

I recently procured an O.MG Elite cable and figured I'd probe the thing and see what the big scare was.

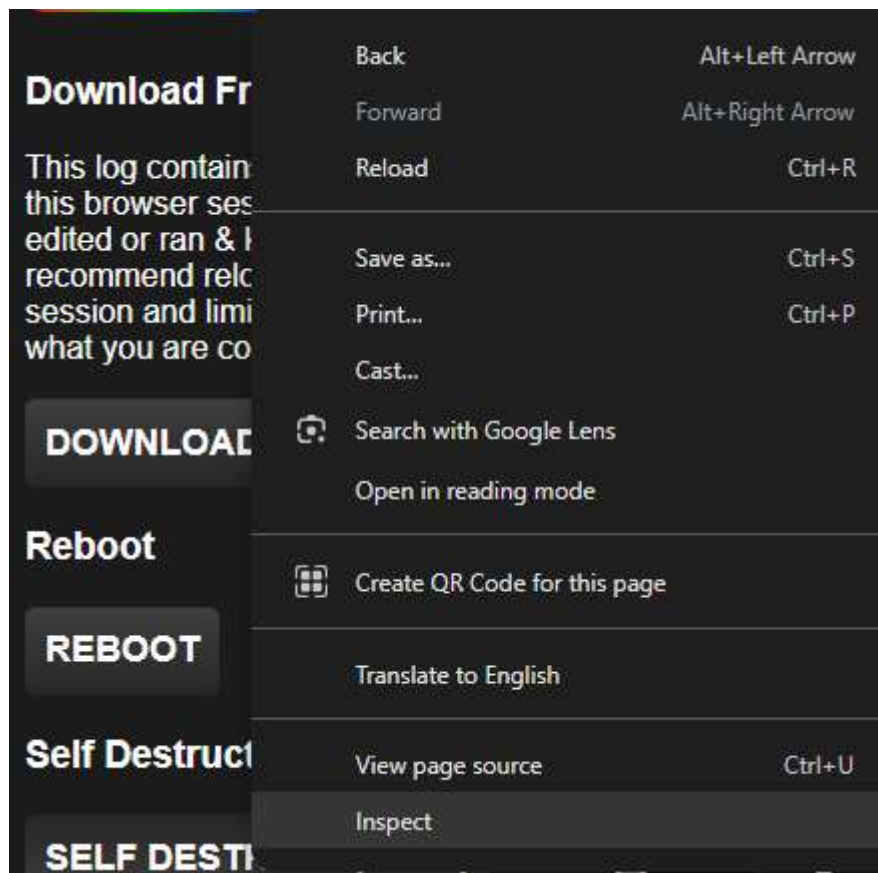
This cable is terrifying if you read the articles and listen to the interviews. It's a ticking time bomb until a company gets hit by this nefarious but inconspicuous cable. But in a recent interview with [CHANNEL] I picked up on a mention from that creator of the cable is okay with it's existence as the stealth features require meticulous configuration.

So I picked one up and started exploring one out of the box. A major feature for red teams and bad actors is the ability to wipe the cable using 2 methods. One option kills the O.MG aspects, leaving a passthrough USB C cable in place. The second method kills communications between the active and passive end, in the hopes the victim will toss it as a dead cable.

Within the UI of the cable I found the button to kill the cable entirely and this is what it looks like:



