0x01 SSRF漏洞常见防御手法及绕过方法

SSRF是一种常见的Web漏洞，通常存在于需要请求外部内容的逻辑中，比如本地化网络图片、XML解析时的外部实体注入、软件的离线下载等。当攻击者传入一个未经验证的URL，后端代码直接请求这个URL，将会造成SSRF漏洞。

具体危害体现在以下几点上：

URL为内网IP或域名，攻击者将可以通过SSRF漏洞扫描目标内网，查找内网内的漏洞，并想办法反弹权限

URL中包含端口，攻击者将可以扫描并发现内网中机器的其他服务，再进一步进行利用

当请求方法允许其他协议的时候，将可能利用gopher、file等协议进行第三方服务利用，如利用内网的redis获取权限、利用fastcgi进行getshell等

特别是这两年，大量利用SSRF攻击内网服务的案例被爆出来，导致SSRF漏洞慢慢受到重视。这就给Web应用开发者提出了一个难题：**如何在保证业务正常的情况下防御SSRF漏洞？**

很多开发者认为，只要检查一下请求url的host不为内网IP，即可防御SSRF。这个观点其实提出了两个技术要点：

1.如何检查IP是否为内网IP

2.如何获取真正请求的host

于是，攻击者通过这两个技术要点，针对性地想出了很多绕过方法。

0x02 如何检查IP是否为内网IP

这实际上是很多开发者面临的第一个问题，很多新手甚至连内网IP常用的段是多少也不清楚。

何谓内网IP，实际上并没有一个硬性的规定，多少到多少段必须设置为内网。有的管理员可能会将内网的IP设置为233.233.233.0/24段，当然这是一个比较极端的例子。

通常我们会将以下三个段设置为内网IP段，所有内网内的机器分配到的IP是在这些段中：

```
192.168.0.0/16 => 192.168.0.0 ~ 192.168.255.255
```

```
10.0.0.0/8 => 10.0.0.0 ~ 10.255.255.255
172.16.0.0/12 => 172.16.0.0 ~ 172.31.255.255
```

所以通常，我们只需要判断目标IP不在这三个段，另外还包括一个 127.0.0.0/8 段即可。

很多人会忘记 127.0.0.0/8 ，认为本地地址就是 127.0.0.1 ，实际上本地回环包括了整个127段。你可以访问**http://127.233.233.233/**，会发现和请求127.0.0.1是一个结果：

所以我们需要防御的实际上是4个段，只要IP不落在这4个段中，就认为是“安全”的。

网上一些开发者会选择使用“正则”的方式判断目标IP是否在这四个段中，这种判断方法通常是会遗漏或误判的，比如如下代码：

```
if re.match(r"^192\.168(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){2}$", ip_address) or \
    re.match(r"^172\.([1][6-9]|[2]\d|3[01])(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){2}$", ip_address) or \
        re.match(r"^10(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){3}$", ip_address):
    raise BaseException("inner ip address attack")
```


