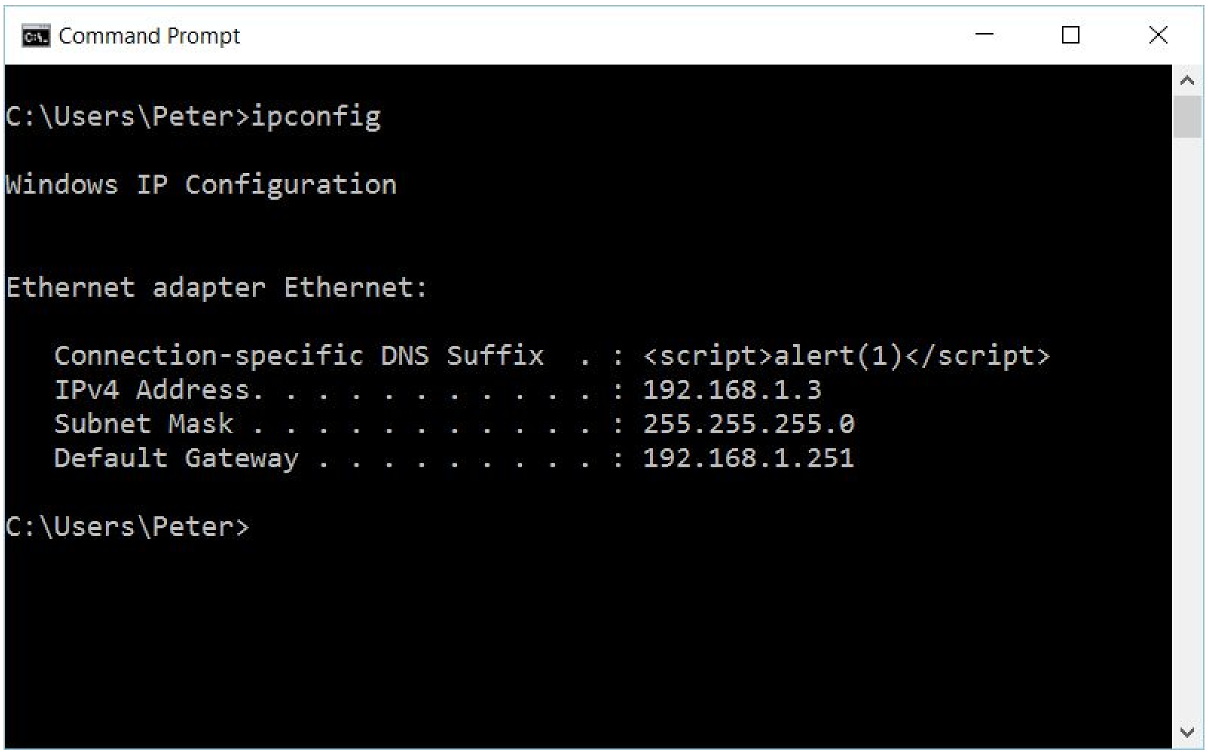
Report Project - Router Hijack

Language-Based Security VT16

Group 5

Peter Eliasson & Joel Severin



```
C:\Users\Peter>ipconfig

Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . : <script>alert(1)</script>
   IPv4 Address. . . . . . . . . . . : 192.168.1.3
   Subnet Mask . . . . . . . . . . . : 255.255.255.0
   Default Gateway . . . . . . . . . : 192.168.1.251

C:\Users\Peter>
```

Table of Contents

Introduction

Many homes today have at least one router, used perhaps primarily for wireless access. However, it is unlikely that all the people who use these home routers are aware that these routers could pose a security risk to devices connected to it. For example, changing the default DNS server used by the router could possibly endanger almost all internet traffic within the network. What makes home routers particularly vulnerable is that they in most cases allows configuring such settings via a web interface. As such, they can potentially be affected by a wide range of attacks targeting such services. Additionally, these routers are often cheap and have a long lifetime, which in many cases makes the likelihood of finding such vulnerabilities higher.

Goal

The goal of the project was to develop and describe an automated proof-of-concept (POC) attack against some home routers. The POC will show how a third-party website can change critical router settings of a visitor, using only an included JavaScript program.