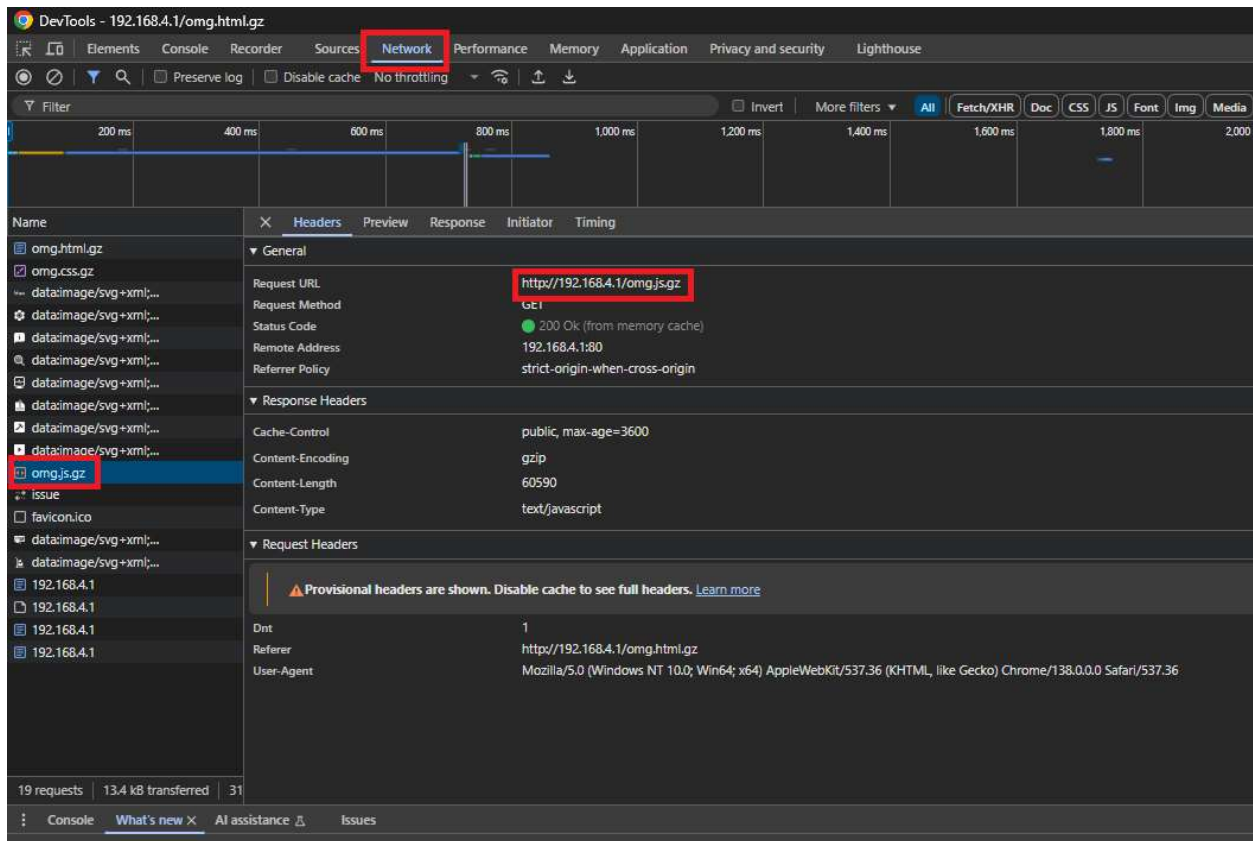
I did a bit more digging by inspecting the DOM element:



