Lecture 2: How Web software breaks

These notes are roughly based on a lecture of Software Security-course on March 19th 2014. Author: Sami Koskinen.

The Web and its interactive applications have become a very important part of information technology. The modern Web applications have become very complex and feature-rich, and at the same time, they will become more interesting targets for attackers. At points where the apps interact with users, attackers can execute injections targeting vulnerabilities in other users’ browsers, Web applications theirselves or backend systems of applications, including databases. Input validation, sanitation and escaping can be used to prevent and mitigate effects of injection-based attacks on Web applications.

History of Web applications

The World Wide Web is currently the most used service in the Internet, and the two terms are even used synonymously. Originally the Web was intented for transfering hypertext, linked documents, but more and more functionality has been implemented in Web browsers. Web browsers and applications have become very feature-rich and ubique, and they can fill the same needs and deliver almost as good an experience as native applications.

Web uses HTTP protocol as its application-level transfer protocol, and the markup language for representing static documents is called HTML. Static and dynamically generated HTML, CSS, JSON and JavaScript files are transferred using HTTP. HTTP is stateless request-response-based plain text protocol. Statelessness mean that every request is responded independently. With some applications, a derived protocol called HTTPS, which provides encryption over HTTP, is used. A feature called cookies have been introduced to maintain state information.

Originally the Web was used to deliver hypertext documents to users. The documents were written in HTML, which was interpreted and rendered
by Web browsers. In the early days, browsers did not support altering the Web page after loading, and all the interactivity had to be implemented by deploying scripts on the server. The scripts usually used, and current Web sites keep using, databases as their backends. The scripts provide dynamically changing HTML to users based on their request parameters.

In 1995 the first implementation of JavaScript was released with Netscape Navigator. JavaScript is a scripting language, which is used to alter the document displayed in a Web browser. Virtually all current Web browsers support JavaScript. By using JavaScript, some of the interactive functionality could be implemented on the client side. Web browsers represent the document as a mutable structure called DOM (Document Object Model). DOM is initialized with HTML and CSS, and the resulting DOM can be modified using JavaScript.

Web pages keep getting more complex, and currently Web applications can possibly be as interactive and complex as native applications. There are even word processing and spreadsheet Web applications available. Because Web applications have more and more functionality and importance, they have to endure increasingly diverse and more frequent attacks.

**Injections with current Web apps**

The execution environment of an Web app and a native app may appear to be different from each other. Native app has been compiled to lowest-level assembly language and is executed in the CPU, while browsers are running scripts written on a high-level language. However, the concept of injection is fundamentally same in both scenarios.

In a traditional binary injection, user-provided data will be interpreted as machine instructions, and run on the CPU. In higher-level Web scheme, the data will be interpreted as a higher-level language like JavaScript or SQL, and run on a another user’s browser or the backend system of a Web site.
Outcome of both schemes is exactly the same: the attacker will be able to make the application function in an unexcepted manner.

An important part of a Web application’s security model is the same-origin policy. The policy is based on a concept called origin, which is defined by scheme, host and port parts of an URL [2]. Intent of the policy is that documents and scripts retrieved from one origin are not allowed to view or modify contexts with different origins. If a script downloaded from URL beginning with https://www.scam.cm/ tries to modify DOM of a page downloaded from http://www.scam.cm/, the browser is supposed to block access because scheme did not match.

Cross-site scripting (XSS) is an injection problem, in which an attacker inserts his own code to another Web page [1]. Other users’ Web browsers will execute the malicious script snippet in the context belonging to a trusted site. By running the script in another context, the attacker may be able to access data it was not supposed to access. Based on the injection vectors used, XSS attacks can be categorised into two categories: stored and reflected XSS.

In stored XSS, the attacker will try to store the malicious script in the backend of a Website, for example by posting it as a message in a discussion forum. If the Web site does not succeed to sanitize and escape the data and does not refuse the malicious request, the injected script will ultimately be
Figure 2: Input interpreted as code may cause unexpected functionality.

run in other users’ Web browsers as its origin was the Web site used in the attack [1]. Stored XSS is also known as persistent XSS.

In reflected XSS, user is made to send a request containing the malicious script to a server [1]. If the server returns some contents of the request to be inserted in DOM, script may get run in user’s browser in an incorrect context, and the attack will succeed.

SQL injections are another common type of attacking Web applications. Like script injections, SQL injections are fundamentally similar to other types of injections. The SQL injections, unlike XSS injections, are run on the Web app backend. Most Web apps use a relational database as a part of their backend, and the database responds to queries written in SQL (Structured Query Language). In an SQL injection, attacker provides SQL statements embedded in data fields of a request, which will be sent to the Web server. If those statements will be executed in the backend database server, the attack
Figure 3: Stored XSS may propagate between users through a database. Malicious user uses a request to store his code to the database, and later, the server will forward the exploit in part of a response to another request. The malicious code will be executed in a context belonging to the legitimate server.

In a trivial example of an SQL injection, the attacker has discovered that the system will insert user input directly to a SQL SELECT query, which is supposed to return a result set matching to user-specified parameters from the backend DB. In that case an attacker could provide "; DROP TABLE Foo;--" as his parameter, which would then cause deletion of table Foo from the running DB. The deletion could cause data loss and downtime for the service.
Mitigating injections

Injections can be mitigated by handling input data with great caution. Good practices include sanitizing and validating inputs, and escaping data, which gets passed to another components or returned to users.

Input sanitization and validation seems really hard to be done right. The main approaches are detecting input patterns known to be harmful (blacklisting) and restricting inputs to a set assumed harmless (whitelisting). Neither approach works by itself, and a successful approach to input validation should include both sides. A good guideline is to not trust any external data, and use several approaches to ensure safety or all received data [4]. The three most lucrative ways of doing it could be input restriction, input tainting and scaping, respectively.

If there is a possibility to restrict the set of valid inputs, it would be the most lucrative approach to input validation. If the valid input looks like a programming language, it is likely too hard to validate and too easy to exploit. A good set of valid inputs would be something that can be expressed in a Backus normal form, or validated with simple regular expressions.

Some programming languages like Perl and Ruby contain a feature called input tainting. In input tainting, a procedure called taint checker marks all the variables that can contain user-modified data. The programmer can review and find statements, where user data is used in dangerous contexts like SQL statements.

The user input should be escaped when it is put out, and proper encoding for the target environment should be applied. Many Internet sources confuse escaping and validating, and thus suggest escaping input in unnecessary ways or places. As a result, unnecessary escaping could introduce extra escape characters or mutate harmless characters. Escaping should be done only, when the target environment is known - for example before database or HTML insertion.
Most of the current Web frameworks, including Django, have features for mitigating SQL injections. In Django, if models and querysets are used, the Django database driver takes care of escaping SQL. However, developers can still write possibly dangerous raw SQL queries [3].

Session stealing

HTTP is a stateless protocol. The HTTP server does not maintain state between HTTP requests, but a typical use case of a modern Web app will require maintaining of state information between requests. Usually cookies or hidden input fields in Web pages are used to implement stateful features, like shopping carts, to Web applications, and the storage for state is called a session. However, the stateless nature causes a need for Web application developers to implement the session functionality or choose from several middlewares offering support for them.

The session functionality is critical for safety, because session information is sensitive and it can be misused. If the integrity protection of session information has flaws or execution of a man-in-the-middle attack is possible, an attacker could invade another user’s privacy or even use their identity in the Web application. Encryption can be used during transfer and storage of session information [4].

References