Scope

The scope of this report and its associated POC will be limited to two routers (Netgear WGT624v3 and Netgear WNDR3700v3, the former being older than the latter). Additional assumptions are also made throughout the report, and will be discussed in more detail where appropriate.

Previous Work

Performing attacks against (home) routers is nothing new [1,2,3,4,5], neither is using Cross-Site Request Forgery (CSRF) as the method of such attacks. Additionally, there are POCs describing how to get the user's internal (behind-NAT) IP address, which happens to be allowed by design in the WebRTC API [6], enabled in many browsers today [7].

[1] https://threatpost.com/d-link-router-vulnerable-to-cross-site-scripting/102892/

[2] http://www.gironsec.com/blog/2015/01/owning_modems_and_routers_silently/

[3] http://security.stackexchange.com/questions/98156/a-possible-router-xss-vulnerability

[4] https://github.com/darkarnium/secpub/tree/master/NetGear/SOAPWNDR

[5] https://github.com/reverse-shell/routersploit

[6] https://bugs.chromium.org/p/chromium/issues/detail?id=333752

[7] http://caniuse.com/#search=webrtc

Description of Work (Result)

In this section, the POC attack will be described. First, a short overview of the overall approach is presented, including methods used when searching for vulnerabilities, as well as a short overview of the POC attack. Following that is a step-by-step description of parts in the attack, ordered by how the the parts would be executed when the attack is run. Each part is described conceptually, and some comments on the code structure for the parts that were implemented is provided.

Overview

Different kinds of known exploits were manually tried against the target routers. These were put together in order to create a complete solution. When no known exploit existed, reverse engineering of the code was conducted in order to find flaws. No automated testing was used.

The POC is written in TypeScript, a typed superset of JavaScript. The code is "compiled" via the TypeScript compiler into one main module ("app.js") and an additional module for each of the payloads. The payload modules are separate as they are included into the context of the router, as opposed to running on the visited domain.

As the main module is loaded, the first few steps all try to identify router(s) that could be targeted in the later stages. The identification is done by first (ab)using WebRTC (Web Real-Time Communication, a fairly recent browser API) to find the victim's local IP address. Possible routers are then found by sending requests to IPs that, based on the victim's local IP, is thought likely to host a router. Successful requests are flagged for fingerprinting, in which the code tries to gather information about router brand, as well as software and hardware configuration.

Following the identification step, a payload is injected in the router configuration interface. In the case of the POC, this means finding and making use of a Cross-Site Scripting (XSS) defect in the router code, so that the payload module for the specific router can be run without the cross-origin restrictions otherwise enforced by the victim's browser. In other words, a request is sent to the router so that some page on the router afterwards contains a script tag referencing the payload that should be run in order to complete the attack.

Finally, the payload is executed by having the victim's browser load the page with the script tag previously injected. As the payload is run on the domain of the router, it can read and write anything a user could do via a browser. Currently, the implemented payload simply submits a form for setting the DNS server address to two Google DNS servers (8.8.8.8 and 8.8.4.4). Setting the DNS servers to ones controlled by the attacker could of course compromise the victim's security by impersonating other sites she visits in the future, for example allowing a phishing attack, posing as her bank requiring her login credentials.

Notably, the POC does not work when run in Microsoft's Internet Explorer or Edge browsers. This due to these browsers disallowing access to local addresses by default, preventing the script from making any types of request to the routers. Neither Mozilla Firefox or Google Chrome implements this type of protection.

Finding Router IP Address Candidates

In order to narrow down the list of possible IP addresses to try in order to find routers on the local network, the victim's own local IP address is found through some (ab)use of the WebRTC protocols. While the router (default route) must be placed on the same network, in order for internetwork communication to function over IP, knowing exactly where it is located or how large a network the victim is part of, is still unknown. It was assumed that the router would be more likely to be at one of the first (.1, .2, ...) or last (..., .253, .254) effectively usable addresses in a /24 net. The solution works also where many home "routers" are used together in the same network (however, strictly, only one of those actually function as a default route for the victim). In the case of the router examined, both were set by default to the address .1.

WebRTC is built on top of a number of standards that negotiate and establish a connection between two peers on the Internet, preferably in a Peer-to-Peer (P2P) configuration. Existing technology initially intended for Voice over IP (VoIP) and similar applications is reused, which leads to a flexible but, according to some [8], insecure-by-design solution [9]. The local IP address is made available to JavaScript through the use of the Session Description Protocol (SDP), a well-known part used in the Session Initialization Protocol (a classic VoIP protocol).

