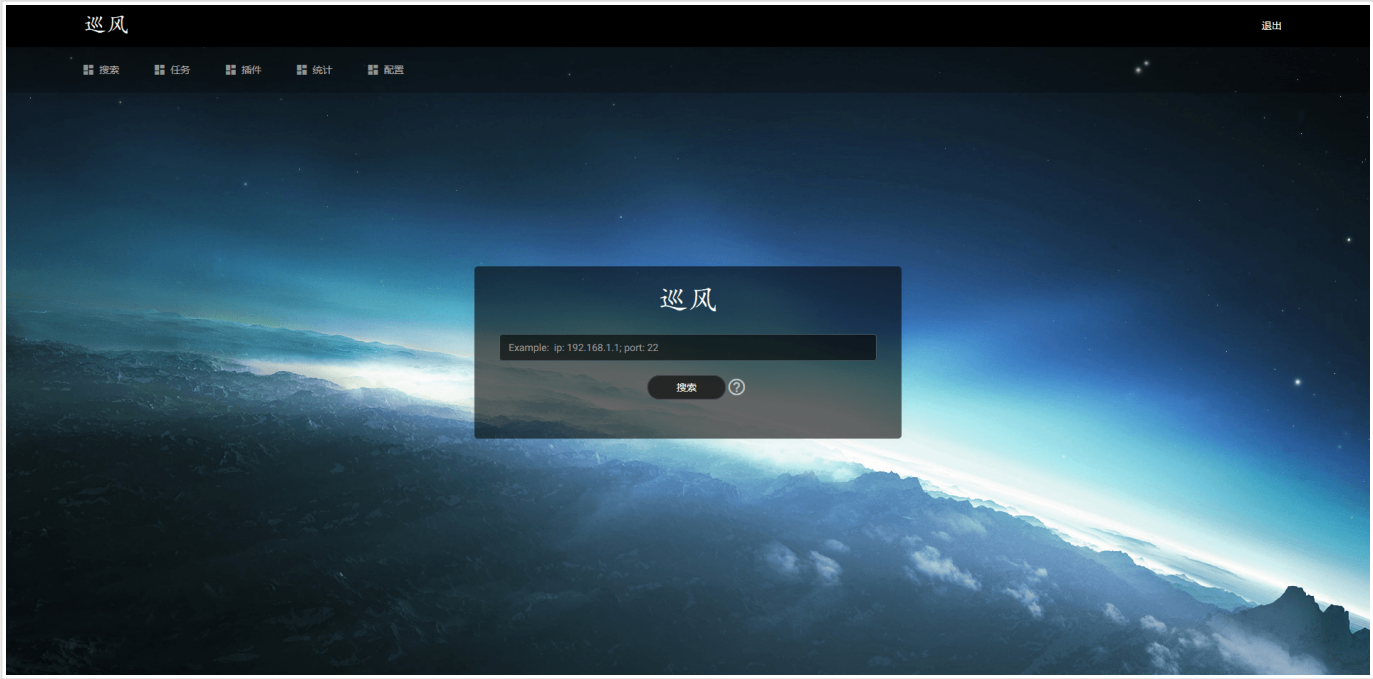
巡风源码浅析之Nascan分析篇

作者: LANDGREY ● 创建时间 2017年11月18日 19:39 ● 更新时间 2017年11月19日 02:00
浏览: 3400 次. ● 标签: #python, #网络安全
您的IP地址: 140.207.23.83



巡风源码浅析之Nascan分析篇 正文

> 巡风是一款适用于企业内网的漏洞快速应急、巡航扫描系统，通过搜索功能可清晰的了解内部网络资产分布情况，并且可指定漏洞插件对搜索结果进行快速漏洞检测并输出结果报表。
> 开源地址： https://github.com/ysrc/xunfeng

0x00: 起因

巡风算是国内开源扫描器中的一个良心产品了，研究下里面的扫描流程和代码逻辑对以后的工作想必大有裨益。

巡风的辅助验证脚本Aider 和我写的 dnstricker 的思路很相似，代码也不复杂，不做单独分析。巡风的项目一直在更新，所以可能在代码上会有些许出入。

0x01: 分析准备

巡风使用python语言开发，基于Flask Web框架和Mongodb数据库。我自己按照代码位置和代码逻辑，将巡风的架构大致分为三个部分，分别是Nascan、Vulscan和Flask web实现。分析篇也大概按照这三个部分介绍。