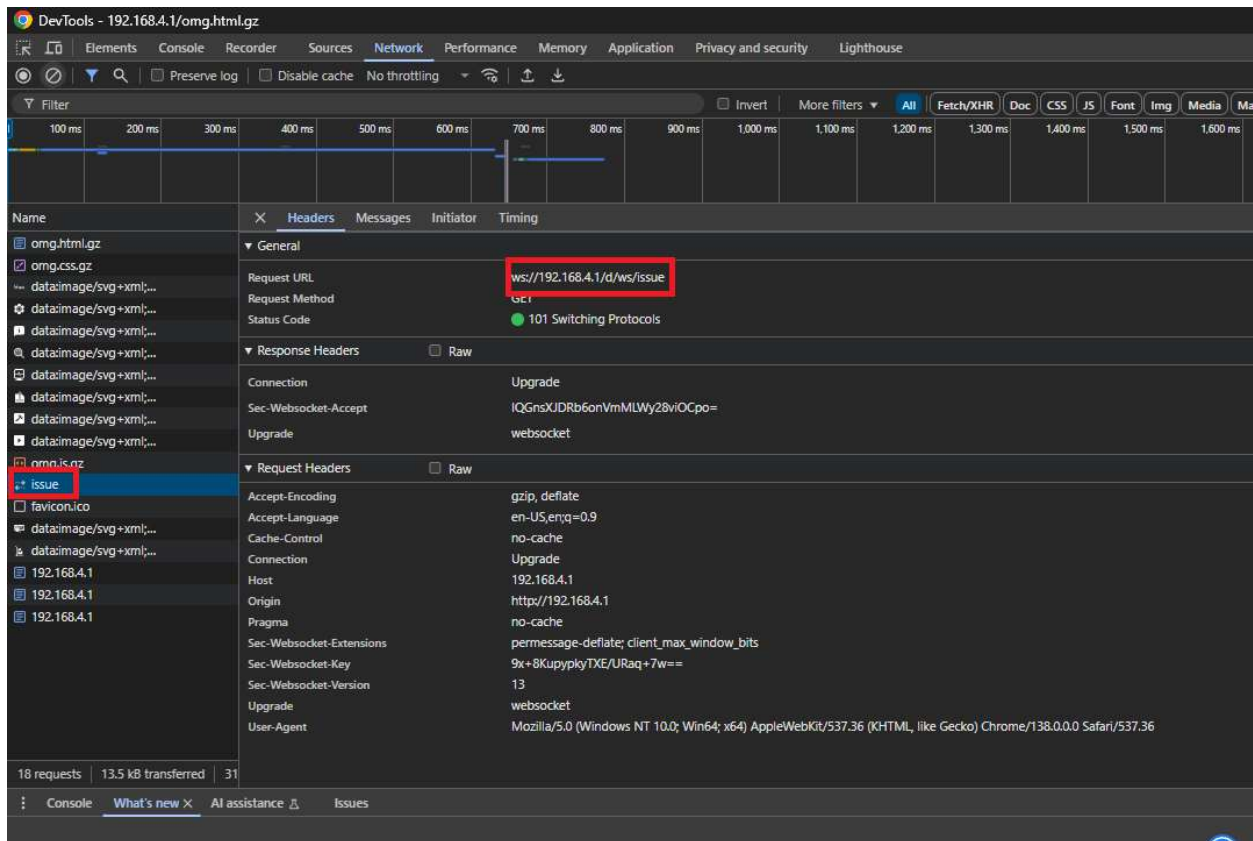
Which lead me to the JavaScript function call to send a message to the cable:

```
<button class="submitButton" id="sidebar_settings_debug_reload" onclick="applyEffect(this); downloadFrontendLog(); ">Download Frontend Log</button> (grid)
<h3>Reboot</h3>
<button class="submitButton" id="sidebar_settings_config_reboot" onclick="applyEffect(this); sendMessage('CR1');">Reboot</button> (grid)
<div class="selfDestruct">
  <h3>Self Destruct</h3>
  <button class="submitButton" id="sidebar_settings_config_selfDestructFull" onclick="applyEffect(this); sendMessage('CD1');">Self Destruct (Full)</button> (grid) == $0
  <aside> </aside>
  <br>
  <button class="submitButton" id="sidebar_settings_config_selfDestructPartial" onclick="applyEffect(this); sendMessage('CD2');">Self Destruct (Partial)</button> (grid)
  <aside> </aside>
</div>
<div class="footer"></div>
```

As I was poking around I came across a few things of interest. I peaked into the Network tab that shows various individual files the server is asking the client to load as well as packets sent to the server. I came upon a JavaScript file the server told the client to load:



Immediately after the JavaScript file I could see the client upgraded a connection to a websocket instead of HTTP. I noted down the URI path for later and moved on:



First was the JavaScript file containing the majority of the code interfacing the client (browser) to the cable (server):

```

// Global Variables
frontendType = "release"; // Options: release, factory
frontendMajorVersion = "3.0";
frontendMinorVersion = "250703";
if (frontendType == "factory") {
  frontendVersionNumber = `${frontendMajorVersion}-${frontendMinorVersion}-${frontendType}`;
} else {
  frontendVersionNumber = `${frontendMajorVersion}-${frontendMinorVersion}`;
}
let debugLevel = "INFO";
let liveLogStatus = "off";
let keylogStatus = "0";
let slotmode = "1";
let firstTimeSetupStatus = "";
// Global Variables - Connection Info
var enabledFeatures = [];
const defaultFeatures = ['global', 'syslog', 'debug'];
let featureFlags = [];
let lastMessage = 0;
var connectionAttempts = 0;
let websocketErrorCount = 0;
let previousUptime = Infinity;
let previousMAC = null;
let uicolor;
let zIndexCounter = 1;

function getAddressParameter() {
  const queryString = window.location.search;
  const urlParams = new URLSearchParams(queryString);
  return urlParams.has('backend') ? urlParams.get('backend') : null;
}

const backendParam = getAddressParameter();
const hostnameMap = {
  // "localhost": connectCustomBackend(),
};
let hostname = backendParam || hostnameMap[window.location.hostname] || window.location.hostname;
if(hostname == "localhost"){
  hostname = "192.168.5.140";
}

function connectCustomBackend() {
  createDialog({
    id: 'ipAddress',
    title: 'O.MG Backend IP Address',
    description: 'Please enter the IP Address of your O.MG Backend Device. <br \><br \><input id="labipaddress"></input><button id="applylabipaddress"
onClick="hostname = document.getElementById('labipaddress').value; console.log(hostname);>Connect</button>',
    closeEnabled: false,
  });
}

// Global Variables - Defaults
const readSize = 1024;
const blockSize = 4096;
let splitPayload = [];
let splitPayloadCounter = 0;
let keymapList = [];
let keymapListPretty = "";