The negotiation of connection parameters used by the WebRTC data streams are managed by the ICE method (RFC 5245 [10]). This method tries to avoid using a middleman server (a TURN server) in the case that any of the peers are behind a NAT, using instead a STUN server for the negotiation of NAT-traversal connection parameters. STUN uses the aforementioned SDP in order to find suitable data stream and connection parameter candidates, which is more or less handed off to the JavaScript directly, as it needs to handle part of the connection itself with the current standard. The local IP cannot be hidden by the browser in this step, as some advanced functionality depends on having it. Such functionality is related to situations where the best path of communication within the network must be chosen, which is not necessarily the default one used by HTTP to get onto the Internet, and includes Quality of Service (QoS) and forced VPN tunneling.

The code used for this part of the POC builds upon the work of Daniel Roesler [11], which sets up a connection to a STUN server in order to negotiate connection details against a remote peer. Note that a connection does not actually need to be established, as the local candidates used are generated according to the needs of SDP, before the actual connection

[8] https://bugs.chromium.org/p/chromium/issues/detail?id=457492#c8

[9] https://bugs.chromium.org/p/chromium/issues/detail?id=333752

[10] https://tools.ietf.org/html/rfc5245

[11] https://github.com/diafygi/webrtc-ips

would be opened (in the normal case of using WebRTC). In the end, the result is that the local IP address(es) of the visitor can be determined.

After having obtained the set of local IP addresses, a first filtering is done in order to speed up the later device fingerprinting stage, as there is no need to fingerprint a device that does not exist. An XMLHTTPRequest (XHR) is performed against port 80 for each IP address, in order to see if it responds. Errors (other than timeouts) are assumed to indicate that the same-origin policy of the browser has responded that the request was forbidden, a result of actually contacting the server and seeing that the remote domain was not allowed to communicate with it due to the same-origin policy.

Running many XHRs in parallel did not work well, as the browser will only handle a few connections at a time, and put the rest in a queue. The current implementation runs the tests in a strictly sequential order. A 2 second timeout is used as a tradeoff between accuracy and performance. The current number of tried IP addresses are 9, leading to a maximum of 18 seconds delay for this stage. Fortunately, most home routers are very predictable with regards to this part of the address space. In order to improve the time of this stage further, using a WebSocket connection instead of XHR was tried, but it did not work as well. The reason for this is unknown, but it might be that these connections are throttled even harder.

Fingerprinting Routers

From the list of possible IP addresses identified in the previous step, the next step is determining which (if any) of these are in fact routers. Additionally, one would also want to try and extract as much information about the router as possible, so that the appropriate payload can be selected in the next step.

Both Netgear routers examined uses HTTP-basic authentication for protecting access to their web interfaces. However, it was discovered that the authentication is only enforced when accessing a server generated page. Static assets, such as images and JavaScript files, could be loaded even without any form of credentials. This (arguably faulty) design made fingerprinting these two models fairly straightforward. As embedding images and JavaScript from another domain[12] is allowed by the same-origin policy, the fingerprinting simply tried including a list of known images and scripts for each router from each IP. By checking the success of loading these files, an IP could be fairly strongly tied to a specific router, or discarded in the case that no list of known files matched the device.

Additionally, by creating a new IFrame element for each script to be included, the result that running said script had on the window object could be individually tested. In the case of the POC, this approach was used for identifying the Netgear WNDR3700 router. The WNDR3700 router has multiple scripts containing global variables used for multi-language support, each variable being assigned the value for some string in another language. Including one of these language mapping scripts allowed for much higher certainty of a

[12] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Cross-origin_network_access

correct identification, as the global translation values set by the script could then also be accessed and checked during the fingerprinting process.

The above method is implemented as part of the fingerprint “package” in the source code. The index.ts file in said package acts as the entry point, delegating each IP to each of the router fingerprinter implementations, and collects the results. The fingerprinter implementations are organized in folders for their respective manufacturer (currently only Netgear). The fingerprinter implementations are kept fairly small, instead using a shared FingerprinterUtil class for shared logic, such as loading images and scripts and checking the results.