这是Sec-News最老版本判断内网IP的方法，里面使用正则判断IP是否在内网的几个段中。这个正则也是我当时临时在网上搜的，很明显这里存在多个绕过的问题：

1. 利用八进制IP地址绕过

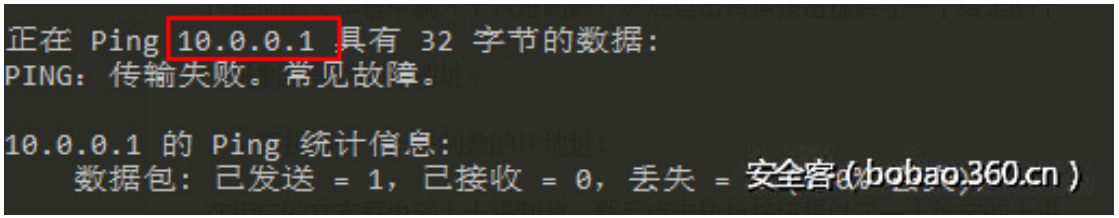
2. 利用十六进制IP地址绕过

3. 利用十进制的IP地址绕过

4. 利用IP地址的省略写法绕过

这四种方式我们可以依次试试：

```
D:\pro\sec-news (master)
λ ping -w 0 -n 1 012.0.0.1

正在 Ping 10.0.0.1 具有 32 字节的数据:
PING: 传输失败。常见故障。

10.0.0.1 的 Ping 统计信息:
    数据包: 已发送 = 1, 已接收 = 0, 丢失 = 1 (100% 丢失),

D:\pro\sec-news (master)
λ ping -w 0 -n 1 0xa.0.0.1

正在 Ping 10.0.0.1 具有 32 字节的数据:
PING: 传输失败。常见故障。

10.0.0.1 的 Ping 统计信息:
    数据包: 已发送 = 1, 已接收 = 0, 丢失 = 1 (100% 丢失),

D:\pro\sec-news (master)
λ ping -w 0 -n 1 167772161

正在 Ping 10.0.0.1 具有 32 字节的数据:
PING: 传输失败。常见故障。

10.0.0.1 的 Ping 统计信息:
    数据包: 已发送 = 1, 已接收 = 0, 丢失 = 1 (100% 丢失),

D:\pro\sec-news (master)
λ ping -w 0 -n 1 10.1

正在 Ping 10.0.0.1 具有 32 字节的数据:
PING: 传输失败。常见故障。

10.0.0.1 的 Ping 统计信息:
    数据包: 已发送 = 1, 已接收 = 0, 丢失 = 1 (100% 丢失),

D:\pro\sec-news (master)
λ ping -w 0 -n 1 0xA000001
```



四种写法（5个例子）：012.0.0.1 、 0xa.0.0.1 、 167772161 、 10.1 、 0xA000001 实际上都请求的是10.0.0.1，但他们一个都匹配不上上述正则表达式。

更聪明一点的人是不会用正则表达式来检测IP的（也许这类人并不知道内网IP的正则该怎么写）。Wordpress的做法是，先将IP地址规范化，然后用“.”将其分割成数组parts，然后根据parts[0]和parts[1]的取值来判断：

```
535     if ( ! $same_host ) {
536             $host = trim( $parsed_url['host'], '.' );
537             if ( preg_match( '#^(([1-9]?\d|1\d\d|25[0-5]|2[0-4]\d)\.){3}([1-9]?\d|1\d\d|25[0-5]|2[0-4]\d)$#', $host ) ) {
538                     $ip = $host;
539             } else {
540                     $ip = gethostbyname( $host );
541                     if ( $ip === $host ) // Error condition for gethostbyname()
542                             $ip = false;
543             }
544             if ( $ip ) {
545                     $parts = array_map( 'intval', explode( '.', $ip ) );
546                     if ( 127 === $parts[0] || 10 === $parts[0] || 0 === $parts[0]
547                             || ( 172 === $parts[0] && 16 <= $parts[1] && 31 >= $parts[1] )
548                             || ( 192 === $parts[0] && 168 === $parts[1] )
549                     ) {
550                             // If host appears local, reject unless specifically allowed.
551                             /**
552                              * Check if HTTP request is external or not.
553                              *
554                              * Allows to change and allow external requests for the HTTP request.
555                              *
556                              * @since 3.6.0
557                              *
558                              * @param bool   false Whether HTTP request is external or not.
559                              * @param string $host IP of the requested host.
560                              * @param string $url  URL of the requested host.
561                              */
562                             if ( ! apply_filters( 'http_request_host_is_external', false, $host, $url ) )
563                                     return false;
564                     }
565             }
566     }
```


其实也略显麻烦，而且曾经也出现过用进制方法绕过的案例（ WordPress <4.5 SSRF 分析 ），不推荐使用。

我后来选择了一种更为简单的方法。众所周知，IP地址是可以转换成一个整数的，在PHP中调用ip2long函数即可转换，在Python使用inet_aton去转换。

而且IP地址是和2^32内的整数一一对应的，也就是说0.0.0.0 == 0，255.255.255.255 == 2^32 - 1。所以，我们判断一个IP是否在某个IP段内，只需将IP段的起始值、目标IP值全部转换为整数，然后比较大小即可。

于是，我们可以将之前的正则匹配的方法修改为如下方法：

```
if ip2long("10.0.0.0") <= ip_long <= ip2long("10.255.255.255") or \
    ip2long("172.16.0.0") <= ip_long <= ip2long("172.31.255.255") or \
    ip2long("192.168.0.0") <= ip_long <= ip2long("192.168.255.255") or \
    ip2long("127.0.0.0") <= ip_long <= ip2long("127.255.255.255"):
    raise BaseException("inner ip address attack")
```


