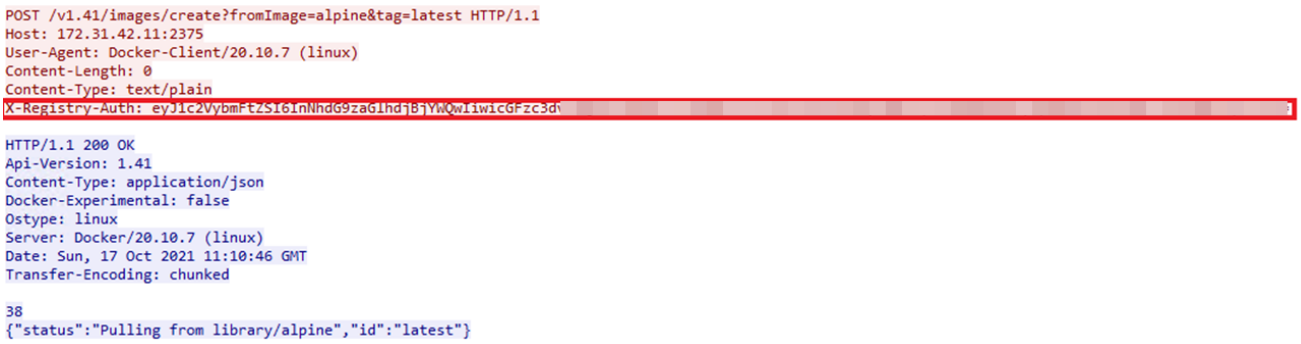


Figure 9. Credentials encoded in Base64

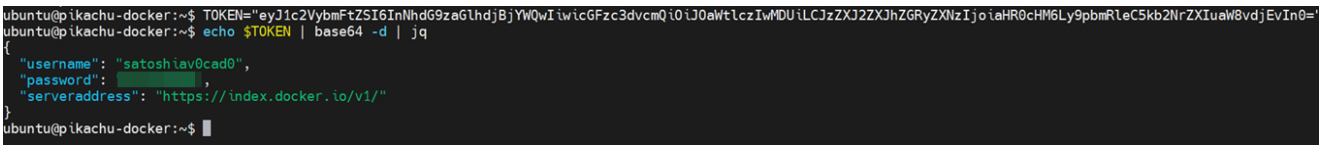


Figure 10. Decoding the credentials from Base64

These are the DockerHub credentials of user "satoshiav0cad0". Upon analysis, the credentials can be seen in the aforementioned header X-Registry-Auth only because the client initiating the request to create a container on a target server had authenticated it to their DockerHub container registry.

As a legitimate use case, a user might want to authenticate their DockerHub repository to create containers based on the images in their private repository. But if the user forgets to log out from DockerHub using “docker logout” and creates containers on untrusted hosts with dockerd exposed over TCP, the user becomes at risk since their usernames and passwords are hard-coded and non-encrypted, not to mention only encoded in Base64.

Credential leak scenarios

In the first scenario, the victim is logged in to their DockerHub registry. An attacker gains access to the victim’s virtual machine (VM) and tries to create a container on a remote server with the dockerd exposed over TCP. If the image that the container attempts for creation does not exist, the image is pulled from DockerHub and the header X-Registry-Auth is populated with the Base64-encoded credentials.

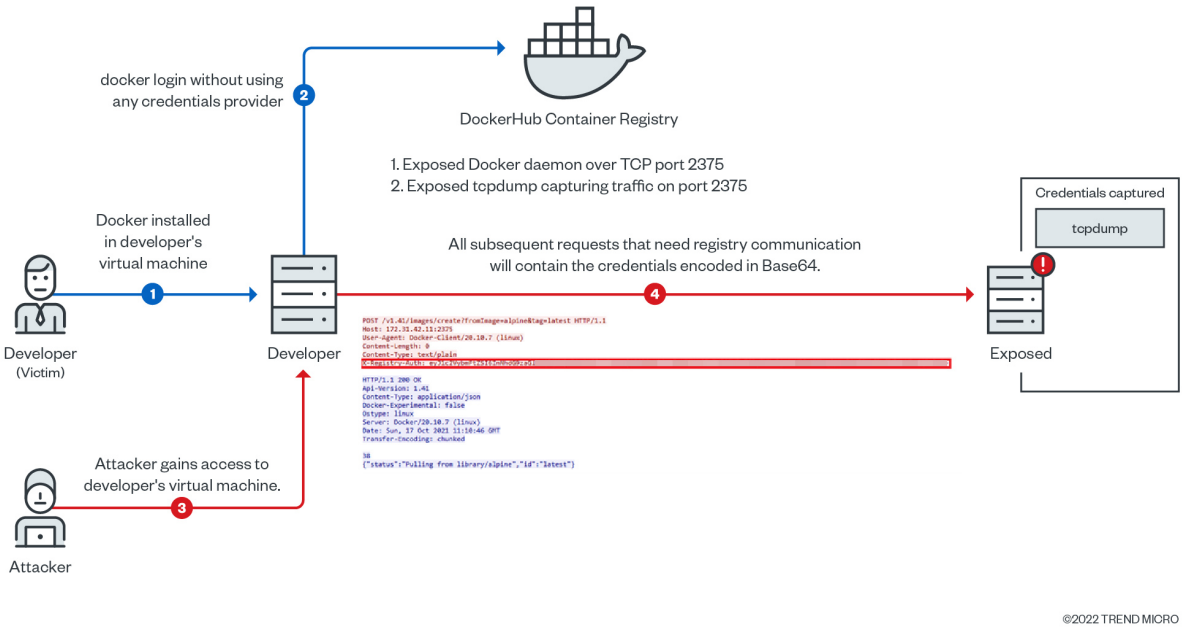


Figure 11. Scenario 1: The image pulled from DockerHub contains the Base64-encoded credentials.