Injecting The Payload

Given the information extracted in the previous step (i.e. router model, hardware and software versions), a method for injecting the payload for the specific model is selected. Injecting the payload is done via a mixture of CSRF and XSS, and as such, the vectors used are likely to vary significantly between different devices. For the WGT624v3 router, one such vector was found in the user editable list of blocked keywords. For the second router, a hidden value, part of the basic internet configuration, was found to be insufficiently filtered.

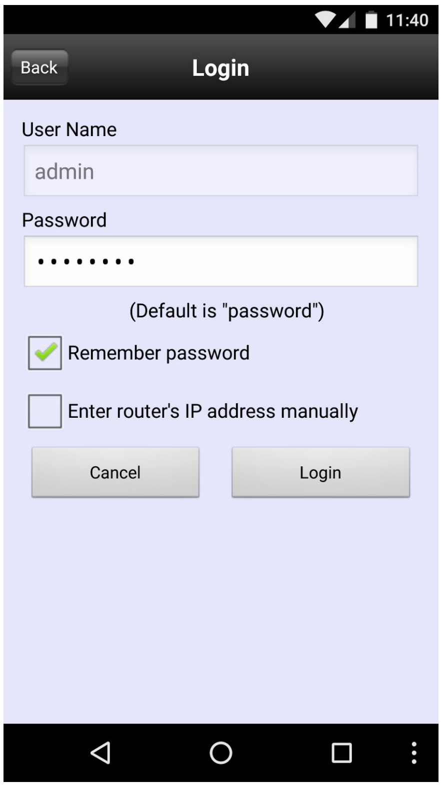
Of the vectors mentioned, the first one is far better suited for the purpose, as adding a content filter to the router is unlikely to affect its operation. Changing the hidden value on the other hand also required setting fields such as ISP connection information - information hard to blindly guess. Overwriting the current configuration with dummy data might in that case disable the Internet connection for the victim, which makes controlling her DNS servers fairly pointless. Disabling the Internet connection could also make the victim suspicious.

Having found a suitable XSS defect, the next problem was bypassing the HTTP-basic authentication that, as previously mentioned, guards each page of the router. By default, HTTP-basic authentication data is sent as headers in each request made to a site. However, due to the same-origin policy, it is not possible to set headers on a cross-origin request, unless the target origin explicitly allows it. The way of allowing such headers would be for the routers to implement CORS, which for a router does not make much sense.

Fortunately, both Firefox and Chrome allows specifying user and password as part of the URL. In combination with submitting a form, rather than using XHR, this method actually bypasses the need for CORS, as the browser sets the appropriate authentication headers automatically. Interestingly enough, this feature is disabled in Internet Explorer [13] and was also previously disabled for some versions of the Chrome browser [14]. As such, this part of the POC is perhaps one of the more likely parts to break in the future, should this feature be disabled again.

[13] https://support.microsoft.com/en-us/kb/834489
[14] https://bugs.chromium.org/p/chromium/issues/detail?id=123150

Finding the password is done via brute forcing, using a list of likely passwords. This list has to be very short, though, as each brute force attempt might take a second or more. Due to this limitation, the list is essentially made up of the default password and an empty password. This is one of the largest assumptions for the POC, that the victim uses the default password of the router. Arguably, this might still be fairly likely following Netgear's stance on security [15], as well as their notes on default password displayed in their router configuration app (see Figure 1 below).



User Name

admin

Password

•••••••••

(Default is "password")

[✓] Remember password

[ ] Enter router's IP address manually

Figure 1. Default password displayed on the Netgear Genie login screen.

The two aforementioned techniques, i.e. brute forcing and including the credentials in the URL of a form, still has one considerable problem. The result is unknown, as again the form is submitted to a different origin, and as such subject to the same-origin policy. However, note that on a successful form submission, the payload will be loaded into the context of the router via the XSS-vector, as previously discussed. Knowing this, each brute force attempt can in fact be validated by having the payload communicate with the main module via channels that are not subject to the cross origin policy. The channel used for the POC is by sending messages over via the postMessage method [16], which is specifically available for cross-origin communication. This requires that after each brute force attempt, the router page that is to contain the payload is loaded, which is done using a simple IFrame.