这就是一个最简单的方法，也最容易理解。

假如你懂一点掩码的知识，你应该知道IP地址的掩码实际上就是(32 - IP地址所代表的数字的末尾bit数)。所以，我们只需要保证目标IP和内网边界IP的前“掩码”位bit相等即可。借助位运算，将以上判断修改地更加简单：

```
from socket import inet_aton
from struct import unpack

def ip2long(ip_addr):
    return unpack("!L", inet_aton(ip_addr))[0]

def is_inner_ipaddress(ip):
    ip = ip2long(ip)
    return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
           ip2long('10.0.0.0') >> 24 == ip >> 24 or \
           ip2long('172.16.0.0') >> 20 == ip >> 20 or \
           ip2long('192.168.0.0') >> 16 == ip >> 16
```

以上代码也就是Python中判断一个IP是否是内网IP的最终方法，使用时调用**is_inner_ipaddress(...)**即可（注意自己编写捕捉异常的代码）。

0x03 host获取与绕过

如何获取"真正请求"的Host，这里需要考虑三个问题：

1. 如何正确的获取用户输入的URL的Host？

2. 只要Host只要不是内网IP即可吗？

3. 只要Host指向的IP不是内网IP即可吗？

如何正确的获取用户输入的URL的Host？

第一个问题，看起来很简单，但实际上有很多网站在获取Host上犯过一些错误。最常见的就是，使用**http://233.233.233.233@10.0.0.1:8080/**、**http://10.0.0.1#233.233.233.233**这样的URL，让后端认为其Host是233.233.233.233，实际上请求的却是10.0.0.1。这种方法利用的是程序员对URL解析的错误，有很多程序员甚至会用正则去解析URL。

在Python 3下，正确获取一个URL的Host的方法：

```
from urllib.parse import urlparse

url = 'https://10.0.0.1/index.php'
urlparse(url).hostname
```

这一步一定不能犯错，否则后面的工作就白做了。

只要Host只要不是内网IP即可吗？

第二个问题，只要检查一下我们获取到的Host是否是内网IP，即可防御SSRF漏洞么？

答案是否定的，原因是，Host可能是IP形式，也可能是域名形式。如果Host是域名形式，我们是没法直接比对的。只要其解析到内网IP上，就可以绕过我们的**is_inner_ipaddress**了。

网上有个服务 http://xip.io ，这是一个“神奇”的域名，它会自动将包含某个IP地址的子域名解析到该IP。比如 **127.0.0.1.xip.io** ，将会自动解析到127.0.0.1，**www.10.0.0.1.xip.io**将会解析到10.0.0.1：

这个域名极大的方便了我们进行SSRF漏洞的测试，当我们请求**http://127.0.0.1.xip.io/info.php**的时候，表面上请求的Host是127.0.0.1.xip.io，此时执行**is_inner_ipaddress('127.0.0.1.xip.io')**是不会返回True的。但实际上请求的却是127.0.0.1，这是一个标准的内网IP。

所以，在检查Host的时候，我们需要将Host解析为具体IP，再进行判断，代码如下：

```
import socket
import re
from urllib.parse import urlparse
from socket import inet_aton
from struct import unpack

def check_ssrf(url):
    hostname = urlparse(url).hostname

    def ip2long(ip_addr):
        return unpack("!L", inet_aton(ip_addr))[0]

    def is_inner_ipaddress(ip):
        ip = ip2long(ip)
        return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
               ip2long('10.0.0.0') >> 24 == ip >> 24 or \
               ip2long('172.16.0.0') >> 20 == ip >> 20 or \
               ip2long('192.168.0.0') >> 16 == ip >> 16

     try:
         if not re.match(r"^https?://.*/.*$", url):
                raise BaseException("url format error")
         ip_address = socket.getaddrinfo(hostname, 'http')[0][4][0]
         if is_inner_ipaddress(ip_address):
              raise BaseException("inner ip address attack")
         return True, "success"
      except BaseException as e:
         return False, str(e)
      except:
         return False, "unknow error"
```

