Avoiding Web Application Flaws In Embedded Devices

Jake Edge
LWN.net
jake@lwn.net
URL for slides: http://lwn.net/talks/elce2008

Embedded Linux Conference Europe 2008
7 November 2008

Overview

Examples – embedded devices gone bad

Brief introduction to HTTP

Authentication bypass

Cross-site scripting (XSS)

Cross-site request forgery (XSRF)

SQL (?) injection

Additional threats

General web and application security

Examples

A-Link WL54AP3 and WL54AP2 – vulnerable to XSS, XSRF

BT (British Telecom) Home Hub – vulnerable to XSS, authentication bypass, XSRF

Cisco Linksys WAG54GS – vulnerable to XSS, XSRF

Xerox WorkCentre – vulnerable to authentication bypass leading to compromise

Router hacking challenge has many examples

Why embedded web apps?

- Typically used for configuration of the device
  - Easy interface that most people are used to
  - Small webservers are easy to implement or obtain
- Double-edged sword
  - Web apps have well known vulnerability types
  - Compromise can lead to a variety of bad results

Compromise can lead to ...

Changing DNS settings – phishing/pharming

Eavesdropping on traffic - network/VOIP

Devices can be repurposed – spam, distributed denial of service

In most cases, the owner will be completely unaware that it has been compromised

Theme ....

Web application security flaws are almost always caused by:

Trusting and/or not validating user input

Anything that comes from users or can be controlled by them is suspect

... and sub-theme

Developers often assume, incorrectly, that input to their web applications always comes from browsers

It is trivial to generate HTTP from programs

Javascript validation is *only* useful as a help to the user. It does **not** provide any server-side protection

HTTP – the web protocol

Browser sends HTTP requests and then displays the results (generally HTML/image)

Very simple protocol, with a few commands:

HEAD – get the headers and dates for the page

GET – get the html (or image or ...), parameters are part of URL (.../foo?id=4&page=3)

POST – encodes form parameters into the request which goes to a specific program named in FORM

There are more, these are the most common

HTTP Example

```
[jake@ouzel ~]$ telnet lwn.net 80
...

GET /talks/elce2008/ HTTP/1.1
Host: lwn.net

HTTP/1.1 200 OK
Date: Mon, 03 Nov 2008 02:32:14 GMT
Server: Apache
Last-Modified: Mon, 03 Nov 2008 01:23:47 GMT
ETag: "b08003-59e-45abecc6faec0"
Accept-Ranges: bytes
Content-Length: 1438
Connection: close
Content-Type: text/html

<html>
<head>
<title>ELCE 2008</title>
...
```

HTML Forms

These are the standard web forms that we fill in all the time:

```
<FORM METHOD=”post” ACTION=”some_action”>
    <INPUT TYPE=”text” NAME=”name”>
    <INPUT TYPE=”password” NAME=”password”>
    <INPUT TYPE=”submit” value=”action”>
</FORM>
```

The parameters (name, password) will get encoded and submitted as a POST

GET vs. POST

HTTP GET requests are not meant to change the state of the application

Classic example: http://somehost.com/delete?id=4

State changes should be done through POST

This is important to protect against trivial XSRF as well as innocent mistakes like the above

Exploits

Often requires more than one vulnerability to fully compromise a device

They often require user action to follow a link or visit a particular web page.

Particular device models can be targeted due to various monocultures (ISPs for example)

Authentication bypass

A variety of techniques to circumvent the username/password of a web app

For apps that check pathnames, aliasing can be a problem. Ex: /path/foo vs. /path//foo

When links to certain pages are only presented post-login, some believe this effectively protects them, but it is easy to guess/know the path

Avoiding authentication bypass

App must be coded such that each privileged page checks auth status whenever accessed

There are too many ways to get to the same page with different looking URLs

Attackers can purchase device to determine what paths are of interest

“Hidden” paths are security through obscurity

If separate program is used to perform the privileged operation, it must also check auth

Cross-site scripting (XSS)

Comes from echoing user input back to browser without properly handling HTML elements

Common mistake is to put user input into error message:

Unknown input <script>alert(“XSS”)</script>

Attacker controls Javascript sent by your app

Can be used to send cookie or other sensitive information to attacker-controlled sites

Avoiding XSS

The main defense is to filter all user input before sending it back to the browser

In particular, it is recommended that these characters: < > ( ) & # be filtered

< > (&#41 & # are substitutes

Usually the language has a function to call to do that for you: htmlentities(), cgi.escape(), etc.

Cross-site request forgery (XSRF)

User follows a link (from email, irc, ...) that quietly causes some action on a different site

For GETs that change the state, the page could have an <img src=deviceURL?DNS1=ip>

Cookies get helpfully sent along by browser

An iframe or other scheme to create a FORM and submit it to deviceURL/form, cookies too

deviceURL can be guessed (192.168.1.1?) for many devices

Avoiding XSRF

Do not use state-changing GETs

For forms, add a randomly named hidden field with a random value, associate those with a session and check them on FORM submission

If app is susceptible to XSS, random name/values can be extracted from forms

For extremely sensitive operations (changing password, others), require re-authentication

SQL (?) injection

Many embedded apps don't use a SQL db

SQLite, file based db being used more

Depending on how data is stored, similar techniques could be used

Abuses SQL queries with crafted data from form variables:

```
SELECT id FROM users WHERE name='$name' AND pass='$pass'
```

if `$pass` is: `' OR 1=1 --`

query becomes:

```
SELECT id FROM users WHERE name='$name' AND pass='' OR 1=1 --'
```

Avoiding SQL injection

Easiest method is to use placeholders in query:

```
db_call("SELECT id FROM users WHERE name=? AND pass=?", $name, $pass)
```

If database API does not allow that, use db-specific quote filter on user input:

```
db_quote($name)
db_quote($pass)
```

Depending on DBMS, stored procedures can also prevent SQL injection

Additional threats

Session hijacking – essentially auth bypass

Sessions that are restricted based on IP address are vulnerable to spoofing

Sessions that use cookies can have cookies stolen via XSS or other means

Sensitive sessions (that allow config changes for example) should be fairly short-lived

Denial of service – crashing the device or otherwise interfering with normal functioning

General web and app security

By default, web servers should only listen on local network, not the internet

All unused services should be disabled

There are Linux security tools that can assist in locking down webservers and devices:

SELinux

AppArmor

SMACK, Tomoyo Linux, grsecurity, RSBAC, etc.

Theme and sub-theme

Any input that can be controlled or influenced by a user (or *attacker)* must be validated carefully. Period.

Validation should use whitelists, not blacklists

Browsers do not generate very much hostile traffic, programs do. Expect the unexpected.

Javascript validation is **not** sufficient

Where to get more information?

My slides and some links
http://lwn.net/talks/elce2008

Wikipedia has some good information on the various flaw types

Open Web Application Security Project (OWASP)
http://owasp.org/

Questions?