For the older WGT624v3 router, this is all that is required to both inject and run the payload. The same can not be applied to the newer WNDR3700 router though, as its settings are protected by what is known as a CSRF token. A CSRF token is some value included in the

[15] https://kb.netgear.com/app/answers/detail/a_id/1024

[16] https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

page by the server for each form, and is required to be sent back with the form for the data to be considered valid. This token should not be predictable from the outside, in theory making the previous attack nearly impossible. However, as described later in the CSRF Token Fixation section, it was found that the token created was far from sufficiently random to fully protect against such attacks. Due to the project time constraints, it was decided to only implement the injection part for the older WGT624v3 device.

The code for this part of the POC is located in the payload “package”. Again, as with the fingerprinting part, the index.ts works as the entry point. It in turn delegates to the implemented device-specific modules. In this case, as already discussed, this having only been done for the WGT624v3 router.

As an ending note, it is worth considering that the injection step could have been skipped altogether. Instead, one could have modified the router settings directly via CSRF. However, it was decided that using an XSS vector would mean the payload could more easily be changed to perform additional steps. This as having the script running on the domain of the router means bypassing security measures, such as CSRF tokens, since they can be read by the script within the same-origin policy. Guessing the CSRF token is then only needed when placing the payload, instead of for every request the payload might need to perform, likely improving reliability.

Payload

The payload constitutes the code injected and run in the web configuration interface of the router. The first thing the payload does is attempting to confirm with the main module that it should run. This is done by posting messages to the topmost JavaScript window object, which if the payload is run as part of the POC should equal the window object where the main module is loaded. The main module uses these messages as a way of confirming that the payload was successfully injected and loaded, and sends a go-ahead message to the payload, signaling to the payload to perform its actions.

The first action of the payload, after having verified that it should run, is making sure that the router is in fact the router it expects. This is done primarily as a safety measure, seeing as the previous fingerprinting step is fairly limited in what information it can obtain, due to the same-origin policy. For both routers investigated, detailed information about both hardware and software versions was easily available on the status page of the routers. This information is simply fetched by the payload via XHR, and the values are confirmed to be what is expected.

Knowing that the router is the correct version, the payload sends additional requests that changes the DNS server settings of the router. This part of the payload is very tightly tied to each specific router. For the older WGT624v3 router, the setting are changed by a single POST request to the request handler for basic router settings. For other devices, this step might require fetching CSRF tokens or performing other multi-step procedures. Notice that fetching CSRF tokens is possible, again due to the payload being injected via XSS on the router domain.

Only the payload for the WGT624v3 router is implemented as code. It is located in the "exploit" directory, and is compiled as a separate JavaScript file during the build process.

CSRF Token Fixation

The WNDR3700 device has a CSRF protection token used as a request parameter for sensitive requests, such as those used to set configuration parameters. It is intended to be used as a nonce, generated by the server (i.e. the router), and is allowed to be used only once. As a way of determining how this token was generated, the firmware for the router was acquired from the Netgear firmware update repository. The firmware utility Binwalk [17] was then used to identify the underlying architecture and extract the files found within it. It seemed like a purpose-built Linux distribution was used, together with some shell scripts and a httpd-executable containing the admin GUI server and its HTML. It was later noticed that the firmware source code was also available from Netgear's support site [18], although still containing multiple pre-compiled binaries (among them the httpd executable for the admin GUI server).

Several tools for disassembling or decompiling the binary httpd-executable were tried, but no really useful tool was found. The router in question used the MIPS architecture, which allows for very hard-to-read code compared to many other architectures. It might be that the code was heavily optimized or even obfuscated by Netgear, but if it was not, the community clearly needs better tools for analyzing MIPS binaries.