首先判断url是否是一个HTTP协议的URL（如果不检查，攻击者可能会利用file、gopher等协议进行攻击），然后获取url的host，并解析该host，最终将解析完成的IP放入**is_inner_ipaddress**函数中检查是否是内网IP。

只要Host指向的IP不是内网IP即可吗？

第三个问题，是不是做了以上工作，解析并判断了Host指向的IP不是内网IP，即防御了SSRF漏洞？

答案继续是否定的，上述函数并不能正确防御SSRF漏洞。为什么？

当我们请求的目标返回30X状态的时候，如果没有禁止跳转的设置，大部分HTTP库会自动跟进跳转。此时如果跳转的地址是内网地址，将会造成SSRF漏洞。

这个原因也很好理解，我以Python的requests库为例。requests的API中有个设置，叫allow_redirects，当将其设置为True的时候requests会自动进行30X跳转。而默认情况下（开发者未传入这个参数的情况下），requests会默认将其设置为True：

```
def get(url, params=None, **kwargs):
    """Sends a GET request.

    :param url: URL for the new :class:`Request` object.
    :param params: (optional) Dictionary or bytes to be sent in the query string for the :class:`Request`.
    :param \*\*kwargs: Optional arguments that ``request`` takes.
    :return: :class:`Response <Response>` object
    :rtype: requests.Response
    """

    kwargs.setdefault('allow_redirects', True)
    return request('get', url, params=params, **kwargs)
```


所以，我们可以试试请求一个302跳转的网址：

```
In [58]: requests.get('http://t.cn/R2iwH6d').status_code
Out[58]: 200

In [59]: requests.get('http://t.cn/R2iwH6d', allow_redirects=False).status_code
Out[59]: 302
```


默认情况下，将会跟踪location指向的地址，所以返回的status code是最终访问的页面的状态码。而设置了**allow_redirects**的情况下，将会直接返回302状态码。

所以，即使我们获取了**http://t.cn/R2iwH6d**的Host，通过了**is_inner_ipaddress**检查，也会因为302跳转，跳到一个内网IP，导致SSRF。

这种情况下，我们有两种解决方法：

1. 设置allow_redirects=False，不允许目标进行跳转

2. 每跳转一次，就检查一次新的Host是否是内网IP，直到抵达最后的网址

第一种情况明显是会影响业务的，只是规避问题而未解决问题。当业务上需要目标URL能够跳转的情况下，只能使用第二种方法了。

所以，归纳一下，完美解决SSRF漏洞的过程如下：

1. 解析目标URL，获取其Host

2. 解析Host，获取Host指向的IP地址

3. 检查IP地址是否为内网IP

4. 请求URL

5. 如果有跳转，拿出跳转URL，执行1

0x04 使用requests库的hooks属性来检查SSRF

那么，上一章说的5个过程，具体用Python怎么实现？

我们可以写一个循环，循环条件就是“该次请求的状态码是否是30X”，如果是就继续执行循环，继续跟进location，如果不是，则退出循环。代码如下：

```
r = requests.get(url, allow_redirects=False)
while r.is_redirect:
    url = r.headers['location']
    succ, errstr = check_ssrf(url)
    if not succ:
        raise Exception('SSRF Attack.')
    r = requests.get(url, allow_redirects=False)
```