首先是先搭建起巡风，正常扫描一次，然后进入数据库，了解数据库结构。

Mongodb数据库的一个特点就是键值对表示。文档(document)组成集合(collection)，集合(collection)再组成数据库(db)，有时在文档和集合间可能还会有子集合。粗略的表示就是

```
document  ->  sub collection  ->  collection  ->  db
```

进入数据库查看，了解大体的集合情况：

```
> show dbs
admin    0.000GB
local    0.000GB
xunfeng  0.001GB
> use xunfeng
switched to db xunfeng
> show collections
Config                  # 各种配置信息
Heartbeat               # 记录程序最新心跳时间
History                 # 删除历史目标的记录信息
Info                    # 存储目标ip，名字，banner，开放端口和服务类型等信息
Plugin                  # 巡风扫描插件
Result                  # 扫描的成功结果
Statistics              # 目标统计信息
Task                    # 用户增加的扫描任务
Update                  # 更新信息
```

0x02: Nascan 介绍

Nascan 主要是做目标的资产识别(信息收集)，算是巡风中逻辑比较复杂的一部分，文件结构如下：

```
|  NAScan.py          # 网络资产信息抓取引擎
|  __init__.py
|
├──lib
|       cidr.py       # CIDR 形式IP地址解析
|       common.py     # 其它方法
|       icmp.py       # ICMP消息发送类
|       log.py        # 控制台信息输出
|       mongo.py      # 数据库连接
|       scan.py       # 扫描与识别
|       start.py      # 线程控制
|       __init__.py
|
└──plugin
        masscan.py    # masscan调用脚本
```

Nascan部分的配置信息在 Config 集合中，可以用命令查看一下数据库内容，分析过程中可以对照数据结构了解数据在程序内部的流转形态。

```
> db.Config.find().pretty()
```

0x03: 具体分析

入口：NAScan.py 文件，

```
if __name__ == "__main__":
    try:
        CONFIG_INI = get_config()  # 读取数据库中整个Config集合数据
        log.write('info', None, 0, u'获取配置成功')    # 输出到控制台
        STATISTICS = get_statistics()  # 读取统计信息
        MASSCAN_AC = [0]     # 值为1表示masscan正在扫描
        NACHANGE = [0]       # 值为1表示能进入扫描阶段
        thread.start_new_thread(monitor, (CONFIG_INI, STATISTICS, NACHANGE))  # /
        thread.start_new_thread(cruise, (STATISTICS, MASSCAN_AC))  # 失效记录删除线
        socket.setdefaulttimeout(int(CONFIG_INI['Timeout']) / 2)    # 设置默认socke
        # 已经完成资产信息收集的时间
        ac_data = []
        while True:
            now_time = time.localtime()
            now_hour = now_time.tm_hour
            now_day = now_time.tm_mday
            now_date = str(now_time.tm_year) + str(now_time.tm_mon) + str(now_day
            log.write('debug', None, 0, now_date)
            # 获取Config集合Cycle子集合(文档)的资产探测周期
            cy_day, ac_hour = CONFIG_INI['Cycle'].split('|')
            log.write('info', None, 0, u'扫描规则: ' + str(CONFIG_INI['Cycle']))
            # 判断是否进入扫描时段或者能直接进入扫描阶段
            if (now_hour == int(ac_hour) and now_day % int(cy_day) == 0 and now_d
                ac_data.append(now_date)
                # 恢复原值,不能再次进入资产探测,直到新的事件触发该值改变
                NACHANGE[0] = 0
                log.write('info', None, 0, u'开始扫描')
                # 具体的资产发现操作
                s = start(CONFIG_INI)
                # masscan扫描状态
                s.masscan_ac = MASSCAN_AC
                s.statistics = STATISTICS
                s.run()
            time.sleep(60)
    except Exception, e:
        print e
```