The binary was in the end decompiled into C code using RetDec [19], an online decompiler from AVG. It was noticed that it in many cases RetDec silently skipped assembly code it did not understand, which constituted a large part of the code. In this particular case, though, it intuitively gave a more clear view of what happened. A snippet of the code that was found interesting is supplied below, together with our comments, marked by /* and */. This function was thought to be interesting with regards to the CSRF token as it was the only function to call rand and srand.

```
// Address range: 0x4175d0 - 0x4176cf
int32_t function_4175d0(int32_t a1, int32_t a2) {
        int32_t v1 = g28 + 0xfbf3d90; // 0x4175d8
        int32_t v2 = g12; // 0x4175e4
        int32_t v3 = g11; // 0x4175e8
        g12 = a1;
        g11 = g57;
/* Seeds the random generator in libc by the current time. */
        srand(time(NULL));
/* Generates a random value, based on the random seed. */
        int32_t v4 = rand(); // 0x41761c
```

[17] http://binwalk.org/

[18] http://kb.netgear.com/app/answers/detail/a_id/2649/~/netgear-open-source-code-for-programmers-(gpl)

[19] https://retdec.com/

```
        int32_t v5 = g55; // 0x417640
        g3 = v5;
/* Some obfuscation follows (some unnecessary). */
        int32_t v6 = (int32_t)(0x17dc65df * (int64_t)v4 / 0x100000000) / 0x800000 -
(v4 >> 31); // 0x417650
        g4 = v6;
        int32_t v7 = 0x55d4a80 * v6; // 0x41765c
        g20 = v7;
        g30 = 0x989680;
        int32_t v8 = v4 - v7 + 0x989680; // 0x41766c
        g29 = v8;
        float32_t v9 = v8; // 0x417674
        g5 = v9;
/* Prints out the value (64 bit float containing a number) to a string. */
        snprintf((char *)(g11 - 0x3a30), 9, (char *)(v5 + 1084), (float64_t)v9);
        g9 = v1;
        g2 = g11 - 0x3a30;
        int32_t v10 = g55; // 0x417690
        g1 = v10;
/* The nonce is saved for a future request. */
        config_set(v10 + 452);
        g9 = v1;
        g2 = g11 - 0x3a30;
        config_set(g12);
        int32_t v11 = g11; // 0x4176b4
        g12 = v2;
        g11 = v3;
        return v11 - 0x3a30;
}
```

The above code was translated into a simplified version that was more easily usable when writing an exploit:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int main(int argc, char** argv) {
        srand(time(0)); // (Remember to adjust for time zone, e.g. +60*60)
        int v4 = rand();

        int v6 = (int)(0x17dc65df * (long)v4 / 0x100000000) / 0x800000 - (v4 >>
31);
        int v7 = 0x55d4a80 * v6;
        int v8 = v4 - v7 + 0x989680;

        float v9 = v8;
        char trunc[9];
        snprintf(trunc, 9, "%f", (double) v9);
        // (Printing v8 directly also gives the same output at the present time)
        printf("Token: %s\n", trunc);