```

I searched for the function mentioned in the onclick action displayed from the DOM inspection of the self-destruct button. Looking over this function it takes the current message and pushes it into a queue and then tells it to process the new message in the queue:

```

const sendMessage = (message, isKeepAlive = false) => {
  try {
    if (!apiMessageQueue.includes(message)) {
      apiMessageQueue.push(message);
    }
    if (!isKeepAlive) {
      config.lastMessageTime = Date.now();
    }
    processMessageQueue();
  } catch (error) {
    logMessage("INFO", `sendMessage error: ${error.message}`);
  }
};

```

So I searched for the function call to process the queue and revealed the encoding used to communicate over the socket:

```
const processMessageQueue = () => {
  if (processMessageQueueRunning || !isWebSocketOpen() || apiMessageQueue.length === 0) {
    return;
  }

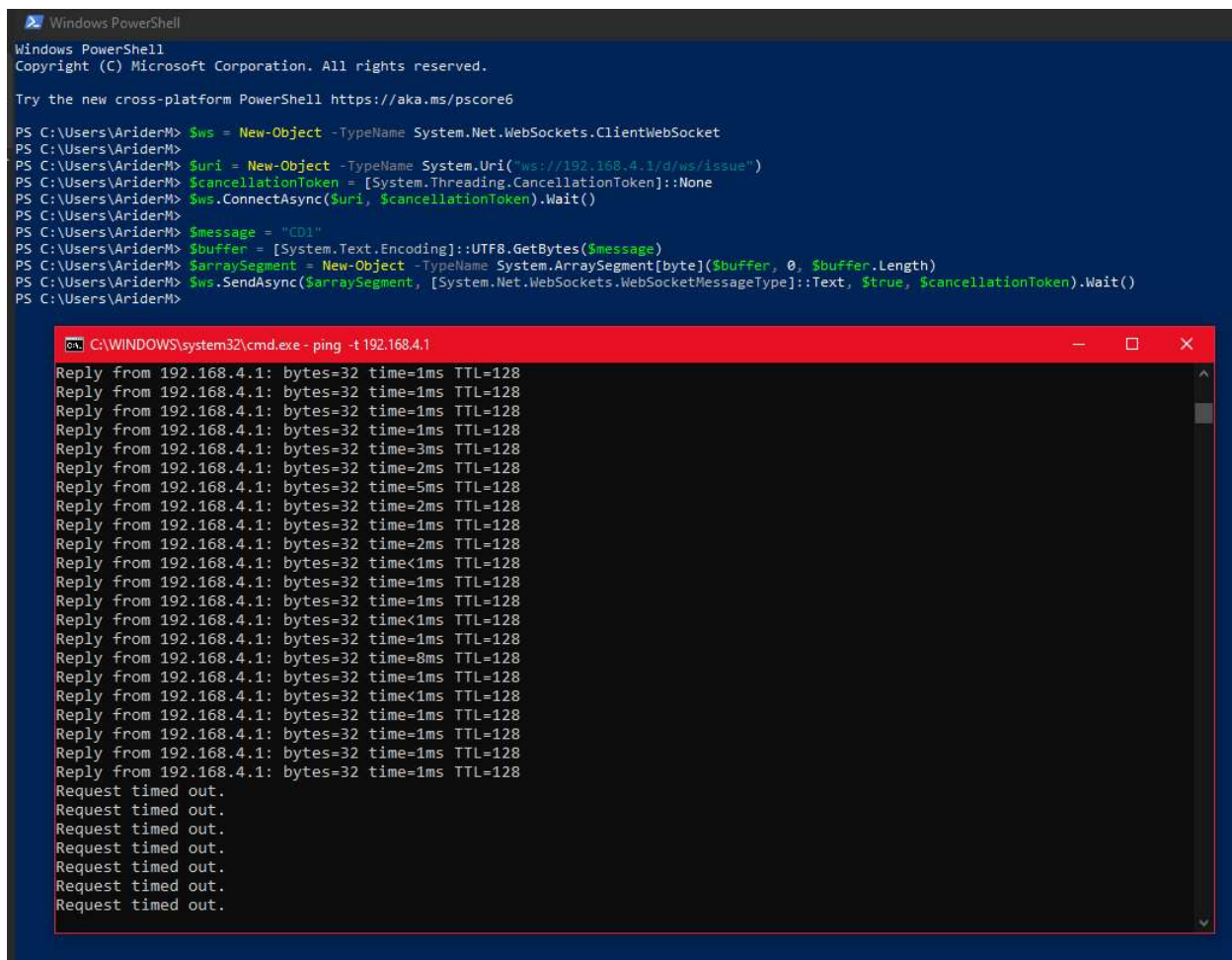
  processMessageQueueRunning = true;
  const msg = apiMessageQueue.shift();
  ws.send(new TextEncoder("utf-8").encode(msg));
  config.lastMessageTime = Date.now();
  logMessage("CRIT", `WebSocket Sent: [${msg}]`);
  networkOut = document.getElementById('networkOut');
  if (networkOut) {
    networkOut.classList.add('networkActive');
    setTimeout(() => {
      networkOut.classList.remove('networkActive');
    }, 250);
  }
};
```