跟进get_config()函数，了解数库中存储的数据是如何取用的，对后面的分析帮助很大：

```
# 信息识别Config集合， 配置统一格式化，返回dict类型
def get_config():
    config = {}
    # Config集合共有vulscan、nascan两个子集合,获取Config集合中的nascan子集合的文档内容
    config_info = mongo.na_db.Config.find_one({"type": "nascan"})
    for name in config_info['config']:
        if name in ['Discern_cms', 'Discern_con', 'Discern_lang', 'Discern_server
            '''
            cms识别、组件容器识别、语言技术识别、端口服务识别 四个部分的文档内容赋值配
            按照事先定义的格式进一步格式化分离数据,方便后续取用.
            '''
            config[name] = format_config(name, config_info['config'][name]['value
        else:
            # 为nascan子集合中name是Scan_list、Port_list、Masscan、Timeout、Cycle、T
            config[name] = config_info['config'][name]['value']
    return config
```

log.write()函数，格式化了输出在控制台界面的信息，并使用了线程锁，防止信息一时间输出过多，导致显示错行。

然后单独开启了两个线程，两个线程通过While True和设定延时，实现了监控资产列表，定时更新数据库、触发扫描、清理失效目标等操作。

```
thread.start_new_thread(monitor, (CONFIG_INI, STATISTICS, NACHANGE)) # 心跳监控线程
thread.start_new_thread(cruise, (STATISTICS, MASSCAN_AC))            # 失效记录删除
```

在monitor 心跳线程中，资产列表数据的base64编码值如果改变，则将 NACHANGE[0] 置为 1，表示可以进入资产收集步骤：

```
# 获得数据库最新的Config集合数据
new_config = get_config()
# 比较扫描目标是否发生了变化， 变化就将值置为1， 表示需要重新扫描
if base64.b64encode(CONFIG_INI["Scan_list"]) != base64.b64encode(new_config["Scan
    NACHANGE[0] = 1
```

然后回到NAscan.py文件中。触发立即进入扫描阶段的判断语句如下：

```
if (now_hour == int(ac_hour) and now_day % int(cy_day) == 0 and now_date not in a
```

一个是判断是否到达资产探测的周期时间，另一个判断就是NACHANGE[0]的值为真(1)，任何一个成立都可以进入重新扫描资产的部分；

所以，如果想修改过目标列表后，立即扫描，可以把monitor函数中的sleep时间设置小一些。

然后跟入start()函数，进入正式的资产发现流程中。在 /lib/start.py中的start类中，__init__初始化了传递过来的配置信息，然后进入run()函数中，处理目标IP地址和使用masscan进行初步扫描等：

```
def run(self):
    global AC_PORT_LIST
    all_ip_list = []
    for ip in self.scan_list:

        # 解析CIDR形式IP地址
        if "/" in ip:
            ip = cidr.CIDR(ip)
        if not ip:
            continue
        # 获得完整目标IP地址列表
        ip_list = self.get_ip_list(ip)

        # 当使用masscan扫描时
        if self.mode == 1:
            self.masscan_path = self.config_ini['Masscan'].split('|')[2]
            self.masscan_rate = self.config_ini['Masscan'].split('|')[1]
            # 默认使用icmp去探测获得存活主机
            ip_list = self.get_ac_ip(ip_list)
            self.masscan_ac[0] = 1  # 可以继续masscan端口扫描

            # 使用Masscan对存活主机进行全端口扫描
            AC_PORT_LIST = self.masscan(ip_list)

            if not AC_PORT_LIST:
                continue
            self.masscan_ac[0] = 0  # 不能再次用masscan进行端口扫描

            # {ip: port}
            for ip_str in AC_PORT_LIST.keys():
                # IP地址加入队列
                self.queue.put(ip_str)
            self.scan_start()  # 开始端口banner获取和banner比对识别等
        # 不使用masscan
        else:
            all_ip_list.extend(ip_list)

    # 不使用masscan时
    if self.mode == 0:
        # 如果启用存活主机探测功能时,会用icmp echo探测存活的主机ip
        if self.icmp:
            all_ip_list = self.get_ac_ip(all_ip_list)
        # IP地址加入队列
        for ip_str in all_ip_list:
            self.queue.put(ip_str)
        self.scan_start() # 开始端口banner获取和banner比对识别等
```

