Lecture 2: Web Application Security

Summary: The lecture deals with basic approaches to web application security as well as some practical examples of security failures.

Fuzzing

The lecture started by continuing the fuzzing theme of the previous lecture. To reiterate, 'fuzzing' refers to using intentionally awkward input in other to test how software behaves when not dealing with utopian data. Often, fuzzing will be done with a tool that uses sample data as a starting point to generate input that breaks observed patterns or commonalities.

When fuzzing input to test an application with, the amount of data will soon reach a point of diminishing returns. As the amount of 'new' data per generated data gets lower with larger amounts of generated data, it's not necessary to cover all possible input cases. Rather, there are some recommendations on how many output cases should be generated.

In the case of random bit flipping, a suggestion of 100,000 test cases was mentioned during the lecture. In addition, the MSDN article on fuzzing suggests that one maintain a 70% CPU load and measure a mean time between failures (MTBF). When the MTBF hits a limit, ranging from four hours to two weeks based on requirements, the test is suggested to be concluded.¹

For smarter tools, such as Radamsa, tens of thousands of cases were given as a likely optimal number. When working with highly specific data, such as file metadata, it was suggested that testing hundreds to thousands of cases was enough.

How web apps break

¹ http://msdn.microsoft.com/en-us/library/cc162782.aspx#Fuzzing_topic10
An application can be defined as two conjoined sets of behaviour. One, which could be considered the 'proper' program, is usually the behaviour which can be reached using valid and intended inputs. However, there is also the set of behaviour which is reached through malformed, malicious or otherwise unexpected input. For this, the term weird machine is introduced. The objective of fuzz tests is to reach the latter.

Typically, valid inputs are not executable program code, but data. The expected functionality interprets the data somehow, and lets it affect execution. The idea behind accessing the weird machine is to understand the effects of 'wrong' input, in a way where it can be considered executable data once its effects on program behaviour are understood.

An implied goal of software development is to minimize the effects of the weird machine, either by limiting the functionality of the weird machine through internal rigor, or by limiting the 'instruction set' of the bad input by limiting the form it may take. At this point, the developer also needs to define what constitutes the weird machine and what doesn't – the program might have functionality that has not been documented or specified but is intended.

**XSS**

The typical method of browser-web server communication, HTTP, is a stateless medium. It only transports data without knowledge of previous requests, usually in the format of HTML structure and content, and CSS presentation. As such, any dynamics in the presentation of data or transfer of data during a web page's life span must be allowed in some other way.

Enter JavaScript, a scripting language capable of modifying the document object model that determines how the HTML content is actually rendered, as well as handling flow of information between the web page and server without a full page load. It is complemented by the web server, typically also running some kind of dynamic web page generation method (as opposed to static web pages). In this way, both the browser and the web server are active parties in modifying content.
Due to the prevalence of JavaScript in virtually all browsers, it is also a large target. One common issue with JavaScript code is cross-side scripting, also known as XSS. In this attack, JS is injected into the web page, either to modify how it's presented to the user or quietly accomplish some other task. Essentially, some modification takes place which was not intended by the web server nor the browser.

To help defuse XSS, browsers use the same-origin policy. This policy dictates that the origin is defined by a triplet (protocol,host,port). In theory, data such as cookies should be restricted only to the origin from which they came. Thus, for a practical XSS attack to work, it must appear as if it came from the same source as the web page itself. In practice, the browser can be fooled by circumventing the restrictions.

There are two main ways to pull this off. In a reflected XSS attack, malicious JavaScript is somehow included in the browser's request for a web page. For example, it might be a GET parameter, or a script stored in the cookie. This method requires that there is also some way in which the script will reappear in the delivered web page, in executable form.

The second way is the persistent (or stored) XSS attack. Arguably worse, this attack sees the server store the malicious script and serve it to some or all users who access a particular page. For example, a badly handled forum post might simply dump its contents into the page where it's displayed and thus inject executable scripts into what should have been normal data.

**SQL injection**

An SQL injection is an attack that exploits flaws in how a web server accepts data and inserts it into an SQL query. If the query is neither properly done nor properly escaped, the semantic structure of the query can be affected by blindly inserting user data in the middle of it. This is due to SQL building the entire query string before parsing it when used naively.
The two main methods of defense versus an SQL injection are the aforementioned query execution and query escaping. The former is normally done by relying on prepared statements, where the parameters are given to the DBC separately from the body of the query. This way, the DBC is aware of what exactly constitutes the parameters. In the latter method, attempts are made to escape dangerous characters and keywords from the input before inserting it into the query, in the hopes that it would prevent them from affecting query execution.

**Validation**

When validating data taken as input for a program, there are two separate stages to consider. Between the points of accepting input and interpreting it, the program is better served by doing strictly semantic analysis (a sanity check, if you will) of the input. After interpretation, if some data is about to be saved, validation is more focused on escaping data, preventing it from having an unintended effect when regurgitated for use later (for example, escaping HTML tags). In this phase, one still needs to keep in mind what the program is intended for. If the data is meant to be emailed out, escaping email control characters is a good idea, while doing an HTML escape might be useful, and escaping SQL characters is just pointless (unless one is emailing SQL queries, in which case there may be more problems than escaping data).

In most programs, input is not something that can be easily defined. In a very fortunate situation, one might be able to use a regular expression or context-free grammar to match against the input. Often, however, these will not do. One solution is to use source/sink analysis, tracking at runtime whether data that has been affected in some way by user input reaches the database before it has been sufficiently sanitized. Data that has been modified by user input without or after being subject to a function labelled as sanitizing is considered tainted. The goal of the analysis is thus to make sure that anything that is stored is untainted.