EMBEDDED IN THIN SLICES

# The Internet of Things (Part 8)

## Security for Web-Enabled Devices

COLUMNS

In this installment, Bob addresses the top five security vulnerabilities in web-enabled devices. He covers how they affect real systems and what you can do to mitigate the risk.

*By Bob Japenga*

Today I got a message on Facebook from a friend highlighting a way that someone can steal your debit card PIN at the grocery store.[1] The idea is that your fingers warm up the keypad in the order that you punch in the PIN. And with a simple infrared camera that attaches to the back of your phone, the thief can get the keys pressed and the order in which you press them.[2] Ingenious!

This is what we are up against in creating secure Internet of things (IoT) devices. Incredibly ingenious hackers who will think outside the box to do whatever malicious activity that they choose. In the previous three articles, I presented three real-world examples (i.e., a car, a medical device, and a security system) and looked at the vulnerabilities of each. I also described how we can guard against such attacks. We need to understand the threats in a systematic way. But even if we close the known vulnerabilities, we need to protect our systems from these incredibly ingenious hackers who are coming up with some surprisingly simple methods of wreaking havoc.

Although we need to recognize the need for outside the box thinking, this month I want to step back and take a systematic look at threats to IoT security. Last month, I mentioned the Mitre website, which enumerates the kinds of security threats that can exist. This month, I'll introduce you to the Open Web Application Security Project (www.owasp.org), which is another resource for you to "tool up" on security issues. One service they provide is a list of the top 10 security vulnerabilities and specific real-world examples of systems that manifested these vulnerabilities.[3] **Table 1** is the current list. Let's look at the first five individually and make sure we know what they are. After we look at a real-world example where they have caused trouble, I'll explain how to prevent them.

## CODE INJECTION

Any web interface that allows unintended code to be introduced via the data stream and subsequently executed creates a code injection vulnerability. We addressed stack overflows in the April article in this series which can be used to create code injection. However, many systems allow simpler mechanisms than stack overflow (where you really need the source code to determine where on the stack your

injected code is located).

One example happens with the widely used SQL. If your IoT device uses SQL, certain features of SQL can be exploited by a hacker to inject code. A simplistic example would be when the user (the hacker) enters a username that contains an escape character (that tells the SQL parser to look for a new command) and an SQL command like `DROP TABLE`. SQL is so powerful and so easy to use that one can get blinded by the need to do proper input validation or range checking on certain fields that are not strongly typed.

The risk is real and can be costly. If the data that you are storing is worth something to you and your customers, hackers can and do target sites for ransom. Or they can show people how to use your data for free. One of the most widely published was the Code Injection hack that came to American Airlines and United Airlines.[4] Usernames and passwords were stolen via SQL code injection and the hackers booked flights for free.

The key to preventing code injection is to securely handle your input and your output. Certain APIs (rather than using raw SQA) provide this for you. If possible, whitelist your input data and be rigorous in enforcing the rule that only data that falls within your whitelist can be used as input. Make sure that no escape sequences get passed to the parser. With SQL, the `mysql_real_escape_string()function` will do that for you. Additionally, if your IoT device allows, you can make sure that code cannot be run in the data space. Also, as we talked about earlier, make sure that only authenticated code can actually be run. There are integrated circuit chips that can help you with that, which could be subject of a future article if there is enough interest.

## BROKEN AUTHENTICATION & SESSION MANAGEMENT

We have addressed this pretty extensively with real world examples in the other two articles so we won't revisit this in detail. Suffice it to say that username and password selection are critical to authenticating usage. We need to authenticate access to critical data as well as authenticating the software update process for our IoT devices.

## CROSS SITE SCRIPTING (XSS)

XSS vulnerabilities are really a specialized form of code injection. They can be classified as Server Side XSS or Client Side XSS. Server Side XSS occurs when the IoT web server injects a script into the client's browser with unintended results. A simple example of Server XSS would be if your IoT allows the user to add comments to your web page. If included in that comment was some executable script, when a user accessed your web page with that comment, if it was not validated or encoded, all kinds of havoc could happen as the script runs on the client.

Server side XSS is best prevented by creating context sensitive output encoding. The OWASP web site has a great check list for helping you do this.[5]

Client Side XSS vulnerability occurs when an unsafe JavaScript function is used to update the data on the browser. The kind of calls that are unsafe in JavaScript are those that allow other JavaScript functions to be added to the web presentation (called a Document Object Model, or DOM). The challenge for us as developers is not only to know which JavaScript functions are unsafe but which libraries contain these unsafe JavaScript functions (like JQuery). The OWASP website provides a PowerPoint presentation with just such a list of unsafe Javascript calls and JQuery APIs.[6] (Jquery is a popular JavaScript library: www.jquery.com.)

This is not for the faint of heart. This requires careful thought, understanding, and clean design. But the risks are immense. Imagine the embarrassment for the UK Parliament to find that a hacker could easily post any YouTube video on its very helpful site and make it look like it's coming from the government.[7] And all because their web server had an XSS vulnerability.

## INSECURE DIRECT OBJECT REFERENCE

A direct object reference is insecure when an IoT device provides access to some internal object like a file. If the design does not provide any authentication check, a hacker could manipulate these objects for destructive ends. An example of such a vulnerability was the Brightbox router supplied in the UK.[8] With virtually no effort, access to security

| Rank | Title |
|---|---|
| 1 | Code Injection |
| 2 | Broken Authentication and Session Management |
| 3 | Cross Site Scripting (XSS) |
| 4 | Insecure Direct Object Reference |
| 5 | Security Misconfiguration |
| 6 | Sensitive Data exposure |
| 7 | Missing Function Level Access Control |
| 8 | Cross-site Request Forgery |
| 9 | Using Components with Known Vulnerabilities |
| 10 | Unvalidated Redirects and Forwards |

**TABLE 1**
The top 10 means of web hacking


### ABOUT THE AUTHOR

Bob Japenga has been designing embedded systems since 1973. In 1988, along with his best friend, he started MicroTools, which specializes in creating a variety of real-time embedded systems. With a combined embedded systems experience base of more than 200 years, they love to tackle impossible problems together. Bob has been awarded 11 patents in many areas of embedded systems and motion control. You can reach him at rjapenga@microtoolsinc.com.


information is available by just using the URL and the file name (e.g. 192.168.1.1/cgi/cgi_status.js).