Once abused, these compromised accounts can be used to view the following information and even pivot in the following ways:

1. **Associated email and reused credentials.** Cybercriminals can check for passwords being reused across different platforms and check for password leaks.
2. **Private repositories and images.** These might contain credentials like API keys and modify private images with backdoors.
3. **Access tokens.** These can be used to maintain persistent access to the account.
4. **Developers’ tools.** Pro features (like Teams, Organization, and Build pipelines) can be used to contaminate the build pipelines based on Docker and lead to supply-chain attacks.

From a different perspective, another attack scenario could involve the attacker's account itself, where the cybercriminals are logged in to their own DockerHub registry and their accounts leak credentials. While legitimate users might not be concerned about stealing threat actors' credentials, we analyzed that the procedure in creating a new container is also applicable to catching threat actors who might have left their respective accounts logged in. They might then attempt to create a container in the honeypot with an exposed dockerd over TCP. Since our honeypot has tcpdump running and captures network traffic as packet capture (PCAP), we can fetch the credentials of the attacker encoded in Base64.

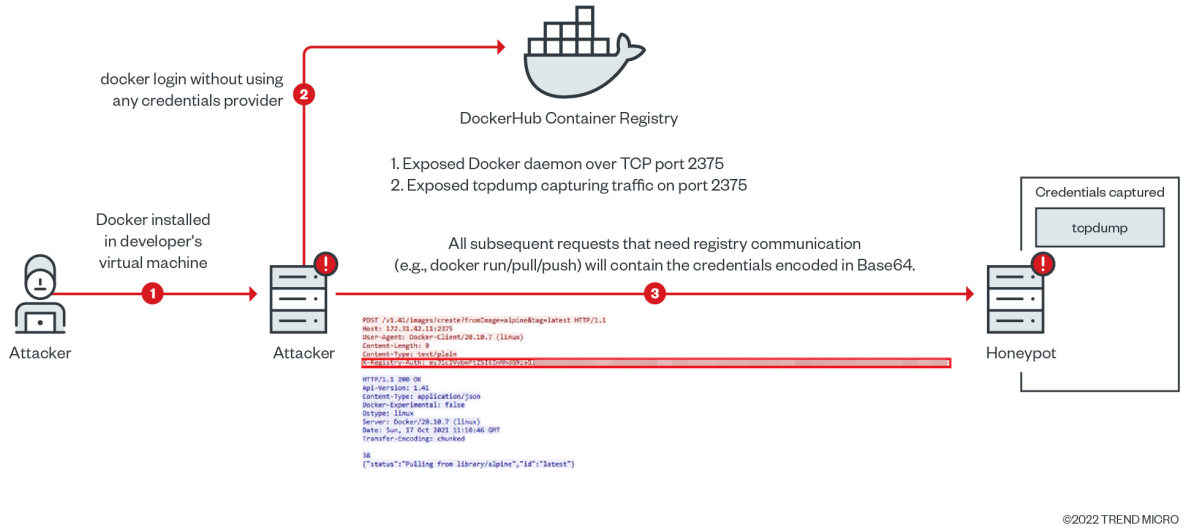


Figure 12. Scenario 2: Attackers' credentials leaked

Conclusion

Developers use containers to aid them in their development and deployments, optimize their workflows, and increase their productivity. Likewise, small gaps such as component misconfigurations have been known to be abused by cybercriminals. Malicious actors can perform damaging activities from these openings, like compromising the host for unauthorized cryptocurrency mining, exfiltrating sensitive information such as keys and credentials, or — at its worst— controlling victim servers to expand botnet malware usage, among other illicit activities. Indeed, cybercriminal groups such as TeamTNT will not stop anytime soon for as long as there are components and accounts that can be abused.

Based on our observations, we were able to identify TeamTNT's accounts because of one of the members' mistake on three occasions. There are three possible scenarios in which the user could have made this error:

1. The threat actors logged in to their DockerHub account using the credentials of alpineos.
2. The threat actors' machines were self-infected and were not using credential helpers.

3. The threat actors didn't log out from their DockerHub account while attacking exposed Docker REST API servers.

We found a total of 30 such accounts that were compromised, the credentials for which were being leaked. The registries for these were DockerHub and Alibaba Cloud Container Registry. While we have acquired this information and have access to the aforementioned credentials that might have been abused by TeamTNT, we did not access these credentials unauthorized. We have also informed Docker about these accounts and are working with them to resolve the matter.

Organizations' security teams need to be aware that developer security is critical considering this type of compromise around developer-centric tools like Docker have been observed being abused by threat actors. We advise that teams create policies for access and credential use, as well as generate threat models of their environments. Security teams can use these to educate developers about what can go wrong. Here are some mitigation practices for organizations and developers:

- While creating containers on a remote host via the Docker daemon REST API, developers should be aware that DockerHub credentials will also be shared if they are creating images from the specified container registry. They should proceed to do so only when the remote host is trusted.
- With the rising number of malicious open-source packages targeting user credentials, users should avoid storing credentials in other components such as environment variables. Instead, they must choose tools such as credential stores and helpers.
- If users need to use the Docker Daemon over REST API via internet, it is recommended that they configure the exposed REST API with TLS (Transport Layer Security) protocol to avoid man-in-the-middle (MiTM) attacks sniffing for credentials.

The full details of this research will be presented at the c0c0n XV Hacking and Cyber Security Conference scheduled on Sept. 21 to 24, 2022.

Indicators of Compromise (IOCs)

For a full list of IOCs, you can visit our blog entries on previous incidents we documented here and here.

sXpIBdPeKzI9PC2p0SWMpUSM2NSxWzPyXTMLIbXmYa0R20xk