跟进到 /plugin/masscan.py的run函数中，调用masscan进行端口扫描。字符串过滤了;|&三个特殊字符，然后拼接到了命令中：

```
# 过滤可能导致命令执行的字符
path = str(path).translate(None, ';|&')
rate = str(rate).translate(None, ';|&')
if not os.path.exists(path):
    return
# rate参数命令执行
os.system("%s -p1-65535 -iL target.log -oL tmp.log --randomize-hosts --rate=%s" %
```

但在Linux平台下，通过重音符"`"，还是可以执行任意命令。漏洞已提Issues，反馈给了巡风项目组。

然后，不管用不用masscan，都会进入scan_start()函数中，跟进后又进入/lib/start.py文件中ThreadNum——一个多线程类中，看下run()函数，把IP地址和端口号列表传到另一个scan()函数中：

```
def run(self):
    while True:
        try:
            # ip地址队列
            task_host = self.queue.get(block=False)
        except:
            break
        try:
            # 如果使用masscan，端口就用扫描到的已经开放的端口
            if self.mode:
                port_list = AC_PORT_LIST[task_host]
            # 没有使用masscan，使用默认端口
            else:
                port_list = self.config_ini['Port_list'].split('|')[1].split('\n
            # 根据banner识别端口开放的服务
            _s = scan.scan(task_host, port_list)
            _s.config_ini = self.config_ini  # 提供配置信息
            _s.statistics = self.statistics  # 提供统计信息
            _s.run()

        except Exception, e:
            print e
        finally:
            self.queue.task_done()
```

进入/lib/scan.py文件，可以看到scan类的run函数。在这个函数中，调用一系列函数，最终完成了资产信息的识别。

```
def run(self):
    self.timeout = int(self.config_ini['Timeout'])
    for _port in self.port_list:
        self.server = ''
        self.banner = ''
        self.port = int(_port)
        # 基础单端口扫描获得开放端口banner
        self.scan_port()
        if not self.banner:
            continue
        # 使用获得的banner进行服务类型识别
        self.server_discern()
        # 测试还剩下的一些没识别出来的端口服务是不是web服务器
        if self.server == '':
            # 尝试获得端口web服务的html源码
            web_info = self.try_web()
            if web_info:
                log.write('web', self.ip, self.port, web_info)
                time_ = datetime.datetime.now()
                # Info 集合更新
                mongo.NA_INFO.update({'ip': self.ip, 'port': self.port},
                                     {"$set": {'banner': self.banner, 'server':
```

函数会遍历端口，scan_port()通过socket套接字连接，获得端口服务返回的banner信息，然后进入server_discern()函数，通过正则表达式，依次比较，获得服务类型:

```
def server_discern(self):
    for mark_info in self.config_ini['Discern_server']:
        try:
            # 服务名    默认端口    识别方法    banner匹配正则表达式
            name, default_port, mode, reg = mark_info
            # 识别模式是default的，只判断端口号
            if mode == 'default':
                if int(default_port) == self.port:
                    self.server = name
            # 识别模式是banner的，正则匹配banner
            elif mode == 'banner':
                matchObj = re.search(reg, self.banner, re.I | re.M)
                if matchObj:
                    self.server = name
            if self.server:
                break
        except:
            continue
    # 处理没识别出来的也不太像(不严谨)web的服务
    if not self.server and self.port not in [80, 443, 8080]:
        for mark_info in self.config_ini['Discern_server']:
            try:
                name, default_port, mode, reg = mark_info
                if mode not in ['default', 'banner']:
                    dis_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                    dis_sock.connect((self.ip, self.port))
                    mode = mode.decode('string_escape')
                    reg = reg.decode('string_escape')
                    dis_sock.send(mode)
                    time.sleep(0.3)
                    dis_recv = dis_sock.recv(1024)
                    matchObj = re.search(reg, dis_recv, re.I | re.M)
                    if matchObj:
                        self.server = name
                        break
            except:
                pass
```

对于没识别出来的服务类型，端口号又不是web服务的常见端口号的，会 重新发包,处理识别mode不是"default"和"banner"的服务类型。也即探测被动式的应答服务，必须发送特定包才会返回应答banner的服务类型。如proxy、mssql、oracle、rdp、redis、postgresql、elasticsearch、memcache、mongodb、zookeeper服务。