        exit(0);
        return 0;
}
```

The router uses libc from the GNU Project, which in turn uses the ISO C implementation of srand() and rand() [20]. By specifying a seed to srand(), the rand() function can then be used to *deterministically* determine the next values in a pseudo-random sequence. It should be noted that for a specific random seed, the exact same sequence is generated every time. The implementation can of course be extracted out and should be simple to port to JavaScript.

The problem with this code is that the current time is not random enough to be hard to guess. This is especially true if the router time is configured to be close to the actual time, which arguably sounds fairly likely. No way of reading the exact server time was found, due once again to the same-origin policy (and the web server not listening on the router's WAN port by default, so a remote server of our control cannot get it either). However, it seems likely that the CSRF token could be bypassed by a brute-force attack, especially since it was noticed that the router did not generate a new nonce, even on requests that fails to pass the CSRF-token match test. Notice also that in C, the time function returns the number of seconds since the Epoch. As such, even a drift of a minute from the current time can be covered by roughly 100 requests.

As a way of verifying the above theory, the router was explored for a page including both a CSRF token and the current timestamp of the router. Fortunately such a page was available on the router, specifically the page displaying the router's logs. Using the simple C program above, taking the known current time as seed to srand(), it was verified that the output matched the CSRF token. In other words, it was confirmed that knowing the current time of the router would make it possible to entirely bypass the CSRF-protection token.

[20] http://www.gnu.org/software/libc/manual/html_node/ISO-Random.html

Result of Test Run

The POC was run successfully on a home network, managing to change change the DNS server setting of the WGT624v3 router. However, the POC is fairly sensitive to timing changes in the responses from the router, and as such does not always successfully execute. As the POC is built against two (and only fully implemented for one) specific router versions, a video showing a test run is available on YouTube [21]. Note that the video shows the script run twice, the first time unsuccessfully and the second one successfully.

[21] https://www.youtube.com/watch?v=MBjTZufinrA

How to Build And Run

The POC is, as stated, written in TypeScript, and must therefore be “compiled” before it can be run. Doing so requires Node.js [22] to be installed on the system. Node.js v4.4.4 was used for development, other recent versions should also work but these have not been tested. With Node.js installed, follow the instructions for building the production version as specificed in the README.md file in the root directory of the project. Once built, host the files located in the /dist/prod folder somehow, and navigate to the index file.

An already built version is also included in the submitted files (dist-prod.tar.gz) as the build process is a bit cumbersome the first time. There is also a version hosted online at our github pages[23].

The script will display a warning message and provide an “OK” button for starting the process. Before clicking the button, it is advisable to bring up the browser console window, as that is where debug output will be printed.

[22] https://nodejs.org/en/
[23] http://verath.github.io/router-hijack/

Discussion

“You can create a very secure network with NETGEAR products, but by default most security is turned off” [24]. This quote, directly from the Netgear support website, seem to quite accurately describe the lack of security focus when it comes to home routers. Our POC shows that even a fairly simple attack, based on XSS and CSRF, is enough to compromise essentially any setting on the router. As evident by the manufacturer’s stance on security, and their seemingly unwillingness to correct even major security flaws[25], it seems likely that router will continue to be a weak point in many networks.

There are of course ways to mitigate some of these issues. Perhaps the most important such advice to end users is is to configure the router with a proper password. Doing so would indeed prevent against the primary purpose of the POC we presented in this report. However, as shown, router fingerprinting might still be possible even without a guessable password. Another advice, that also applies to most software, is to keep the router firmware up to date. Looking through the firmware updates for routers, it is evident that we are far from the only ones to have discovered an issue in a router firmware.

Looking instead at what can be learnt for developing software, we once again see that implementing custom crypto, this time in the form of a severely broken CSRF token, is best to avoid. There are specialised libraries (and OS utilities) for these kinds of purposes, and we would highly recommend using those instead. In addition, seeding the random function is best done once, as opposed to before each use. Doing so makes predicting the values slightly harder, as the sequence would be unknown using a good generator.

Another issue that struck us during the project was the browser vendors’ stance on different security issues. As mentioned in the beginning of the report, it was quickly noticed that both of Microsoft’s browsers were not (by default) vulnerable to this type of attack. Additionally, some features required by our attack had been previously disabled in Chrome, but was then again re-enabled. This difference could perhaps be explained by Microsoft having enterprises as their primary users, whereas Chrome and Firefox are more targeted towards home users. By more strictly blocking features, such as access to private ip addresses, Microsoft might be more likely to prevent abuse. However, such preventions does come at the potential cost of breaking functionality. Which stance is more correct depends on the circumstances, but knowing what features are available or not should perhaps be given more thought when it comes to choosing what web browser to use.

[24] https://kb.netgear.com/app/answers/detail/a_id/1024
[25] https://github.com/darkarnium/secpub/tree/master/NetGear/SOAPWNDR

Future Work

The current implementation may be improved by performing the individual parts of the attack more in parallel, up to the point where the browser starts to throttle the amount of concurrently open connections. Also, actually implementing the CSRF mitigation technique should be fairly straightforward. The list of supported home routers might also be extended, where many of them probably share some of the weaknesses already demonstrated. As the POC is built as a framework to facilitate the automated testing of many devices, this should simplify the process and allow faster execution due to code sharing (both runtime and static).

Looking beyond the techniques demonstrated so far, there are other areas of improvement concerning the workflow of actually finding a fully working exploit for a particular router. We propose primarily two areas of improvement: automated testing and development of better tools. One path would be to use automated fuzzy testing, which would try to inject manicious code everywhere it could in order to break the application. Another, perhaps more burdensome, but also more accurate way, would be to write a usable decompiler to the MIPS code running behind the scenes. This would also, at the same time, allow for developing automated static analysis tools. The opcodes need a meaning.

Appendix A. Who did what?

We did everything mostly sitting together. Peter spent more time writing the POC script, because he found TypeScript fun. Joel spent more time on reverse engineering the router firmwares, because he found that fun.