Earlier I noted the client was upgrading a connection to a websocket and did a search for the URI using just the path parameters and came across this:

```
const manageWebSocket = (status) => {
  try {
    if (status === 'open' && (!ws || ws.readyState !== WebSocket.OPEN)) {
      ws = new WebSocket(`ws://${hostname}/d/ws/issue`);
      ws.binaryType = "arraybuffer";
      ws.onopen = wsOnOpen;
      ws.onmessage = wsOnMessage;
      ws.onclose = wsOnClose;
      ws.onerror = wsOnError;

      keepAliveIntervalId = setInterval(() => {
        if (Date.now() - config.lastMessageTime > 4000) {
          sendMessage("SVStatus", true);
        }
      }, 2000);
      config.websocketIdleTime = Date.now();
    } else if (status === 'close' && ws) {
      clearInterval(keepAliveIntervalId);
      ws.close();
      ws = null;
    }
  } catch (error) {
    logMessage("INFO", `manageWebSocket error: ${error.message}`);
  }
};
```

Using all of this information we now know the web client will open a socket to the cable and issue commands over the socket with UTF-8 encoding. I didn't see any authentication over this socket and figured I would give it a shot in PowerShell. I'll replace the variable in the web socket URI with the full address and try to connect. Then send the command found in the original DOM element, "CD1", and see what happens while keeping a ping to the cable running:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/powershell

PS C:\Users\AriiderM> $ws = New-Object -TypeName System.Net.WebSockets.ClientWebSocket
PS C:\Users\AriiderM>
PS C:\Users\AriiderM> $uri = New-Object -TypeName System.Uri("ws://192.168.4.1/d/ws/issue")
PS C:\Users\AriiderM> $cancellationToken = [System.Threading.CancellationToken]::None
PS C:\Users\AriiderM> $ws.ConnectAsync($uri, $cancellationToken).Wait()
PS C:\Users\AriiderM>
PS C:\Users\AriiderM> $message = "CD1"
PS C:\Users\AriiderM> $buffer = [System.Text.Encoding]::UTF8.GetBytes($message)
PS C:\Users\AriiderM> $arraySegment = New-Object -TypeName System.ArraySegment[byte]($buffer, 0, $buffer.Length)
PS C:\Users\AriiderM> $ws.SendAsync($arraySegment, [System.Net.WebSockets.WebSocketMessageType]::Text, $true, $cancellationToken).Wait()
PS C:\Users\AriiderM>