这个代码思路大概没有问题，但非常简陋，而且效率不高。

只要你翻翻requests的源代码，你会发现，它在处理30X跳转的时候考虑了很多地方：

所有请求放在一个requests.Session()中

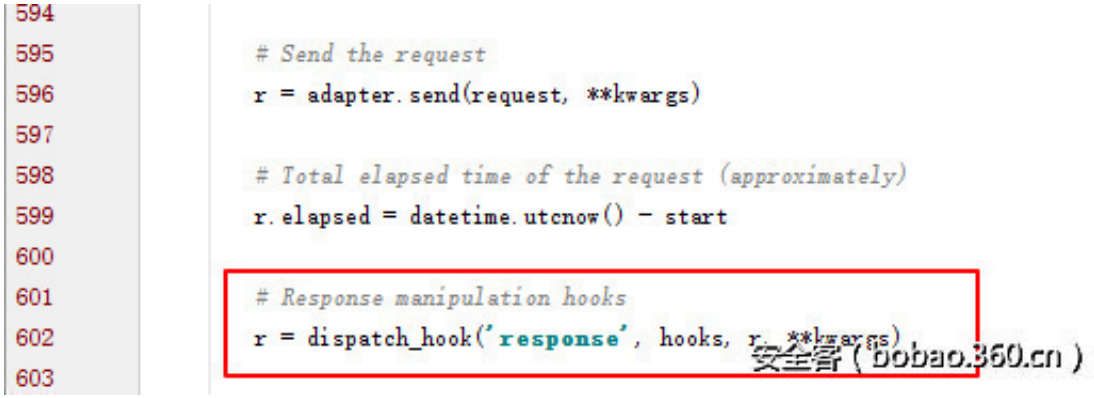
跳转有个缓存，当下次跳转地址在缓存中的时候，就不用多次请求了

跳转数量有最大限制，不可能无穷无尽跳下去

解决307跳转出现的一些BUG等

如果说就按照之前简陋的代码编写程序，固然可以防御SSRF漏洞，但上述提高效率的方法均没用到。

那么，有更好的解决方法么？当然有，我们翻一下requests的源代码，可以看到一行特殊的代码：



hook的意思就是“劫持”，意思就是在hook的位置我可以插入我自己的代码。我们看看dispatch_hook函数做了什么：

```
def dispatch_hook(key, hooks, hook_data, **kwargs):
    """Dispatches a hook dictionary on a given piece of data."""
    hooks = hooks or dict()
    hooks = hooks.get(key)
    if hooks:
        if hasattr(hooks, '__call__'):
            hooks = [hooks]
        for hook in hooks:
            _hook_data = hook(hook_data, **kwargs)
            if _hook_data is not None:
                hook_data = _hook_data
    return hook_data
```

hooks是一个函数，或者一系列函数。这里做的工作就是遍历这些函数，并调用：**_hook_data = hook(hook_data,**kwargs)**

我们翻翻文档，可以找到hooks event的说明 http://docs.python-requests.org/en/master/user/advanced/?highlight=hook#event-hooks ：

Event Hooks

Requests has a hook system that you can use to manipulate portions of the request process, or signal event handling.

Available hooks:

`response`:
    The response generated from a Request.

You can assign a hook function on a per-request basis by passing a `{hook_name: callback_function}` dictionary to the `hooks` request parameter:

```
hooks=dict(response=print_url)
```

That `callback_function` will receive a chunk of data as its first argument.

```
def print_url(r, *args, **kwargs):
    print(r.url)
```

If an error occurs while executing your callback, a warning is given.

If the callback function returns a value, it is assumed that it is to replace the data that was passed in. If the function doesn't return anything, nothing else is effected.

Let's print some request method arguments at runtime:

```
>>> requests.get('http://httpbin.org', hooks=dict(response=print_url))
http://httpbin.org
<Response [200]>
```


文档中定义了一个**print_url**函数，将其作为一个hook函数。在请求的过程中，响应对象被传入了**print_url**函数，请求的域名被打印了下来。

我们可以考虑一下，我们将检查SSRF的过程也写为一个hook函数，然后传给requests.get，在之后的请求中一旦获取response就会调用我们的hook函数。这样，即使我设置**allow_redirects=True**，requests在每次请求后都会调用一次hook函数，在hook函数里我只需检查一下**response.headers['location']**即可。

说干就干，先写一个hook函数：

```
def _request_check_location(r, *args, **kwargs):
    if not r.is_redirect:
        return
    url = r.headers['location']

    # The scheme should be lower case...
    parsed = urlparse(url)
    url = parsed.geturl()

    # Facilitate relative 'location' headers, as allowed by RFC 7231.
    # (e.g. '/path/to/resource' instead of 'http://domain.tld/path/to/resource')
    # Compliant with RFC3986, we percent encode the url.
    if not parsed.netloc:
        url = urljoin(r.url, requote_uri(url))
    else:
        url = requote_uri(url)

    succ, errstr = check_ssrf(url)
    if not succ:
        raise requests.exceptions.InvalidURL("SSRF Attack: 安全客(bobao.360.cn)
```

当**r.is_redirect**为True的时候，也就是说这次请求包含一个跳转。获取此时的**r.headers['location']**，并进行一些处理，最后传入**check_ssrf**。当检查不通过时，抛出一个异常。