What this means is that someone who is on your home or office Wi-Fi network could, without authentication or authorization, get all kinds of details about your login on the LAN network. And you thought that you only gave them your WPA key!

The implications for your IoT devices are significant if it is designed like the Brightbox router. Your user could access a lot of files that you never intended for their eyes.

Most of these vulnerabilities can be resolved as part of your configuration of your web server on the IoT device. The Apache server has literally hundreds of configuration parameters to tailor access on your device. You need to understand what each actually does. And that leads to the fifth and final vulnerability that we will talk about this month.

## SECURITY MISCONFIGURATION

If your embedded web server or web browser is not configured correctly, you could be exposing your device to some significant security vulnerabilities. All of the major web servers have pretty complicated configurations. And if this isn't something you do every day, a lot will seem mysterious and mind boggling. It is important to go through all of the configuration parameters and make sure that you have it set to the security level you desire. Once you have it like you think you want it, you can test your web server or web browser (not as common on the IoT devices we design) on a test site like Qualys SSL Labs (www.ssllabs.com).

Let's walk through the configuration of NgInx (www.nginx.com), which is one web server that we use quite often on our IoT devices. Out of the box, NgInx is quite secure. But let me walk you through some steps that will make it even more secure.

Since much of what we use is open source, this means that hackers have the ability to discover soft underbellies of the software. But open source also means a plethora of versions. Although securing your headers doesn't ensure a bulletproof IoT device, it is a step in the right direction. Out of the box, a query of your web server will reveal something like this:

```
HTTP/1.1 200 OK
Server: nginx/1.3.0
```

Why give the hacker a head start? Change the `server_tokens` parameter in the configuration file to off and hackers will not know what version you are running.

```
HTTP/1.1 200 OK
Server: nginx
```

It would be great if we could eliminate the "nginx," but marketing has its ways.

Another header that you want to prevent from being disclosed is the:

```
X-Powered-By:
```

Again, NgInx doesn't allow you to not disclose this, but you can prevent the backend engine from disclosing it to NgInx. For example, if you are using PHP, you would turn `expose_php` to off in the php.ini file. Reducing the size of the headers

circuitcellar.com/ccmaterials

### REFERNCES


[1] Infrared PIN detector http://tiphero.com/scam-alert-how-to-keep-your-atm-code-safe-from-a-new-threat/?utm_source=fbbp&utm_medium=dk&utm_campaign=atm-scam-alert-dk

[2] Check out the FLIR One http://www.wired.com/2014/08/a-review-of-the-iphone-infrared-camera-the-flir-one/

[3] OWASP Top Ten https://www.owasp.org/index.php/OWASP_Top_10/Mapping_to_WHID

[4] United and American offer free flights http://www.nydailynews.com/news/national/thousands-american-united-airlines-accounts-hacked-article-1.2075162

[5] OWASP XSS Cheat sheet, www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet

[6] Power Point presentation which identifies unsafe JavaScript functions, https://www.owasp.org/images/c/c5/Unraveling_some_Mysteries_around_DOM-based_XSS.pdf

[7] UK Parliament Hacked http://www.telegraph.co.uk/technology/internet-security/10673520/Revealed-key-UK-websites-vulnerable-to-hackers.html

[8] S. Helme, " EE BrightBox Router Hacked—Bares All if You Ask Nicely," 2014," https://scotthelme.co.uk/ee-brightbox-router-hacked/.

| Parameter | Description |
|---|---|
| ssl_protocols TLSv1.2; | Since you are generally in control of your device, why not limit this to the latest (1.2). |
| ssl_ciphers | We have tested the difference between AES128 and AES256 and found very little degradation in performance with the higher encryption. Thus we favor EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH which define the key exchange algorithm and the bulk encryption algorithm (AES256). |
| ssl_prefer_ server_ciphers | This should be configured to "on" for maximum security since it forces the client to use the server's ciphers. |

**TABLE 2**
Some of the best parameters for the IoT environment

has the positive side effect of reducing the bandwidth. This is important to many of our designs which obtain very cheap data plans with under 1 MB per month.

You want to configure your system to give as little information as necessary. Most web servers, including NgInx include some pre-canned error pages for HTTP errors 401, 403, etc. Changing the error_page parameter to only disclose what you want disclosed to the user is a step in the right direction. For the most part, silently ignoring errors works for the client. You can still obtain the errors from the logs from most web servers if need be.

Out of the box, the NgInx's SSL is not configured as tightly as we would like. **Table 2** shows some parameters that we consider best for the IoT environment.

In many of our designs, the IoT device is only going to be talked to by our customer's server. Their customers will access the data only through that server. So it makes sense to limit the access to only your customer's server. With NgInx this is done with the allow parameter. This is an often overlooked security feature that can drastically improve the security of your device.

## FIRST LINE OF DEFENSE

Although it is important to pursue outside the box ideas for protecting the security of your IoT device, systematically addressing the major issues where others have failed is really your first line of defense. Next time we will look at the last five vulnerabilities from the OWASP website. But of course, only in thin slices.