C:\WINDOWS\system32\cmd.exe - ping -t 192.168.4.1
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=3ms TTL=128
Reply from 192.168.4.1: bytes=32 time=2ms TTL=128
Reply from 192.168.4.1: bytes=32 time=5ms TTL=128
Reply from 192.168.4.1: bytes=32 time=2ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=2ms TTL=128
Reply from 192.168.4.1: bytes=32 time<1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time<1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=8ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time<1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Reply from 192.168.4.1: bytes=32 time=1ms TTL=128
Request timed out.
Request timed out.
Request timed out.
Request timed out.
Request timed out.
Request timed out.
```

I didn't capture it here, but Windows even complained about the cable being unrecognized after the kill chain was initiated.

This isn't a catch all, as this is an O.MG fresh out of the box with stable firmware applied and no changes. There's no magic bullet for this cable as it can be fully configured. This means the USB Vendor ID can be changed, MAC address, SSID, and the list goes on.

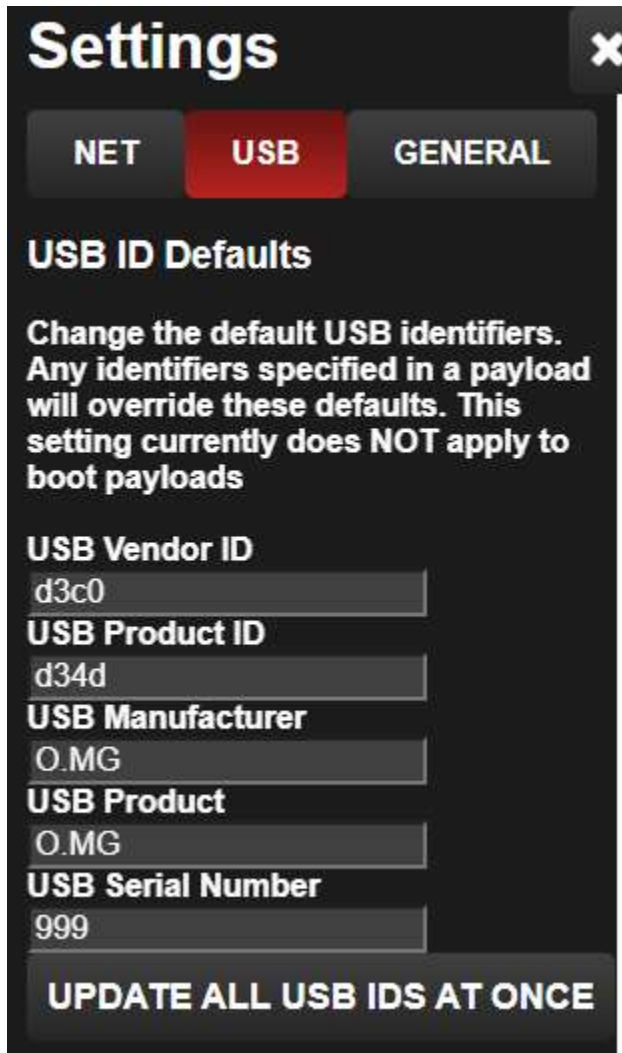
This article aims to demonstrate how to neuter a cable in the field assuming a bad actor didn't meticulously comb over every setting. There are various methods of detection and remediation but a couple things I recommend noting:

- 1) I could not find the default USB Vendor ID in any online database. Below I will include a screenshot of what my cable reported to me while setting it up. EDR should potentially block this vendor ID to prevent payloads from having a chance to launch an attack.

- 2) Out of box and flashed, the SSID was pretty obvious. If you have the means to have a wifi radio in monitor mode that can keep an eye out and alert on detection of “O.MG” as an SSID there’s a chance to intervene using the default password for the SSID.
- 3) I’m not sure if EDR has this built it yet, but measuring keystrokes per second and limiting the USB device sending them could prevent a large number of these attacks. It looks like keystrokes might be missed if rate limited, causing a payload to fail.

Through all of this I’m going to work with my security team to implement a detection / alert / remediation plan for at least all of the out of box defaults for this cable. It’s not perfect, and a skilled bad actor can find a way, but by capturing all of the basics we can elevate our detection beyond the out of box functions of our current NDR.

Out of box USB Vendor ID information:



The screenshot shows a 'Settings' window with a dark theme. At the top, there are three tabs: 'NET', 'USB' (which is highlighted in red), and 'GENERAL'. Below the tabs, the section is titled 'USB ID Defaults'. A descriptive text states: 'Change the default USB identifiers. Any identifiers specified in a payload will override these defaults. This setting currently does NOT apply to boot payloads'. Below this text are six input fields, each with a label and a value: 'USB Vendor ID' with 'd3c0', 'USB Product ID' with 'd34d', 'USB Manufacturer' with 'O.MG', 'USB Product' with 'O.MG', and 'USB Serial Number' with '999'. At the bottom of the settings panel is a large button labeled 'UPDATE ALL USB IDS AT ONCE'.

Setting	Value
USB Vendor ID	d3c0
USB Product ID	d34d
USB Manufacturer	O.MG
USB Product	O.MG
USB Serial Number	999