然后编写一个请求函数**safe_request_url**，意思是“安全地请求一个URL”。使用这个函数请求的域名，将不会出现SSRF漏洞：

```
def safe_request_url(url, **kwargs):
    def _request_check_location(r, *args, **kwargs):
        if not r.is_redirect:
            return
        url = r.headers['location']

        # The scheme should be lower case...
        parsed = urlparse(url)
        url = parsed.geturl()

        # Facilitate relative 'location' headers, as allowed by RFC 7231.
        # (e.g. '/path/to/resource' instead of 'http://domain.tld/path/to/resource')
        # Compliant with RFC3986, we percent encode the url.
        if not parsed.netloc:
            url = urljoin(r.url, requote_uri(url))
        else:
            url = requote_uri(url)

        succ, errstr = check_ssrf(url)
        if not succ:
            raise requests.exceptions.InvalidURL("SSRF Attack: %s" % (errstr, ))

    success, errstr = check_ssrf(url)
    if not success:
        raise requests.exceptions.InvalidURL("SSRF Attack: %s" % (errstr,))

    hooks = dict(response=_request_check_location)
    kwargs['hooks'] = hooks
    return requests.get(url, **kwargs)
```


我们可以看到，在第一次请求url前，还是需要**check_ssrf**一次的。因为hook函数**_request_check_location**只是检查30X跳转时是否存在SSRF漏洞，而没有检查最初请求是否存在SSRF漏洞。

不过上面的代码还不算完善，因为**_request_check_location**覆盖了原有（用户可能定义的其他hooks）的hooks属性，所以需要简单调整一下。

最终，给出完整代码：

```
1  import socket
2  import re
3  import requests
```

```
from urllib.parse import urlparse
from socket import inet_aton
from struct import unpack
from requests.utils import requote_uri

def check_ssrf(url):
    hostname = urlparse(url).hostname

    def ip2long(ip_addr):
        return unpack("!L", inet_aton(ip_addr))[0]

    def is_inner_ipaddress(ip):
        ip = ip2long(ip)
        return ip2long('127.0.0.0') >> 24 == ip >> 24 or \
               ip2long('10.0.0.0') >> 24 == ip >> 24 or \
               ip2long('172.16.0.0') >> 20 == ip >> 20 or \
               ip2long('192.168.0.0') >> 16 == ip >> 16

     try:
         if not re.match(r"^https?://.*/.*$", url):
             raise BaseException("url format error")
         ip_address = socket.getaddrinfo(hostname, 'http')[0][4][0]
         if is_inner_ipaddress(ip_address):
             raise BaseException("inner ip address attack")
         return True, "success"

       except BaseException as e:
           return False, str(e)
       except:
           return False, "unknow error"

def safe_request_url(url, **kwargs):
    def _request_check_location(r, *args, **kwargs):
        if not r.is_redirect:
            return

        url = r.headers['location']

        # The scheme should be lower case...
        parsed = urlparse(url)
        url = parsed.geturl()

        # Facilitate relative 'location' headers, as allowed by RFC 723
1.
        # (e.g. '/path/to/resource' instead of 'http://domain.tld/path/to/
resource')
        # Compliant with RFC3986, we percent encode the url.
        if not parsed.netloc:
            url = urljoin(r.url, requote_uri(url))
        else:
            url = requote_uri(url)

        succ, errstr = check_ssrf(url)
        if not succ:

```

```
            raise requests.exceptions.InvalidURL("SSRF Attack: %s" % (errs
tr, ))

    success, errstr = check_ssrf(url)
    if not success:
        raise requests.exceptions.InvalidURL("SSRF Attack: %s" % (errst
r,))

    all_hooks = kwargs.get('hooks', dict())
    if 'response' in all_hooks:
        if hasattr(all_hooks['response'], '__call__'):
            r_hooks = [all_hooks['response']]
        else:
            r_hooks = all_hooks['response']

        r_hooks.append(_request_check_location)

    else:
        r_hooks = [_request_check_location]

    all_hooks['response'] = r_hooks
    kwargs['hooks'] = all_hooks
    return requests.get(url, **kwargs)
```

外部程序只要调用**safe_request_url(url)**即可安全地请求某个URL，该函数的参数与requests.get函数参数相同。

完美在Python Web开发中解决SSRF漏洞。其他语言的解决方案类似，大家可以自己去探索。