最后还剩下一部分端口识别不出来的服务，基本就剩一些web服务和没有预先定义banner的不知名服务了；所以，web服务是在最后才识别的。放入try_web()函数，看看是不是web服务，是的话就保存下来，不是就不管了。

在try_web()中，可以看到，web服务的banner为整个HTTP数据包：

```
web_banner = str(header) + "\r\n\r\n" + html
self.banner = web_banner
```

然后用内置map()函数，调用discern()函数，识别cms类型、网站容器、使用语言三个方面的信息，最后用filter()函数过滤一下没结果的部分，就返回了。

```
tag = map(self.discern, ['Discern_cms', 'Discern_con', 'Discern_lang'], [url, url
return filter(None, tag)
```

web指纹识别，discern()函数如下：

```
# 网站容器、语言、cms类型识别
def discern(self, dis_type, domain):
    file_tmp = {}
    if int(domain.split(":")[1]) == 443:
        protocol = "https://"
    else:
        protocol = "http://"
    try:
        req = urllib2.urlopen(protocol + domain, timeout=self.timeout)
        header = req.headers
        html = req.read()
    except urllib2.HTTPError, e:
        html = e.read()
        header = e.headers
    except Exception, e:
        return
    for mark_info in self.config_ini[dis_type]:
        # 根据 http header内容判断cms或框架等类型
        if mark_info[1] == 'header':
            try:
                if not header:
                    return
                if re.search(mark_info[3], header[mark_info[2]], re.I):
                    return mark_info[0]
            except Exception, e:
                continue
        # 根据服务器文件、路径判断
        elif mark_info[1] == 'file':
            # 根据文件类型(.asp,.php,...),判断服务器使用何种语言
            if mark_info[2] == 'index':
                try:
                    if not html:
                        return
                    if re.search(mark_info[3], html, re.I):
                        return mark_info[0]
                except Exception, e:
                    continue
            # 根据特定网站路径返回内容判断容器、cms等类型
            else:
                if mark_info[2] in file_tmp:
                    re_html = file_tmp[mark_info[2]]
                else:
                    try:
                        re_html = urllib2.urlopen(protocol + domain + "/" + mark_
                                                  timeout=self.timeout).read()
                    except urllib2.HTTPError, e:
                        re_html = e.read()
                    except Exception, e:
                        return
                    file_tmp[mark_info[2]] = re_html
                try:
```

```
                    if re.search(mark_info[3], re_html, re.I):
                        return mark_info[0]
                except Exception, e:
                    print mark_info[3]
```

0x04: 总结

总体识别逻辑:

1. 解析用户设置的目标IP地址范围,获得完整IP地址列表;
2. 选择用masscan进行端口扫描时:
   强制先探测存活主机,只保留存活主机的IP,
   然后对存活主机进行全端口扫描,
   保存存活主机ip地址对应的开放端口号;
3. 不使用masscan扫描端口时:
   如果选择先探测存活主机,则只保留存活主机的IP,使用默认配置的端口号列表
   否则保留所有ip地址和默认配置的端口号列表;
4. 统一进行TCP端口连接扫描获得banner
5. 识别mode是default类型的，只判断默认端口号即确定服务类型
6. 识别mode是banner类型的，正则匹配banner确定服务类型
7. 没识别出来服务的,主动发特定数据探测包,看是不是特定的被动式应答类型服务
8. 还识别不出的，最后尝试是不是web服务，是的话保存相关信息，不是就不管了

值得借鉴的地方:

1. 心跳线程. 很少用到，但用在复杂的工程确实可以节省很多麻烦
2. web基本指纹的简单识别思路. 看discern()函数就可以发现，合适的指纹识别数据结构配合一小段代码，就可以完成比较头疼

一些改进建议:

1. 应该修复因为没有校验好数值，直接拼接命令导致的Linux平台下的任意命令执行漏洞;
2. 可能是为了减少代码复杂度，代码中多处对web识别部分的https协议和http协议请求考虑不周;默认只在443端口尝试https协i

blog comments powered by Disqus

<